Fault tolerant filter implemented using N-modular redundancy

Project – Fault tolerant systems

Miloš Arbanas

Table of Contents

1 Introduction

The goal of the project is described in following section.

Create a fault-tolerant FIR filter using the N-modular redundancy hardware fault tolerance approach. Let the FIR filter be a 10^{th} order filter, and the number of modules in the redundant system, N, should be equal to 5.
VHDL model of the FIR filter must allow the user to specify the values of the filter coefficients (b_i) as well as the location and the type of the fault that is present in the filter (if any). Use the following block diagram to develop the VHDL model of the filter.

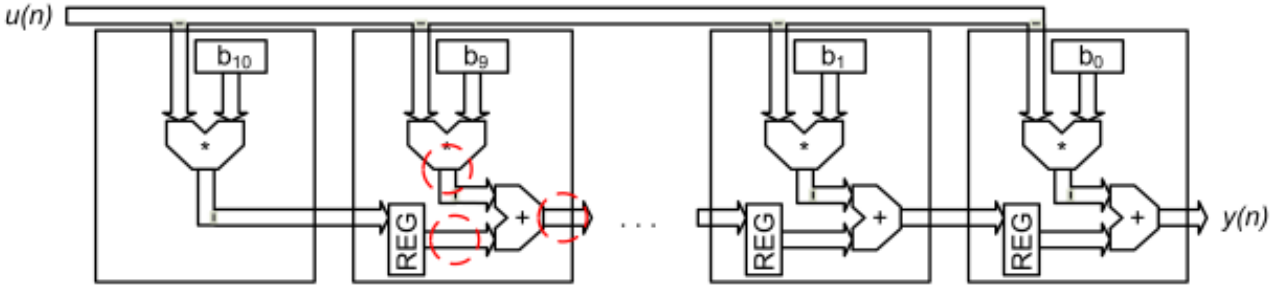


Figure 1. Architecture of the 10^{th} order FIR filter

Locations of the possible faults that should be modeled are highlighted by the red circles. Please note that faults in these locations can be present in every module that makes the filter.
When developing the fault tolerant FIR filter the N-modular redundancy hardware fault tolerance approach should be used. Fault tolerant architecture that should be developed is presented in the Figure 2.

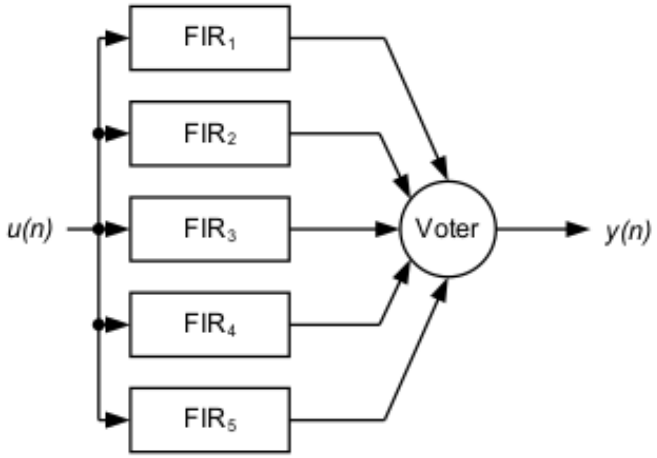


Figure 2. Fault tolerant FIR filter architecture

Fault tolerant architecture has 5 identical FIR modules that can be switched on and off depending on the decision that is made inside the voter module.

Tasks that must the done in order to finish the project:

1. Develop a VHDL model of the 10th order FIR filter based on the architecture in Figure 1 that enables the user to specify the coefficient values and to insert the fault at the desired location inside the filter.
2. Develop a VHDL model of the fault tolerant FIR filter system based on the architecture in Figure 2.
3. Develop a testbench that will test the operation of the fault tolerant FIR filter architecture in case when faults are inserted randomly into the system. Simulation should clearly illustrate the operation of the fault masking mechanism that is the basis of the N-modular redundancy hardware fault tolerance approach.

2 Methodology

First, the **ordinary filter** should be developed, then **fault modeling** should be added, and, at the end, **redundancy** to detect and handle faults.

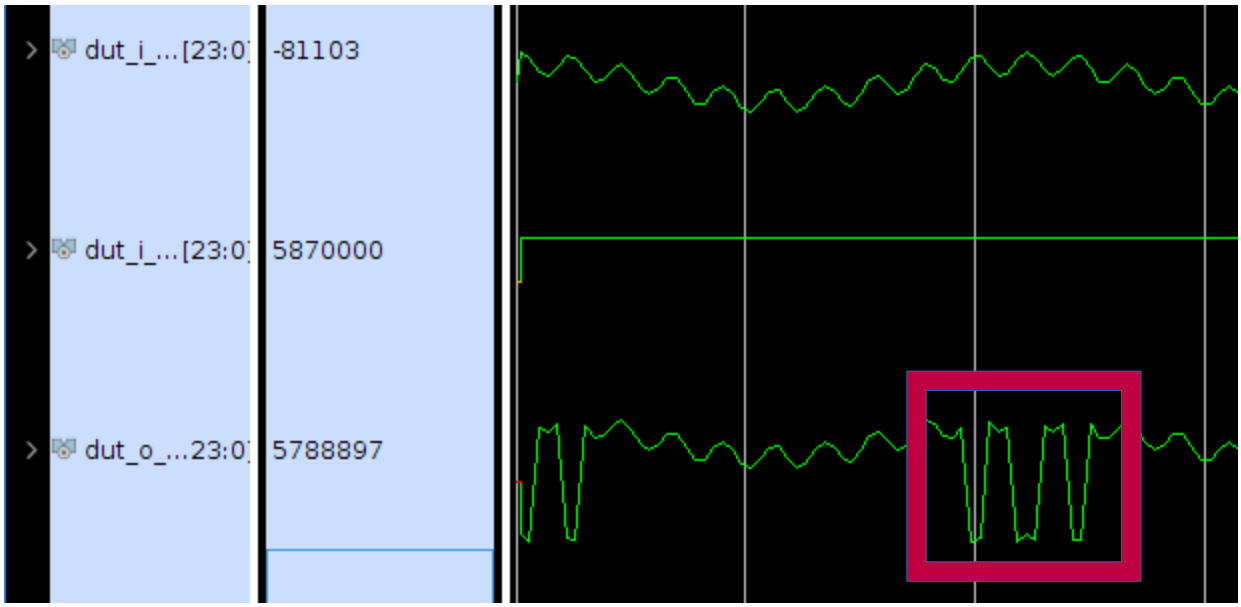
2.1 Developing VHDL model of a generic order FIR filter

This is fairly simple and only few notes will be taken.

Data representation used for calculations is **Fixed Point Q1.23**.

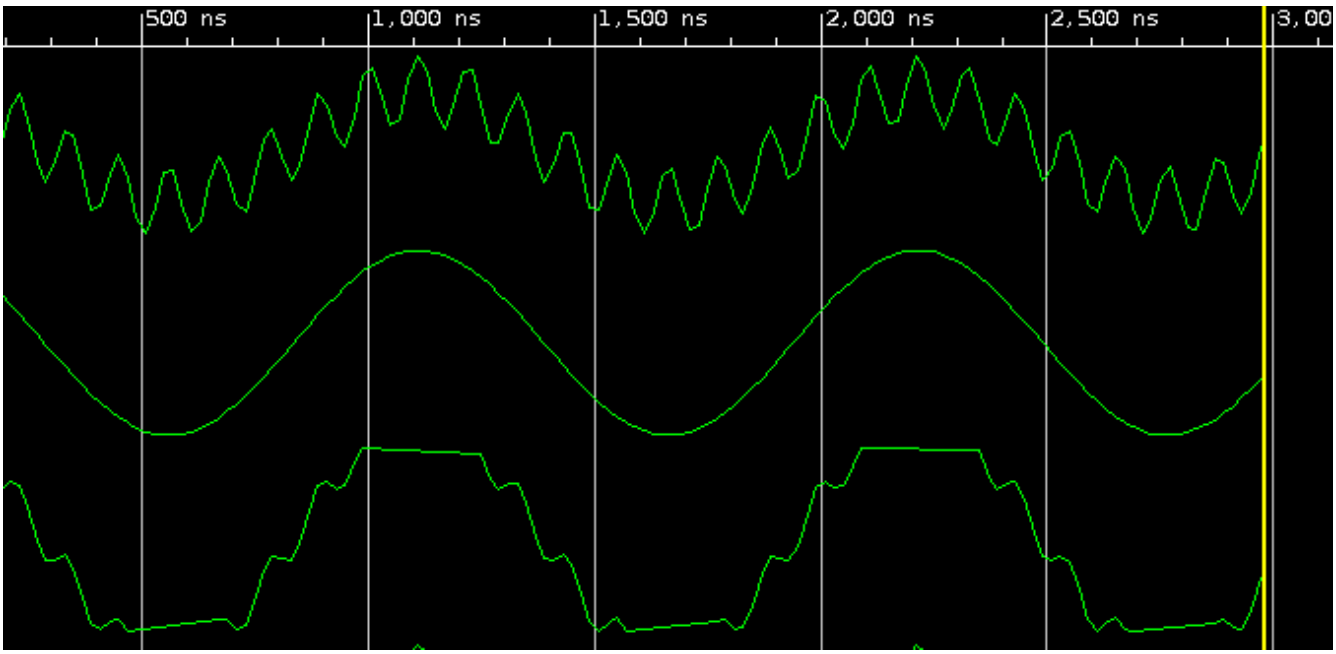
Both additions and multiplications are done using FPGA DSPs. In Xilinx FPGA these are **48 bit DSPs**.

Addition: we should be aware of the possibility for Overflow to happen. Like on the picture below:



Simplest way to handle this situation is to saturate output of a adder.

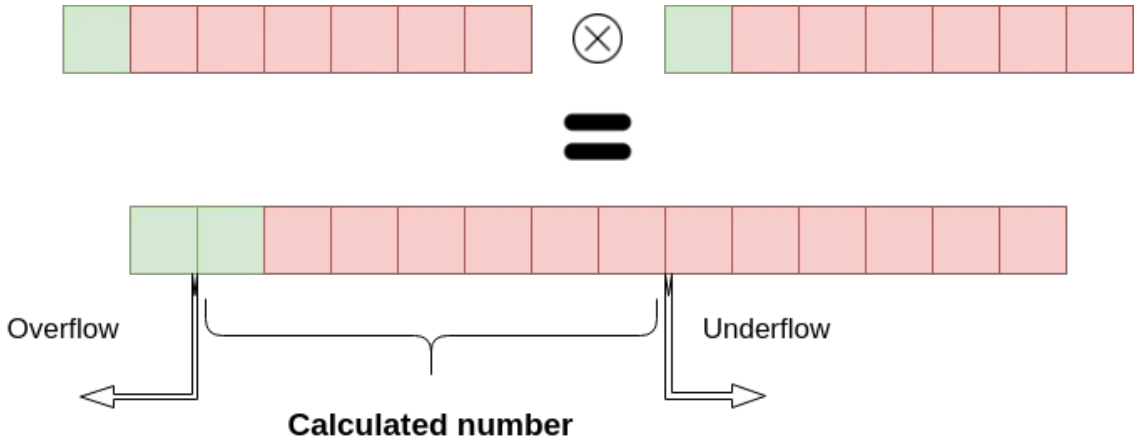
Here is the picture with saturation:



Multiplication:

All of the numbers are less than one, so there should be no problems.

Multiplying 2 x Q1.23 numbers gives Q2.46 number, and than one should get last bit from Whole part and first 23 bits from decimal part. Like shown in picture:



2.2 Verification strategy

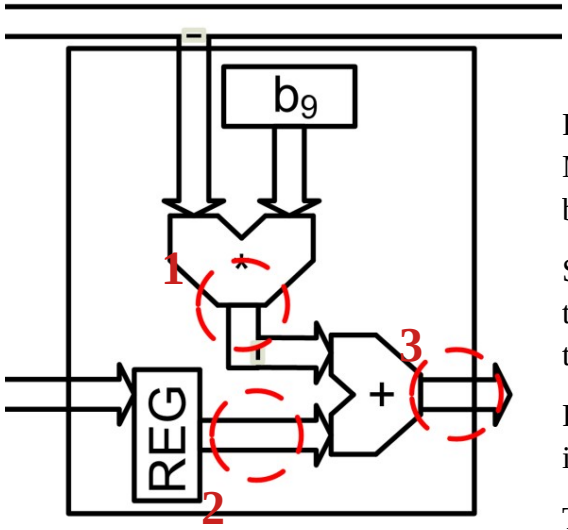
For the verification of the whole filter, the signal is going to be fed into the filter (slow sinusoidal plus faster sinusoidal). Output of the filter should pass just slow sinusoidal signal.

There is going to be referent signal from the Octave, and “assert” will be written to check if the difference between output of the filter and referent signal is under 5%.

If the difference is higher than 5%, the ERROR message is written in report.

2.3 Fault modeling

In order to test fault tolerant design, there is a need of inducing artificial faults into the system.



From this picture we can see that outputs of a Multiplication component, Adder and Register should be able to emulate Faults.

So normal components should have additional inputs that enable injecting of the fault instead of providing the calculated value.

First idea, for **behavioral fault** modeling, is to add input that will be forced to the output of the modules.

This faults emulates error in calculation.

Another idea is to try to emulate **Structural fault** modeling. Although we don't have control of the system at logic level, we can emulate that by inducing fault on behavioral level that resembles faults that commonly happen on structural level.

We can either flip some bit (modeling "soft", transient, faults that mostly environment causes).

Or we can model Stuck-At Fault by holding some bit to be always "1" (SA1) or "0" (SA0) which commonly represents faulty silicon (most frequently transistors) and it is probably permanent fault.

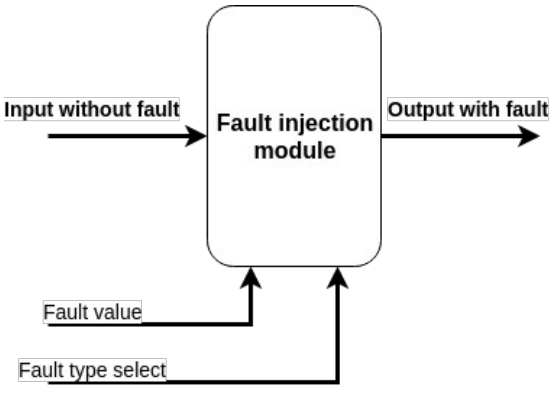
In order to work at the behavioral level, the new component **Fault injection module** will be introduced. Outputs of adder, multiplier and register will be passed through this module.

There are 10 MAC modules in the filter, and each module can have 3 different places where the fault might occur. So there are 30 different places combined.

Fault injection module

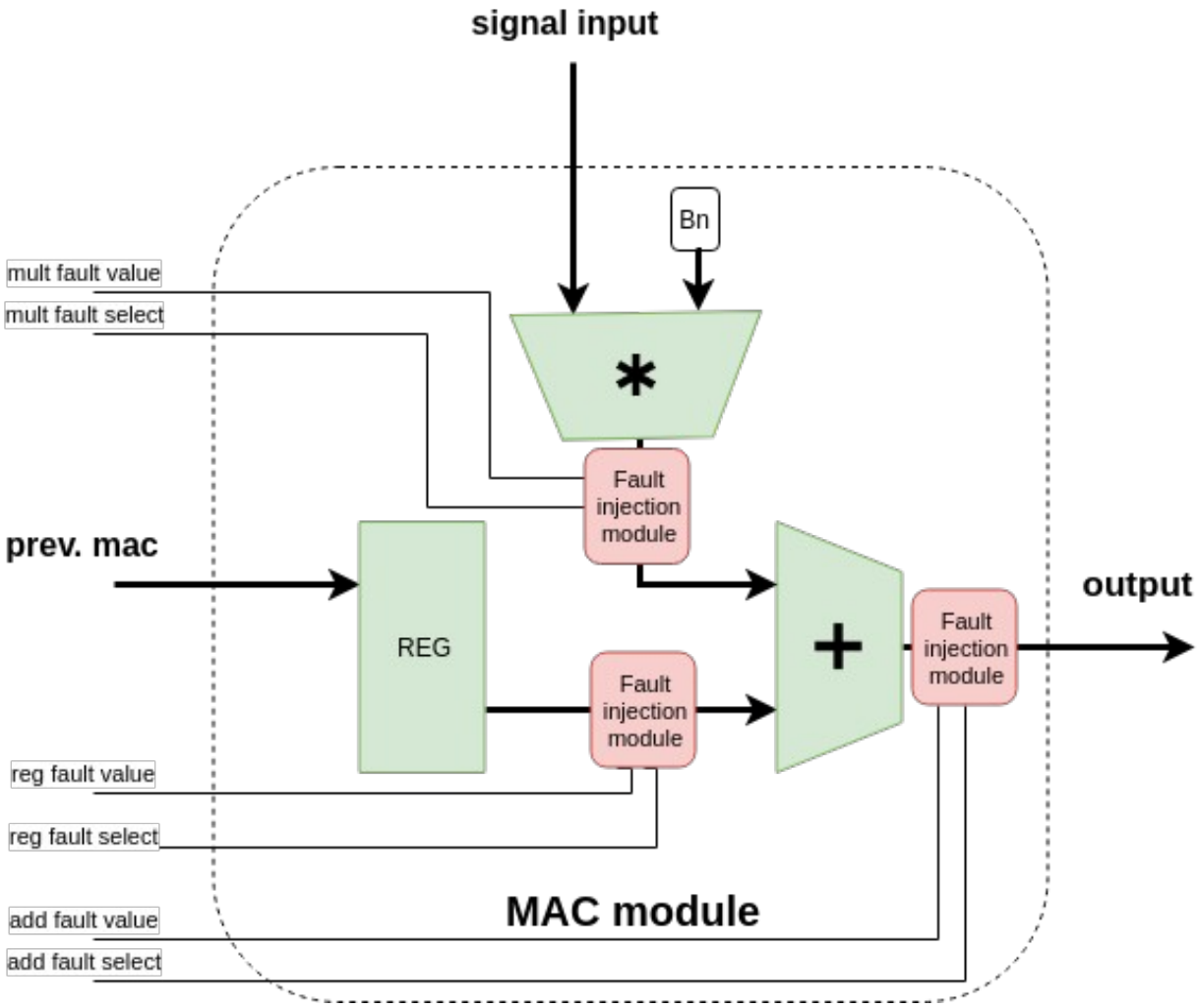
This module changes data in order to induce a fault of different types. **Fault value** gives the input vector that is used and changes input signal as described:

1. behavioral fault - output is equal to fault value
2. flip bit fault - flips bits where fault value is "1"
3. SA1 - sets bits to "1" where fault value is "1"
4. SA0 - sets bits to "0" where fault value is "1"



Fault type is chosen with **Fault type select** input, by indexing fault type shown above.

Final component for fault injection is shown above and whole MAC might look something like this:



Fault injection methodology

The next important question is how to index where to place the fault???

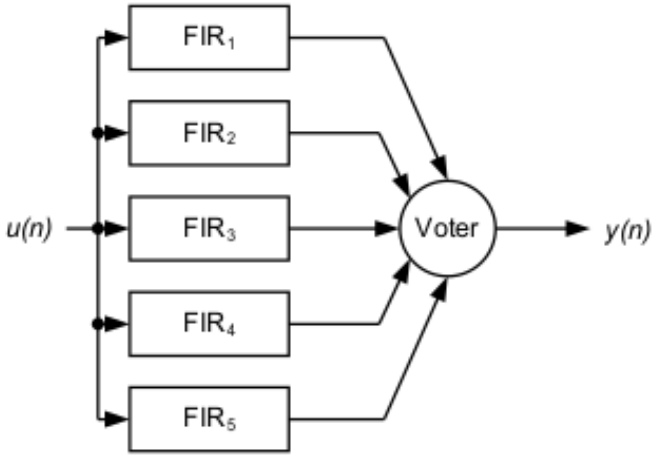
There are 5 Filters, each with 10 MAC modules and 3 potential faulty modules. So there are 5*10*3 = 150 places where faults might happen.

So one fault should be indexed in which filter and in which MAC is particular fault occurring.

ModelSim can force signals in MAC, so we can index each individual reg, mult or add fault.

For example if we want reg fault in 8th MAC of 1st filter we would force signals reg_fault of FIR(1)/MAC(8) using ModelSim's signal_force function.

2.4 N-modular redundancy



Next element that should be implemented is Voter. The voter is implemented in combinational circuit that satisfies following table (if there are 3 or more ‘1’ or ‘0’ then output is ‘1’ or ‘0’ respectivelly :

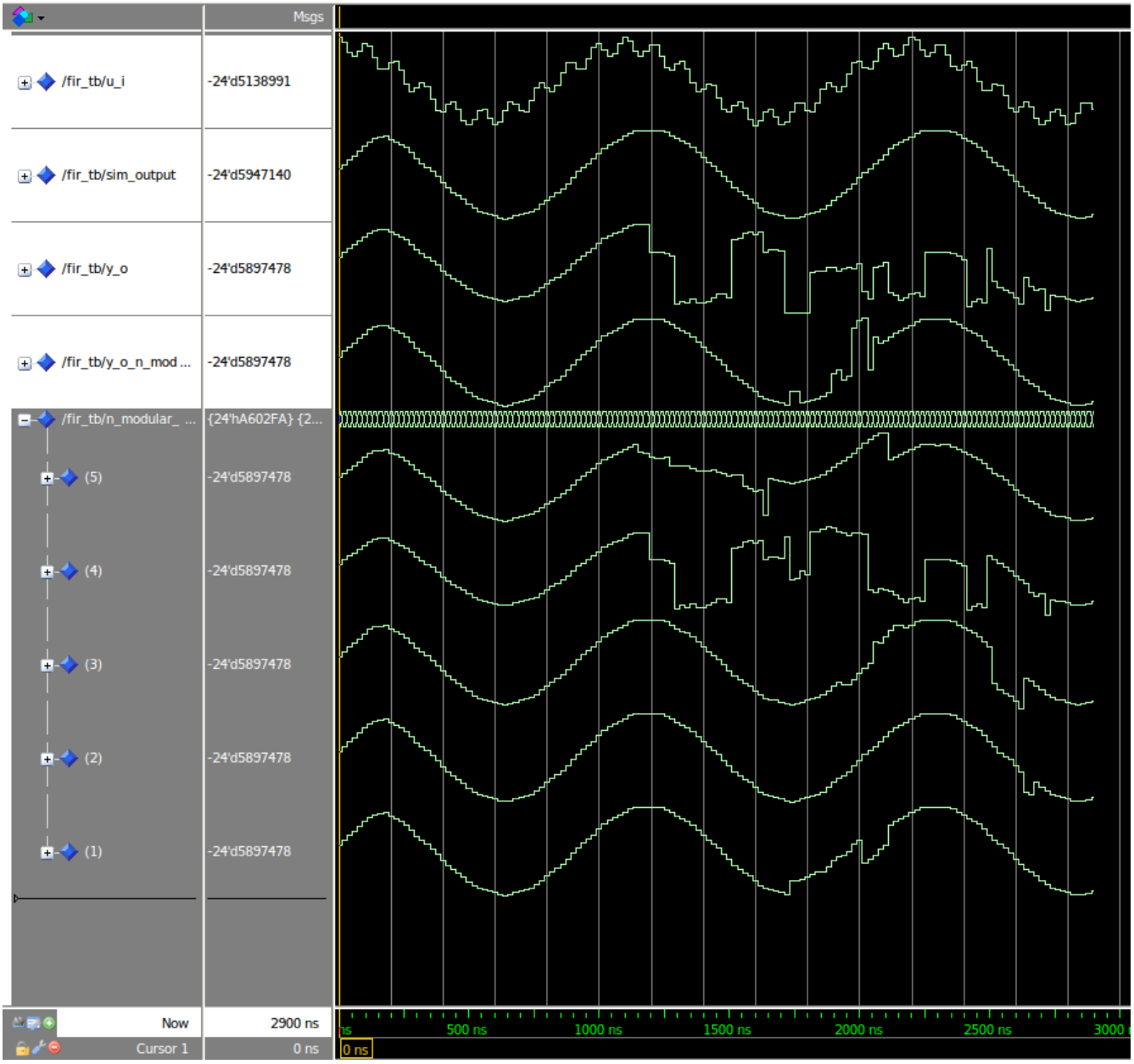
i_u1	i_u2	i_u3	i_u4	i_u5	o_majority
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	1	1	0
0	0	1	0	0	0
0	0	1	0	1	0
0	0	1	1	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	0	1	0
0	1	0	1	0	0
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	0	1	1
0	1	1	1	0	1
0	1	1	1	1	1
1	0	0	0	0	0
1	0	0	0	1	0
1	0	0	1	0	0
1	0	0	1	1	1
1	0	1	0	0	0
1	0	1	0	1	1
1	0	1	1	0	1
1	0	1	1	1	1
1	1	0	0	0	0
1	1	0	0	1	1
1	1	0	1	0	1
1	1	0	1	1	1
1	1	1	0	0	1
1	1	1	0	1	1
1	1	1	1	0	1
1	1	1	1	1	1

Using Logisim, the minimized formula is:

o_majority = i_u3 i_u4 i_u5 + i_u2 i_u4 i_u5 + i_u2 i_u3 i_u5 + i_u2 i_u3 i_u4 + i_u1 i_u4 i_u5 + i_u1 i_u3 i_u5 + i_u1 i_u3 i_u4 + i_u1 i_u2 i_u5 + i_u1 i_u2 i_u4 + i_u1 i_u2 i_u3

The Voter is implemented easily in VHDL with this in mind.

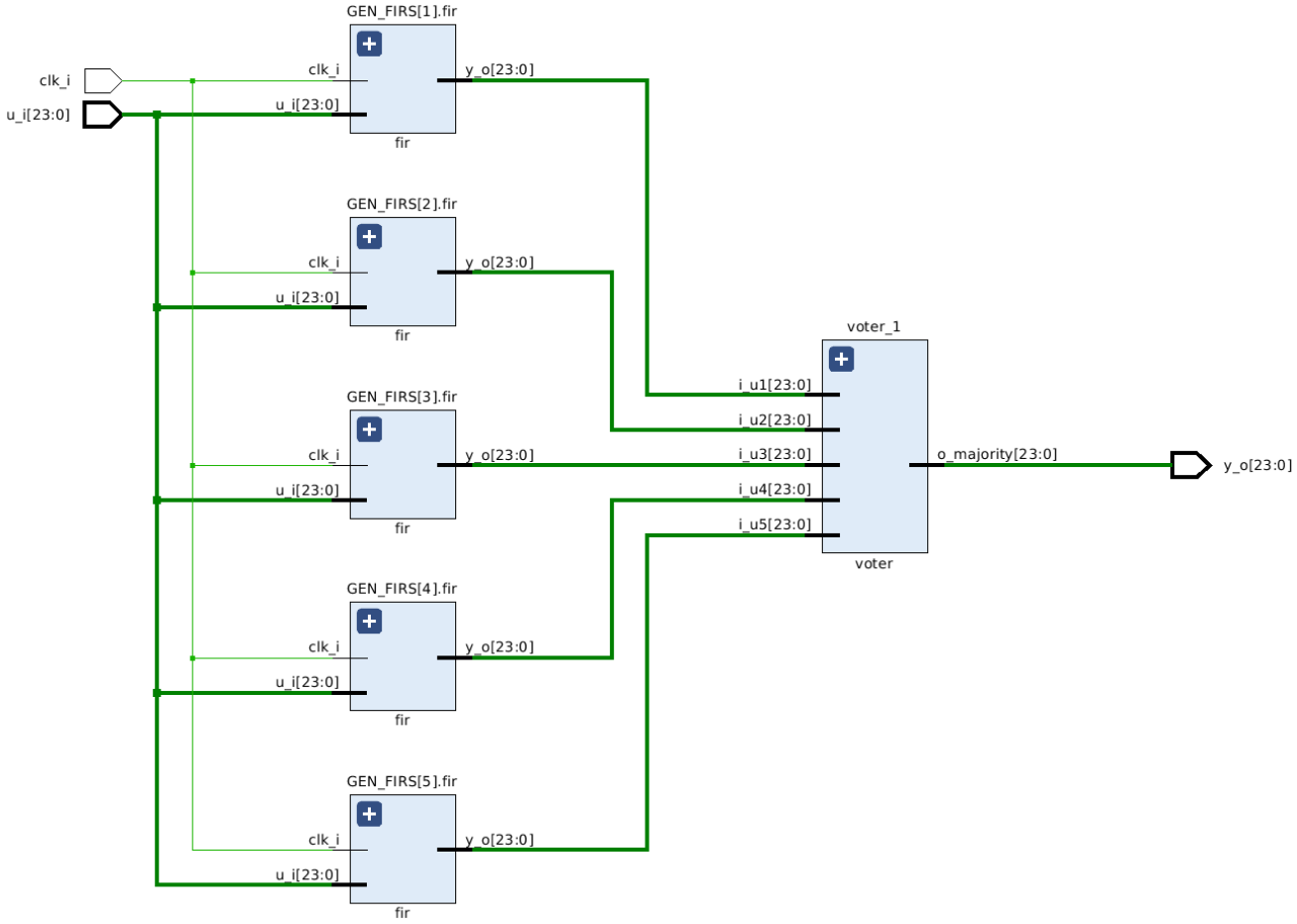
3 Results



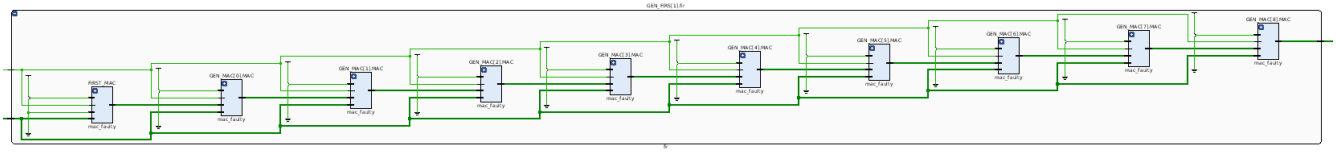
We can see that up to two FIRs can fail, and the system masks them.
If there are 3 modules that have some fault, than the system just passes result from FIR1.

4 Appendix 1 – Schemes from Vivado

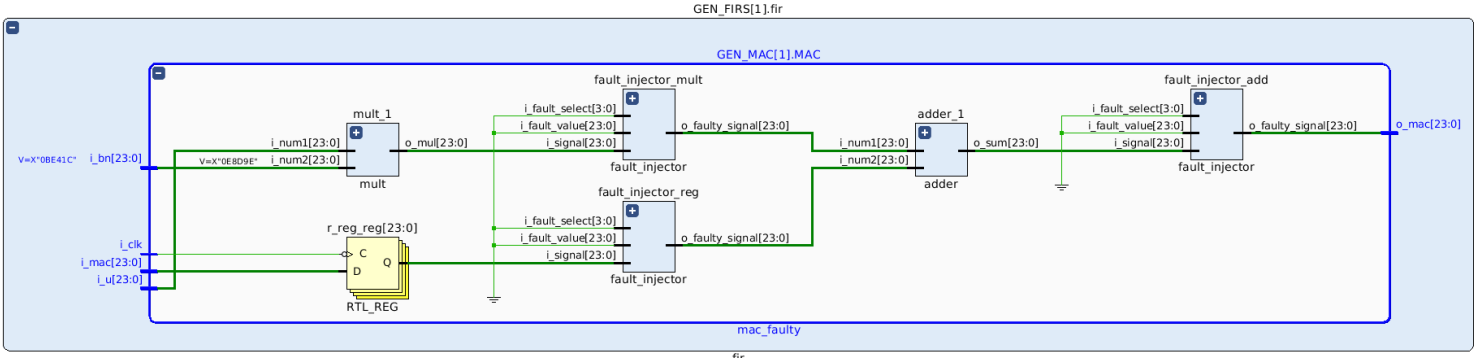
4.1 Top level N-modular redundancy



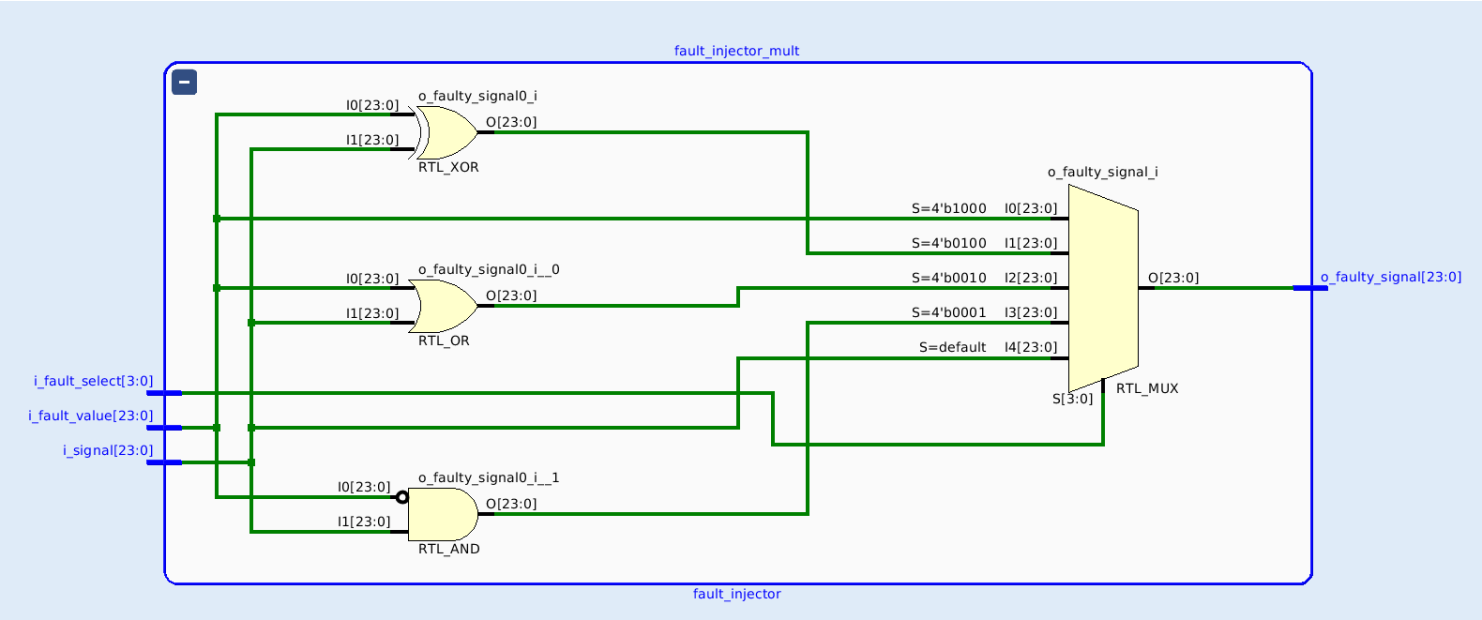
4.2 One FIR Filter



4.3 MAC module



4.4 Fault Injector module



4.5 Adder module (with saturation)