Fault coverage analysis through programmable MISRs integrated into a RISC-V core

Alice Afragoli
s324967

Martina di Leo
s322461

Giulia Solito
s329160

I. Introduction

This paper describes a fault coverage analysis on a benchmark circuit with the introduction of two programmable MISRs.

As a benchmark, the cva6 circuit was used, a 6-stage, single-issue, in-order CPU which implements the 64-bit RISC-V instruction set.

II. misr implementation

A. Structure

A MISR is a multiple input LFSR (Linear Feedback Shift Register) that is used mainly as an output data evaluator (ODE) for a circuit. Like any ODE, its main task is to compact the sequence of output values, called primary outputs(PO(t)), into a single vector of n bits, i.e. a signature S. At the end of the test session, the computed signature is compared to the expected one.

The implemented MISR is the one shown in Fig. 1.

The characteristic polynomial of the MISR is implemented through the coefficients $c_{\{0\}}, c_{\{1\}}, ..., c_{\{n\}}$, which are connected to an AND gate together with the feedback branch. The signature is read in parallel and stored in a dedicated register, accessible to the software.

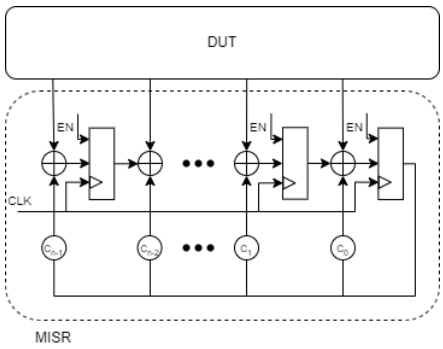

Fig. 1: Structure of the MISR

B. Programmability

The main advantage brought by this project is the programmability of the MISRs, which are seen as peripherals of the microcontroller. This is possible due to the presence of three control status registers:

- *Control register*, with two bits, programmable via software, that tell the MISR when to start/stop the test, and when to reset
- *Coefficient register*, where the user stores, via software, the primitive polynomial the MISR must refer to
- *Signature register*, from where the user can read, via software, the current value of the signature

The structure of the wrapper containing both MISR and control registers is shown in Fig. 2.

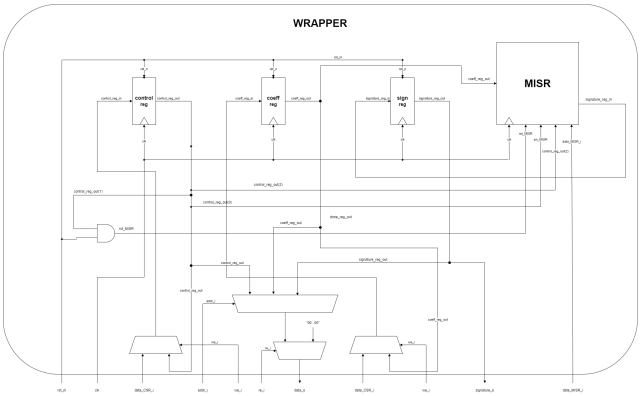

Fig. 2: Connections between MISR and relative control registers.

III. misr placement inside the core

A. Chosen input signals

The first MISR (with its relative registers) was instantiated close to the Branch Prediction Unit, between the ID and EXE stages, and its input data signal is the branch predict address. The second MISR (with its relative registers) was instantiated close to the ALU, inside the EXE stage, and its input data signal is the data computed by the ALU.

Their control status registers are memory mapped inside the architecture, and were assigned a free address range.

B. Interface

In order to let the software be able to program the MISRs, it was decided to add some signals to the interface of the core; specifically, the address given by the SW, together with the data to be read or written, and some enable signals. The signatures computed by the MISRs were also added to the interface for testing purposes, so that they could be constantly monitored. Moreover, a multiplexer was added in order to choose with which MISR the software wants to communicate, based on the given target address.

The whole architecture has been compiled, synthesized and tested after these changes.

IV. Testbench

In order to program the MISRs via SW, an external bus architecture called AXI is used.

This bus was already connecting the cva6 core to an SRAM, through an *axi2mem* module capable of translating the protocol of the bus to the signals needed by the SRAM.

An address decoder was added inside this testbench in order to manage the addresses coming from the software that are not related to the SRAM. If the decoder detects that the given address belongs to the SRAM, it will disable both reading and writing for the MISR, and will enable the correct operation for the SRAM; the opposite will happen in case the address belongs to the range chosen for the MISRs. If the software is indeed trying to read/write the MISR registers, the signals forwarded to the cva6 by the address decoder are the following:

- *re_misr_o*, read enable (one bit for each MISR)
- *we_misr_o*, write enable (one bit for each MISR)
- *addr_MISR_o*, address referring to the specific register of one of the two MISRs
- *data_MISR_o*, data that the sw wants to write inside the registers (could be coefficients, enable or reset)

Moreover, to ease the test, a `$monitor` function was added inside the testbench to check the signatures computed by the MISRs.

V. Software

A. Drivers

To manage the requests coming from the software related to the MISRs' control registers, some drivers were implemented, responsible of writing to or reading from said registers:

- `MISR_start()` and `MISR_stop()` respectively set to 1 or 0 the enable bit of the *Control register*, while `MISR_get_enable_value()` allows to read said bit
- `MISR_reset()` and `MISR_clear_reset()` respectively set to 0 or 1 the reset bit of the *Control register*, while `MISR_get_reset_value()` allows to read said bit
- `MISR_set_coefficients()` and `MISR_get_coefficients()` allow to write and read the *Coefficient register*
- `MISR_init()` initializes the MISR by clearing its reset bit, setting the coefficients given as argument and enabling the computation of the signature
- `MISR_get_signature()` allows to read the *Signature register*

All the drivers require to specify in the argument the base address of the MISR that must be targeted; this operation is eased by the creation of a *define* for each of the base addresses.

B. SBST applications

As for the SBST application, a C code computing a Fibonacci sequence was chosen; before starting the computations, both MISRs were set and enabled, and their resulting signatures were read and printed at the end of the computation. This was done using the previously described drivers.

An assembly code performing the same computations was also written, as it allowed to better control the operations performed by the microprocessor rather than by using a high level approach.

VI. Simulation

The final architecture was simulated using the previously described testbench and software, to make sure that the behavior of the embedded MISRs was the expected one.

Fig. 3 and Fig. 4 show the waveforms of one of the two MISRs, obtained during the simulation. In particular, Fig. 3 shows what happens when the software clears the reset bit of the *Control register*, writes in the *Coefficient register* and finally enables the MISR, thus allowing it to start the computation of the signatures. On the other hand, Fig. 4 shows the end of the simulation, where the enable bit of the MISR is cleared and the signature is read by the software.

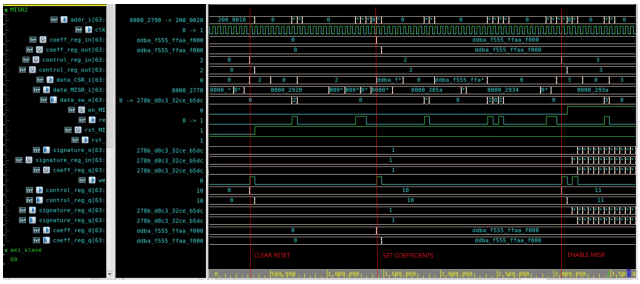

Fig. 3: Simulation waveforms showing a MISR being reset, loaded with the coefficients and enables

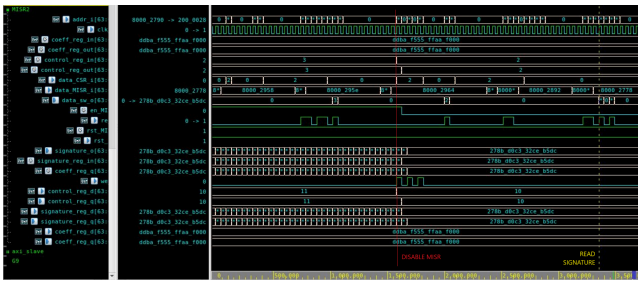

Fig. 4: Simulation waveforms showing a MISR being disabled and the reading of the computed signature

VII. Fault coverage results

The final step was to run a fault simulation on the whole circuit, both with and without the added MISRs, in order to compute the fault coverage.

Both the C and assembly codes were tested to see which was able to achieve a higher fault coverage; it was found that the best results were provided by the assembly, therefore it was chosen as SBST for the fault simulation.

Table I show the results of the fault simulations, respectively with and without the added hardware of the MISRs. As expected, the number of fault increased in the architecture with the MISRs with respect to the one without them, as more hardware means more possible fault locations. The values of the observational coverage only differ of a $0,01\%$, meaning that the introduction of the MISRs did not bring a significant

improvement of the coverage. This could be due to the fact that the chosen SBST was very simple, and only meant to show the correct functionality of the MISRs embedded in the core.

TABLE I: Results

	Total number of faults	Untestable faults	Observational coverage
architecture without misrs	64688	144	27.76%
architecture with misrs	64758	144	27.75%

VIII. Possible improvements

A possible future improvement would be to write a more in-depth SBST application, which performs more operations aimed at triggering the faults present in the core, specifically in the sections where the MISRs were placed. This would lead to a better fault coverage.

Another possibility would be to change the way the software is able to program the control registers of the MISRs. This would imply that, instead of using the AXI bus to communicate with the peripherals, the internal bus of the core could be directly accessed. This more elegant solution would allow to remove all the added signals to the core's interface, which should be kept to a minimum.

A final improvement would be to add more MISRs inside the architecture, in other strategic locations where many important computations are performed, thus allowing to increase the visibility inside the processor.