Politecnico di Torino

Testing and fault tolerance

Technical report

Samuele Yves Cerini, s256813
Federico Pozzana, s254758
Professor Matteo Sonza Reorda

January 16, 2020

Contents

0 Project objective **2**

1 ASM program **2**
1.1 ATPG generated instructions 2
1.2 LFSR-like generated instructions 3
1.3 LAB4-based generated instructions 4

2 Results and considerations **4**

0 Project objective

The objective for this assignment is to develop a functional testbench in the form of a single program, either written in `C` or in `asm`, able to target single transition delay faults for the execution unit of a *RISCY* CPU. Some assumptions made for this project are:

- Single transition delay faults
- I/O signals observable and controllable at any clock cycle

The default flow provided during the lab sessions has been used, only the `tmax.tcl` file has been modified to evaluate the faults in the execution unit and to look for delay faults instead of stuck-at faults.

1 ASM program

In order to achieve a low execution time (thus keeping the entire test-phase as fast as possible) we decided to make use of the *RISCV* assembly language. This decision also allowed us to manipulate each instruction of the processor at a finer grain as we will explain further in this report. Finally, using the assembly language allowed us to follow the exact same flow involving the use of *TetraMAX*'s *ATPG* functionalities, as done previously in one of the course's labs (lab #4). The assembly program developed for this assignment can be divided into three macro categories, namely:

- ATPG generated instructions
- LFSR-like generated instructions
- LAB4-based generated instructions

All the assembly files generated have been included into the provided `main.S` assembly program. We will now carefully analyze the aforementioned three main categories for our `asm` files.

At first, we started the evaluation of the execution unit by splitting it into sub-components like the `ALU` (which also contains the `DIV` unit) and the `MUL` unit. As we can see by looking at the entire `EXE` fault list the majority of faults (72735 faults in total) is split almost equally between the `ALU` unit and the `MUL` unit.

The first approach we used involved the use of *TetraMAX*'s *ATPG*, as we explain in the following section.

1.1 ATPG generated instructions

Hence, we started with the goal to achieve a decent fault coverage for the `ALU` module. To do that we started by setting a constraint on the `OPCODE` input signals of the `ALU` module and let the ATPG to generate automatically all the couples of input patterns. Once the `.stil` file containing the ATPG's patterns has been produced, we only had to parse it using the same `python` script provided with lab #4.

Since the parser was initially intended for *stuck-at faults*, we needed to modify it accordingly: when using the *single-transition delay faults* model, the `ATPG` generates two distinct `_PIs` patterns inside the resulting `.stil` file. The first pattern is used to excite the fault while the second one is used to propagate it. In order to transmit this behavior also to our assembly program, we needed to modify the parser accordingly. This simple modification is structured as follow: when parsing the first `_PI` line of the actual set of `_PI + _PI + _PO` pattern the goal is to translate this sequence into the corresponding operation (for example, an `ADD`) between two operand registers (whose value is also parsed from the first `_PI` sequence). The result of this operation is then stored into the stack, altough not strictly necessary since the first `_PI` sequence is used to control the fault, not to observe it.

Following the same idea, we parse also the second `_PI` sequence into the same operation as before. This time, we must store the value produced into the stack since this value will allow us to observe the transition delay fault.

This approach led to a fault coverage of almost 45% for the `ALU` using the majority of the `ALU`-related instructions (the `OPCODE` for each operation can be found in the `riscv_defines.sv` file). Many cases and `ATPG` sessions were made in order to constraint correctly the input signals in order to obtain a increasingly higher fault coverage. As an example: we found that forcing the `vector_mode_i[1:0]` imput signals to the value `"11"` in order to enable the 8-Bit vector mode (to perform four operations on the 4 bytes inside a 32-bit word at the same time) led us to a similar fault coverage to the one obtained with the vector mode disabled but with a lower number of patterns produced. This allowed us to obtain up to $2x \div 3x$ improvement on the total execution time.

Finally, we decided to build a new python script that, given all the `OPCODE`s of the instructions we wanted to generate `ATPG` patterns with, could automatically launch `TetraMAX` for each instruction and parse the provided `.stil` file into the related `atpg_patterns_$(OPCODE).S` assembly file. This allowed us to avoiding launching

manually TetraMAX for each operation when a minimal modification was made to the constraints or to the parser script.

However, even when we covered the majority of the OPCODEs found in the riscv_defines.sv file we found that the resulting fault coverage was still not sufficient (i.e. too low), especially considering the huge amount of patterns generated (and the consequent overall test execution time required). Since the overall result was not satisfying enough, we decided to move and test a completely different approach, as explained in the next section.

1.2 LFSR-like generated instructions

The following approach we adopted required a complete change of paradigm: we stopped using the ATPG and we started generating patterns using a randomized approach. The first goal was to build a *LFSR-like* assembly program. The program, given two random register values (created manually or using the python-shell hex(random.randint(2, pow(2, 32)-1)) command) completes various operations using these two registers, saving the results into the stack, "scrambling" the two operand registers (also using the newly-obtained results for an additional randomization) and finally repeating the process with a new set of instructions, using the freshly scrambled operands.

For each operation, four combinations of the input operands have been tested (where possible):

- *Operand1* OP *Operand2*
- *Operand2* OP *Operand1*
- *Operand1* OP *Operand1*
- *Operand2* OP *Operand2*

The main idea is to enclose the entire program around a loop that iterates again and again the same operations by always keeping random and different operands registers (as the second iteration will use the scrambled registers obtained when executing the last operations of the first iteration). This approach was applied to all the possible ALU, MUL and DIV operations we found in both the general *RISC-V* Manual and in the *RI5CY* Manual (including nearly-all the custom *RI5CY* instructions).

As we discovered later, this assumption cease to be true after some iterations of the main loop (we found a cap around the fourth iteration of the loop: any additional iteration will have produced additional patterns, of course, but no benefits in terms of fault coverage). This is due to the intrinsic random-resistance property of circuits. Still, this approach led to greatly improved performances in terms of fault coverage, while keeping at the same time a lower number of patterns produced (with respect to the ATPG-based approach) and thus a reduced time needed to complete the fault simulation.

In the following section of code, you can see an example of two operations included into the *LFRS-like* MUL file. Each result is stored into a different stack location. The two input operands x18 and x19 are finally scrambled using the byte-vectorial XOR operation among the freshly obtained results.

```
p.mac   x22, x18, x19
p.mac   x23, x19, x18
p.mac   x24, x18, x18
p.mac   x25, x19, x19

p.msu   x26, x18, x19
p.msu   x27, x19, x18
p.msu   x28, x18, x18
p.msu   x29, x19, x19

sw  x22, 4(sp)
sw  x23, 8(sp)
sw  x24, 12(sp)
sw  x25, 16(sp)
sw  x26, 20(sp)
sw  x27, 24(sp)
sw  x28, 28(sp)
sw  x29, 32(sp)

pv.xor.sc.b x18, x23, x25
pv.xor.sc.b x19, x24, x22
```

To overcome the aforementioned limitations due to the increasing resistance to randomization, we had to split the unique file we made for the `ALU` module into multiple files, with a different internal architecture. Each block that initially composed every single file was in fact copy-pasted 3-4 times inside the same file (in a loop-unrolling fashion): each macro-block of code this time has its set of randomized operands, thus avoiding depending on the previous macro-block of code (for an additional randomization). This approach has been finally applied only to the files concerning `ALU` module (that showed a higher resistance to randomization with respect to the `MUL` and `DIV` modules.)

This approach finally allowed us to reach the cap of the 80% fault coverage on the entire Execution Unit. Due to lack of time we decided to keep the original files with the aforementioned modifications and accepting a fault coverage slightly above the 80% instead of implementing ex-novo a *Python* script that could solve finally the problem of the random resistance (such script will have completely unrolled the loops, randomizing the two operands after each operation thus requiring no more deterministic scrambling), hence increasing drastically the fault coverage and minimizing the test execution time.

1.3 LAB4-based generated instructions

A new approach has also been taken into consideration to further improve the fault coverage obtained with the `LFSR`-like generated instructions; following the tips provided in the 4th laboratory pdf a loop-based assembly algorithm was implemented, which consists in performing the operation between operands that have a repetition of four by four bits.

Adding the lab4 generated instructions improved the fault coverage by a significant amount without affecting considerably the test time, unlike the `ATPG` generated instructions.

2 Results and considerations

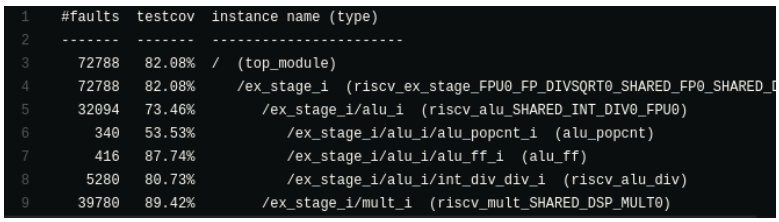
For our final simulation run we decided to abandon completely the patterns obtained with the `ATPG` and include only the following files (`LFSR`-like and `LAB4`-based):

- `ALU (includes DIV)`
  - `LFSR_ALU_Vect_Comparison.S`
  - `LFSR_ALU_General.S`
  - `LFSR_ALU_Vect_Ops.S`
  - `LFSR_ALU_Bit_Manipulation.S`
  - `LFSR_ALU_Riscv_Classic.S`
  - `LFSR_DIV_Smart_Patterns.S`
  - `LFSR_DIV_Unit.S`
- `MUL`
  - `LAB4_patterns_mul.S`
  - `LAB4_patterns_mulh.S`
  - `LFSR_MUL_Unit.S`

By keeping a similar (just slightly modified) approach to our *LFRS-like* allowed us to reach a fault coverage cap that sadly cannot be overcome by just increasing the number of iterations of the assembly program (due to the random-resistance of the logic). To overcome that cap only a new *python* script with an unrolled approach would have been successful, that sadly we couldn't implement due to time limitations involving the members of this group. However, as a drawback, such approach based on a python script increases dramatically the code size of the program since no loop is implemented on the assembly side (the entire code is unrolled since this is the only way we found that allows to re-load the operands with completely new and random hexadecimal values after each operation). This must be taken into consideration especially when the memory reserved to code is limited.

Our approach, on the other side, sacrifices a little bit the final fault coverage and the execution time (which is higher with respect to the *completely-unrolled* solution we mentioned above) but heavily reduces the code size (since the loop is actually implemented on the assembly domain).

Both these solutions must be carefully evaluated by the test engineer, considering the final processor he/she has to take into consideration. The three main trade-off parameters that he/she must take into consideration are: fault coverage, test execution time and code size.



```
#faults  testcov  instance name (type)
-------  -------  ----------------------
  72788   82.08%  /  (top_module)
  72788   82.08%    /ex_stage_i  (riscv_ex_stage_FPU0_FP_DIVSQRT0_SHARED_FP0_SHARED_D
  32094   73.46%      /ex_stage_i/alu_i  (riscv_alu_SHARED_INT_DIV0_FPU0)
    340   53.53%        /ex_stage_i/alu_i/alu_popcnt_i  (alu_popcnt)
    416   87.74%        /ex_stage_i/alu_i/alu_ff_i  (alu_ff)
   5280   80.73%        /ex_stage_i/alu_i/int_div_div_i  (riscv_alu_div)
  39780   89.42%      /ex_stage_i/mult_i  (riscv_mult_SHARED_DSP_MULT0)
```

Figure 2: Fault coverage and hierarchy obtained with the final fault simulation.

Below we report the results obtained with our solution. In the following figure we finally report the fault coverage achieved.

For further references please visit the GitHub Repo (will be made accessible after the project deadline).

- Patterns generated: 42849
- Code size: 147.2 KiB (sbst.hex)
- Test time: 10158 seconds