Politecnico di Torino

Testing and Fault Tolerance

SW Assignment: March Test & SBST

Assignment Report

Master degree in Electronics Engineering

Referents: Prof. Matteo Sonza Reorda, Prof. Riccardo Cantoro, Prof. Nikolaos Deligiannis

Authors: Group 22

Castagneri Dario 277967, Damiani Manuel 277996

January 14, 2022

CHAPTER 1

March Test

The goal of this assignment is the development of a functional unit-oriented testing firmware able to test some blocks of a RISC-V architecture (RI5CY), compute their fault coverage and determine the overall test execution time. The involved units are the RAM, the register file, the multiplier and the ALU.

Functional testing is intended to be executed at the power-up of a bare-metal system. For this reason, the entry point is the *crt*0.*S* (which stands for "C Runtime 0") file. It is a set of execution startup routines that perform the first required initializations before calling the program's main function. After initializing the register file, the RAM testing starts.

Functional testing of memories allows to detect permanent faults that cause memory misbehaviors. One category of test algorithms is that of March Tests, composed of a sequence of March Elements which perform on each cell of the memory the same sets of read and write operations. These March Elements are executed either in ascending, descending or in indifferent address order.

There are different proposed solutions, each with a different degree of complexity; the choice can depend for instance on the available test time. In this assignment, the March C algorithm was chosen, as a good compromise between test complexity and effectiveness.

1.1 March C Algorithm

The March C algorithm consists of seven march elements, which can be seen below in Figure 1.1. Each element is repeated on all the cells of the RAM, in the desired address order. The operations "w0" and "w1" consist in writing, respectively, 0 or 1 in the whole memory; instead, operations "r0" and "r1" means reading the whole memory and checking that the correct value is present.

However, the RAM is not composed of 1-bit cells: each one has a 32-bit length. For this reason, we use the concept known as "Data background". This means that the whole March C Algorithm is repeated once for every different pattern corresponding to a 0 or a 1. In the first iteration, 0 corresponds to all 0's and 1 to all 1's; in the second iteration, 0 corresponds in the upper halfword equal to 0, and the lower halfword equal to 1, and viceversa for the pattern corresponding to 1, and so on. In our initialied memory space to store the March patterns, inside section "TEST_RODATA" (which is a

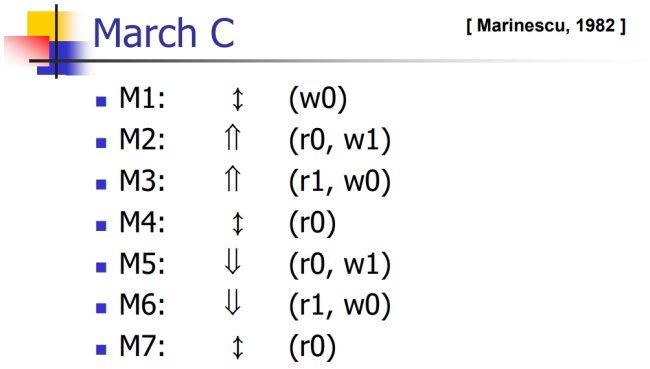

Figure 1.1: The March C Algorithm

read only section), we only stored the patterns corresponding to the value 0. To obtain the patterns for 1 we simply perform, at the start of every iteration, a XORI operation between the current value for 0 and immediate "-1", which essentially implements a NOT, obtaining the required pattern for 1.

The whole March C algorithm can be found inside *march_algorithm.S*. To test the RAM in its entirety, we iterated every March Element between addresses "__RAM_START" and "__RAM_END", which are constants found in the linker script corresponding to the starting and ending address of the Data RAM. To perform the write operations, a simple store operation is done on every memory cell. Instead, to implement the r0 and r1 operations, we load the value in the current address and perform a subtraction between the read pattern and the expected one. If the result is zero, it means that they are identical, and the test can continue; otherwise, the program jumps to "march_error", which sets register x31 to 1 and returns to crt0. If the whole test procedure reaches the end, x31 is set to 0 and we return to crt0. After we tested the entire March Algorithm to ensure its correctness and evaluate the total test time, for easier usage we also created a shorter version by simply going through cells between addresses $\_\_RAM\_START$ and $\_\_RAM\_START + 400$.

After returning to crt0, the initialization of the global pointer and the stack pointer take place. After the stack pointer initialization, we store the March Test result contained in x31 inside the stack. The reason behind this is that the we need to let the memcpy operations take place before we stop the program in case of error, otherwise the dataram region is not copied in the "copyram" region and an error for "Writing outside DRAM" occurs. After this is done, the March Test result is loaded back into x31: in case of success, the program moves on to the SBST procedure; otherwise, the SBST and the main program are skipped and the *test_fail.c* function is called, which simply returns an "EXIT_FAILURE" error code.

The March C Algorithm was tested by running a gate-level simulation with QuestaSim to evaluate the total execution time: the result is $243'820'480ns$, corresponding to roughly $243.8ms$.

CHAPTER 2

Software-Based Self-Test

The next step in the startup test consists in the execution of the Software-Based Self-Test (SBST) routines. This technique consists in testing the functional units of the processor core by running some code on it with some data values. In this way, fault simulation can be performed in order to evaluate the fault coverage (FC) for each functional unit. To achieve a good target FC, not only the basic RISC-V instruction set was tested, but also some of its extensions. The base integer instruction set is called RV32I: in addition, we also targeted the "M" extension for integer multiplication and division, the "C" extension for compressed instructions and the "Xpulpv2" extension specifically supported by the RI5CY core tested in this assignment. The functional units that we decided to target in our SBST procedure are the Register File, the Multiplier, the Divider and the ALU.

To perform the test, the *sbst_boot.S* program is launched: at the start, the content of the non-volatile registers is stored in the memory, and the interrupts are disabled (if they were enabled). Then, based on the value of a word of flags stored in the "TEST_RODATA" section, called "ENABLE_FLAGS", the various routines are either executed or skipped. In particular, bit 0 of this value contains the enable for the Register File, bit 1 enables the Multiplier, bit 2 enables the Divider and bit 3 enables the ALU. Of course, any combinations can be executed. At the end of each SBST routine, the obtained signature is compared with the golden one obtained from a fault-free simulation: if they are different, the content of the non-volatile registers is restored, the interrupts are re-enabled and we return to crt0. Then, the main is skipped and $test\_fail.c$ returns EXIT_FAILURE. In case all tests are successful, the value 0 is loaded in register t4, so that crt0 runs the *main.c* program as intended.

To respect the constraints on the SBST subroutine length, each routine for each functional unit is split in multiple subroutines of at most 1000 clock cycles. After running the simulation for each functional unit separately, we executed TestMAX by running the provided shell script, adding to the list of faults only the ones regarding the unit currently being tested. The complete and detailed results of all these procedures can be found in folder "results".

2.1 Register File

The Register File is composed of 32 registers, and it is used to retrieve source operands in the Decode stage for the various operations, and also in the Writeback stage to store the result, if needed. To

test it, we loaded each register with some random values and summed them all together to compute the signature. This was repeated with different patterns and also with different types of load/store instructions: not just LW and SW, but also loading/storing bytes (LB, LH, SB, etc...), operations to load immediates (LI, LUI), and post-increment load/store instructions (like p.lw and p.sw, provided by the Xpulpv2 extension). For the fault simulation, we added all faults relative to the Register File. The final result was $FC = 79.57\%$, just below our desired value of 80%.

2.2 Multiplier

The second unit we tested is the Multiplier, which is outside of the ALU and performs both integer multiplications and, with the Xpulpv2 extension, also vectorial ones (like pv.dotsp.h, pv.dotup.h, and so on). The test is composed of two nested loops: the outer loop grabs a first pattern, then the inner loop keeps the first operand constant and executes all the instructions by taking as the second operand all other patterns, while also accumulating the signature by summing the results. Once the inner loop is finished, the outer loop index is increased and we repeat. After running the fault simulation on the Multiplier faults, we achieved a fault coverage $FC = 90.11\%$.

2.3 ALU

The ALU is composed of several different units: an adder/subtractor, a shifter, a comparator, a shuffle unit, a unit for bit-counting operations, one for bit manipulation, and a divider unit. In order to test the ALU, we first decided to focus, as an example, on only one of its units, the divider. Once we evaluated the FC of the divider, we included its instructions, along with other ones for the various ALU subunits, in the ALU test.

The algorithm used for the DIV and ALU test follows the one used for the Multiplier: there are two nested loops, performing all the different instructions on different couples of operands and accumulating the signature. The various subroutines work on different operands and different instructions.

2.3.1 Divider

The Divider test is composed of four simple instructions: DIV, DIVU, REM and REMU. The resulting fault coverage is $FC = 91.24\%$.

2.3.2 Complete ALU

After the DIV test, we incorporated its instructions and a lot of other ones in the ALU SBST. These include additions and subtractions, logical and arithmetic shifts, comparisons, shuffling operations, bit counting ops, bit manipulations (like p.ror, p.extract, etc...), and of course also the division operations listed before. The fault simulation for the complete ALU achieves a fault coverage $FC = 81.16\%$.

To conclude the tests, we ran simulation of all SBST routines to evaluate the total execution time. The result is $136890ns$. This leads to an overall (March Test + SBST) test duration $t = 243'820'480ns +$

$136890ns = 243'957'370ns$, equal to about $243.9ms$. This means that the total test time depends almost solely on the March Test duration.

2.4 Summary of Results

Test	**Execution Time**
March Test	243'820'480 ns
SBST	136'890 ns
Total	243'957'370 ns

Table 2.1: Test execution time

SBST	**Fault Coverage %**
Register File	79.57%
Multiplier	90.11%
Divider	91.24%
ALU	81.16%

Table 2.2: Fault coverage results