Politecnico di Torino
Testing and Fault Tolerance

REPORT: SW ASSIGNMENT
MARCH ALGORITHM AND SBST

December 21, 2021

Matteo Battilana s281389
Gabriele Salvatore La Greca s281589

1 Introduction

The software assignment was based on two portions of code that are used to:

- A March test algorithm on the RAM
- A set of SBST routines that are scheduled in portion of 1000 clock cycles, used to test three functional units: ALU, Multiplier and Register File.

1.1 March Algorithm

The March test has been put at the startup of the system, in order to perform a complete check of the data memory. The method, called `_run_march_algorithm` that is defined in the `march_algorithm.S` file, is called in the `crt0.S` just after the initialization of all registers to 0.

The base idea of the March Test is to perform a sequence of March Elements. Each March Element is composed of a sequence of write/read operations that must be performed in a cells and also for all the others.

The algorithm used in this assignment is the March LA, developed by Van de Goor, in 1996 has a complexity of $22n$ and it's able to detect Stuck-at-faults, Address-faults, Transition-faults and Coupling-faults, linked in any way. It composed of the following March elements:

- $\updownarrow$ (w0)
- $\Uparrow$ (r0, w1, w0, w1, r1)
- $\Uparrow$ (r1, w0, w1, w0, r0)
- $\Downarrow$ (r0, w1, w0, w1, r1)
- $\Downarrow$ (r1, w0, w1, w0, r0)
- $\Downarrow$ (r0)

Moreover in order to make the test complete, the algorithm does not simply write 0 and 1 in the memory locations, but uses $\lfloor \log_2(m) \rfloor + 1$ patterns, where m is the word size. This allows to test also the intra-word coupling faults. So, the total complexity of the march test is:

$$6 \cdot 22 \cdot n = 132n$$

For this reason the `_run_march_algorithm` procedure acts only as a wrapper in order to call another procedure, called `_march_LA`.

This last procedure, is used to perform entirely the March LA algorithm, by passing one parameter on the `a0` register that is used to define the 0 pattern that will be used when performing *w0* and *r0*. The 1 pattern is simply computed by the produced, by negating `a0`.

This allows to not duplicate the code and execute the March LA for all the 6 patterns.

In order to speed up the end of the test, in case of error, the procedure terminates as soon as possible, returning 1 in case of an error and 0 otherwise.

In order to perform the test over the entire data ram, two sections have been defined:

1. ___RAM_START
2. ___RAM_END

The first one, is used to define the starting point of the March LA while the second one its end. The end address is loaded into the `t2` register but it can't be used as is. In fact, since it define the end point, the loop must avoid to write in that precise location (outside the dataram); for this reason, a subtraction of 4 is performed. The algorithms is 1:1 the implementation of the March LA, and by using `t1` and `t2` as starting and ending references, all the locations are read/written accordingly to the March algorithm.

1.1.1 Problems

One of the C library used in the linker script (newlib) creates in the .elf file a section in .data with a struct called `__rent` whose address is defined by the symbol ___global_impure_pointer. The struct contains information about the errno, used by the ANSI C standard, stdout, stin and stderr and other callbacks. At the beginning, the .elf loaded during the simulation contains that information, and running the March LA destroys everything. In particular, the crt0 executes the function `memset` for the initialization of the bss segment, that reads data from the `__rent` structure, that has been erased by the memory test. The problem is that the RAM sections are (improperly) initialized by the testbench at power-on and this has been solved by copying at the startup all sections into a portion of a Flash memory. In this way, at the end of the march algorithm, data saved into the flash are moved into the RAM.

1.2 Results

As stated before, the March LA is a powerful algorithm to test memories that has the drawback of being complex from the point of view of the computational complexity. Moreover, the complexity increases even more, if the intra-coupling faults are tested. Having a RAM, which size is 0x40000, the time needed to test it with this implementation is: 212992660ns, that corresponds to more or less 200ms.

2 Test scheduler

Beside the March test, the Software-Based Self-Test needs to be split into smaller subroutines that do not last more than 1000 clock cycle. This means that, for testing a functional, more than one subroutine will be needed. In order to let the user to specify which subroutine of which module to run, a 32 bit flag register is used in the following way: In fact, the ALU is composed

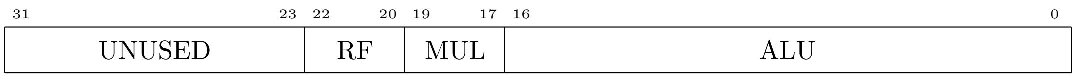

Figure 1: *Test flag register*

of 17 subroutines, the multiplier by 3 as for the register file; the other bits are left unused. So, by setting that register to 0x03, only the first three subroutines for the ALU test are executed.

In order to work correctly, a zone in memory has been defined in the `.TEST_RODATA` called *TEST_SIGNATURES*, that stores the expected signatures for all the subroutines enabled, in order to check the result after executing them. The memory locations assumes first the ALU signatures, then the multipliers one and in the last position the register file ones.

The starting memory location is passed as parameter to the main test routing, assuming first 11 signatures for The entry point of our test routines for the functional units and the register file, is the `_scheduler_test` procedure that is contained into the *test_scheduler.S* file. This functions takes the following parameters:

1. **a0**: Vector of flags for the test subroutines to be enabled, refer to Figure 1;
2. **a1**: Pointer to the current in the vector of test signatures. Used to check the expected signature;
3. **s0**: Stores the expected signature

Based on the signature returned by the procedures and the vector of test signature given as input to this procedure, the scheduler will decide to halt the CPU or not if the test doesn't succeed.

As described before, the register that contains the test flags taken and the subroutine `i` is executed only if `a1[i]` is set to one. At this point, the selected procedures of each module are executed and, the signature computed for each one of them, are merged together to perform a single signature for each unit by resorting to the XOR.

In order to check the test result, the signatures coming from each one of the units, must be compared with the expected one. For this reason, the *test_scheduler.S* includes a procedure called `_get_sig` that takes the test flag register and the starting location of the expected signature of all test subroutines. At this point, the procedure computes the expected signature for that functional unit by performing the XOR logical operation by taking the signatures of the enabled subroutines. The test scheduler on the main program, will check that the generate signatures from the test routines are the same with the respect to the expected ones. In case of a mismatch, the processor is halted by using an endless loop.