Politecnico di Torino
Testing and Fault Tolerance

REPORT: SW ASSIGNMENT MARCH ALGORITHM AND SBST

December 21, 2021

Matteo Battilana s281389
Salvatore Gabriele 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 unit, 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:

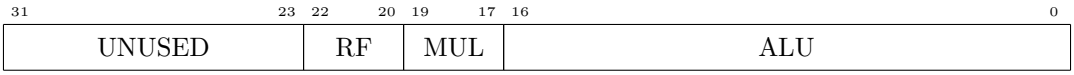

Figure 1: *Test flag register*

In fact, the ALU is composed 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.

TO COMPLETE DESCRIPTION

3 Test Routines

As said, the three functional units are tested by means of routines. Their aim is to test as much as possible the functional unit by executing some instructions. These instructions are not random generated and the process of generation of these routines is different if we consider the ALU or the MUL/MAC unit and if we consider the RF unit.

All the process of generation of the assembly code, compilation and simulation is automatized thanks to different scripts. In particular, there are 3 precreated files `alu.S`, `mul.S` and `rf.S` that contains the base structure of the assembly code and the parsers will insert the generated instructions between the two lines `##- START -##` and `##- END -##`.

Moreover, in order to compute the time of exectuion of each routine, the testbench has been modified in order to log in the stdout the time when the register `x30` is loaded with the value `0x12345678` while when the register `t4` is loaded with the same value, the testbench will log both the time and the content of the register `t3` that contains the computed signature.

3.1 Routines generation for ALU and MUL/MAC

The first step in order to generate the routines for the ALU and the MUL/MAC units is to generate the test patterns by means of an ATPG (*TetraMAX*). The ATPG is configured in order to generate the patterns only for that specific units, at the end a `.STIL` file is generated containing all the patterns.

The second (and most important) step is to generates some assembly instructions starting from the patterns. The problem here is that obviously we don't have any direct access to the ALU (or MAC/MUL)'s entity from the software prespective, but we can only use the assembly instructions that the overall processor offers to us in order to mimic the patterns.

In fact, for each instruction, the core's decode unit will configure in a certain manner the inputs of the entity. So, the most difficult part here is to understand how the input of the entity are mapped to each instruction offered by the processor.

After a deep analysis of the decode unit's verilog RTL description, together with the PULP's user manual in order to understand all the added hardware extensions that within our functional units, a set of instructions has been created. All these instructions are used more than once with different operands (up to 3 different operands) according to each generated pattern. The downside here is that not all the ALU (or MUL/MAC)'s control signals are directly controllable by assembly instructions so the coverage will not be exactly as the one obtained by the ATPG but it'll be sligthly lower.

There can be indentified two big sets of instructions (thanks to PULP's extensions): scalar instructions and vectorial instructions. The first ones are the classical ones (for example an add of two operands on 32 bits). The latter one regards vectorial operations, means that for example a vectorial add will consider the two 32 bits operands as groups of 8 or 16 bits so it will perform 2 or 4 adds in parallel by means of a single instruction.

Finally, the last step is to generate the assembly code having the patterns generated at step 1 and a set of possible instructions generated at step 2. A script has been developed in order to accomplish this job automatically. The script is in charge of parsing the `.STIL` file and for each pattern find the most situable instruction and generate the assembly code. The script is pretty complicated and internally instructions are divided into subgroups (classical operations, branch instructions, single operand instructions, single immediate operand instructions, triple operands instructions, triple operands with 2 immediate instructions) and for each of them there may be a scalar or vector implementation of it. Howewer, the base assembly code generated for each of them is:

1. loading the operands into temporary registers
2. execute the instruction

3. compute the cumulative signature on the result

The script is in charge to automatically generate not only the entire test routine but either the subroutines, each of them with a duration < 1000 ccs. So the script will add, beside the classical test instructions, control instructions like which subroutine needs to be executed according to the flags given as input by the scheduler and to compute the cumulative signature.

3.2 Routine for Register File

A different approach has been adopted in order to test the Register File. Here the process of generation is a bit more complicated because of the nature on the Register File itself and its implementation in the given processor under test.

The test is based on a MARCH algorithm with different data backgrounds, the most simple one that performs the following march elements:

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

Unfortunately, this is not enough to achieve a high fault coverage. This is because this implementation has three read ports and two write ports.

In order to apply for example the march element *(r0)*, the *(r0)* must be applied on all the three read ports, and from them the same value must be read. So in order to accomplish this, we need a triple operand instructions, and it's `p.mac t0, t1, t2` that performs $t0 = t0 + t1 \cdot t2$.

Actually the most difficult part is to test the two write ports. In order to understand how the two write ports are used by the processor, another deep study of the decode unit and in general of the entire pipeline of the processor has been done.

What happens is that one of the write port is used during the write back stage that occurs after the memory stage so means that in order to test one of the two write ports we need a load instruction.

A bit harder is to understand how the second write port is used and it's used when a forwarding path is enabled.

At the end, after considering all these aspects, a new script with the same purpose of the ones used for ALU/MAC can be implemented. It's in charge of creating for each data background the three march element of the march algorithm described before and for each march action it takes care of applying the concepts explained in order to test all the ports of the register file.

At each step a signature is computed but it can't be simply stored in the register file becuse it's under test with a march algorithm that destroys the content of the RF. So everytime, at each step, a load and store of a signature is performed from the memory.

4 Obtained coverage

The following described coverages are guaranteed to be obtained only by enabling all the subroutines for each functional unit.

Functional Units Coverage		
ALU	MUL/MAC	RF
91.68%	96.60%	92.72%