Politecnico di Torino

Testing and Fault Tolerance

Assignment 1: Testing

Final Report

Master's degree in Computer Engineering

Authors: Group 20

Giuseppe Silvestri s307792, Lorenzo Marino s317703

1 Introduction

The goal of the assignment was to produce some functional stimuli that target a chosen subset of small delay faults of the OpenHW RISC-V-based *cv32e40p* processor. The purpose of this report, in turn, is to explain the work done: how the module to be tested was chosen, how the assembly code used to excite the faults was written and the results that were obtained.

2 SDD fault subset selection

The selection of the module to be tested started from the `.gsf` file generated after the synthesis, which reports the slacks for both the rising and falling transitions of every cell in the post-synthesis netlist. The ideal candidates for SDD testing are those fault sites with low slacks, which would be more sensible to set-up/hold time violations in case a small extra delay were to affect the site in question. The aforementioned timing report shows that many critical sites are in the fetch and decode stages, which are hardly controllable from assembly instructions. While the Arithmetic Logic Unit was a feasible candidate, it is often extensively tested and thus somewhat uninteresting to work on. As a result, our choice fell on the *LSU* (Load Store Unit), which exhibits a good number of slack-critical fault sites with respect to the total number of faults: we calculated that the fault sites in the LSU are 3045 (which is the 4.36% of the total) and that their average slack is 1.37 ns (compared to the clock period used to synthesise the processor, which is 5 ns, and the clock period used by the testbench, which we set to 6 ns instead of 10 ns to better simulate the at-speed behaviour of the system). Moreover, this unit is easy to control via assembly code. Once we decided that we would focus on the LSU, we generated a fault list containing only the sites present in our chosen module, in order to better target it and analyse the results of the fault campaigns.

3 STL development and analysis

A summary of the developed tests routines and their results can be found in Figure 1 and Table 1. The first attempt consisted in sequentially using of all the variants of the load and store instructions: each of them (`sw`, `sb`, `sh`, `lw`, `lh`, `lhu`, `lbu`, `lb`) is executed repeatedly using a per-instruction loop, resulting in a total of 8 loops. The main goal is to cause transitions from 0 to 1 and from 1 to 0 for (ideally) every fault site, which is why we write `0xFFFFFFFF` (or its truncated forms, if we store half-words or bytes) in the memory. In order to make our routine more closely resemble an actual SBST, we included a counter incremented after every reading mismatch (which never changes during the logic simulation, since it's fault-free). After testing the code with a RISC-V simulator[1] and QuestaSim, the fault simulation and coverage computation have been performed with Z01X. As for the SDD faults, the fault simulation is performed many times with varying degrees of defect seriousness, represented by the factor K: the delay of a fault is multiplied by K, so for higher factor values the slack decreases, which could possibly result in a timing violation. In order to cover a good amount of possible defect cases, the fault simulation is run with K values in the following ranges:

- range [1, 2], with a step of 0.1 (the most realistic);
- range [3, 10], with a step of 1;
- range [20, 94] with a step of 10 (the least realistic).

In Figure 1 the red line represents the coverage obtained for the SDD fault simulations of this first version of the routine, revealing a local maximum in $K = 1.7$ corresponding to 48.87% of fault

[1]RARS by TheThirdOne: https://github.com/TheThirdOne/rars

Test Version	**Features**	**TAT [cc]**	**TDF coverage [%]**	**SDD coverage [min % - max %]**
I	8 loops, one for each instruction variation	1076	55.12	40.73 - 54.67
II	single store loop, multiple load loop	989	61.74	49.04 - 63.84
III	2 additional misaligned addr. store loop, multiple load loops	1257	63.65	52.22 - 64.33
IV	single store loop, 2 load loop with and without misaligned addr.	1600	64.67	50.41 - 66.14
V	single store loop and single load loop	1250	64.37	45.67 - 66.48

Table 1: Summary of the test versions and their results

coverage. The maximum coverage reached by this routine is 54.67%, but it is achieved with the extremely unrealistic value of $K = 90$ (which corresponds to a defect increasing the delay of each cell by hundreds of nanoseconds), even if similar coverage values start from $K = 4$.

With the purpose of increasing the number of transitions, the second version of the code incorporates a single write loop performing all the store variants, while the reading loops remain separated and unchanged. This resulted in an improvement in both coverage and the test application time, as reported in Table 1 and depicted in Figure 1 (purple line).

A single loop performing both the store and load instructions was also tested but that only worsened the results, so the decision was made to keep the operations loops separate.

Upon inspection of the RTL code, it was discovered that this architecture also supports misaligned accesses to memory; therefore, this feature is exploited in the third version of the code by adding two loops performing misaligned stores with different offsets. As shown in the coverage plot (green line) the results are best for $K < 1.7$

In the fourth and fifth versions of the code, the misaligned accesses are expanded to load operations: the former adds one loop to perform the misaligned reads (meaning there is one loop for aligned and one for misaligned reading), the latter performs both aligned and misaligned reading operations in one single loop. We initialized a section of the memory with special patterns in such a way that all the different types of sign extension (zero-extension and one-extension) are triggered. The results of these last two iterations of the routine differ from our expectations: the coverage is lower than the third version for $K < 1.7$ but higher for greater values of the parameter, which are less significant for the purpose of analysing SDDs. The difference is approximately 2%.

3.1 Possible limitations

By furtherly inspecting the RTL code and the original OpenHW documentation we found out that we may have encountered some limitations: we were not able to test the atomic memory operations because the RV32A extension is no longer supported, but the RTL code implementing it has been partially left in the architecture. Furthermore, there are several special memory operations with

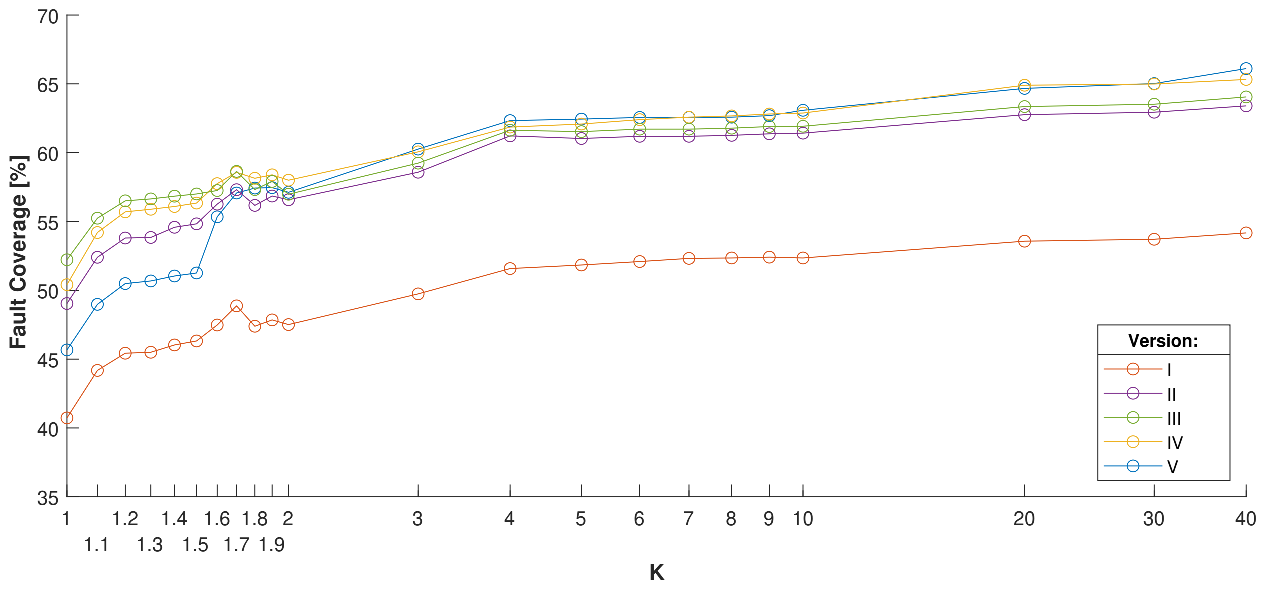

Figure 1: Comparison between the different versions of the routine

support for pre- and post-incrementing the accessed memory addresses using different modes (register-register, register-immediate, etc.), but due to toolchain compatibility issues we were not able to use them.

4 Conclusions

Table 1, along with the SDD coverage results, reports the *TAT* (test application time), measured in clock cycles (*cc*), and the transition delay fault coverage. As previously mentioned, the best results associated to lower value of the K factor are achieved with the third version of the code. It is worth noting that the SDD fault coverage for extreme K values is higher than the TDF coverage. This, coupled with the observation that longer and more extensive simulations do not necessarily increase the coverage of SDD faults (as seen in the fourth version of the test), further highlights the differences between the two fault models.