Testing and Fault Tolerance

SBST Contest Report

Andrea Floridia, 224906, Robert Margelli, 224854

11/01/17

1. The *selftest* program

The selftest program consists of a single assembly file whose content was firstly generated by means of a C program and then extra sections were added by writing assembly instructions. Overall the program is divided into the following sections:

1. Register file March algorithm;
2. LFSR loop;
3. SFR read/writes;
4. Simple and extended jump tests;
5. Test on all addressing modes;
6. Nested CALL instructions.

The so-called LFSR algorithm was our starting idea which stemmed from what was covered during class hours on functional test. In fact, by adopting a strategy that leverages on pseudo-random values it is known that a reasonably high coverage can easily be achieved (around 80% in our case). This code is composed of a loop in which two numbers are generated with two LFSRs (one the reverse of the other) and then assigned to the two registers (R6 and R7) used in the second part of the loop where all ALU and multiplier operations are performed.

```
/* taps: 16 14 13 11; feedback polynomial: x^16 + x^14 + x^13 + x^11 + 1 */
/* generate two random numbers  */

    bit1  = ((lfsr_1 >> 0) ^ (lfsr_1 >> 2) ^ (lfsr_1 >> 3) ^ (lfsr_1 >> 5) ) & 1;
    lfsr_1 =  (lfsr_1 >> 1) | (bit1 << 15);
    bit2  = ((lfsr_2 >> 0) ^ (lfsr_2 >> 2) ^ (lfsr_2 >> 3) ^ (lfsr_2 >> 5) ) & 1;
    lfsr_2 =  (lfsr_2 >> 1) | (bit2 << 15);
```

Figure 1 - Code for the LFSR number generation.

The number of loop iterations can be limited by setting a value into R15 instead of letting the LFSRs generate all possible values (i.e. 2^16 = 65536, this was an extremely useful decision as detailed in the conclusions). From this, we started writing the other sections aiming at specific modules in the OpenMSP430 core by referring to the *report_faults.txt* file generated by TetraMax after each fault simulation.

The MATS+ March algorithm on the register file provided some extra coverage on such module but was not enough for our purpose. This is mainly due to that R0-R2 were not modified, since any change in these registers may alter the processor status. Furthermore, there are some other circuits that are in charge for implementing different addressing modes and we focused on exciting them. In order to take this value over 90% we exploited the tests on the autoincrement addressing mode written by the OpenMSP430 author (*two-op_autoincr.s43* and *two-op_autoincr.s43*).

For exciting faults in the SFR (Special Function Registers) simple reads and writes in this memory area was enough to achieve a high coverage on them but did not impact the overall one as only 210 faults are here present (a negligible amount if we compare it to other modules; for instance, the execution unit has 14544 faults).

Further tests on jumps instructions have been added. These test have been partially developed by us, since we exploited already existing benchmarks (*c-jump_x.s43*) that address specific modules either in the frontend and execution unit.
Additional tests on most of the core's modules were done by exploiting all addressing modes (taken from *two-op_add.s43*). This provides some coverage on the frontend, register file, memory backbone and execution unit as well.

To complete the test on all instructions (excluding the RETI) the final part of the program is a series of nested function calls in all possible addressing modes. To fetch from the memory as many different addresses as possible, functions have been placed in specific memory locations by means of `.org` directive. We first extracted the address of the last instruction in the program without function calls (by looking at the `pmem.l43` file). Then, we computed the new addresses by adding for each function an offset of 506. It is worth noting that, if for any reason a `CALL` or `RET` instructions do not succeed due to faults, an alternative path is taken and a different output is provided (this technique is adopted also for testing jump instructions). With this procedure, we obtained an additional coverage of approximately 1%, enough to achieve our target of 90%.

2. Conclusions

By adopting the above strategy, we were successfully able to reach a fault coverage of 90.14% by iterating the main loop 10 times. We did notice that with lower values we achieved a coverage lower than the target one (86.99% with 2 iterations and 89.49% with 5). Conversely, by raising this number to a higher value (500) we obtained a coverage of 90.46% at the expense of extremely lengthy simulation times (approximately 4 hours solely for the logic simulation).
The resulting assembly file (*selftest.s43)* is 3829 lines long and the memory file *pmem.elf* 26Kbytes.
The logic simulation lasted 757975 ns, while the fault simulation around 8 minutes. We find these times very acceptable given the coverage we were able to reach.