Testing Microcontroller Software Simulators

Thomas Reinbacher
Embedded Computing Systems Group
Vienna University of Technology
Treitlstr. 3, 1040 Vienna, AT
reinbacher@ecs.tuwien.ac.at

Dominique Gückel, Stefan Kowalewski
Embedded Software Laboratory
RWTH Aachen University
Ahornstraße 55, 52074 Aachen, DE
lastname@embedded.rwth-aachen.de

Martin Horauer
Dept. of Embedded Systems
Univ. of Applied Sciences Technikum Wien
Höchstädtpl. 5, 1200 Vienna, AT
horauer@technikum-wien.at

Abstract: Software simulators that emulate equivalent behavior of physical microcontrollers play an important role in the process of software development for embedded systems from an early development stage (e.g. when no target hardware is available) to the final verification process (e.g. used in combination with formal methods). Thus, much reliance is put on the correctness of these simulators. This paper presents a practicable approach to test auto-generated and custom microcontroller simulators (both closed and open-source) against a physical device. We show how to set up a test oracle that allows to run the simulators in parallel, validate individual runs based on a comparison of their accumulated state-space, and – in case an error is found – finger-point to the root cause of the error, thus giving valuable support for fixing the discrepancies. A case study shows that the presented testing framework was able to reveal non-trivial bugs in several implementations.

1 Introduction

A microcontroller (MCU) software simulator emulates the functional and (occasionally) temporal execution environment offered by a physical microcontroller. In practice, MCU software simulators are used both by computer scientists and electronics engineers throughout different product development stages for various different activities (e.g., debugging, profiling, testing and verification). For example, firmware development can start prior to the availability of a functional physical target board, using the simulator for debugging purposes. An orthogonal example of an MCU simulator application is the [MC]SQUARE binary code verification framework, cf. [Sch08]. The model-checker of [MC]SQUARE relies on dedicated microcontroller simulators to construct the state-space out of a given binary program image. The state space is then traversed to prove the validity of given properties.

Even though there is no theoretical limitation to precisely imitate a target microcontroller by a software simulator (by the Church-Turing thesis), a faithful simulator is hard to obtain in practice. Thus, a vexing question arises in the context of simulation based verification: *How "correct" is the simulation of the respective physical microcontroller?*

1.1 Behavioral equivalence among heterogeneous MCU implementations

The execution environment of a MCU simulator is typically made up of a linear address space that contains: general purpose CPU registers, the stack, special function registers (for the peripherals, the I/O subsystem and the interrupt controller), and the program and data memory (RAM, EEPROM, Flash, etc.). The simulator executes every instruction of a program in this emulated execution environment, mimicking the behavior of the instructions of the physical MCU. Note that, the simulator does not necessarily need to emulate all the internals of a physical implementation (e.g. how a specific timer is implemented internally); it is usually sufficient to have an equivalent behavior on the register level accessible via the programs' instructions. In contrast, however, an MCU simulator often implements additional means, e.g., to model the environment and thus provide input to the MCU simulator from the outside.

We evaluate the equivalence of a physical MCU and the respective simulator by their configurations $\mathcal{C}_\mathcal{P}$ and $\mathcal{C}_\mathcal{S}$ as follows: Let $\mathsf{Addr} = \{0 \leq x < |\mathsf{Mem}| : x \in \mathbb{N}_0\}$ denote the set of memory locations of the microcontroller, where Mem represents the (linear) address space of the microcontroller memory. We assume a memory mapped I/O architecture (e.g. Intel MCS-51), thus, registers and special function registers reside within Mem. In the following, let $\mathbb{N}_k = \{0, \dots, k-1\}$. A configuration $\mathcal{C}$ of a microcontroller is a tuple $\langle pc, m \rangle \in \mathsf{Locs} \times (\mathsf{Addr} \to \mathbb{N}_{2^w})$, where Locs is a finite set of program counter values, and $m : \mathsf{Addr} \to \mathbb{N}_{2^w}$ is a map from memory locations (with bit-width w) to memory configurations. The state space of the program is thus a subset of $\mathsf{Locs} \times (\mathsf{Addr} \to \mathbb{N}_{2^w})$. The initial microcontroller configuration $\mathcal{C}^0$ is $\langle \texttt{0x00}, m^0 \rangle$ where m^0 represents the configuration of all memory locations after power-up and `0x00` is the assumed reset vector. Two arbitrary configurations $\mathcal{C}$ $\langle pc, m \rangle$ and $\mathcal{C}'$ $\langle pc', m' \rangle$ are considered equivalent if the program counters as well as the memory configurations of $\mathcal{C}$ and $\mathcal{C}'$ are identical, thus $pc \equiv pc' \wedge m \equiv m'$ holds for all combinations of valid and invalid instruction executions.

1.2 Related Work

Commercial and freely available MCU simulators are either manually coded – see e.g. [LDG$^+$08] – or auto-generated from some high-level description, see e.g. [GBK10]. Applications of software simulators in practice are manifold, e.g.:

- debugging when no target hardware is available, e.g., in an early development stage to shorten the time-to-market
- firmware profiling to optimize performance, lower memory or power consumption, or for execution time analysis to verify real-time properties, see e.g. [MOZB10]

- evaluation of sensor-networks with regard to their timing behavior, power consumption, interoperability, etc., see [TLP05, EÖF$^+$09]
- evaluation of multiprocessors with regard to performance, etc., see [GKK$^+$08]
- formal verification as used by the [MC]SQUARE model-checker, to verify properties of binary firmware applications, see [RHS$^+$10]

From the above list it becomes obvious that an incorrect emulation by the MCU simulator of the physical implementation may eventually lead to hazardous outcomes.

Related work in [MPRB09] reports on testing IA-32 CPU emulators, i.e., QEMU, VALGRIND, PIN and BOCHS. Their approach relies on fuzz-testing, that is they set both $\mathcal{C}_P$ and $\mathcal{C}_S$ to a synthetic state, execute one random instruction, and compare the resulting state. Using this approach, they were able to uncover several behavioral deviations in all considered CPU emulators.

A method for verifying instruction-level simulators via step-by-step register state comparison to a hardware implementation is reported in [GL01]. They were able to verify two SPEC benchmarks on a MIPS 12000 CPU simulator.

Other related literature on MCU simulator verification shows that, even carefully engineered CPU simulators do deviate from their silicon counterpart [Fer06, RKK07]. Formal verification, such as model checking [CGP99, BK08], equivalence checking by e.g. multiway decision graphs [BT98], or interactive theorem proving [Bon10], seems to be well suited to cope with this kind of verification problem, however, often fail when applied to reasonably-sized real-life instances. Besides, formal approaches often need a clear understanding and insight into verification goals and the desired correctness criteria.

[Mon00] proposes a black-box testing method that relies on hardware analysis for test-case generation. The validation method aims at both determining the accuracy of the simulator and to pinpoint simulator errors for improving the accuracy. Based on their method they were able to improve the NEC V850 CPU core simulator from an average error rate of 11.2% down to 1.3%.

1.3 Contribution

In contrast to the related work, we propose a lightweight verification framework for microcontrollers that allows to run various simulators against each other and a hardware implementation as oracle that allows to vote over the individual simulation runs. Whenever a deviation is detected, the framework is able to highlight differences in the corresponding data memory dumps, guiding the test engineer to the root cause of the error. We use a mature microcontroller IP-core as a reference device along with a dedicated test-interface to perform the information exchange with a host computer. The presentation in this paper targets the Intel MCS-51 microcontroller architecture, however, is easily transferable to other platforms.

To summarize, the paper makes the following contributions:

- a fully automated testing methodology targeting MCU simulators
- an effective algorithm to generate test-programs stressing the simulators under test
- a prototype implementation of our testing methodology for Intel MCS-51 simulators
- an extensive testing of an auto-generated, several open-source and one commercial closed-source Intel MCS-51 simulators that resulted in the discovery of several defects and several serious bugs

In the remainder of this paper we first describe our test-framework in section 2 before we present a case-study to evaluate our approach in section 3. Section 4 concludes our paper.

2 Test Framework

The proposed framework consists of two modules, i.e., (i) a hex-file generator to provide the respective (random) test cases, and (ii) the execution and evaluation unit.

2.1 Test-case Generation

To generate test cases we apply two strategies (a) *random program generation* and (b) *custom program generation*. The output of both approaches are executable hex files, containing the test cases as a sequence of individual assembly instructions. Whereas in (a) the instructions and data are randomly generated, (b) adopts a test-pattern known from CPU testing, with the aim of adopting a strategy known from everyday software testing, i.e., equivalence classes partitioning, cf. [AO08, p. 150].

2.1.1 Random Input Generation

Random input generation is a well known paradigm in the area of software testing [BM83, CS04]. A microcontroller instruction typically consists of an opcode followed by one or more operands. Especially on CISC based machines, the total length of an instruction (opcode and operands) may vary. In our approach, we first use a random generator to generate an opcode. An internal data structure provides a lookup-table to map to each opcode the total length of the instruction as well as the structure of the remaining operands. For example consider the arithmetical instruction for the Intel MCS-51 microcontroller family:

```
ADD A, #C
```

that adds a constant `#C` to the accumulator `A`. The datasheet [Int94] states that the opcode for the instruction is $b0010.0100$ and the constant (i.e., the operand) is allowed to be in the range of $0 \leq$ `#C` $< 2^n$, where n is the bitwidth of the microcontroller, which in case of the Intel MCS-51 is 8. Whenever the random generator creates the opcode $b0010.0100$, the random generator is constrained to return a number in the range of $0 \leq$ `#C` $< 2^n$.

Pure random testing does not require any manual work, however, it comes with some well-known drawbacks:

- Many sets of input values (instructions and data) may lead to exactly the same observable behavior and are thus redundant.
- The probability of selecting particular inputs that cause buggy behavior may be very low [OH96].
- It performs poor when inputs need to adhere to a certain structure or pattern. Consider the code fragment in Lst. 1 that triggers on specific input data. The probability of detecting the erroneous function `ERROR()` drastically depends on the size of the integer data type, which varies for different target platforms. Suppose the integer parameter `x` is 32 bits wide, the probability of detecting the error evaluates to $p_e = \frac{1}{2^{32}}$.

```c
void function_to_test(int x){
  if (x == 12345) {
    ERROR();
  }
}
```

Listing 1: A C-code snippet subject to random testing.

For our MCU verification framework we implemented a random hex file generator for test-case generation. In contrast to fuzz-testing, our random generator takes the syntactical specification of the respective microcontroller's instruction-set into account. Thus, only valid programs are created that, when parsed correctly, can be executed on the target. Figure 1 depicts the idea of the actual implementation.

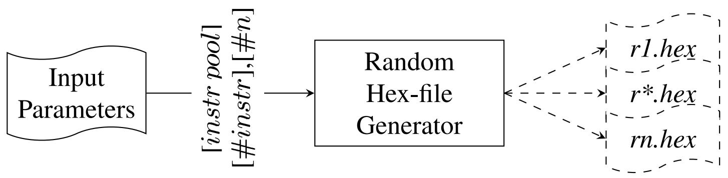


Figure 1: Random hex file generator

For the sake of portability we implemented the random hex file generator as a command line application in JAVA. The arguments are $instr\ pool$ (a list of opcodes the random generator should choose from; in case when this argument is not given the full instruction set is used), $\#instr_{max}$ (an upper bound for the randomly chosen number of total instructions to be included in the hex-file), and $\#n$ (the total number of hex-files to be generated).

2.1.2 Custom Input Generation

In contrast we also experimented with handcrafted input generation. Handcrafted input generation may outperform random input generation in cases where the faulty behavior only exhibits as a consequence of some previously taken actions. On the downside, a cleverly designed handcrafted program requires a skilled test engineer with a deep insight into the particularities of the target MCU, and may therefore be time intensive to create. Handcrafted input generation may easily take the side-effects of an instruction into account, such as setting and deleting status flags, stack pointer modifications, etc. To illustrate, consider the following instruction sequence:

```
ADD  A, #C
ADDC A, #C2
```

where again the first line adds the constant `#C` to the accumulator and the second one performs an *add with carry* instruction. The semantics of the `ADDC` instruction additionally takes the *carry flag* into consideration when calculating the sum, thus, the result of the operation depends on whether the preceding `ADD` instruction causes an overflow of the accumulator or not. Such implications are hard to check with random input sampling.

2.2 Execution and Evaluation Unit

The execution unit is basically a set of scripts for different OS platforms that streams the previously generated instruction snippets in form of hex-files to the respective MCU simulator or the physical MCU via a suitable interface and controls their execution, cf. Figure 2. The configurations of the MCU simulators or physical MCU are dumped to files and evaluated for equivalence by subsequent evaluation scripts. On success the respective tests are marked as 'pass'; otherwise 'fail' is logged along with the information allowing to pin-point to the discrepancies.

Algorithm 1 Single stepping for one MCU simulator or physical MCU.

```
1: for $l = 1 \rightarrow \#n + \#m$ do
2:   $\mathcal{C} \leftarrow \mathcal{C}^0$;
3:   $create(file^l)$;
4:   for $i = 1 \rightarrow \#instr_{max}$ do
5:     $\mathcal{C} \leftarrow simStep()$;
6:     $file^l \ll \mathcal{C}$;
7:   end for
8: end for
```

For $\#n$ random and $\#m$ custom generated hex-files, the algorithm first resets the simulator or physical MCU to a defined 'initial' configuration $\mathcal{C}^0$ and than executes repeatedly (up to $\#instr_{max}$) one instruction at a time ($simStep()$) and dumps the program counter, the registers, and the memory to a log-file (where '$\ll$' denotes concatenation), cf. lines

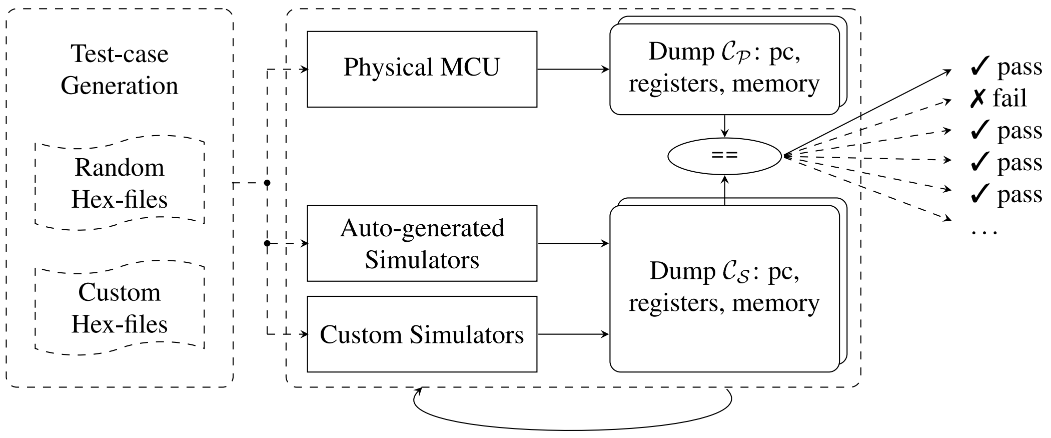

Test-case Generation
Random Hex-files
Custom Hex-files
Physical MCU
Auto-generated Simulators
Custom Simulators
Dump $\mathcal{C}_{\mathcal{P}}$: pc, registers, memory
Dump $\mathcal{C}_{\mathcal{S}}$: pc, registers, memory
==
✓ pass
✗ fail
✓ pass
✓ pass
✓ pass
...

Figure 2: Execution and Evaluation Unit

4-7 of Algorithm 1. Herein, the first set of instructions of every test-program perform some startup-sequence that initializes the memory with random bit-patterns. We execute this sequence for the physical MCU and every simulator under test. Afterwards, we use standard *diff* tools to cross-check the log-files for discrepancies, see Algorithm 2. Hereby, our test approach outputs a verdict that either marks the differences as 'fail' with a referral to the respective assembly instruction of the respective test-program (line 6) or when all dumps are equal a simple 'pass' statement. The entire test execution is fully automated, with the exception of reasoning and categorization of the findings.

Algorithm 2 Comparing simulator and physical MCU memory dumps.

```
1:  for $l = 1 \rightarrow \#n + \#m$ do
2:      $out \leftarrow$ "$pass$";
3:      $load(file^l_{\mathcal{P}}, file^l_{\mathcal{S}})$;
4:      for $i = 1 \rightarrow \#instr_{max}$ do
5:          if $\mathcal{C}^i_{\mathcal{P}} \neq \mathcal{C}^i_{\mathcal{S}}$ then
6:              $out \ll$ "$fail$" $+ i$;
7:          end if
8:      end for
9:      $print(out)$;
10: end for
```

2.3 Register and Memory Dumps

One restriction of our automated approach is that the simulators under test should possess an interface to control execution of the test-programs and dump the registers and memory information to a log-file. This is easily accomplished when some command-line interface is provided, requires, however, some GUI automation when this is not the case. All dumps are written in a common format, following the one used by the Unix HEXDUMP utility.

For likewise execution and evaluation of the test-programs on the physical MCU (an IP-core running on an FPGA platform) we re-use some test-circuitry that interfaces via a high-speed USB 2.0 interface to our host, see [RSM+11] for details.

3 Case Study

In the following, we show the feasibility of our framework by a case study in which we instantiate seven different implementations of the Intel MCS-51. All experiments were performed on a Intel Core i5 CPU equipped with 4 GB of RAM, running Linux. We have conducted the experiments with the expressed aim of answering the following questions:

Q1 Is the framework presented a practicable method to increase confidence in the correctness of MCU simulators?

Q2 What kind of errors can be found with (a) random input programs and (b) handcrafted test programs?

3.1 Simulators

Even though our framework is applicable to a wide range of simulators, we based our experiments on the Intel MCS-51 architecture. The main motivation for selecting this MCU was to increase confidence in the respective simulators that are used within [MC]SQUARE. These simulators are used to create state spaces from binary programs, which are later checked by the actual model checking algorithm for property violations. Hence, an incorrect simulation would result in incorrect state spaces, and could therefore culminate in either errors being missed by the model checker, or false alarms for actually sound programs.

For our experiments in this section, we examined the following set of simulators:

- [MC]SQUARE
  - **C51SIM** ($\mathcal{S}_i$) A handwritten, thoroughly tested simulator [Rei09] that was created with the intention of building state spaces for model checking Intel MCS-51 binary programs.
  - **MCS51SIM** ($\mathcal{S}_{ii}$) A synthetic simulator created from a hardware description, compiled by the simulator compiler [GBK10] contained in [MC]SQUARE. During its development, it was compared with the already existing C51SIM by running several case study programs simultaneously in both simulators.

- Open Source
  - **μCSIM** ($\mathcal{S}_{iii}$) A console based simulator that supports various microcontroller targets. We used the Intel MCS-51 port, version 0.5.4, which is part of the SDCC framework. Available from `http://sdcc.sourceforge.net`
  - **GSIM51** ($\mathcal{S}_{iv}$) A console based simulator originating from a student project, version 1.1. Available from `http://gsim51.sourceforge.net`
  - **EMU8051** ($\mathcal{S}_{v}$) This console based simulator, of which we used version 1.1.0, has been under development since 1999. Available from `http://www.hugovil.com/fr/emu8051`
- Industrial Strength
  - **KEILC51** ($\mathcal{S}_{vi}$) A commercial, industrial-strength simulator, shipped with the Keil μVision3 v 3.23 IDE. Further details about this tool are available at `http://www.keil.com/c51`
- FPGA based
  - **OREGANO** ($\mathcal{S}_{vii}$) The Oregano Systems 8051 IP core is a parameterizable, synthesizable circuit description of the Intel MCS-51 intended to run on an FPGA. Available from `http://www.oregano.at`

3.2 Benchmarks

We used two distinct benchmark sets Γ_R and Γ_H with the expressed aim to cover a wide range of valid opcodes.

Random Γ_R: A set of random hex files generated by the random hex file generator introduced in section 2.1.1.

Handcrafted Γ_H: We adopted a benchmark set[1] which was generated at the University of California at Riverside for a synthesizable VHDL Model of the Intel MCS-51 in course of their Dalton project [VG01].

Note that, [MC]SQUARE has no explicit real-time model, thus the simulators C51SIM and MCS51SIM do not implement timers and interrupts in the same manner as stated in the datasheet [Int94]. For example, an expired timer will generate two successor states, one representing the entry into the interrupt service routine and the second for the state following the call to the interrupt handler, therefore, deviate from common simulator implementations. We thus did not aim to find errors depending on certain interrupt configurations, as they are not comparable to the [MC]SQUARE simulators in a 1:1 manner.

[1] Available at `http://www.cs.ucr.edu/~dalton/i8051/i8051syn`

The benchmarks $\mathit{\Gamma}_R$ and $\mathit{\Gamma}_H$ are subsequently loaded into the execution unit of our framework. Test cases that become tagged with a 'fail' verdict by the execution unit are manually inspected to figure out the root cause of the deviation.

3.3 Results

Table 1 states the results generated by the framework when running benchmark sets $\mathit{\Gamma}_R$ and $\mathit{\Gamma}_H$. The entry #TC is the number of test cases generated, i.e., the number of hex files and #Instr is the number of instructions traversed by the test case. The column Dump size states the size of the corresponding memory dumps for a single simulator, whereas Time is the total execution time of the test run.

Benchmarks			**Test cases passed**						**Dump size**	**Time**
Name	# TC	# Instr	$\mathcal{S}_i$	$\mathcal{S}_{ii}$	$\mathcal{S}_{iii}$	$\mathcal{S}_{iv}$	$\mathcal{S}_v$	$\mathcal{S}_{vi}$	[MB]	[s]
$\mathit{\Gamma}_R$	20	2531	16	12	13	12	11	20	2.0	2:27
$\mathit{\Gamma}_H$	3	952	2	0	0	0	0	3	0.7	0:41
Total	23	3483	18	12	13	12	11	23	2.7	3:08

$\mathcal{S}_i$	...	C51SIM	$\mathcal{S}_{iii}$	...	μCSIM	$\mathcal{S}_v$	...	EMU8051
$\mathcal{S}_{ii}$	...	MCS51SIM	$\mathcal{S}_{iv}$	...	GSIM51	$\mathcal{S}_{vi}$	...	KEILC51

Table 1: Experimental results for benchmarks $\mathit{\Gamma}_R$ and $\mathit{\Gamma}_H$.

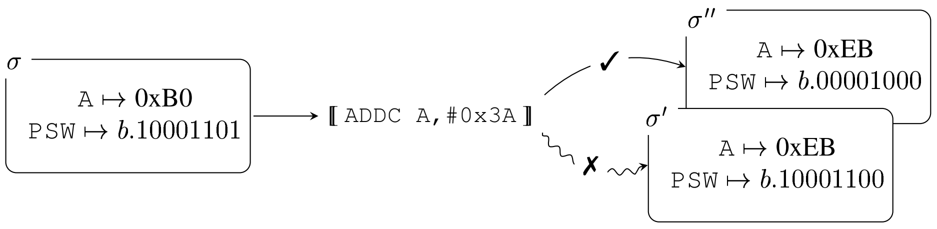
At a first glance the number of 23 different test-programs seems rather low – and is in fact far from exhaustive. Nevertheless, we found several errors and deviations compared to the physical MCU in all simulator implementations except one, the KEILC51 simulator. Both benchmark sets were able to reveal implementation bugs. We classify the found deviations into (a) wrong status flag updates, (b) illegal state modifications, (c) corner cases, and (d) exceptions. In the following we will give some examples of the bugs being found by our framework.

Wrong status flag updates. The Intel MCS-51 architecture holds status flags that reflect additional information about the executed instruction (such as over- and underflow information) in the program status word (`PSW`)[2]. Missed or wrongly set status flag may have severe effects on both the control flow as well as the configuration. For example consider the instruction `JNZ CY, C:0x100` that jumps to address `0x100` iff the carry flag `CY` is set. Suppose a simulator wrongly clears the `CY` flag after, say, an overflowing `ADD` instruction, then execution will continue at the program counter location following the `JNZ CY, C:0x100` instruction, thus, skipping parts of the intended control flow.

- *Overflow Flag (OV) not set.* The instruction `ADDC A,B` simultaneously adds the content of register `B`, the carry flag and the accumulator contents, leaving the results

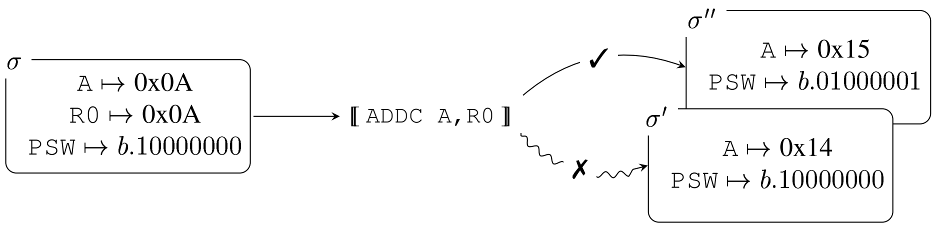
[2]The respective flags in the `PSW` are: [`CY`|`OV`|`F0`|`RS1`|`RS0`|`OV`|`-`|`P`]

in the accumulator [Int94]. Consider the following instance of the instruction, where the values `0xB0` and `0x3A` together with the carry flag are added: While the result



of the addition is correct, however, the remaining bit values in the `PSW` are not. The faulty implementation does not clear neither the `CY` nor the `OV` flag. The semantic definition of status flag updates of the `ADDC A,B` reads as follows: *"The carry and the auxiliary carry flags are set, respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise … `OV` is set if there is a carry-out of bit 6 but not out of bit 7, or a carry-out of bit 7 but not out of bit 6, otherwise `OV` is cleared".* Analyzing the addition on hardware level, reveals that there is only a carry out from bits 4 and 5, thus, the correct implementation needs to clear both the `CY` as well as the `OV` flag.

- *Add with Carry without Carry.* An akin deviation we found, is that – occasionally – the carry flag does not enter the addition, causing the accumulator to hold the value of (`A+R0`) instead of (`A+R0+CY`).

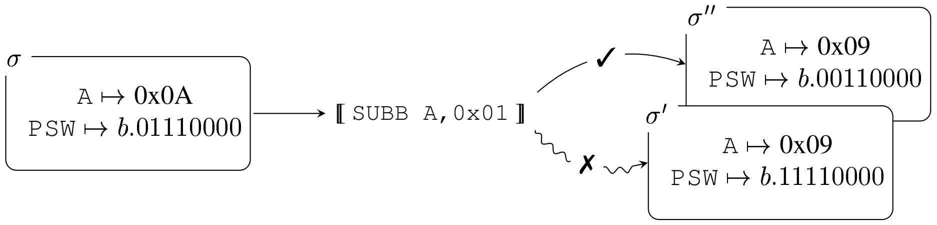


It is notable that this is a particularly delicate bug. One of the test cases contained the sequence:

```
(1)  ADDC  A, R0
(2)  SUBB  A, #0x15
(3)    JZ  C:0064
(4)    ...
```

In the faulty implementation, executing (1) yields the state σ' as above, (2) subtracting the constant `#0x15` from `#0x14` causes an overflow and leaves `0xFF` in the accumulator. Hence, the faulty implementation will reach (3) with $\sigma'_3\langle A \mapsto \texttt{0xFF}\rangle$ whereas a correct implementation will yield $\sigma''_3\langle A \mapsto \texttt{0x00}\rangle$. Obviously, the latter will take the *true* branch of the conditional jump at location (3), whereas the former will erroneously take the *false* branch.

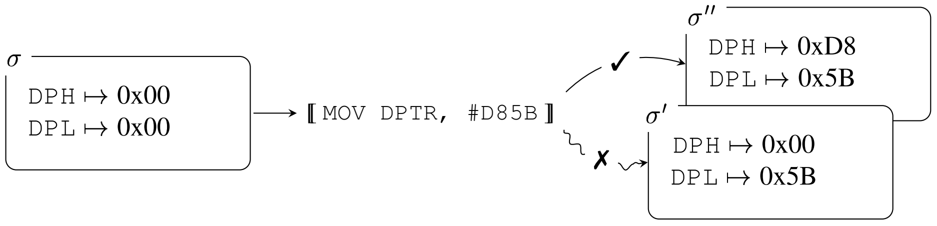
- *Subtraction using addition.* Subtraction can be implemented by using two's complement and an addition operation. Hence, it is tempting for simulator developers to reuse the existing code for addition. However, this has undesired side effects on the flags. For instance, on the 8 bit architecture MCS-51, the two's complement of `0x01` is `0xFF` $-$ `0x01` $+$ `0x01` $=$ `0xFF`. Adding `0xFF` to `A` will therefore have the same effect on the data as `A` $- 1$, but will cause an overflow, and therefore may result in flags being set differently.



In the scenario above, the value of `A` is decremented from `0x0A` to `0x09`. Similarly to the previous example, the incorrect values of the carry and auxiliary carry flag in σ' may result in data errors after subsequent arithmetic operations, and finally lead to incorrect control flow.

Illegal state modifications. Wrong or missing status flag updates mostly cause a change of the program flow (as the conditional of branches are often status flags). However, we also found severe bugs in various implementations that lead to real data faults. In the following, we will show some representative examples:

- *Writing the data pointer.* The data pointer on the Intel MCS-51 architecture is a double word, thus, internally stored in two distinct 8 bit registers, i.e., `DPH` and `DPL`. Moving a constant to the data pointer requires the simulator to load the high byte of the constant into `DPH` and the low byte into `DPL`. Even though this seems to be easily implemented, it is indeed the case that one of the simulators wrongly ignores the high byte of the constant that is meant to be shifted into the register `DPH`. Wrongly setting the data pointer may have several severe consequences. Switch

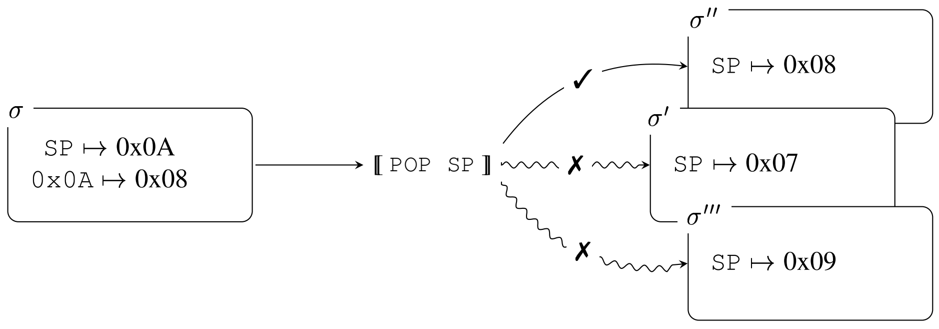


case statements are often efficiently translated to jump tables. The entries of the tables (e.g. the comparison value, the jump targets) are then loaded at run time from the program memory where the program memory is addressed by the sum of the accumulator value and the data pointer. Another scenario is when one loads, e.g., a value table of a trigonometric function from the program memory into the RAM. If the data pointer is corrupted, the involved instruction will likely interpret some random program bytes as the value table.

- *Writing automatically updated memory*. Certain memory locations are updated automatically on the physical device. For instance, the *parity flag* is set to 1 after each instruction iff the number of 1 bits in `A` is odd. Thus, instructions writing the parity flag should have no effect. A few simulators did not abide by this, creating states that are logically inconsistent. Programs relying on the value of this flag would therefore operate on flawed data.

Corner Cases.

- *Popping a byte into the SP*. After popping a value from the stack the stack pointer (`SP`) is decremented to point to the next element on the stack. The Intel MCS-51 instruction set explicitly allows to pop a value from the stack into the register holding the `SP`. In this case the stack pointer is overwritten with the value from the stack and *not* decremented afterwards [Int94]. However we could reveal two different implementations of the `POP SP` command.



One implementation wrongly decremented the stack pointer yielding σ', another one even incremented the stack pointer yielding σ'''.

Exceptions. Loading the HEX files we used for the tests was possible with all simulators except for one. For the generated simulator, we discovered a problem with programs starting at addresses greater than 0. The simulator always expected this start address and would yield an exception in any other case, thus preventing simulation. Therefore, we implemented a workaround that allowed this simulator to execute our tests.

4 Conclusion & Future Work

In this paper we present a methodology to check the functional equivalence of microcontroller simulators with its physical counterpart. Our (mostly automated) approach relies on auto-generated random and/or custom hex-files that are fed to both the simulators and the physical implementation, respectively. After the execution of every single instruction the actual configuration, consisting of the program counter, the registers and the data memory is dumped to log-files. After the execution of the test-cases, every simulators' log-files are compared with the physical MCU's log-files that serve as test-oracle. Differences are marked and cross-referenced with the involved instructions, allowing the test-engineer to determine the actual cause(s) thereof. Experiments conducted with auto-generated, closed- and open-source Intel MCS-51 simulators revealed several flaws and even severe bugs in almost all implementations.

Future work needs to work on mechanisms to ease the interpretation of the results and to rule out repetitive occurrences of the same encountered bugs that show-up multiple times under different input conditions. Furthermore, the presented approach will benefit by the provision of a test-coverage, a more efficient handling of the state spaces (e.g., by relying on state-changes rather than entire states), and a speed-up of the execution. One obstacle for the latter, however, is the lack of standardized interfaces to control the simulators (especially when only a GUI is available).

Acknowledgement The work of Thomas Reinbacher has been supported within the FIT-IT project CEVTES managed by the Austrian Research Agency FFG under grant 825891. The work of Dominique Gückel and Stefan Kowalewski has been, in part, supported by the DFG Cluster of Excellence on *Ultra-high Speed Information and Communication*, German Research Foundation grant DFG EXC 89 and by the DFG research training group 1298 *Algorithmic Synthesis of Reactive and Discrete-Continuous Systems*.

References

[AO08] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. ISBN: 0521880386.

[BK08] C. Baier and J. P. Katoen. *Principles of Model Checking*. The MIT Press, 2008. ISBN 026202649X.

[BM83] D. L. Bird and C. U. Munoz. *Automatic generation of random self-checking test cases*. IBM Systems Journal, 22(3):229 –245, 1983.

[Bon10] M. P. Bonacina. *On theorem proving for program checking: historical perspective and recent developments*. In PPDP, pages 1–12. ACM, 2010.

[BT98] S. Balakrishnan and S. Tahar. *Modeling and formal verification of a commercial microcontroller for embedded system applications*. In *ICM*, pages 107 –110, 1998.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999. ISBN 0262032708.

[CS04] C. Csallner and Y. Smaragdakis. *Jcrasher: an automatic robustness tester for Java*. Software: Practice and Experience, 34:2004, 2004.

[EÖF+09] J. Eriksson, F. Österlind, N. Finne, N. Tsiftes, A. Dunkels, T. Voigt, R. Sauter, and P. J. Marrón. *COOJA/MSPSim: Interoperability Testing for Wireless Sensor Networks*. In SimuTools, 2009.

[Fer06] P. Ferrie. *Attacks on Virtual Machine Emulators*. Technical report, Symantec Advanced Threat Research, 2006.

[GBK10] D. Gückel, J. Brauer, and S. Kowalewski. *A System for Synthesizing Abstraction-Enabled Simulators for Binary Code Verification*. In SIES, pages 118–127, 2010.

[GKK+08] L. Gao, K. Karuri, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. *Multiprocessor performance estimation using hybrid simulation*. In DAC, pages 325–330. ACM, 2008.

[GL01] B. Glamm and D.J. Lilja. *Automatic verification of instruction set simulation using synchronized state comparison*. In Proc. of Annual Simulation Symposium, pages 72 – 77, 2001.

[Int94] Intel Cooperation. *MCS 51 Microcontroller Family User's Manual*, 1994. Order No.: 272383-002.

[LDG+08] M. Lv, Q. Deng, N. Guan, Y. Xie, and G. Yu. *ARMISS: An Instruction Set Simulator for the ARM Architecture*. In ICESS, page 548, July 2008.

[Mon00] S. Montan. *Validation of Cycle-Accurate CPU Simulators against Real Hardware*. Master's thesis, Uppsala University, Department of Computer Systems, 2000.

[MOZB10] W. Mueller, M.F. Oliveira, H. Zabel, and M. Becker. *Verification of real-time properties for Hardware-dependent Software*. In IEEE International High Level Design Validation and Test Workshop, page 154, June 2010.

[MPRB09] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. *Testing CPU emulators*. In ISSTA, pages 261–272. ACM, 2009.

[OH96] J. Offutt and J. Hayes. *A Semantic Model of Program Faults*. In ISSTA, pages 195–200. ACM Press, 1996.

[Rei09] T. Reinbacher. *Model Checking and Static Analysis of MCS-51 Assembly Code*. Master's thesis, Department of Embedded Systems, University of Applied Sciences Technikum Wien, June 2009.

[RHS+10] T. Reinbacher, M. Horauer, B. Schlich, J. Brauer, and F. Scheuer. *Model Checking Embedded Software of an Industrial Knitting Machine*. IJITCC, 1(2):186–205, 2010.

[RKK07] T. Raffetseder, C. Kruegel, and E. Kirda. *Detecting System Emulators*. In ISC, 2007.

[RSM+11] T. Reinbacher, A. Steininger, T. Müller, M. Horauer, J. Brauer, and S. Kowalewski. *Hardware Support for Efficient Testing of Embedded Software*. In MESA, 2011. to appear.

[Sch08] B. Schlich. *Model Checking of Software for Microcontrollers*. Dissertation, RWTH Aachen University, Aachen, Germany, June 2008.

[TLP05] B. Titzer, D. K. Lee, and J. Palsberg. *Avrora: Scalable sensor network simulation with precise timing*. In IPSN, pages 477–482, April 2005.

[VG01] F. Vahid and T. Givargis. *Platform Tuning for Embedded Systems Design*. IEEE Computer, 34(3):112–114, 2001.