# Randomised testing of a microprocessor model using SMT-solver state generation

Brian Campbell and Ian Stark

LFCS, School of Informatics, University of Edinburgh
Brian.Campbell@ed.ac.uk
Ian.Stark@ed.ac.uk

**Abstract.** We validate a HOL4 model of the ARM Cortex-M0 microcontroller core by testing the model's behaviour on randomly chosen instructions against a real chip.
The model and our intended application involve precise timing information about instruction execution, but the implementations are pipelined, so checking the behaviour of single instructions would not give us sufficient confidence in the model. Thus we test the model using sequences of randomly chosen instructions.
The main challenge is to meet the constraints on the initial and intermediate execution states: we must ensure that memory accesses are in range and that we respect restrictions on the instructions. By careful transformation of these constraints an off-the-shelf SMT solver can be used to find suitable states for executing test sequences.

**Keywords:** Randomised testing, microprocessor models, HOL, SMT

Mechanised formal models of instruction set architectures provide a basis for low-level verification of software. Obtaining accurate models can be difficult; most architectures are described in large reference manuals consisting primarily of prose backed up by semi-formal pseudo-code. Once a model is produced it is common to test individual instructions against hardware to gain confidence in the model (for example, [7]), but some effects may require a sequence of several instructions to appear.

This is relevant for the intended application of our model. We wish to extend existing low-level verification work using Myreen's decompilation [9] to include timing properties. We want to use a realistic processor with a relatively simple cost model, because our technique is largely orthogonal to the low-level timing analysis and we do not wish to spend resources recreating a complex worst-case execution time analysis (such as those surveyed in [13]).

While the ARM Cortex-M0 has a simple cost model, even this design has a non-trivial microarchitecture in the form of a three stage pipeline (fetch, decode and execute). Hence to build confidence in our model, and in particular the timing information contained in the model, we wish to test sequences of instructions in order to exercise the pipeline.

However, finding a processor state in which an arbitrary sequence of instructions can be executed without faulting is not always easy. Consider the sequence

```
ldrsh   r0, [r1, r2]   ;  load r0 from r1+r2 (16 bits, sign-extended)
lsls    r0, r0, r2     ;  shift r0 left by r2
bcs     +#12           ;  branch if carry set

add     r0, r0, r2     ;  add r2 to r0
ldr     r3, [r0, #0]   ;  load r3 from r0
```

**Fig. 1.** An example test sequence of M0 code

in Figure 1. The final instruction reads a word from memory at an address stored in the `r0` register. We must ensure that this address is a valid location in memory, which makes up a tiny fraction of the processor's address space.

Moreover, `r0`'s contents is a result of several operations: a half-word load, sign-extension, shift and bitvector addition, where `r2`'s value is used in several different ways. Finding solutions to such constraints is a natural application for an SMT (Satisfiability Modulo Theories) solver with good bitvector support.

The solutions provided by the solver will specify parts of the registers and memory, including the placement of instructions. For example, if the `bcs` branch is taken then the next instruction must be placed 12 bytes later. Whether the processor takes the branch is also decided by a non-trivial calculation due to the shift instruction; again, we leave this to the SMT solver.

Note that the constraints come from two sources; some, such as instruction alignment, come directly from the model in the form of hypotheses to a theorem. However, the model is not intended to be comprehensive, especially where details about particular implementations are concerned. Thus we need to add additional constraints to capture these details, such as the size and layout of memory.

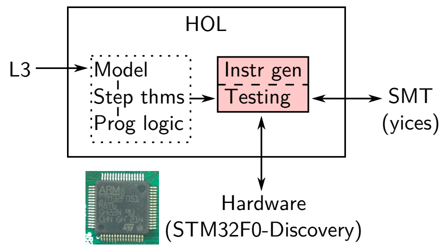

**Fig. 2.** Outline of the testing system

The testing system is outlined in Figure 2. The model is written in the L3 specification language, and the generated HOL version is accompanied by tool libraries, shown in the dashed box. Instruction generation is separated from the main testing code, and can be overridden manually. An SMT solver is invoked to help find a pre-state using an adaption of the HolSmtLib library [12], and the hardware is invoked over a USB link. Our software is available online[1].

[1] `https://bitbucket.org/bacam/m0-validation`

Our contributions are to demonstrate that symbolic evaluation with SMT constraint solving is an effective way to test a formalised microprocessor model, and that the ability to add additional constraints is useful for bridging the gap between a model and an implementation, and for checking hypotheses about deviations from the model.

Section 1 briefly describes the M0 model and accompanying tools, then the description of the testing system begins with the generation of instructions in Section 2 and continues in Section 3 with the construction of pre-states which satisfy the restrictions on successful execution. Section 4 discusses the practicalities of running the tests on the hardware, and Section 5 presents the outcomes of testing. We consider the results and variations of the system in Section 6, then consider related work before concluding.

## 1 The processor model

The ARM Cortex-M0 model we use was developed by Anthony Fox in his L3 domain specific language [6]. It is a greatly simplified adaption of his ARMv7 model, with the addition of instruction cycle timings from the ARMv6-M reference manual [1]. L3 provides a specification language with imperative features that allows models to closely follow the pseudo-code typically found in such manuals.

An automatic translation produces a version for the HOL4 proof assistant [8], together with Standard ML versions of the instruction decoder and encoder. The main interface to the model itself is a step function,

$$\mathsf{NextStateM0} : \mathit{m0\_state} \to \mathit{m0\_state}\ \mathit{option},$$

where the type *m0_state* is a record containing register values, memory content, flags, and other miscellaneous information about the processor state. The memory is represented as a HOL function from 32-bit words representing addresses to 8-bit contents.

```
[Aligned (s.REG RName_PC,2),     s.MEM (s.REG RName_PC) = 3w,
 Aligned (s.REG RName_0 + 0w,4), s.MEM (s.REG RName_PC + 1w) = 104w,
 ¬s.AIRCR.ENDIANNESS, ¬s.CONTROL.SPSEL, s.exception = NoException]
⊢ NextStateM0 s = SOME (s with
      <|REG := (RName_PC =+ s.REG RName_PC + 2w)
              ((RName_3 =+ s.MEM (s.REG RName_0 + 0w + 3w) @@
                           s.MEM (s.REG RName_0 + 0w + 2w) @@
                           s.MEM (s.REG RName_0 + 0w + 1w) @@
                           s.MEM (s.REG RName_0 + 0w)) s.REG);
        count := s.count + 2; pcinc := 2w|>)
```

**Fig. 3.** Example step theorem for `ldr r3,[r0,#0]`

Additional tools are provided in two HOL libraries. The first, `stepLib`, provides symbolic evaluation of the step function for individual instructions. An example is shown in Figure 3. The hypotheses assert that the program counter and source address are correctly aligned, that the instruction is present in memory and that certain control flags are set correctly. The conclusion states that the model's step function will succeed with an updated state, where the program counter is moved forward, the result of the load is present and two ticks of the processor's clock have passed. (The HOL term `(`$k$ `=+` $v$`)` $m$ denotes a map which returns $v$ at $k$ and is $m$ everywhere else.) For branches, which are the only conditional instructions in the ARMv6-M architecture, two theorems are returned: one for when the branch is taken, and one for when it is not.

The second library provides separation-logic-like specifications for instructions. The principal intended use for the model is to provide low-level verification using Myreen's decompilation technique [9], and these specifications provide the interface between the model and the decompilation library.

We could test the model directly, or test one of these two libraries. In order to determine what constraints on the state are necessary for successful execution we need to perform symbolic evaluation of the instructions. If we used the separation-logic specifications we would have to make memory aliasing decisions before we can obtain the preconditions. Thus by testing `stepLib` we obtain the symbolic evaluation and can leave the aliasing decisions to the SMT solver.

The model does not currently support interrupts and exceptions, which are not necessary for the verification work we intend to do in the near future. Hence we leave these for further work, briefly discussing them in Section 6.

## 2 Instruction sequence generation

We produce instruction sequences by randomly picking instructions that are supported by the model. The M0's subset of the Thumb instruction set is fairly small; the reference manual lists 77 instructions [1, §A6.7] of which the model only supports the user-level instructions. Thus we take the opportunity to provide a fresh list of the instruction formats for cross checking against the model. In Section 6 we will consider alternative approaches.

Figure 4 gives a datatype for fragments of M0 instruction formats and a few sample formats. While the datatype has a few generic constructors for literal and immediate bit strings, the rest are specialised for targeting the M0. Registers

```
datatype instr_format =  Lit of int list    | Imm of int
    | Reg3             | Reg4NotPC          | Reg4NotPCPair
    | CmpRegs          | RegList of bool (* inc PC/LR *)
    | STMRegs          | Cond               | BLdispl

val instrs = [
( 1,([Lit [0,0,0,1,1,0,0], Reg3, Reg3, Reg3],           "ADD (reg) T1")),
(14,([Lit [1,1,0,1], Cond, Imm 8],                      "B T1")),
( 1,([Lit [0,1,0,0,0,1,1,1,0], Reg4NotPC, Lit [0,0,0]], "BX")),
( 1,([Lit [1,1,1,1,0], BLdispl],                        "BL")),
...
```

**Fig. 4.** Language for instruction formats and sample formats

come in three-bit and four-bit representations[2], with several further variants for pairs of registers and specific instructions.

Each format consists of a list of fragments. For example, the `ADD` instruction format in Figure 4 has a constant prefix, the destination register and two source registers. The integer before the format is a weighting, which is used here to make the conditional branches appear more often than other instructions. A name is given to each format for debugging and logging purposes.

The Branch with Link instruction format `BL` is the most unusual. It is the only 32-bit instruction supported by the model (the rest are system instructions) and the offsets for the jump are very large (up to 16MB) compared to the amount of memory available (kilobytes). Thus almost all `BL` instructions are unusable, and untestable. To produce useful instructions the `BLdispl` fragment picks and encodes branch displacements with magnitude bounded by the size of SRAM.

### 2.1 Sanity checks

We perform several automated tests of our generator. The first is an internal consistency check which ensures that every format is either 16 or 32 bits long.

To cross check the generator against the model, we split the list of instruction formats into the instructions that we expect the model to support, and those that we believe the model does not. We then ask the model for theorems about several instances of each format, ensuring that they are present for all supported instructions and absent for all other instructions. We also check that only the conditional branches generate multiple theorems.

Following the discovery of an extra instruction in the model, we ensured that no further extra instructions were present by performing a manual check of the format list against `stepLib`. It may be possible to devise automated checks of this, but the small instruction set makes a manual comparison the most effective use of time. More details, and the other testing results, can be found in Section 5.

## 3 Generating pre-states for testing

Having chosen a sequence of instructions we can obtain theorems describing the behaviour of each one from `stepLib` and combine them into a single result by taking the theorem for the final instruction and repeatedly instantiating the start state with the step theorem for the previous instruction, simplifying at each step to keep the symbolic state a reasonable size. Also, to keep this process manageable we randomly pick whether to take each branch so that we only have one step theorem for each instruction.

Recall the example sequence in Figure 1. We saw the step theorem for an `ldr` instruction in Figure 3. After combining the step theorems the conditions become more complex due to the symbolic execution. For example, the requirement from the `ldr` instruction that the address for the load in `r0` is aligned is now expressed in terms of the initial state, reflecting how `r0` is calculated:

[2] Many instruction formats only use `r0` to `r7` to keep the instructions compact.

```
Aligned (s.REG RName_2 +
         sw2sw (s.MEM (s.REG RName_1 + s.REG RName_2 + 1w) @@
                s.MEM (s.REG RName_1 + s.REG RName_2))
         <<~ w2w ((7 >< 0) (s.REG RName_2)), 4)
```

where `(7 >< 0)` selects the low byte of `r2` and `<<~` performs the shift.

These hypotheses make up the bulk of the constraints the pre-state must satisfy, but we must also meet additional requirements that are imposed by the hardware, and translate the constraints into a form suitable for the solver.

### 3.1 Additional requirements

For successful execution on the device, the state must satisfy requirements about areas where the model is deliberately incomplete (often because they vary between implementations):

– self-modifying code must be forbidden;
– the memory map and its restrictions must be respected;
– a test harness is required to stop execution cleanly; and
– the model's implicit invariant that stack pointers are always aligned must be enforced.

The last point is the result of starting execution from a state loaded by the debugger; the model (and the manual's pseudo-code) establish the alignment of the stack pointers on reset and maintain it throughout execution. We must respect that invariant in our states, or the debug interface will reject them.

To implement the restrictions on self-modification and the memory map we need to know the symbolic positions of every instruction and memory access. Recovering this information after combining the step theorems would be difficult at best. For example, if we throw away the result of a load,

```
ldr r0, [r1, #0]
ldr r0, [r2, #0]
```

then the use of the address in `r1` will disappear in the combined theorem. Instead, we record the symbolic addresses in each step theorem by adding free variables for the set of instruction locations and accessed memory locations, and hypotheses to give symbolic values to them. The symbolic value for the instruction is given by the program counter, and we can find all memory accesses by finding the terms which consult the memory field of the state, `s.MEM`.

For example, for the final instruction in the example, we add a hypothesis for the instruction location, two for the memory access for the instruction, and another four for the word that is loaded:

```
instr_start 4 = s.REG RName_PC,
memory_address 4 = s.REG RName_PC,
memory_address 5 = s.REG RName_PC + 1w,
memory_address 0 = s.REG RName_0,
memory_address 1 = s.REG RName_0 + 1w,
memory_address 2 = s.REG RName_0 + 2w,
memory_address 3 = s.REG RName_0 + 3w,
```

Combining the extended step theorems as before performs symbolic evaluation of the earlier instructions, restating these addresses in terms of the initial state:

```
instr_start 4 = s.REG RName_PC + 8w,
memory_address 4 = s.REG RName_PC + 8w,
memory_address 5 = s.REG RName_PC + 9w,
memory_address 0 = s.REG RName_2 +
                   sw2sw (s.MEM (s.REG RName_1 + s.REG RName_2 + 1w) @@
                          s.MEM (s.REG RName_1 + s.REG RName_2))
                   <<~ w2w ((7 >< 0) (s.REG RName_2)),
 ...
```

The remaining addresses are similar. We can then add constraints requiring all accesses to be in the range of the target device's SRAM. For self-modification we can discover the symbolic locations written to by examining the memory field of the post-state record, then constrain them to be disjoint from the instructions.

By extracting the expression for the post-state program counter we can synthesise locations for the harness instructions, and so add constraints requiring them to be present and unmodified by execution. In most of our tests the harness consists of a single software breakpoint instruction[3].

## 3.2 Practical HOL

The technique outlined above of combining step theorems by instantiation and simplification is intended to keep the symbolic state manageable. However, there are two further issues we need to address to achieve this.

The main problem was that when a memory update is distributed to the uses of memory by evaluation, subsequent memory lookups will be partially evaluated into aliasing checks. These grow quickly, especially when load-multiple and store-multiple instructions are involved; every address written to is compared against every address read. We could attempt to avoid exposing these aliasing checks by curtailing evaluation, but we chose the more compact solution of introducing free variables for the intermediate memory states. The updates from store instructions are moved to hypotheses which constrain the free variable to be exactly the intermediate memory state.

For example, storing a single byte with `strb r0, [r1,#0]` updates the memory field of the state record,

```
s with <|MEM := (s.REG RName_1 =+ (7 >< 0) (s.REG RName_0)) s.MEM;
         REG := (RName_PC =+ s.REG RName_PC + 2w) s.REG;
         count := s.count + 2; pcinc := 2w|>
```

so we add the new hypothesis

```
Abbrev (mem_step_0 = (s.REG RName_1 =+ (7 >< 0) (s.REG RName_0)) s.MEM)
```

using HOL's `Abbrev` mechanism to prevent the definition being used during simplification, and put the variable in the state record:

[3] Hardware breakpoints would also work, but are slightly harder to use on our main target device.

```
s with <|MEM := mem_step_0;
         REG := (RName_PC =+ s.REG RName_PC + 2w) s.REG;
         count := s.count + 2; pcinc := 2w|>
```

The second problem is that unconstrained evaluation will reveal implementation details about the model and HOL libraries, which it is important to avoid for the SMT translation described in the next section, and to keep the size of terms reasonable. Restricting the computation rules used during evaluation prevents this, along with careful choices of simplification and rewrite rules.

## 3.3 Preparing for SMT solving

The constraints we obtain consist largely of bitvector operations, first-order logic and a little natural number arithmetic, which suits the abilities of a number of SMT solvers. HOL4 already comes with the HolSmtLib package for interfacing with Yices 1 and Z3 [12]; we used Yices because HolSmtLib's translation for Yices supports a greater range of HOL types and terms. We do not expect that using another solver would pose any major problems.

Normally, HolSmtLib proves goals where the free variables are universally quantified. To use an SMT solver in this way the library must negate the goal and check that that is *unsatisfiable*. We thus have to adapt the library, because we wish to check that the constraints are *satisfiable*, treating the free variables as *existentially* quantified and using the satisfying assignment returned for them to construct the pre-state for testing. Hence we adapted the library to remove the negation and parse the satisfying assignments.

However, the subset of HOL supported by the translation to Yices' input language is still rather small, and the definitions used in the model do not always fit within it. For example, the `Aligned` predicate is defined by

```
Aligned (w,n) = (w = n2w (n * (w2n w DIV n)))
```

which is not well supported as it switches between bitvectors and natural numbers using `n2w` and `w2n`. Thus during the process of combining the step theorems above we are careful not to unfold definitions like `Aligned`. Instead, we use HOL theorems about them to rewrite them into the supported subset. For example, the model already provides the result

```
(∀a : word32. Aligned (a, 1)) ∧
(∀a : word32. Aligned (a, 2) = ~word_lsb a) ∧
(∀a : word32. Aligned (a, 4) = ((1 >< 0) a = 0w:word2)) ∧
(∀a : word8.  Aligned (a, 4) = ((1 >< 0) a = 0w:word2))
```

which provides a bitvector interpretation for all of the necessary cases. We prove simple results to ensure that bit selection, shifts and addition are also in the expected form, and slightly more complex ones to obtain the overflow and carry bits for addition.

By using HOL theorems we know that the transformations are *sound*; the worst case for an error is that we end up with constraints that the SMT solver cannot handle. We also extended the transformation itself for a few terms that are awkward to deal with in HOL: right rotation, map updates, 8-bit bitvector to natural conversion, and variable bit indexing. The latter two are defined in a brute-force fashion by large `if-then-else` trees, but perform well in practice. These do not benefit from any soundness guarantees, but are not critical to the soundness of the testing procedure because we check in HOL that the generated assignments satisfy the hypotheses from the model when we use them.

To find the above set of rewrites that are required to target the SMT-friendly subset of HOL we performed a survey of the step theorems produced by the model. For a randomly chosen set of instructions from each format we attempted to translate the step theorem into Yices' input language. As each unsupported definition was discovered, we added a new rewrite, until no more were found.

One example of this is the carry bit from the `lsls` instruction in the example, where it decides whether the branch is taken. The step theorem from the model calculates it from the registers,

```
if w2n ((7 >< 0) (s.REG RName_2)) = 0 then s.PSR.C else testbit 32
  (shiftl (w2v (s.REG RName_0)) (w2n ((7 >< 0) (s.REG RName_2))))
```

saying that the old value is used if no shift is done, otherwise the correct bit is extracted from `r0`. However, the shift and test are done using *bitstrings* rather than bitvectors, a different but related HOL4 type which is not supported by the SMT translation. This was discovered during the survey of instruction formats described above. Thus we proved a small theorem,

```
∀x : word32. testbit 32 (shiftl (w2v x) n) =
               if (n > 0) ∧ (n <= 32) then x ' (32 - n) else F
```

that expresses the shift and test as a bit selection, and added it to the set of rewrites applied to the step theorems. The resulting theorems still need some of our extensions to the Yices translation. Note that these transformations can be sensitive to changes in the model; in particular, if more step theorems exposed bitstring operations like this, then we may need to perform more rewriting.

Once all of the constraints have been translated, Yices is invoked and if successful returns a satisfying partial assignment. We can form a partial pre-state from this; for our example this is (eliding some irrelevant details):

```
<| MEM :=
     (0x20000000w =+ 136w) ((0x20000001w =+ 94w) ((0x20000002w =+ 144w)
    ((0x20000003w =+ 64w)  ((0x20000004w =+ 4w)  ((0x20000005w =+ 210w)
    ((0x20000006w =+ 16w)  ((0x20000007w =+ 68w) ((0x20000008w =+ 3w)
    ((0x20000009w =+ 104w) ((0x2000000Aw =+ 0w)  ((0x2000000Bw =+ 190w)
    ((0x200002F0w =+ 7w)   ((0x200002F1w =+ 0w)  rand_mem))))))))))));
    PSR := rand_flags with C := F;
    REG := (RName_PC =+ 0x20000000w) ((RName_1 =+ 0xFFFFE9F8w)
          ((RName_2  =+ 0x200018F8w) rand_regs));
    count := 0 |>
```

Instantiating the *rand_* free variables with random data provides a full pre-state.

## 4 Test execution and comparison

We can now instantiate the combined theorem that describes the behaviour of the instructions with the pre-state we have discovered, and hence obtain a concrete post-state to compare against the device. This step is complicated slightly by the abstraction of the intermediate memory values (Section 3.2) and the size of the term describing the whole memory. For the former we progressively instantiate each variable using the definition we recorded in the hypothesis, then use evaluation to make the next memory value concrete. For the latter we split up the state record before supplying the concrete values; this prevents duplication of the large memory term in several places where it is projected away.

At the end of this process we should obtain a theorem with no hypotheses giving the model's concrete prediction of the behaviour of the device. By checking that the hypotheses have been discharged we do not have to trust the SMT translation and solver.

To perform the test we used the OpenOCD debugging tool [10] to write the pre-state to the microcontroller on the development board and start execution. To encode the processor flags into the binary Application Processor Status Register (APSR) we use the HOL function provided by the model. If the test is successful, the device will halt at the breakpoint instruction in the test harness. The same tool is used to read the post-state back from the board.

Precise cycle timings are obtained by directly setting up and reading back the device's SysTick counter timer from the debugger. This timer only ticks while the processor is running, so it stops once the breakpoint is reached. We then have to correct for the time spent in the harness. However, the reference manual does not provide a cycle count for breakpoints, but experimentally we measured a consistent overhead of 3 cycles. (This matches the reasonable hypothesis that there will be 2 cycles overhead to fill the pipeline at the start, and one more at the end to execute the breakpoint.)

The post-state retrieved from the board is then compared to the model's prediction, checking the register contents (adjusting the PC for the extra instructions in the test harness), the SRAM, the APSR and the cycle count.

Some failed tests might not reach the breakpoint instruction; for example, a branch might not be taken when it is expected to, or a memory access may fault. In these cases the comparison will fail because the program counter will be incorrect, and usually other parts of the state too.

## 4.1 Scaling up testing

To perform large test runs, and to debug the testing code, we designed logging with several features in mind:

- Ease of rerunning tests, if necessary with the same random background data for memory, flags and registers.
- Classification into successes and failures (impossible combination of instructions, SMT solver returned 'unknown', exception due to bug in model or test code, or genuine mismatch).
- Data about the differences between the post-state and the model.
- Text that can be read manually.

This is implemented in a straightforward manner, where there is one file per classification with one line per test. Each line contains a numerical identifier, the instruction sequence, branch choices, the test harness used, and details about the failure, if any. The random data used in each test case is stored in a file named using the identifier, so that it can be reproduced precisely when necessary. Instruction placement and profiling of the testing code were added to the logs later to provide more detail.

## 5 Outcomes from testing

First, the consistency tests for Section 2's instruction generation revealed a missing instruction pattern for `ldrsb` in `stepLib`, with the result that `ldrsb` instructions were present in the model but code using them could not be verified. A few minor bugs when generating step theorems were also found.

Recall that these consistency tests do not detect *extra* instructions in the model. Our manual check for these was prompted by Fox discovering one such instruction. Fortunately, as we can bypass the instruction generation we could immediately test for this instruction and confirm its absence from the device.

Similarly, there are alternative forms of the `bx` and `blx` instructions which the reference manual marks as UNPREDICTABLE, i.e., they might work but should not be used. L3 already features the syntax for indicating this, but doesn't act upon it, so the variants are present in the model. Again, if we bypass the instruction generator we can test these variants, finding that a few of these instructions behave normally on the hardware we have.

Moving on to the results from the randomised testing, we tried sequences of 5 to 10 instructions long. Test failures with `bx` and `blx` instructions uncovered a bug where the check on the Thumb mode bit was reversed. The lack of support for self-modifying code can be seen if we turn off the additional constraints to prevent it, as can the invariant about the alignment of the stack pointer.

Finally, we encountered anomalies with the timing model where some test sequences would take one cycle longer than predicted. Several test cases which produced these anomalies were examined, revealing that in each case the SMT solver had chosen to place instructions in the last word of SRAM. This was reinforced by testing manually chosen sequences with extra constraints which forced instructions to be placed in or around the last word. More intensive evidence was provided by adding constraints to require or forbid an instruction placed in the final word for a full test run.

This explanation fits with the processor's pipeline design: executing an instruction from the last word implies attempting to fetch the next word, but this does not exist. Presumably it is handling this corner case that consumes an extra processor cycle.

### 5.1 Performance

To give a general indication of the performance of the system, we performed a test run of 1000 sequences of 5 instructions for this paper. Of the 1000, 105 had no possible pre-state, 882 matched the model's prediction, and 13 did not match; each instruction format was used in at least 34 viable tests. All of the non-matching cases differed only in execution time, and had an instruction in the last word. None of the successful tests did.

To compare the effects of increasing the sequence length, $n$, we present the per-test rate of impossible cases and mean time in seconds for each stage:

| $n$ | Impossible | Generation | Combination | Pre-state | SMT | Instantiation | Testing |
|---|---|---|---|---|---|---|---|
| 5 | 0.105 | 0.037 | 0.769 | 2.418 | 0.313 | 4.795 | 3.272 |
| 7 | 0.145 | 0.053 | 1.585 | 5.909 | 1.101 | 8.662 | 3.437 |
| 9 | 0.320 | 0.038 | 2.917 | 11.239 | 5.114 | 12.956 | 3.516 |

The rate of impossible combinations increases as the probability of incompatible instructions and branch choices increases. The main time costs are the stages involving the full 8kB of memory; it may be possible to reduce this by restricting the memory to the test's footprint.

These were measured on a dual-core 8GB Intel Core i5-3320M using a development version of HOL4 from March 2014 running on PolyML 5.5.1. The generated HOL and SML for the M0 model and tools is present in the HOL4 distribution[4]. An STM32F0-Discovery board was used as the target.

## 6 Discussion

For a simple microprocessor like the Cortex-M0 we can ask whether testing sequences of instructions rather than individual instructions is worth the extra effort. Indeed, the mistakes in the model and tools could have been detected by single instructions, and even the timing anomaly could be detected despite appearing to be the result of the processor's pipeline.

However, it is only through testing sequences that we know this. Moreover, by adjusting the additional constraints that we add we can also witness the lack of support for self-modifying code, and if we wanted to extend the model to support that, or to support timing when executing from slower memory (such as the flash memory on the STM32F0), only sequences of instructions would explore the relevant behaviour.

*The form of the generated code.* We generate the sequences of instructions to be executed by picking each instruction individually. We do not expect this code to reflect real application code, but one feature that is worth considering in detail is the likelihood of generating loops. First, note that the execution cannot stop inside a loop because a breakpoint is used to halt execution. More importantly, we must generate the same instructions several times with an appropriate branch. For example, we manually tested the following sequence to show that the extra timing penalty for placing instructions in the last word of memory could build up over time, only taking the `bcs` branch to the breakpoint at the end:

[4] In the directory `examples/l3-machine-code/m0`.

```
run_test_code debug
  `adds    r0, r0, r2     bcs     -#12     b       -#4
   adds    r0, r0, r2     bcs     -#12     b       -#4
   adds    r0, r0, r2     bcs     -#12`
  (SOME [0, 1, 0, 0, 1, 0, 0, 0])
  (Basic Breakpoint) [``instr_start 2 = 0x20001ffew : word32``];
```

The SMT solver then picks a state in which the loop runs the correct number of times. However, the chances of generating a sequence like this randomly are vanishingly small. One possible area for future work is to produce structural features like loops in conjunction with the instruction generation.

*Generating instructions directly from the model.* By writing our own instruction generator we duplicated some of the information contained in the model: the set and encodings of supported instructions. To apply the testing more widely it would be helpful to use the model directly. (Unsupported instructions are less of an issue because missing instructions do not affect the soundness of verification.)

L3 models feature a datatype for instructions, a decoding function and (optionally) an encoding function. An easy approach is to generate members of the datatype, then use the encoding function to produce binary code to test. However, we have few guarantees about these functions: the most we can expect is that decoding reverses encoding. The opposite may not be true.

For example, if we took this approach for the M0 model we would still not test the extra UNPREDICTABLE forms of the branch instructions, because they have the same representation in the datatype as the normal version, so the encoder will not produce them, but the decoder handles them.

We suggest two possibilities to consider: analysing the decoder function to guide generation of binary instructions; or generating the decoder and encoder functions from a single, more abstract, definition. An example of the latter approach would be a more principled version of our instruction format language.

There is another alternative which is suitable for some targets, including the M0: the Thumb instruction set is sufficiently dense that you can simply pick values at random, then check whether it is a valid instruction. The downside to this approach is that you cannot bias which instructions are chosen or the values used (such as the branch distance for `bl` instructions mentioned in Section 2).

*The form of generated pre-states.* The parts of the state most relevant to the execution are provided by the SMT-solver. These are certainly not randomly sampled, but can be pleasingly daemonic: for example, we have seen useful biases towards reusing locations and using the top and bottom of SRAM, which reveal self-modifying code (if allowed) and the timing anomaly we found. If we wished to generate a wider variety of pre-states we could investigate adding further constraints to force the solver to behave differently.

*Potential extensions.* More complete microprocessor models include system features such as interrupts and exceptions. We expect our model-driven approach would adapt well to these because we can generate tests where the event occurs on a particular instruction. For example, to produce a fault on a load instruction the SMT solver can be asked to ensure that the given address is invalid, and to interrupt at a given instruction the timing knowledge can be used to initialise a timer. Examining the behaviour of sequences of instructions would be vital in this context; even a relatively simple processor like the Cortex-M0 has complex interrupt handling features such as nesting and tail chaining.

## 7 Related work

SMT solvers have already been used in dynamic test case generation, Bounimova et al. [3] have described deploying their SAGE fuzzer at scale to test Microsoft products. This derives new test cases by tracing the execution of existing cases, finding a set of constraints on the input that should alter the execution in an interesting way, then using Z3 to solve those constraints. These are then run against a runtime checker. Our testing is model-driven rather than trace-driven; and the constraint solving is for producing meaningful test cases rather than being the source of fresh test cases. They also perform much more engineering to scale up to mass testing on a large variety of products.

Brucker et al. [4] performed model-based test generation for Verisoft's VAMP architecture, using the HOL-TestGen tool for the Isabelle/HOL proof assistant. The tool takes theorem-like specifications for tests and produces 'abstract' test cases, plus test input data which fills in the details (selected randomly, or using Z3). In principle the whole pre-state could be generated by regarding it as an input, but they used an empty initial configuration because their representation of the model allows ill-formed states to be generated. Their work focused on test case generation and did not test against hardware.

Fox and Myreen [7] validated a previous ARMv7 model using single instruction randomised testing, successfully covering a large portion of the instruction set and revealing several bugs. As they did not have to solve complex constraints to construct the state they had more freedom to pick extreme values while searching for bugs.

A higher standard for models is a formal proof that they match a hardware design. There has been work in this area for decades; one particularly relevant example is Fox's verification of the ARM6 microarchitecture with respect to an ISA definition [5]. Proofs of processor designs can be difficult, and this is unusual for its high coverage of a commercial processor's instruction set. Similarly, Beyer et al. [2] have verified the VAMP gate-level design in the PVS system.

On a related note, Moore [11] promotes the usefulness of symbolic execution of executable processor models, and mentions the importance of testing even for low-level RTL models by recalling that testing an AMD processor model was crucial for persuading managers that the model was valid and worth investigating. This work involved executing existing test cases, whereas we have synthesised them by solving constraints revealed by symbolic execution.

## 8 Conclusion

We have gained confidence in our cycle-accurate Cortex-M0 model using this model-driven randomised testing. An off-the-shelf SMT solver handled the constraints involved in forming a valid pre-state well, and we used additional constraints to respect the model's limitations and validate our hypothesis about the only timing anomaly found. A key question for future work is whether we can make the testing easy to reuse with L3-generated models for other architectures.

**Acknowledgements.** Support for this work was provided by the EPSRC Programme Grant EP/K008528/1, Rigorous Engineering for Mainstream Systems (REMS). Our thanks go to Anthony Fox, Magnus Myreen, Peter Sewell and the other members of the REMS project for their assistance.

## References

1. ARM: ARMv6-M Architecture Reference Manual (2010), `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0419c/index.html`, document DDI 0419C
2. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.J.: Putting it all together — formal verification of the VAMP. International Journal on Software Tools for Technology Transfer 8(4-5), 411–430 (2006)
3. Bounimova, E., Godefroid, P., Molnar, D.: Billions and billions of constraints: Whitebox fuzz testing in production. In: Proceedings of the 2013 International Conference on Software Engineering. pp. 122–131. ICSE '13, IEEE (2013)
4. Brucker, A., Feliachi, A., Nemouchi, Y., Wolff, B.: Test program generation for a microprocessor. In: Veanes, M., Vigan, L. (eds.) Tests and Proofs, LNCS, vol. 7942, pp. 76–95. Springer (2013)
5. Fox, A.: Formal specification and verification of ARM6. In: Basin, D., Wolff, B. (eds.) Theorem Proving in Higher Order Logics, LNCS, vol. 2758, pp. 25–40. Springer (2003)
6. Fox, A.: Directions in ISA specification. In: Beringer, L., Felty, A. (eds.) Interactive Theorem Proving, LNCS, vol. 7406, pp. 338–344. Springer (2012)
7. Fox, A., Myreen, M.O.: A trustworthy monadic formalization of the ARMv7 instruction set architecture. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving, LNCS, vol. 6172, pp. 243–258. Springer (2010)
8. HOL4, `http://hol.sourceforge.net/`
9. Myreen, M.O., Gordon, M.J.C., Slind, K.: Decompilation into logic - improved. In: Cabodi, G., Singh, S. (eds.) FMCAD. pp. 78–81. IEEE (2012)
10. Open on-chip debugger (2014), `http://openocd.sourceforge.net/`
11. Strother Moore, J.: Symbolic simulation: An ACL2 approach. In: Gopalakrishnan, G., Windley, P. (eds.) Formal Methods in Computer-Aided Design, LNCS, vol. 1522, pp. 530–530. Springer (1998)
12. Weber, T.: SMT solvers: New oracles for the HOL theorem prover. International Journal on Software Tools for Technology Transfer (STTT) 13(5), 419–429 (2011)
13. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem—overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. 7, 36:1–36:53 (May 2008)