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

Brian Campbell Ian Stark

LFCS,
School of Informatics,
University of Edinburgh, UK

Rigorous Engineering for Mainstream Systems (REMS) project
(Edinburgh, Cambridge, Imperial)

12th September 2014

Test

random instruction sequences

to build confidence that the

functionality and timing

specified in a formal processor model
matches a real chip

Why are we testing a processor model?

Want to extend Myreen's decompilation approach to verification

- e.g., used for seL4 binary correctness proof
- uses Fox's processor models written in L3 DSL

Add execution time to model, use in verification.

- Focus is on the decompilation/verification
- Don't want full worst-case timing analysis
- Use a simple processor

Even simple microcontrollers are pipelined
$\Rightarrow$ testing single instructions not enough

Overview

Test instruction sequences to build confidence in functionality and timing

Test Cortex-M0 model against an STM32F0 chip

Need to find pre-states where instructions will not fault:

- some constraints from the model
- extra hardware specific constraints

We show you can transform constraints into solver-friendly form

- SMT solver provides critical parts of pre-state
- fill in the rest randomly

Successfully show model predicts chip behaviour, except for a couple of (fixed) bugs and a corner case.

The microcontroller core

ARM Cortex-M0 looks simple enough:

> Table 3-1 shows the Cortex-M0 instructions and their cycle counts. The cycle counts are based on a system with zero wait-states.

Table 3-1 Cortex-M0 instruction summary

Operation	Description	Assembler	Cycles
Move	8-bit immediate	`MOVS Rd, #<imm>`	1
	Lo to Lo	`MOVS Rd, Rm`	1
	Any to Any	`MOV Rd, Rm`	1
	Any to PC	`MOV PC, Rm`	3

Has 3-stage pipeline: fetch, decode, execute

Compact, cut-down Thumb2 instruction set

L3 model of the Cortex M0

Fox's L3 is a DSL for specifying ISAs

- Allows definitions close to manual's pseudo-code
- Tool produces definitions for HOL4 theorem prover

Cortex-M0 model is cut-down version of ARMv7 model

+ ARMv6M specific parts
+ Cortex-M0 instruction timings
− No exceptions or system instructions

HOL4 definitions are functional

Accompanying tools provide more convenient versions

Tools accompanying L3 model

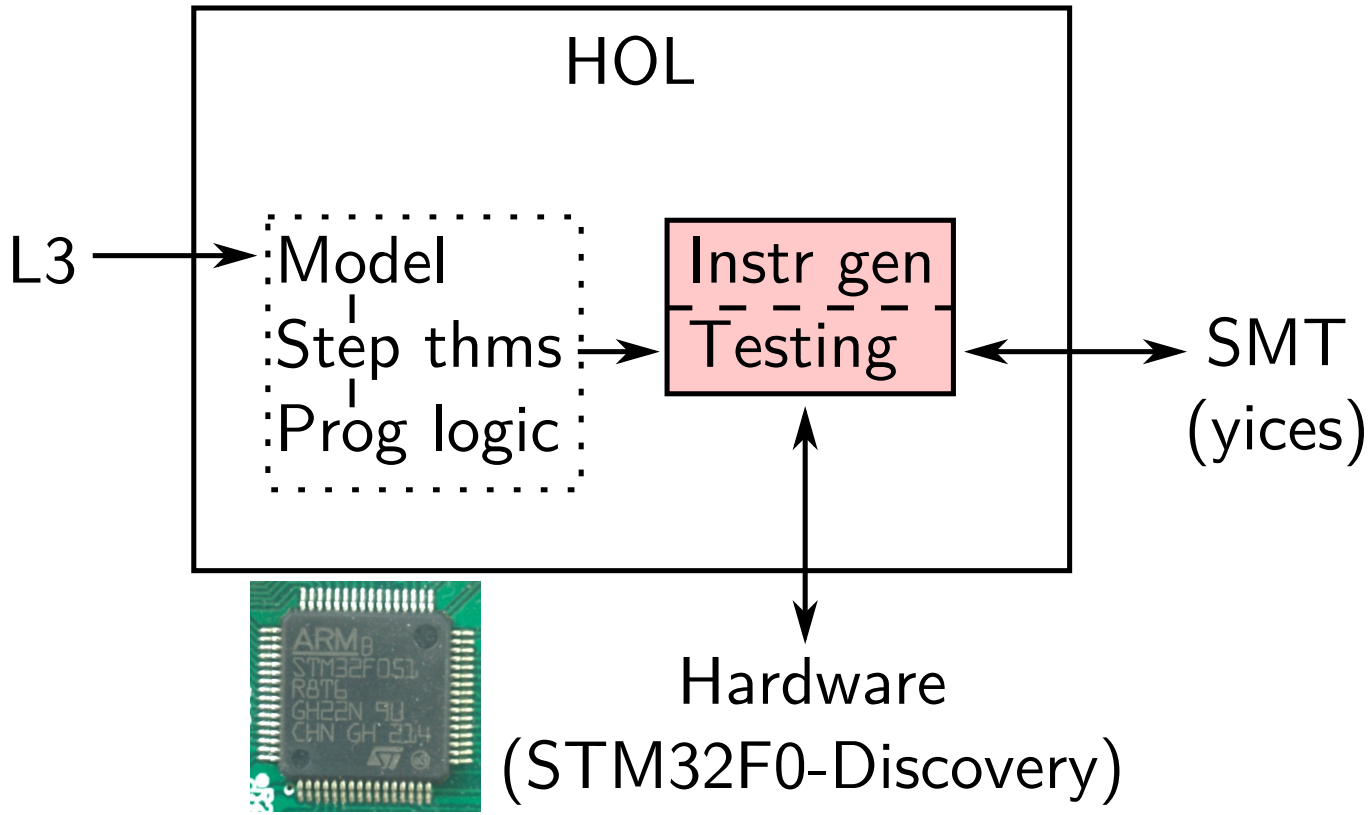
1. Assembly and disassembly
2. `stepLib` describes behaviour for individual instructions
3. `progLib` provides separation logic triples for verification

Step theorem for `ldr r3,[r0,#0]`:

```
[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|>)
```

- `Aligned` address prevents fault

Testing system



- Need the SMT solver to find suitable pre-state

Instruction sequence generation

Cross-check against model; could generate directly instead

Data structure for instruction formats

```
datatype instr_format =
  Lit of int list
| Reg3
| Reg4NotPC
| ...

val instrs = [
  (1, ([Lit [0,0,0,1,1,1,0], Imm 3, Reg3, Reg3],     "ADD (imm) T1")),
  (14,([Lit [1,1,0,1], Cond, Imm 8],                  "B T1")),
  (1, ([Lit [0,1,0,0,0,1,1,1,1], Reg4NotPC, Lit [0,0,0]], "BLX")),
  ...
```

Sanity checks:

- Well formed
- Supported subset matches model

‘Example’ test

```
ldrsh   r0, [r1, r2]   ;   load 16 bits at r1+r2 into r0
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
```

Choose up-front whether to branch

- Reduces complexity of finding pre-state
- But some choices will be impossible

Combine step theorems by repeated instantiation and simplification

Accumulate hypotheses needed for successful execution

- Constrain possible instruction locations

‘Example’ test

```
ldrsh   r0, [r1, r2]   ;  load 16 bits at r1+r2 into r0
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
```

- Model requires, e.g.,

```
Aligned
  (s.REG RName_0 + 0w,4)
```

‘Example’ test

```
ldrsh   r0, [r1, r2]   ;  load 16 bits at r1+r2 into r0
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
```

- Model requires, e.g.,

```
Aligned
  (s.REG RName_0 + s.REG RName_2,4)
```

‘Example’ test

```
ldrsh   r0, [r1, r2]   ;   load 16 bits at r1+r2 into r0
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
```

- Model requires, e.g.,

```
Aligned
  (s.REG RName_2 +
   s.REG RName_0 <<~ w2w ((7 >< 0) (s.REG RName_2)),4),
```

‘Example’ test

```
ldrsh   r0, [r1, r2]   ;  load 16 bits at r1+r2 into r0
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
```

- Model requires, e.g.,

```
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)
```

  featuring bitvector addition, concatenation, sign extension, shifting and alignment.

'Example' test

```
ldrsh   r0, [r1, r2]   ;  load 16 bits at r1+r2 into r0
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
```

- In addition to model's preconditions, hardware requires that

```
(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)))
```

  is a valid location in RAM

- Constraints to add test harness (BKPT)
- ...

Manipulating theorems from the model

While combining into single theorem about whole run:

- Record extra information, e.g., to prevent shadowing

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

  Naïve combination of step theorems forgets about `r1`
  — but need `r1` to be a valid location

- Extra information useful for user-supplied constraints
  e.g., restricting instruction locations

Careful management of large terms:

- separate intermediate memory states
  — otherwise aliasing checks cause quadratic blow-up
- keep appropriate abstractions
  — underlying definitions can be larger, plus . . .

SMT-friendly HOL

Existing HolSmtLib translation only handles small subset of HOL.

- Difficult definition:

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

- Good theorem:

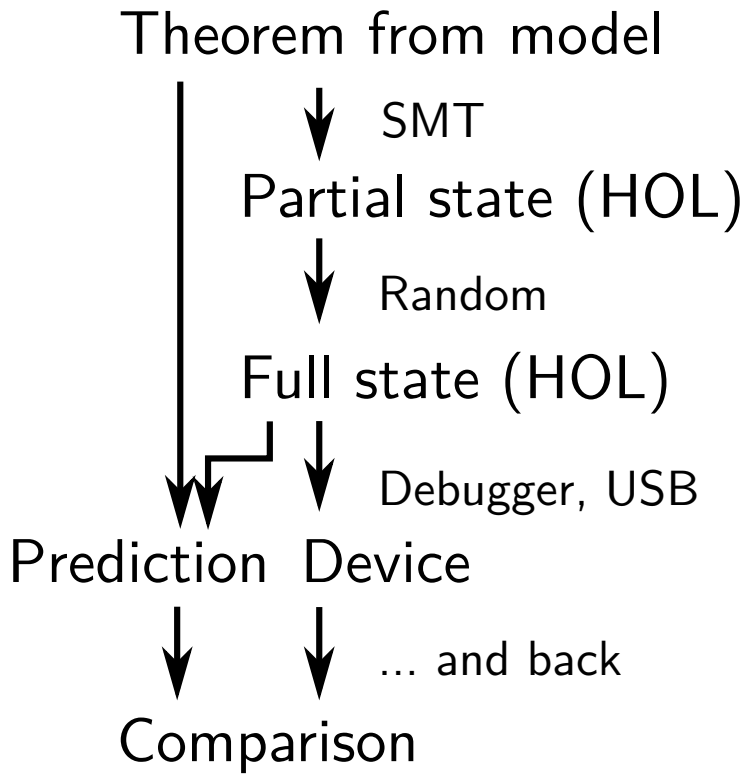
```
(∀a : word32. Aligned (a, 1)) ∧
(∀a : word32. Aligned (a, 2) = ~word_lsb a) ∧
(∀a : word32. Aligned (a, 4) = ((1 >< 0) a = 0w:word2)) ∧
(∀a : word8.  Aligned (a, 4) = ((1 >< 0) a = 0w:word2))
```

Similarly: shifts, flags for addition, ...

Systematic sweep of theorems for each instruction type discovered unsupported terms.

Extended HolSmtLib to construct HOL terms for satisfying assignment from Yices solver.

Running the tests



Rough idea of performance:

1000	candidate tests (5 instrs)
105	no possible pre-state
882	matched model
13	non-matching
4hrs 3mins	
$>$ 3.5	useful tests a minute

Testing outcomes

Functionality:

- `LDRSB` present in model, but step theorem missing
- negated constraint for `BX`

Aligned stack pointers are implicit invariant of TRM's pseudocode

Self-modifying code is visible (remove using constraints)

Timing anomaly:

- extra cycle for placing instruction in last word of memory

Semi-automatic mode useful for:

- testing specific instructions
- confirming timing anomaly accumulates in a loop

Later: more implementations

3 more Cortex-M0 chips, making 4 total:

1. STM32F051
2. Infineon XMC1100
3. NXP LPC11U14
4. Cypress PSoC 4 4125

All have the end-of-memory penalty
— masked by odd memory mapping on 3

3 and 4 have one cycle penalty on store instructions located on odd half-words

Checked by script which corrects model's predictions

Conclusions

Hence,

- SMT good way to find test states (with interesting biases)
- Extra constraints bridge model/reality gap
- Extra constraints useful for hypothesis checking
- Formal setting allows sound transformations into SMT-friendly HOL
- High confidence in M0 cost model for STM32F0

Further work:

- Generalising to other L3 models
- Testing MIPS model against experimental design

Code: `https://bitbucket.org/bacam/m0-validation`

Extras: loops

```
run_test_code debug
  `adds    r0, r0, r2
   bcs     -#2
   adds    r0, r0, r2
   bcs     -#2
   adds    r0, r0, r2
   bcs     -#2
   adds    r0, r0, r2
   bcs     -#2
   adds    r0, r0, r2
   bcs     -#2`
  (SOME [0, 0 (* take first time *),
         0, 0,
         0, 0,
         0, 0, 0, 1 (* pass last time *)])
  (Basic Breakpoint) [];
```

- Must loop because of backward branch
- SMT solver chooses state to run correct number of times
- Vanishingly unlikely to generate by chance

Extras: Scaling it up

Add logging:

- what did we run
- what happened
- enough to reproduce each case exactly

Categorise by outcome:

- Impossible sequence (e.g., branching opposite ways on a flag)
- No suitable pre-state exists (e.g., SMT returned UNSAT)
- Unable to find pre-state (SMT returned UNKNOWN)
- The testing code threw an exception
- 'Proper' failure — post-state did not match prediction
- Success

Extras: Why are some sequences impossible?

- Early decisions about branching may be incompatible:

```
bcs     +#12  ; randomly choose to take branch
bcs     +#12  ; randomly choose not to
```

- Creating a valid address may be impossible:

```
ldrb    r1, [r0,#0]  ; so r1 is a byte
str     r2, [r1,#0]  ; store r2 at r1
```

  but $0 - 256$ is flash memory

- Variations of this involving (e.g.) branch displacements:

```
bl      +#2048
...
bl      +#2048
```

- Restrictions on operands:

```
lsls    r0, r0, #1   ; left shift
bx      r0           ; requires bit 0 set
```