Randomised testing of a HOL 4 microprocessor model

Brian Campbell

REMS project

LFCS, University of Edinburgh

31st January 2014

Why are we testing a processor model?

Want a good model for:

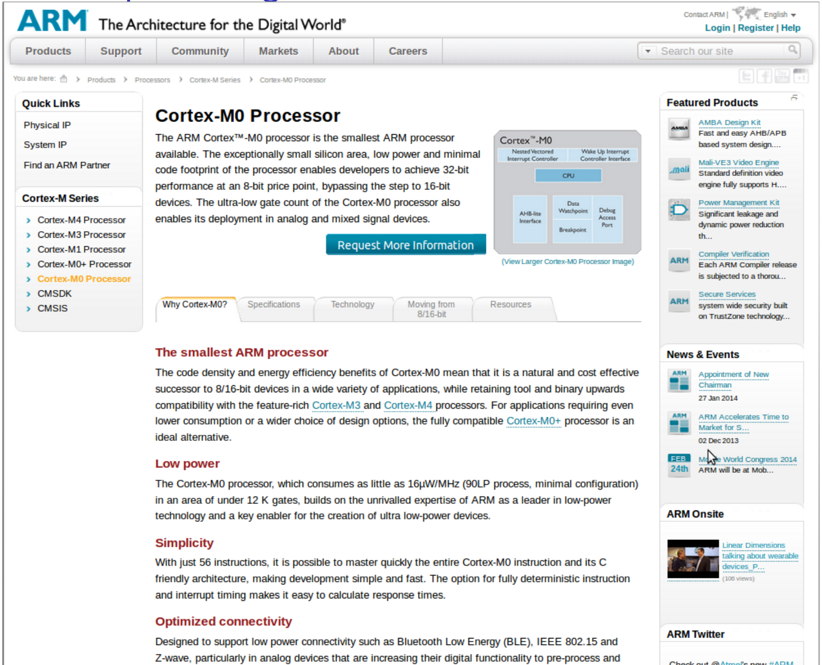
Cost-lifting decompilation

$$\begin{tikzcd}
\text{C} \arrow[r] \arrow[d, "\text{gcc}"'] & \text{HOL} \arrow[d, phantom, "\wr"] \\
\text{binary ARMv6-M} \arrow[r] & \text{HOL}
\end{tikzcd}$$

1. Compile C code normally
2. Simple timing analysis on ARM code for basic blocks
3. Use decompilation to
   - check functional equivalence
   - attach timing annotations on to C source

Based on [Sewell, Myreen, Klein, PLDI'13]

Need a simple timing model...


ARM The Architecture for the Digital World®

Contact ARM | English
Login | Register | Help

Products | Support | Community | Markets | About | Careers | Search our site

You are here: › Products › Processors › Cortex-M Series › Cortex-M0 Processor

Quick Links

- Physical IP
- System IP
- Find an ARM Partner

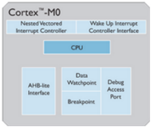
Cortex-M Series

- › Cortex-M4 Processor
- › Cortex-M3 Processor
- › Cortex-M1 Processor
- › Cortex-M0+ Processor
- › Cortex-M0 Processor
- › CMSDK
- › CMSIS

Cortex-M0 Processor

The ARM Cortex™-M0 processor is the smallest ARM processor available. The exceptionally small silicon area, low power and minimal code footprint of the processor enables developers to achieve 32-bit performance at an 8-bit price point, bypassing the step to 16-bit devices. The ultra-low gate count of the Cortex-M0 processor also enables its deployment in analog and mixed signal devices.

Request More Information



Cortex™-M0
Nested Vectored Interrupt Controller
Wake Up Interrupt Controller Interface
CPU
AHB-lite Interface
Data Watchpoint
Breakpoint
Debug Access Port

(View Larger Cortex-M0 Processor Image)

Why Cortex-M0? | Specifications | Technology | Moving from 8/16-bit | Resources

The smallest ARM processor

The code density and energy efficiency benefits of Cortex-M0 mean that it is a natural and cost effective successor to 8/16-bit devices in a wide variety of applications, while retaining tool and binary upwards compatibility with the feature-rich Cortex-M3 and Cortex-M4 processors. For applications requiring even lower consumption or a wider choice of design options, the fully compatible Cortex-M0+ processor is an ideal alternative.

Low power

The Cortex-M0 processor, which consumes as little as 16µW/MHz (90LP process, minimal configuration) in an area of under 12 K gates, builds on the unrivalled expertise of ARM as a leader in low-power technology and a key enabler for the creation of ultra low-power devices.

Simplicity

With just 56 instructions, it is possible to master quickly the entire Cortex-M0 instruction and its C friendly architecture, making development simple and fast. The option for fully deterministic instruction and interrupt timing makes it easy to calculate response times.

Optimized connectivity

Designed to support low power connectivity such as Bluetooth Low Energy (BLE), IEEE 802.15 and Z-wave, particularly in analog devices that are increasing their digital functionality to pre-process and

Featured Products

- AMBA Design Kit — Fast and easy AHB/APB based system design....
- Mali-VE3 Video Engine — Standard definition video engine fully supports H....
- Power Management Kit — Significant leakage and dynamic power reduction th...
- Compiler Verification — Each ARM Compiler release is subjected to a thorou...
- Secure Services — system wide security built on TrustZone technology...

News & Events

- Appointment of New Chairman — 27 Jan 2014
- ARM Accelerates Time to Market for S... — 02 Dec 2013
- FEB 24th — Mobile World Congress 2014 — ARM will be at Mob...

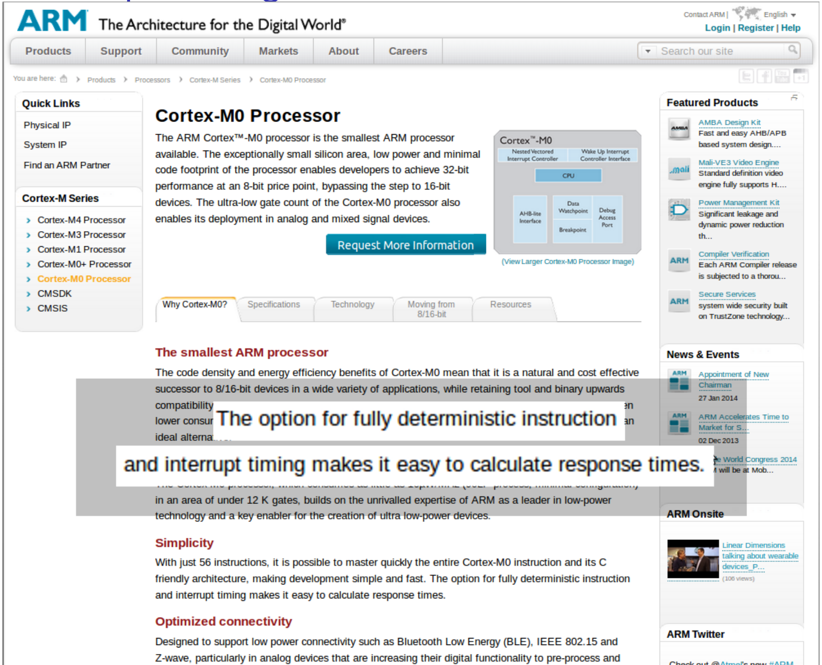
ARM Onsite

Linear Dimensions talking about wearable devices P... (106 views)

ARM Twitter

Check out @Atmel's new #ARM

Need a simple timing model...



ARM The Architecture for the Digital World®

Contact ARM | English
Login | Register | Help

Products | Support | Community | Markets | About | Careers | Search our site

You are here: › Products › Processors › Cortex-M Series › Cortex-M0 Processor

Quick Links

- Physical IP
- System IP
- Find an ARM Partner

Cortex-M Series

- › Cortex-M4 Processor
- › Cortex-M3 Processor
- › Cortex-M1 Processor
- › Cortex-M0+ Processor
- › Cortex-M0 Processor
- › CMSDK
- › CMSIS

Cortex-M0 Processor

The ARM Cortex™-M0 processor is the smallest ARM processor available. The exceptionally small silicon area, low power and minimal code footprint of the processor enables developers to achieve 32-bit performance at an 8-bit price point, bypassing the step to 16-bit devices. The ultra-low gate count of the Cortex-M0 processor also enables its deployment in analog and mixed signal devices.

Request More Information

Cortex™-M0
Nested Vectored Interrupt Controller
Wake Up Interrupt Controller Interface
CPU
AHB-lite Interface
Data Watchpoint
Breakpoint
Debug Access Port

(View Larger Cortex-M0 Processor Image)

Why Cortex-M0? | Specifications | Technology | Moving from 8/16-bit | Resources

The smallest ARM processor

The code density and energy efficiency benefits of Cortex-M0 mean that it is a natural and cost effective successor to 8/16-bit devices in a wide variety of applications, while retaining tool and binary upwards compatibility ... en lower consur ... an ideal alterna...

... in an area of under 12 K gates, builds on the unrivalled expertise of ARM as a leader in low-power technology and a key enabler for the creation of ultra low-power devices.

Simplicity

With just 56 instructions, it is possible to master quickly the entire Cortex-M0 instruction and its C friendly architecture, making development simple and fast. The option for fully deterministic instruction and interrupt timing makes it easy to calculate response times.

Optimized connectivity

Designed to support low power connectivity such as Bluetooth Low Energy (BLE), IEEE 802.15 and Z-wave, particularly in analog devices that are increasing their digital functionality to pre-process and

Featured Products

- AMBA Design Kit — Fast and easy AHB/APB based system design....
- Mali-VE3 Video Engine — Standard definition video engine fully supports H....
- Power Management Kit — Significant leakage and dynamic power reduction th...
- Compiler Verification — Each ARM Compiler release is subjected to a thorou...
- Secure Services — system wide security built on TrustZone technology...

News & Events

- Appointment of New Chairman — 27 Jan 2014
- ARM Accelerates Time to Market for S... — 02 Dec 2013
- ...e World Congress 2014 ...will be at Mob...

ARM Onsite

Linear Dimensions talking about wearable devices_P... (106 views)

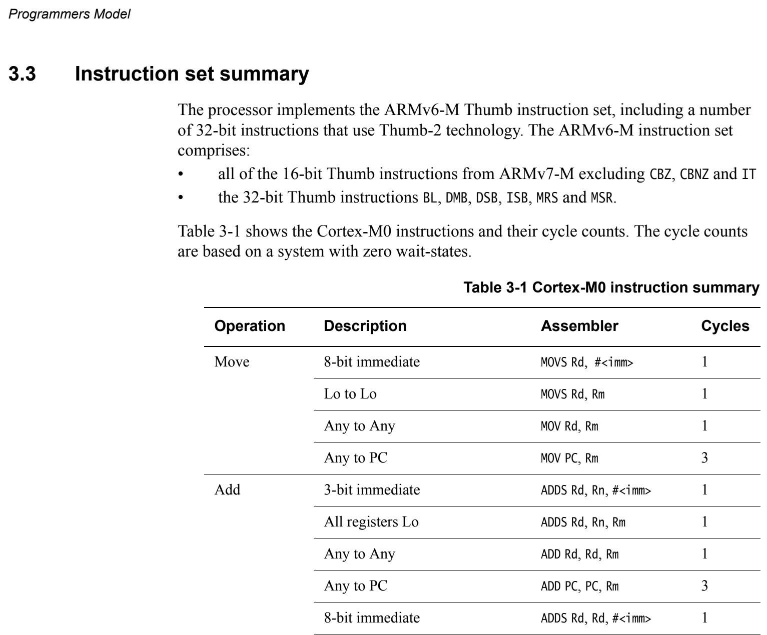
ARM Twitter

Check out @Atmel's new #ARM

The option for fully deterministic instruction

and interrupt timing makes it easy to calculate response times.

Need a simple timing model...



Programmers Model

3.3 Instruction set summary

The processor implements the ARMv6-M Thumb instruction set, including a number of 32-bit instructions that use Thumb-2 technology. The ARMv6-M instruction set comprises:

- all of the 16-bit Thumb instructions from ARMv7-M excluding `CBZ`, `CBNZ` and `IT`
- the 32-bit Thumb instructions `BL`, `DMB`, `DSB`, `ISB`, `MRS` and `MSR`.

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
Add	3-bit immediate	`ADDS Rd, Rn, #<imm>`	1
	All registers Lo	`ADDS Rd, Rn, Rm`	1
	Any to Any	`ADD Rd, Rd, Rm`	1
	Any to PC	`ADD PC, PC, Rm`	3
	8-bit immediate	`ADDS Rd, Rd, #<imm>`	1

Is the simple timing model real?

ARM Cortex-M0 processors

+ nice timing table in reference manual
+ none of the usual caveats in the manual
+ no cache / write-back buffer concerns
+ implementations have fast enough SRAM

Is the simple timing model real?

ARM Cortex-M0 processors

+ nice timing table in reference manual
+ none of the usual caveats in the manual
+ no cache / write-back buffer concerns
+ implementations have fast enough SRAM
− manufacturers very quiet about timing
− three stage pipeline

  (fetch, decode, execute)

Overview

- Have a HOL 4 model of ARM Cortex-M0 processor
- which includes timing

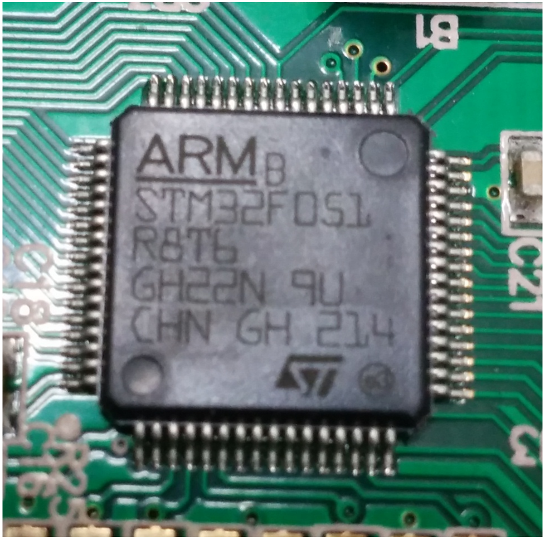
Want to check

1. suitable for verification (sound)
2. timing is correct

So we take random sequences of instructions.

Test predictions the model makes against real chip(s).

A real chip



The HOL4 model [Fox]

$$\begin{tikzcd} & L3 \arrow[d, "\textit{(automated)}"'] & & \\ & \text{Model} \arrow[r] & \text{Step} \arrow[r] & \text{Prog} \end{tikzcd}$$

- Written in L3 DSL [Fox, ITP'12]
- Automatic translation to HOL 4
- Model — step function
- Step — per-instruction behaviour of step fn
- Prog — separation-logic-like triples

Main purpose: program verification
Not modelled: target memory, exceptions, self-modifying code, . . .

- ⋆ Includes simple timing model

The HOL4 model [Fox]

$$\begin{tikzcd}
\text{L3} \arrow[d, "\textit{(automated)}"'] & & \\
\text{Model} \arrow[r] & \text{Step} \arrow[r] & \text{Prog}
\end{tikzcd}$$

```
instruction DecodeARM(w::word) = {
...
     case '1010 : imm24' =>
        if Take (cond, true) then
        {  imm32 = SignExtend (imm24 : '00');
           Branch (BranchTarget (imm32))
        }
        else
           Skip ()
...
define Branch > BranchTarget
   ( imm32 :: bits(32) )
   =
   BranchWritePC (PC + imm32)
```

The HOL4 model [Fox]

$$\begin{tikzcd}
L3 \arrow[d, "\textit{(automated)}"'] & & \\
\text{Model} \arrow[r] & \text{Step} \arrow[r] & \text{Prog}
\end{tikzcd}$$

```
DecodeThumb2 h =
...
 else if ¬b'14 ∧¬ b'12 then
   (if
      FST (ConditionPassed (v2w [b'25; b'24; b'23; b'22]) state)
    then
      Branch (BranchTarget
         (sw2sw
            (v2w [b'26] @@ v2w [b'11] @@ v2w [b'13] @@
             v2w [b'21; b'20; b'19; b'18; b'17; b'16] @@
             v2w [b'10; b'9; b'8; b'7; b'6; b'5; b'4; b'3;
                b'2; b'1; b'0] @@ 0w)))
    else NoOperation (),state)
```

The HOL4 model [Fox]

$$\begin{tikzcd}
\text{L3} \arrow[d, "\textit{(automated)}"'] & & \\
\text{Model} \arrow[r] & \text{Step} \arrow[r] & \text{Prog}
\end{tikzcd}$$

```
   d5f8   bpl -12
```

```
[Aligned (s.REG RName_PC,2), ¬s.AIRCR.ENDIANNESS, ¬s.PSR.N,
 s.MEM (s.REG RName_PC) = 248w, s.MEM (s.REG RName_PC + 1w) = 213w,
 s.exception = NoException]
|- NextStateM0 s =
   SOME
     (s with
      <|REG := (RName_PC =+ s.REG RName_PC + 4w + 0xFFFFFFF0w) s.REG;
        count := s.count + 3; pcinc := 2w|>)
```

The HOL4 model [Fox]

$$\begin{tikzcd} & L3 \arrow[d, "(automated)"'] & & \\ & \text{Model} \arrow[r] & \text{Step} \arrow[r] & \text{Prog} \end{tikzcd}$$

```
d5f8   bpl -12
```

```
[] |- SPEC M0_MODEL
     (m0_count count        * m0_PSR_N n * m0_CONFIG (F,spsel) *
      m0_PC pc          * cond ( ¬n))

     {(pc,INL 54776w)}

     (m0_count (count + 3) * m0_PSR_N n * m0_CONFIG (F,spsel) *
      m0_PC (pc - 12w)                )
```

The HOL4 model [Fox]

$$\begin{tikzcd} & \text{L3} \arrow[d, "\textit{(automated)}"'] & & \\ & \text{Model} \arrow[r] & \text{Step} \arrow[r] & \text{Prog} \end{tikzcd}$$

We choose to work with Step

- Would need to do equivalent work anyway
  - To specialise to specific instructions
  - To isolate preconditions
- Prog requires up-front decisions about separation

Small danger to validity:

- Prog may not use Step in the same way as us

Testing overview

Several distinct stages:

1. Instruction sequence generation
2. Combining step theorems
3. Constructing suitable pre-state
4. Instantiate theorem to get prediction
5. Run sequence on hardware and compare

Instruction sequence generation

Want

- to pick randomly
- but bias selection of instructions, registers, values

Could reuse L3's knowledge of instructions, but

- Small instruction set, so
- opportunity to cross-check

Instruction sequence generation

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")),
```

Instruction sequence generation

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:

- Supported instructions have Prog triples
- Unsupported ones don't

`LDRSB` was missing from Step!

Combining step theorems

Get Step theorem for each instruction

- randomly picking whether to take a conditional branch

```
... |- NextStateM0 s = SOME (s with |< ... >|)
... |- NextStateM0 s = SOME (s with |< ... >|)
    ...
```

Combining step theorems

Get Step theorem for each instruction

- randomly picking whether to take a conditional branch

```
... |- NextStateM0 s = SOME (s with |< ... >|)
... |- NextStateM0 s = SOME (s with |< ... >|)
    ...
```

Progressively instantiate each `s` with previous step theorem.

- Simplify as we go
- Record symbolic memory accesses, instruction locations
  - otherwise we forget about accesses whose value is discarded

```
...
 instr_start 1 = s.REG RName_PC + 2w,
 memory_address 0 = s.REG RName_PC + 8w,
 s.MEM (s.REG RName_PC) = v2w [F; F; T; F; T; F; T; F],
 ...
|- NStatesM0 5 s =
   s with <| ... |>
```

Finding a pre-state — requirements

Model

- only tells us about successful executions
- gives preconditions
- doesn't cover everything

We need more:

- Memory accesses in range
  - must hit 8kB memory in 4GB address space
- no self-modification
- add test harness (BKPT instruction)
- align stack pointer registers

Add these to model's preconditions as HOL terms

Finding a pre-state — constraint solving

Constraints may be complex

```
0:  5e88     ldrsh   r0, [r1, r2]    ; load r0 from r1+r2  (16 bits)
2:  4090     lsl     r0, r0, r2      ; shift r0 left by r2
4:  1880     add     r0, r0, r2      ; add r2 to r0
6:  6803     ldr     r3, [r0, #0]    ; load r2 from r0
```

Constraints involving bitvector adds, shifts, sign extension, repeated variables and inequalities.

Finding a pre-state — constraint solving

Constraints may be complex

```
0:  5e88     ldrsh   r0, [r1, r2]    ; load r0 from r1+r2  (16 bits)
2:  4090     lsl     r0, r0, r2      ; shift r0 left by r2
4:  1880     add     r0, r0, r2      ; add r2 to r0
6:  6803     ldr     r3, [r0, #0]    ; load r2 from r0
```

Constraints involving bitvector adds, shifts, sign extension, repeated variables and inequalities.

- Requirements fit SMT solving well
- Existing HolSmtLib targets Yices, Z3
  - but for proving (via negation)
  - adapted Yices part for constraint solving

Constraint solving with SMT appears to be unusual for interactive theorem proving.

Finding a pre-state — constraint solving

The adapted HolSmtLib will translate subset of HOL into Yices format.

- Need to fit preconditions into HOL subset

1. Sound rewriting of unsupported definitions
   `Alignment`, shifts, `add_with_carry`
2. ensure supported form of bitvectors is used
3. some mixed bitvector/nat operations unsupported
   rewrite away, or implement limited version (e.g., 8-bit)

Finding a pre-state — constraint solving

The adapted HolSmtLib will translate subset of HOL into Yices format.

- Need to fit preconditions into HOL subset

1. Sound rewriting of unsupported definitions
   `Alignment`, shifts, `add_with_carry`
2. ensure supported form of bitvectors is used
3. some mixed bitvector/nat operations unsupported
   rewrite away, or implement limited version (e.g., 8-bit)

Discovering what to do isn't easy:

- Tried preconditions for every instruction type to detect all unsupported
- Have to be careful not to undo rewrites when simplifying

Instantiate theorem to get prediction

Translating the SMT results into HOL terms gives us a partial state

- Fill in the blanks with random choices
- Instantiating the theorem derived earlier should
  - Discharge all hypotheses
  - predict final state

Instantiate theorem to get prediction

Translating the SMT results into HOL terms gives us a partial state

- Fill in the blanks with random choices
- Instantiating the theorem derived earlier should
  - Discharge all hypotheses
  - predict final state

HOL isn't entirely happy with a list of 8192 8-bit bitvectors. Careful handling required.

Run sequence on hardware and compare

$$\begin{tikzcd}
\text{HOL state} \arrow[d, "\textit{extract}"] \\
\text{memory, registers, flags} \arrow[d, "\textit{IPC}"] \\
\text{OpenOCD debugger driver} \arrow[d, "\textit{USB}"] \\
\text{STMF0-Discovery board} \arrow[d, "\textit{USB}"] \\
\text{OpenOCD debugger driver} \arrow[d, "\textit{IPC}"] \\
\text{Final state}
\end{tikzcd}$$

Check memory, registers, flags and onboard SysTick timer.

Run sequence on hardware and compare

$$\begin{tikzcd}
\text{HOL state} \arrow[d, "\textit{extract}"] \\
\text{memory, registers, flags} \arrow[d, "\textit{IPC}"] \\
\text{OpenOCD debugger driver} \arrow[d, "\textit{USB}"] \\
\text{STMF0-Discovery board} \arrow[d, "\textit{USB}"] \\
\text{OpenOCD debugger driver} \arrow[d, "\textit{IPC}"] \\
\text{Final state}
\end{tikzcd}$$

Check memory, registers, flags and onboard SysTick timer.

If processor goes off-sequence, end up in Fault state with huge time.

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

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

Future: gather statistics on coverage.

Results so far

Results so far

Surprised SMT solver isn't returning UNKNOWN in practice.

Results so far

Surprised SMT solver isn't returning UNKNOWN in practice.

Some bugs:

1. Missing `LDRSB` in Step (plus minor issues)
2. Inverted check for `BX` in Model

Both would be found by single-instruction testing.

Results so far

Surprised SMT solver isn't returning UNKNOWN in practice.

Some bugs:

1. Missing `LDRSB` in Step (plus minor issues)
2. Inverted check for `BX` in Model

Both would be found by single-instruction testing.

Some sequences take one cycle too long:

- `add pc, r0` at end
- Some mixtures of branch and memory operation

Results so far

Surprised SMT solver isn't returning UNKNOWN in practice.

Some bugs:

1. Missing `LDRSB` in Step (plus minor issues)
2. Inverted check for `BX` in Model

Both would be found by single-instruction testing.

Some sequences take one cycle too long:

- `add pc, r0` at end
- Some mixtures of branch and memory operation

Appear to be hitting some subtle PolyML GC / HOL incompatibility..!

Future

Near future:

- Investigate timing anomalies
  - Different timing harnesses, add padding, ...
- Longer runs
- Longer instruction sequences
- Investigate coverage

Future

Near future:

- Investigate timing anomalies
  - Different timing harnesses, add padding, . . .
- Longer runs
- Longer instruction sequences
- Investigate coverage

Further ahead:

- New REMS DSL for processor models, SAIL
  - Integrate testing?
  - Reproducible test cases for weak memory models

How much of this can we commoditise?

Future

Near future:

- Investigate timing anomalies
  - Different timing harnesses, add padding, . . .
- Longer runs
- Longer instruction sequences
- Investigate coverage

Further ahead:

- New REMS DSL for processor models, SAIL
  - Integrate testing?
  - Reproducible test cases for weak memory models

How much of this can we commoditise?

Other opportunities:

- Test bigger processors with proper WCET analyses?
- Can we choose pre-states more randomly?

Conclusion

1. Took a HOL processor model
2. Test sequences of instructions for functional and timing bugs
3. Used SMT solving to ensure successful executions
4. Preliminary signs of success

Main technical difficulty:

- Getting preconditions into SMT friendly form.
- ⋆ Formal system makes doing this soundly easier

Sort out timing anomalies
$\Rightarrow$ sound basis for cost-preserving decompilation