# The State of Sail

Alasdair Armstrong

Joint work with: Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, Peter Sewell

13th September, 2019

**This work was partially supported by EPSRC grant EP/K008528/1 (REMS), and an ARM iCASE award. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 789108). Approved for public release; distribution is unlimited. This research is sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8650-18-C-7809 ("CIFV"). The views, opinions, and/or findings contained in this article/presentation are those of the author(s)/presenter(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.**

## Sail ISA description language

- Imperative first-order language for ISA specification
- Lightweight dependent types for bitvectors (checked using Z3)

```
function get_width() -> {|32, 64|} = ...

function example(xs: bits(16)) -> unit = {
  let 'width = get_width();
  let ys: bits('width) = zero_extend(xs);
  if length(ys) == 32 then {
    // __prove is special compile-time assert
    __prove(constraint('width == 32)) // always true
  }
}
```

## Sail semantics

- Very simple imperative semantics (no aliasing, pointers, etc)
    - *Very amenable to static analysis*
- Behavior of memory actions left to external memory model
    - Can plug in separate semantics for relaxed memory concurrency

## Example RISC-V instruction

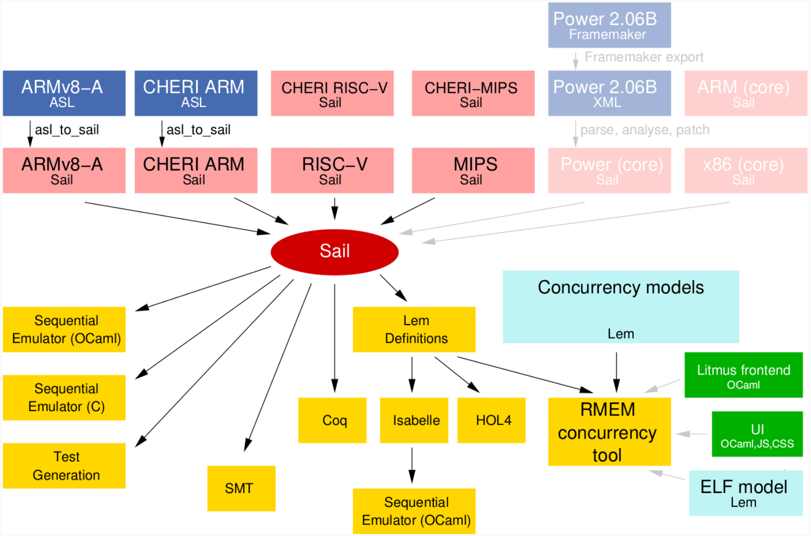
Example instruction in Sail:

```
function clause execute (ITYPE (imm, rs1, rd, op)) = {
  let rs1_val = X(rs1);
  let immext : xlenbits = EXTS(imm);
  let result : xlenbits = match op {
    RISCV_ADDI  => rs1_val + immext,
    RISCV_SLTI  => EXTZ(rs1_val <_s immext),
    RISCV_SLTIU => EXTZ(rs1_val <_u immext),
    RISCV_ANDI  => rs1_val & immext,
    RISCV_ORI   => rs1_val | immext,
    RISCV_XORI  => rs1_val ^ immext
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

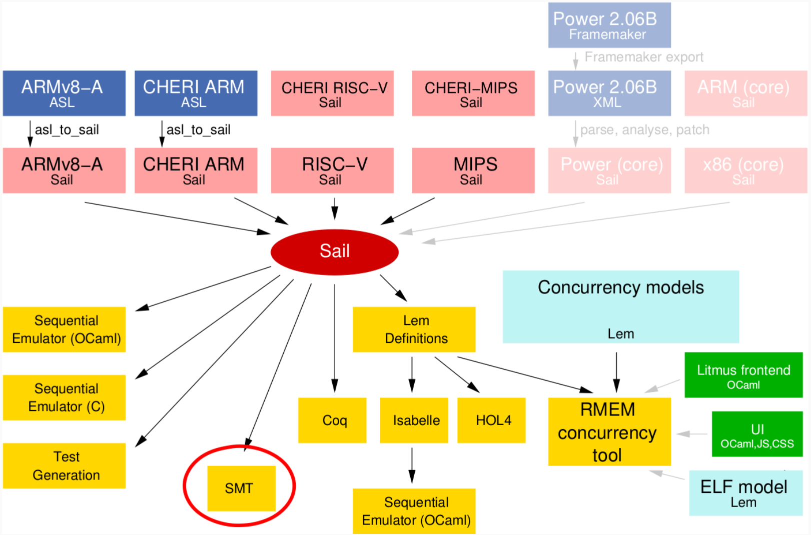
## Bi-directional mappings for assembly and encode/decode

```
mapping encdec_iop : iop <-> bits(3) = {
  RISCV_ADDI  <-> 0b000,
  RISCV_SLTI  <-> 0b010,
  RISCV_SLTIU <-> 0b011,
  RISCV_ANDI  <-> 0b111,
  RISCV_ORI   <-> 0b110,
  RISCV_XORI  <-> 0b100
}

mapping clause encdec = ITYPE(imm, rs1, rd, op)
  <-> imm @ rs1 @ encdec_iop(op) @ rd @ 0b0010011

mapping clause assembly = ITYPE(imm, rs1, rd, op)
  <-> itype_mnemonic(op) ^ spc() ^ reg_name(rd) ^ sep() ^ reg_name(
      rs1) ^ sep() ^ hex_bits_12(imm)
```

## Sail Overview



## Sail Overview



## Sail ISA Models

| Model | Source | KLoS | KIPS | Provers | Boots |
|---|---|---|---|---|---|
| ARMv8.5-A | ASL | 125 | 200 | All | Linux, Hafnium |
| MIPS | hand | 2 | 800 | All | FreeBSD |
| CHERI MIPS | hand | +2 | 400 | All | FreeBSD, CheriBSD |
| RISC-V | hand | 5 | 150 | All | † |
| CHERI RISC-V | hand | +2 | | All | |
| CHERI ARM | ASL | lots | | | |
| ARMv8 (small) | hand | 6 | | | |
| IBM POWER fragment | hand | 6 | | | |
| x86 fragment | hand | 2 | | | |

† FreeBSD, Linux, FreeRTOS, Hafnium

## RISC-V

Open ISA, developed by broad industrial and academic community

- Test system features by booting seL4, FreeBSD, Linux, and others
- Validated against RISC-V conformance tests, and via trace comparison with Spike simulator
- Led to contributions to original ISA specification, e.g.
  - description of page-faults in page-table walks
  - ambiguities in the specification of interrupt delegation
  - bug fixes in Spike simulator
- Involvement in the RISC-V formal working group
- Integration with RMEM operational concurrency tool
  - Used with the 6874 litmus tests for the RISC-V memory model

## ARMv8-A

Described in ARM's ASL executable pseudocode language

- Used within ARM for documentation, hardware validation, and architecture design

Authoritative, vendor-supplied semantics

- Automatic translation from ASL into Sail via **asl-to-sail** tool
- Features: Floating-point, address translation & page-table walks, synchronous exceptions, hypervisor mode, crypto instructions, vector instructions (NEON and SVE), memory partitioning and monitoring, pointer authentication, etc. . .

Such a complete vendor-supplied architecture description not previously publicly available for formal reasoning

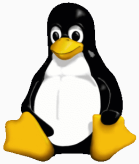
# ARMv8-A Model Validation

ARM's Architecture Validation Suite (AVS)

- Internal test suite within ARM
- Sail v8.3 model passed 99.85% of 15 400 tests as compared with ARM ASL

Linux booting

- Useful sanity test for system features
- Only covers about 25% of the 64-bit model even without vector instructions!



Also experimented with running Linux under Hafnium hypervisor

## ARMv8.5-A

64-bit part of v8.5 specification contains:

- 66558 LOS for all 64-bit instructions
- 3825 Sail functions
- 561 registers
- 981 instructions (each may be multiple assembly mnemonics)

When evaluated each instruction performs around 800 calls to auxiliary functions, and 500 primitives

Key question: Is such a large specification actually useable for verification and proof?

## Example Proof for ARMv8-A

Address translation: Most complex part of ARMv8 model!

- 9000 lines of specification required
- Page table walk: Over 500 LOS excluding helper functions
  - ...and there are *lots* of page table helper functions
- Involves iteration, variable-length bitvectors, memory effects, nondeterminism, ...

## Example Proof for ARMv8-A

We defined a simple characterisation of address translation suitable for reasoning about non-system code

About 500 lines of Isabelle total

**Theorem**

*Simplified address translation is equivalent to full ARMv8 address translation under certain useful assumptions*

*user mode, no virtualisation, valid translation tables, hardware updating of translation table flags*

Uncovered a few small bugs in the ASL specification

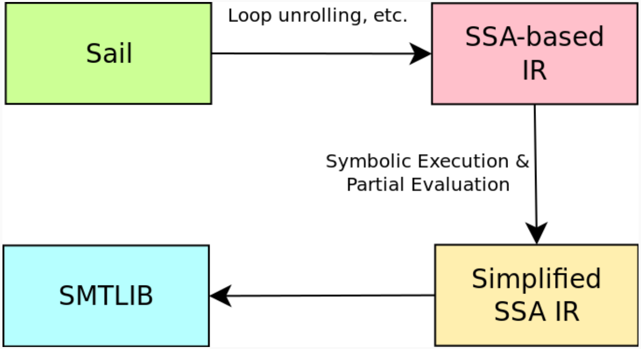
## Bounded model-checking for Sail

Interactive proof is great, but can we have something more automated?

- Translate Sail source into SMT problems
- Allows verifying properties of Sail functions
- For example, used to verify capability-manipulation properties in CHERI-RISCV

```
$property
function prop_set_bounds_exact
    (c : Capability, base : bits(64), top : bits(65)) -> bool = {
  let (exact, c’) = setCapBounds(c, base, top’);
  let (base’, top’) = getCapBounds(c’);
  ~(exact)  | (unsigned(base) >= unsigned(top))
  | (base’ == unsigned(base) & top’ == unsigned(top))
}
```

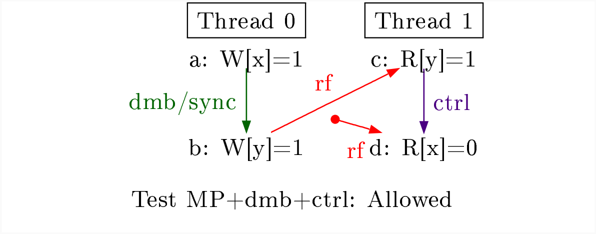
## Bounded model-checking for Sail



Similar to existing tools such as CBMC

Now rather than just model-checking properties of Sail functions, we want to analyse litmus tests for ARMv8, RISC-V (and others)

**Example (observable speculative execution):**



MP+dmb+ctrl ARM

| Thread 0 | Thread 1 |
|---|---|
| MOV R0,#1 | LDR R0,[R3] |
| STR R0,[R2] | CMP R0,R0 |
| DMB | BNE LC00 |
| MOV R1,#1 | LC00: |
| STR R1,[R3] | LDR R1,[R2] |
| Initial state: 0:R2=x, 0:R3=y, 1:R2=x , 1:R3=y | |
| Allowed: 1:R0=1, 1:R1=0 | |

## Axiomatic concurrency semantics for Sail

- Concurrency semantics not in Sail ISA description - that only covers sequential intra-instruction behaviour
- Expressed in two ways:
  - *operational*: abstract-microarchitectural abstract machine describing allowed speculation etc.
  - *axiomatic*: predicate on candidate complete executions (graphs of memory events, rf relation, etc.)
- Both approaches made executable as test oracle (to compute all allowed behaviours of litmus tests) by:
  - Lem operational semantics, integrated with Sail ISA semantics (albeit not full ARMv8.5-A yet) in RMEM tool
  - Herd, Memalloy, ... - but these have hardwired and limited ISA semantics support

## Axiomatic concurrency semantics for Sail

- Problem: how can we compute model-allowed behaviour w.r.t. axiomatic models wrt general (Sail) ISA semantics? And how can we do that in the presence of self-modifying code or other interesting architectural features?
- Idea: Use the above infrastructure for Sail$\rightarrow$SMT to build a tool similar to herd

- Inputs:
  - A litmus file
  - A Sail model for the instruction set used in the litmus file
  - An axiomatic memory model specified in the cat language (used by Herd)

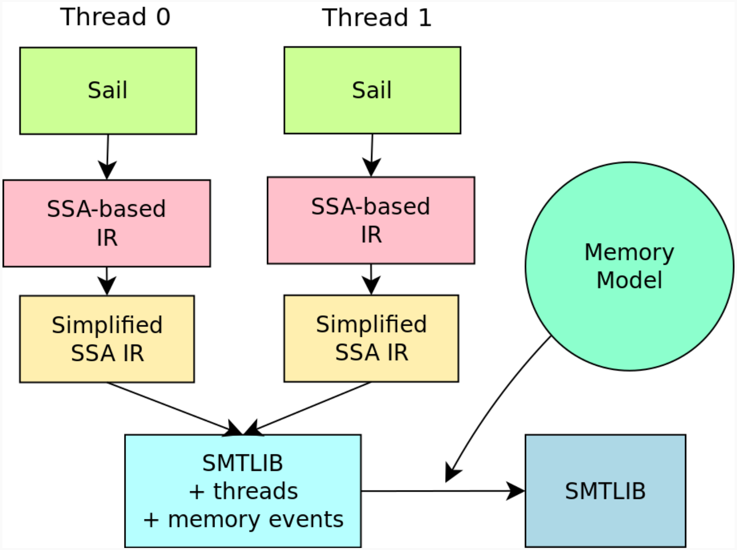
- Related work: Dartagnan, Cerberus-BMC

## Axiomatic concurrency semantics for Sail

```
0:X1=x; 0:X3=y;
1:X1=y; 1:X3=x;
}
 P0          | P1          ;
 LDR W0,[X1] | LDR W0,[X1] ;
 MOV W2,#1   | MOV W2,#1   ;
 STR W2,[X3] | STR W2,[X3] ;
exists
(0:X0=1 /\ 1:X0=1)
```

Load-buffering litmus test for ARMv8-A

## Axiomatic concurrency semantics for Sail



- We unfold the Sail definitions for the each instruction in the litmus test
    - Sail specification also specifies instruction syntax, so the litmus file parser is dynamically derived from the specification
- Larger tests can generate SSA graphs with 500 000+ basic blocks
- Dependent on partial evaluation and symbolic execution to reduce those graphs
    - Most values are concrete, but values returned by memory reads can be symbolic
- Hard to guarantee alignment and other architectureal restrictions for symbolic addresses (quite commonly used in litmus files)
- Large SSA graph usually reduced to about 20-100 lines of SMT per thread

## Axiomatic concurrency semantics for Sail

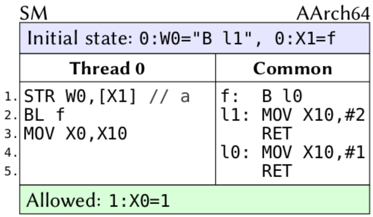
- So far we have tested on approximately 100 litmus tests for ARMv8 using a simplified model
- More engineering work is required to support the full ASL derived spec
- Get the same result as existing RMEM operational model on these tests
- Most execution time taken up by this simplification process, not by the SMT solver

## Axiomatic concurrency semantics for Sail

- With such complete semantics we can look at features non considered by other tools
- Often very important for systems code (operating systems)
- One interesting aspect of the architecture we have been exploring with this tool is the relaxed-memory behavior of self-modifying code
- Relevant for JIT compilers (e.g. Javascript engines!), dynamic linkers, etc

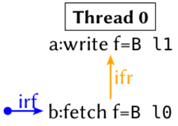
Joint work with: Ben Simner, Shaked Flur, Christopher Pulte, Jean Pichin-Pharabod, Luc Maranget, Peter Sewell

# Self-modifying code



SM AArch64

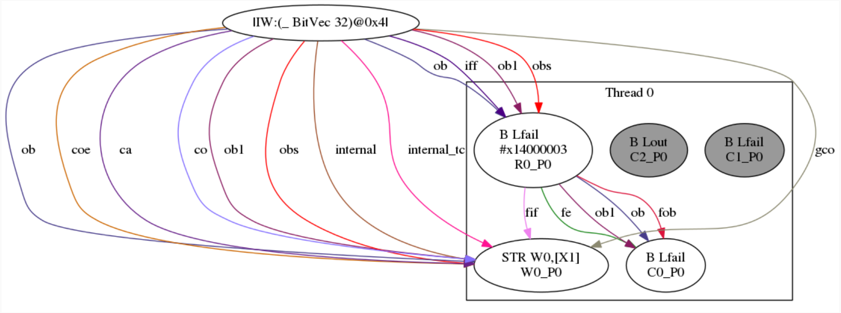
| Initial state: `0:W0="B l1", 0:X1=f` | |
|---|---|
| **Thread 0** | **Common** |
| `STR W0,[X1] // a` | `f:  B l0` |
| `BL f` | `l1: MOV X10,#2` |
| `MOV X0,X10` | `    RET` |
| | `l0: MOV X10,#1` |
| | `    RET` |
| Allowed: `1:X0=1` | |



$$\begin{tikzcd} & \boxed{\textbf{Thread 0}} \\ & \text{a:write f=B l1} \\ \bullet \arrow[r, "irf"] & \text{b:fetch f=B l0} \arrow[u, "ifr"'] \end{tikzcd}$$

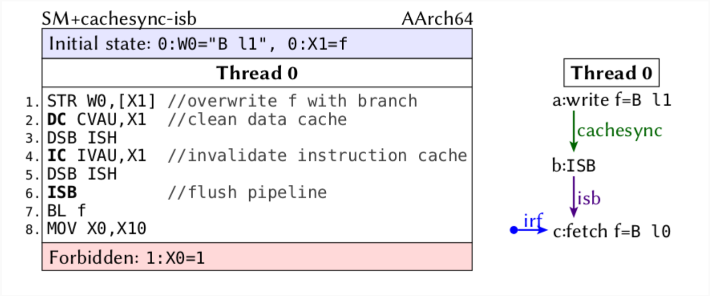
- Store can overwrite `B l0` with `B l1` at `f`
- But it might not have taken effect by the time we execute f, so `X0 = 1` is allowed
- `X0 = 2` is also allowed

# Self-modifying code



(Output graph drawing still very much work-in-progress)

## Self-modifying code


SM+cachesync-isb AArch64

Initial state: `0:W0="B l1", 0:X1=f`

**Thread 0**

```
STR W0,[X1] //overwrite f with branch
DC CVAU,X1  //clean data cache
DSB ISH
IC IVAU,X1  //invalidate instruction cache
DSB ISH
ISB         //flush pipeline
BL f
MOV X0,X10
```

Forbidden: `1:X0=1`

$$\begin{tikzcd}
 & \boxed{\text{Thread 0}} \\
 & \text{a:write f=B l1} \arrow[d, "\text{cachesync}"] \\
 & \text{b:ISB} \arrow[d, "\text{isb}"] \\
\bullet \arrow[r, "\text{irf}"] & \text{c:fetch f=B l0}
\end{tikzcd}$$

- Not just a single barrier!
- . . . need `DC` (data) and `IC` (instruction) cache maintenance instructions
- . . . and both `DSB` and `ISB` barriers

Permitted behavior is very weak!

## Conclusion

- Rigorous semantics for real production architectures
- Usable for interactive proof and complete with system features
- Can build a tool capable of exploring relaxed memory behaviors of these specifications using axiomatic descriptions of memory models
- Many other exciting projects enabled by such ISA specifications

Sail: **`https://github.com/rems-project/sail`**

ARMv8.5-A: **`https://github.com/rems-project/sail-arm`**

RISC-V: **`https://github.com/rems-project/sail-riscv`**

REMS