Isla: Integrating full-scale ISA semantics and axiomatic concurrency models (extended version)

Alasdair Armstrong[1], Brian Campbell[2], Ben Simner[1], Christopher Pulte[1], and Peter Sewell[1]

[1] University of Cambridge, Cambridge, UK
[2] University of Edinburgh, Edinburgh, UK

Abstract. Architecture specifications such as Armv8-A and RISC-V are the ultimate foundation for software verification and the correctness criteria for hardware verification. They should define the allowed sequential and relaxed-memory concurrency behaviour of programs, but hitherto there has been no integration of full-scale instruction-set architecture (ISA) semantics with axiomatic concurrency models, either in mathematics or in tools. These ISA semantics can be surprisingly large and intricate, e.g. 100k+ lines for Armv8-A.

In this paper we present a tool, Isla, for computing the allowed behaviours of concurrent litmus tests with respect to full-scale ISA definitions, in Sail, and arbitrary axiomatic relaxed-memory concurrency models, in the Cat language. It is based on a generic symbolic engine for Sail ISA specifications, which should be valuable also for other verification tasks. We equip the tool with a web interface to make it widely accessible, and illustrate and evaluate it for Armv8-A and RISC-V.

By using full-scale and authoritative ISA semantics, this lets one evaluate litmus tests using arbitrary user instructions with high confidence. Moreover, because these ISA specifications give detailed and validated definitions of the sequential aspects of *systems* functionality, as used by hypervisors and operating systems, e.g. instruction fetch, exceptions, and address translation, our tool provides a basis for developing concurrency semantics for these. We demonstrate this for the Armv8-A instruction-fetch model and self-modifying code examples of Simner et al.

1 Introduction

A processor architecture should define, for any initial machine state, the set of all architecturally allowed observable executions — thus specifying the basic assumptions for programming and for software verification, and the correctness criterion for hardware verification. Architecture specifications have two main parts: the sequential and relaxed-memory concurrent aspects of instruction behaviour, each of which have been studied in previous work. For Armv8-A and RISC-V, Armstrong et al. have established full-scale sequential models in Sail [10,14], a domain-specific language for instruction-set architecture

(ISA) specification, that are complete enough to boot real-world operating systems such as Linux. For Armv8-A this model is automatically derived from the authoritative Arm-internal specification [23], while for RISC-V it has been hand-written and adopted by RISC-V International. On the concurrency side, relaxed-memory semantics can be specified in two main styles: either as *abstract-microarchitectural operational* models, characterising observable behaviour with explicit out-of-order execution and buffering, or as *axiomatic* models, expressed as a predicate over complete candidate executions represented as graphs of memory events. For Armv8-A and RISC-V "user" concurrency, both exist [21,7,1,8], along with a "Promising ARM" variant [22]. For Armv8-A they have been proved equivalent [21,20]; the authoritative vendor definition is the axiomatic one.

However, while an architecture *should* define the set of allowed executions for arbitrary programs, hitherto there has been no integration of full-scale ISA definitions with axiomatic concurrency models, either in mathematics or in tools (for operational models, this has only been done for RISC-V; other operational models have used small ISA fragments). Research and industry practice for relaxed memory semantics rely on making the semantics *executable as a test oracle*: not just a paper definition (in prose or mathematics), but tool-supported definitions that for small litmus-test examples can *compute* the set of all allowed executions, that can then be compared against experimental data. Many tools have been developed for operational and axiomatic architectural concurrency models [31,19,29,30,25,18,24,13,16,6,4,8,11,28,27,17], with axiomatic tools notably including the Herd tool of Alglave and Maranget [6,4,8], that can evaluate litmus tests w.r.t. axiomatic memory models specified in a relational-algebra style in the Cat language [2]. However, all of these previous tools for axiomatic models have (at best) used hard-coded ISA semantics that cover only small fragments of the complete architecture. For example, Zhang et al [31] use a SMT solver based approach for SoC verification, with a user-specified memory model (TSO or SC), however the instruction level abstractions (ILAs) they use are much more abstract than the ISA semantics we consider.

In this paper we describe a tool, Isla, that integrates full-scale ISA specifications, in Sail, with arbitrary axiomatic models, in the Cat language. We first build a generic symbolic execution library for Sail specifications—which should also be valuable for other verification tasks. We use this to construct a tool for symbolically running binary litmus tests for any Sail ISA under any (non-recursive) Cat axiomatic memory model, using an SMT solver. We equip it with a web interface to make it widely accessible, and illustrate and evaluate all this for Armv8-A and RISC-V. Isla is available at https://isla-axiomatic.cl.cam.ac.uk and https://github.com/rems-project/isla. This is an extended version of the paper, including appendices showing the main parts of the full Sail/ASL semantics of a sample Armv8-A instruction (`add x4, x3, #1`); the Armv8-A axiomatic concurrency model (combining the official Arm specification for user concurrency [9,12] with the additions for instruction fetch semantics by Simner et al. [26]); and examples of the latter.

Our approach has several key advantages, which all follow from the fact that mainstream industry ISAs are surprisingly large and intricate. The Armv8-A ISA specification is around 100k lines. It defines the sequential behaviour of the full instruction set in all its detail, including e.g. instruction decoding, behaviour at each exception level, register banking, floating-point, vector instructions, system registers, exceptions, address translation, virtualisation, security extensions, and a host of optional architectural features. Simple litmus tests developed to investigate user concurrency have historically used only very few instructions and very little of this, and hand-written ISA models have sufficed, but even a 'simple' `ADD` instruction can, in reality, involve surprisingly much of the specification. If one wants to examine arbitrary compiler-generated code one needs many more instructions; and to develop systems concurrency semantics, e.g. covering the concurrency behaviour of instruction fetch, exceptions, or address translation, one might need any of the specification — and it would be exceedingly laborious and error-prone to reproduce it by hand in a hard-coded semantics. By handling the full authoritative Armv8-A ISA, we automatically support litmus tests that use arbitrary instructions, and we enable research on systems concurrency, with high confidence that the ISA follows the vendor specification. We demonstrate this by applying our tool to the model and examples for self-modifying code by Simner et al. [26], and our integration has also identified several places where the ISA specification needs modifications to correctly give the intended behaviour in a concurrent setting, e.g. to remove or enforce additional ordering. Because this is based on authoritative Arm and RISC-V ISA specifications, the work should enable relaxed-memory behaviour to be included in the standard test-edit-debug cycle used in the development of such large and critical specifications.

2 Implementation

Axiomatic relaxed-memory concurrency models, being expressed as logical constraints over candidate execution graphs, lend themselves to solver-based tool implementations. For the instruction-semantics part of such a tool, the most direct approach would be to translate the ISA semantics (for the instructions that occur in a litmus test) directly into SMT and combine that with the axiomatic-model constraints, roughly along the lines of Alglave et al. [3]. That approach was followed by Simner et al. [26], who compiled Sail directly into SMT to test an axiomatic model for instruction-fetch tests, but using a small handwritten Arm fragment, rather than the full Sail model derived from the Arm-internal model. The problem with this direct approach is one of scale: as one covers more of the Arm semantics, the resulting SMT problem simply becomes too large to be practicable. For example, for a load instruction, the virtual address must be translated into a physical address, which is a complex process with a great deal of configurability—there may be zero, one, or two stages of address translation, the page size may vary, the number of levels used in the page table may differ, etc. This approach also required the top level fetch-execute-decode loop to be

handled specially, as one cannot translate such an unbounded loop directly into SMT, which imposes significant constraints on the shape of allowable tests.

In contrast, here we build and use a generic symbolic evaluation for Sail definitions using the Z3 SMT solver, which lets us compute the possible symbolic thread-local traces of each instruction, and hence of each thread (treating memory read values as unknowns, left to the concurrency model constraints). It also lets us use the same fetch-decode-execute loop that is used for emulation and co-simulation (which embodies various architecture-specific subtleties).

2.1 Symbolic execution for Sail

Sail is attractive for symbolic execution for several reasons. First, it is an intentionally simple language, lacking many of the features found in general-purpose languages. Second, it has to support very few programs, just the specifications of major ISAs, so (unlike tools for conventional programming languages) we can tune the execution to them. Third, almost all of the loops in these programs are bounded. Our starting point is the translation of Sail to C, for emulation, by Armstrong et al [10]. This goes via a simple goto-language intermediate representation which is already well-suited for this task.

Static function linearisation Our symbolic execution always creates a new task when we hit a branch, and we do not ever merge these tasks at join points. This is a good strategy for instruction semantics, as it simplifies the symbolic execution engine significantly, but it does mean some code can cause unnecessary branching. To avoid this we have a static rewrite that can take a function with if statements and rewrite it into a 'linear' form, e.g. as below:

```
var x = 2;
if undefined {
  x = x + 1
} else {
  x = x + 2
};
return x
```

```
let x0 = 2;
let b = undefined;
let x1 = x0 + 1;
let x2 = x0 + 2;
let x3 = ite(b, x1, x2);
return x3
```

This works by translating the body of the function into SSA form, then replacing the ϕ-functions with if-then-else (ite) functions that translate into the SMT `ite`. This results in a more complex SMT expression, but less branching in the symbolic execution, so it is a trade-off, but often worthwhile.

Per-thread candidate executions For each litmus-test thread this symbolic execution will produce a number of *candidate executions*, each of which is a sequence of memory events (memory reads and writes, fences, register accesses, and so on) with the symbolic values of these events potentially being constrained by

some SMT formula for the overall execution. For example, consider the Armv8-A instruction `add x4, x3, #1`. Its multi-page semantics is given in Appendix A, from which our symbolic evaluator generates an execution:

```
(declare-const input (_ BitVec 64))
(read-reg |R3| nil input)
(define-const output (bvadd input #x0000000000000001))
(write-reg |R4| nil output)
```

where the SMTLIB formula is defined by the `declare-const` and `define-const` statements, with `read-reg` and `write-reg` effects indicating which variables in the SMT formula correspond to the values read and written to registers (which are otherwise just global variables) by the instruction. We simplify here for brevity, omitting the negative, zero, carry and overflow flags that the model computes. For more complex instructions, there are additional effects for memory accesses, cache maintenance events, barriers, and so on.

2.2 Checking a litmus test

Fig. 1 shows the overall process of checking a litmus test. Tests can be supplied either in the `.litmus` format of previous axiomatic and operational tools [5,4,13], reusing the parser from [4], or as a TOML file (a standard configuration file format, with libraries available for most languages). We first assemble the test with a conventional assembler into an ELF binary and load it into the representation of memory that will be used, before initialising the model with the program counter set to the entry point for each thread, then we symbolically execute the instructions in each thread separately, using the Sail semantics for each instruction, plus the same fetch-execute-decode loop in Sail we would use for emulation, to produce sets of per-thread traces as above. Treating litmus tests essentially as binaries, rather than the more-or-less ad hoc fragments of assembly abstract syntax used by earlier tools, accommodates the fact that the Armv8-A model does not define an abstract syntax, and reduces the gap between what the tool evaluates and what is run in experimental testing. Note that the Arm assembly in Fig. 1, as well as subsequent assembly snippets in this paper, use the standard Arm convention that `x0` and `w0` refer to the same register, where `w0` refers to the lower 32-bits of the register, and `x0` refers to the full 64-bit width.

We then generate an SMT problem for every combination of the candidate executions of each thread. This problem consists of the per-thread SMT formulae concatenated together (renaming variables as necessary to avoid name-clashes), combined with the axiomatic memory model (described in more detail below).

Finally, we need to generate some 'glue' SMT that connects the per-thread semantics with the memory model. For every effect in the per-thread SMT semantics we generate an enumeration of *events*, e.g. for an execution with two reads and two writes:

```
(declare-datatypes ((Event 0)) (((R1) (R2) (W1) (W2) (IW))))
```

The event `IW` is a special write event that represents the initial state. We generate relations such as `value-of` that relate events to their values as determined by the effects in the per-thread semantics, so if the second read event `R2` read the value `#xABCD`, `(value-of R2 #xABCD)` would be true. We generate *syntactic dependency relations* for address, data, and control dependencies, discussed in more detail in Section 2.3. Finally, there is a constraint on the final state of each test which specifies values expected in registers and memory after all threads have executed.

The Cat language represents axiomatic memory models as definitions of relations over the above events, and constraints over those relations, e.g. that specific relations are irreflexive, acyclic, or empty (or the negation of any of these). Relations are defined in a point-free relation-algebraic style, in terms of standard relational operators such as composition, intersection and union. The memory models we consider are all multi-copy-atomic, and all recursion in their definitions can trivially be replaced with (reflexive)-transitive closure. Herd's let rec construct computes the least solution to a set of equations [2], which is tricky to represent in SMT, so we do not support it. We believe even relations such as Power's (mutually recursive) preserved program order are nevertheless representable as SMT, so this limitation is mostly in our translation from Cat—we would likely want to use a different syntax to represent these relations for Isla. The Armv8-A axiomatic model is reproduced in Appendix B for reference.

A satisfiable solution to the overall SMT problem described above thus represents an execution permitted by the architecture. Parsing the model generated by the SMT solver allows us to generate a graph of the execution by instantiating each relation in the model with the various events. If all generated SMT problems are unsatisfiable for every combination of per-thread candidate executions then there are no permitted executions. If desired we can repeatedly ask the SMT solver for additional distinct models until we have all permitted executions.

2.3 Syntactic Dependency Analysis

Axiomatic memory models for relaxed hardware architectures rely heavily on notions of address, data, and control dependencies between instructions. For example, consider the following assembly:

```
    ldr  w0, [x1]   // load 32 bits from address in x1 into x0
    cbnz w0, LC01   // compare and branch if non-zero to LC01
  LC01:
    mov  w2, #1     // load 1 into x2
    str  w2, [x3]   // store 32 bit-value in x2 to the address in x3
```

Here there is a control dependency between the load (`ldr`) and the store (`str`), as the value read by the load is used to determine whether the branch instruction `cbnz` that precedes the store is taken or not. This control dependency exists irregardless of whether the branch is taken or not—its existence is purely determined by the syntactic structure of the above code.

In general, existing ISA descriptions do not cover this aspect of the architecture well, as they are principally developed only to describe the sequential be-

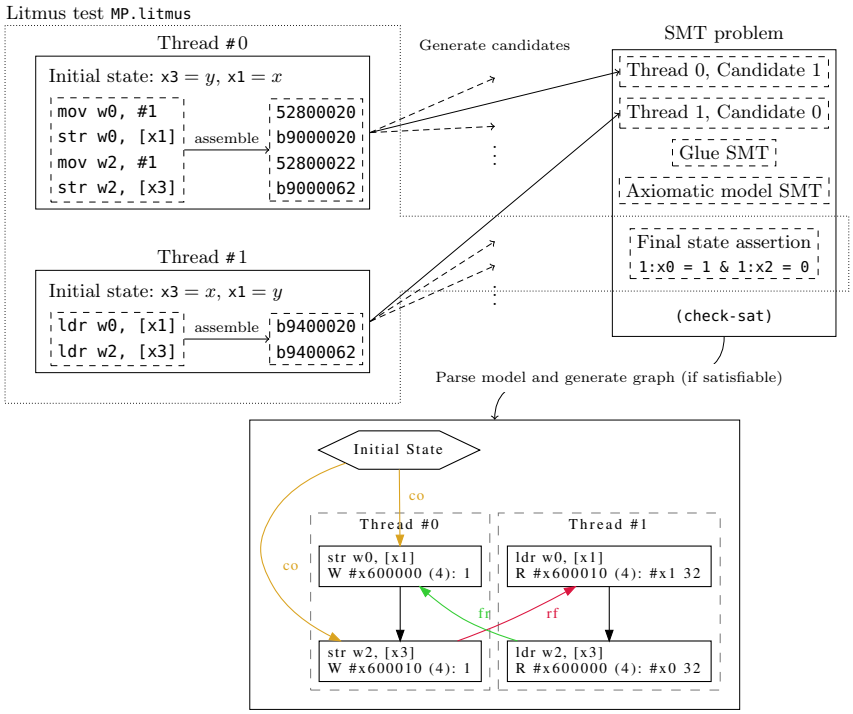


Fig. 1. Overview of process for checking the allowed executions of a litmus test

haviour. Previous tools have either hand-coded dependency information, which is acceptable for cut-down ISA models but too laborious and error-prone at the scale of the ISA models we use, or used a heavyweight taint-tracking interpreter [14]. Our approach avoids both of these. It is similar to the latter, computing dependencies from the ISA specification, but building the footprint analysis atop our symbolic execution library requires only around 500 LoC.

To express dependencies, we need to associate each event in our candidate executions with the syntactic instruction/opcode that generated them. To do this we use a Sail function `__instr_announce(opcode)`, called in each architecture's fetch-decode-execute loop just after fetching an instruction; this adds a special effect to the candidate execution recording the instruction opcode. We also have another special effect that delimits each fetch-decode-execute cycle, so each effect such as `read-mem` and `write-mem` that would give rise to an event can be associated with an opcode, as well as an index in the program order relation for its thread.

For each instruction we also need to know its *footprint*: data about the instruction including which input registers it reads, which output registers it writes, whether it is a branch instruction, and so on. It also contains *taint* information—we need to know which registers writes may contain data 'tainted' by a memory read performed by a load, or which input registers 'taint' data

written to memory. The Sail ISA specifications do not explicitly describe this footprint, so we are forced to derive it from the specification.

To do this we symbolically evaluate each opcode independently in a suitably unconstrained environment so as to capture all its possible behaviours. This can be computationally expensive due to the number of possible behaviours some instructions have, so we build a footprint cache to avoid re-computing this where possible. It turns out to be hard to distinguish ordinary branches from instructions that can cause an exception to occur, so we add a special branch address announce effect, created by a Sail function `__branch_address_announce` that we call in branch instructions. This also enables the taint tracking for branch addresses we need for control dependencies as described above. The taint tracking is achieved simply by looking at what sub-expressions in the generated SMT problem contain variables that also appear in the various effects in each trace.

Once we have this footprint information we can analyse it for the opcodes between each read and write effect and derive the necessary dependency relations over their events. Note that this dependency relation must be exact. If we under-approximate, we will allow executions that should be forbidden, and if we over-approximate we will forbid executions that should be allowed.

In some cases the current Arm-provided ISA specification does not include enough information to identify the architecturally respected dependencies, and our dependency analysis would identify a dependency when there should not be one. To solve this we add some special Sail functions that give fine-grained control of the dependency calculation. For example, in indirect branches we ignore any dependency between the target register `X`n and the link register `X30` by including a function in the Sail definition that tells the footprint analysis to ignore any relation it finds between the two registers.

```
if branch_type == BranchType_INDCALL then {
    ignore_dependency_edge(n, 30);
    X(30) = PC() + 4
};
```

This works by adding a special annotation in the candidate execution trace which can be used by the footprint analysis—for all other purposes it is a no-op. This information should properly become part of the architecture specification, as mistakes in the dependency calculations could be a source of soundness bugs. The lack of support for this information in existing ISA specifications can partly be explained by the lack of tooling to properly explore the integration of ISA specifications with concurrency, something we hope a tool such as ours can address.

2.4 Web Interface

Fig. 2 shows the web interface we have developed for our tool, based on the web interface for the C memory model tool Cerberus-BMC by Lau et al. [15]. This can either be run locally, or via a website, https://isla-axiomatic.cl.cam.ac.uk.

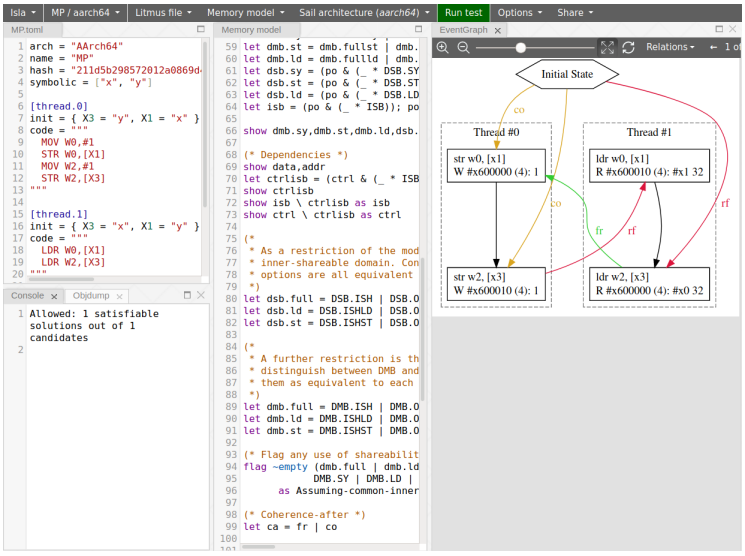


Fig. 2. Web interface for the tool

3 System Litmus Tests

As mentioned previously, one advantage of our tool is that, because it supports the full sequential ISA, it enables easy experimentation with tests and models outside the scope of previous tools, e.g. involving new systems features. For example, Simner et al. developed semantics for Arm instruction fetch and I/D cache maintenance [26]. Consider the litmus test in Fig. 3 [26, §3.3], a simple test involving self-modifying code. In order to run this test and the others in [26] our tool required only minimal changes: we had to add support for data-cache and instruction-cache maintenance events and relations for them in our Cat to SMT translation. Additionally we needed to generalise how we generated the `rf` (reads-from) relation to generate both the regular `rf` relation and the new `irf` (instruction-reads-from) relation. Because our tool already runs tests using a fetch-execute-decode loop, all the instruction fetch events were already available—we in fact filter them out when running user-mode tests.

When generating candidate executions for a thread we normally do not assume anything about what other threads may be doing, but for self-modifying code this would clearly be problematic, as it would imply that any other thread could modify any of this thread's instructions arbitrarily. We therefore mark the memory locations that contain instructions that can be modified and provide in advance all the possible values they might take.

```
1        str w0, [x1]
2        dc cvau, x1
3        dsb ish
4        ic ivau, x1
5        dsb ish
6        isb
7        bl f
8        mov w2, w10
9        b Lout
10   f:  b l0
11   l1: mov w10, #2
12       ret
13   l0: mov w10, #1
14       ret
15   Lout:
```

In the initial state register `x1` contains the address of the label `f`, and register `w0` contains the opcode for the branch instruction `b l1`. Without the highlighted cache-maintenance and barrier instructions on lines 2–6, the write of that opcode to `f` performed by the store on line 1 may or may not be observed before the instruction fetch for `f`, so at the end of the test the register `w2` can contain either 1 or 2, depending on whether we branched to `l1` or `l0`.

The highlighted instructions on lines 2–6 are a sequence of data-cache (`dc`) and instruction-cache (`ic`) maintenance instructions with requisite data and instruction barriers that must occur to guarantee that the write is observed by the instruction fetch, as documented by the Armv8-A architecture reference manual [7] and captured by the axiomatic model of Simner et al. [26], which is reproduced in Appendix B

Appendix C shows a screenshot of our tool executing this test.

Fig. 3. Self-modifying code litmus test SM+cachesync-isb

4 Results and Comparisons

We evaluate our tool for correctness and performance with respect to Herd using previous corpora of tests.

We select 3798 litmus tests for both Armv8-A and RISC-V to compare between our tool and Herd—these tests include a representative set of features such as barriers and atomics, while exercising all of the basic litmus test shapes. All tests were run on a 2.6GHz Intel Xeon Gold 6240 CPU with 36 physical cores and 400GB of RAM. The tests are split into rough categories based on the contents of the tests. We ran 36 concurrent instances of both our tool and Herd across each set of tests, running Herd with the `-speedcheck fast` flag which causes it to stop enumerating executions when it resolves the final assertion in each test, which is the closest behaviour to how our tool behaves by default.

To assess correctness, we use a set of golden references for these above tests, for all of which the previous operational RMEM [13] and axiomatic Herd models and tools agree, and which have been extensively validated against hardware implementations. We confirm that our tool produces the same expected results as those models for all the litmus tests, including when run in exhaustive mode.

To assess performance, the table below gives the total real execution time for each batch of tests.

Test set	Number of tests	Isla	Herd
Armv8-A basic 2-thread	1377	49s	11s
Armv8-A basic 3-thread	161	11.7s	1.2s
Armv8-A exclusives	23	20.2s	1.5s
Armv8-A DMB/LD	70	7.4s	0.7s
Armv8-A PPO	2020	3m29.3s	16.2s
RISC-V basic 2-thread	36	0.7s	0.2s
RISC-V AMOs	111	2s	0.7s

In general Herd is faster for nearly all tests, but this is not surprising given the amount of detail in the full-scale instruction semantics that we are using, particularly for Armv8-A. Our goal is not to be faster, but to support those full-scale ISA semantics while remaining fast enough for practical purposes. We achieve this: most tests take only a second or so to run, which is perfectly usable interactively. For example, given the Armv8-A basic 3-thread tests, for a single sequential run of the tests, the shortest took 872ms to run, while the longest took 1231ms. The above batch times are similarly perfectly usable for (e.g.) regression testing while editing a model.

We also evaluate our tool with respect to that of Simner et al., for the instruction-fetch tests (which are currently not supported by Herd) in Section 6 of their paper. Our tool returns the expected results for all these tests, including the two tests (FOW and SM.F+ic) that were unsupported by their tool; Appendix C shows a screenshot of our tool executing FOW. In terms of performance, we note that their tool took 30 minutes to run just 90 of the 1377 basic 2-thread tests above, which is awkwardly slow for using a tool in practice, whereas when limiting our tool to 8 cores (to more closely match their experimental setup) our tool will execute all 1377 in under 3 minutes. We were additionally able to provide further validation that the Simner et al. model behaves as the standard Armv8-A model for non-self-modifying tests by showing that it behaves identically for all 3798 of the non-self-modifying tests above.

Acknowledgements This work was partially supported by the UK Government Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme, to deliver a DSbDtech enabled digital platform (grant 105694), ERC AdG 789108 ELVER, EPSRC programme grant EP/K008528/1 REMS, an Arm iCASE award, Arm, and Google. Approved for public release; distribution is unlimited. This work was supported 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 report are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

A One "Simple" Arm instruction: `add x4, x3, #1`

To give a sense for the complexity of the full Armv8-A ISA specification, in this appendix we give the main parts of the definition for one of the simplest instructions, `add x4, x3, #1`, that adds one to the value in register `x3` and puts the result in register `x4`. Instructions that touch memory are much more complex than this, e.g. with address translation potentially involving multiple page-table walks and many access checks (all of which is covered by full ISA specification that we use).

We give the Sail form of the semantics, which is automatically translated from the authoritative Arm-internal ASL definition, and which has been independently validated against the Arm-internal Architecture Validation Suite. The Sail and ASL correspond very closely, except that the Sail form has richer type

information, using a lightweight dependent type system for bitvector length and integer range types. This instruction is actually an instance of the family of instructions:

```
ADD <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

C6.2.4 ADD (immediate)
Add (immediate) adds a register value and an optionally-shifted immediate value, and writes the result to the destination register.

A.1 Full ASL/Sail semantics: Decoding, 1/2

First, the following clause of the `decode64` function pattern matches against a 32-bit opcode, extracts the opcode fields, and calls `addsub_immediate_decode`. From that point on, the ADD (immediate) and SUB (immediate) instructions are handled together.

```
val decode64 : bits(32) -> unit
  effect {configuration, escape, undef, wreg, rreg, rmem, wmem}

function clause decode64
  ((_:bits(1) @ 0b0010001 @ _:bits(24) as op_code) if SEE<1066) = {
    SEE = 1066;
    Rd : bits(5) = op_code[4 .. 0];
    Rn : bits(5) = op_code[9 .. 5];
    imm12 : bits(12) = op_code[21 .. 10];
    shift : bits(2) = op_code[23 .. 22];
    S : bits(1) = [op_code[29]];
    op : bits(1) = [op_code[30]];
    sf : bits(1) = [op_code[31]];
    addsub_immediate_decode(Rd, Rn, imm12, shift, S, op, sf)
}
```

A.2 Full ASL/Sail semantics: Decoding, 2/2

This second phase of decoding converts the opcode fields to the appropriate types, e.g. computing the mathematical integer corresponding to the immediate-value opcode field. It also throws a Sail exception in one case where a specific encoding has been used for a different family of instructions; the top-level handles that and decodes and executes the opcode as that family if need be. Decoding ends by calling the `addsub_immediate` execute function.

```
val addsub_immediate_decode :
    (bits(5), bits(5), bits(12), bits(2), bits(1), bits(1), bits(1))
    -> unit
    effect {escape, rreg, undef, wreg}
```

```
function addsub_immediate_decode(Rd, Rn, imm12, shift, S, op, sf) = {
    __unconditional = true;
    let 'd = UInt(Rd);  let 'n = UInt(Rn);
    let 'datasize = if sf == 0b1 then 64 else 32;
    let sub_op = op == 0b1;  let setflags = S == 0b1;
    imm : bits('datasize) = undefined : bits('datasize);
    match shift {
      0b00 => { imm = ZeroExtend(imm12, datasize) },
      0b01 => { imm = ZeroExtend(imm12 @ Zeros(12), datasize) },
      0b10 => { throw(Error_See("ADDG, SUBG")) },
      0b11 => { ReservedValue() }
    };
    __PostDecode();
    addsub_immediate(d, datasize, imm, n, setflags, sub_op)
}
```

A.3 Full ASL/Sail semantics: Execution

The main execute function reads the source register values, calls an auxiliary `AddWithCarry` to compute the mathematical result, including new `NZCV` flag values, and writes the target register value and (if the opcode requires it) those flag values. It handles subtraction by negating `operand2` and setting `carry_in` before doing an addition.

```
function addsub_immediate(d, datasize, imm, n, setflags, sub_op) = {
    result : bits('datasize) = undefined : bits('datasize);
    let operand1 : bits('datasize) = if n == 31 then SP() else X(n);
    operand2 : bits('datasize) = imm;
    nzcv : bits(4) = undefined : bits(4);
    carry_in : bits(1) = undefined : bits(1);
    if sub_op then {
        operand2 = ~(operand2);
        carry_in = 0b1
    } else {
        carry_in = 0b0
    };
    (result, nzcv) = AddWithCarry(operand1, operand2, carry_in);
    if setflags then {
        (PSTATE.N @ PSTATE.Z @ PSTATE.C @ PSTATE.V) = nzcv
    };
    if d == 31 & ~(setflags) then { SP() = result }
    else { X(d) = result }
}
```

A.4 Full ASL/Sail semantics: Auxiliary register-access functions

What look like register accesses in the above, e.g. `SP()` and `X(n)`, are actually indirected via register getter and setter functions, to handle the fact that in Armv8-A the stack pointer register `SP` is *banked*: there is a different `SP` register for each exception level. These functions therefore have to do another register read, not obvious from the opcode, of the register that holds the current exception level.

```
function aset_SP(value) = {
  assert('width == 32 | 'width == 64);
  if PSTATE.SP == 0b0 then {
      SP_EL0 = ZeroExtend(value)
  } else {
      match PSTATE.EL {
        el if el == EL0 => SP_EL0 = ZeroExtend(value),
        el if el == EL1 => SP_EL1 = ZeroExtend(value),
        el if el == EL2 => SP_EL2 = ZeroExtend(value),
        el if el == EL3 => SP_EL3 = ZeroExtend(value)
      }
  }
}
val aget_X : forall 'width 'n, 0 <= 'n <= 31 & 'width in {8, 16, 32,
    64}).
  (implicit('width), int('n)) -> bits('width) effect {rreg}

function aget_X(width, n) =
  if n != 31 then slice(_R[n], 0, width) else Zeros(width)
```

A.5 Full ASL/Sail semantics: Execution auxiliary functions

Finally we come to the actual (pure) arithmetic and computation of flag values, which is done over mathematical integers.

```
val AddWithCarry : forall ('N : Int), ('N >= 0 & 'N >= 0).
  (bits('N), bits('N), bits(1)) -> (bits('N), bits(4))

function AddWithCarry (x, y, carry_in) = {
  let 'unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
  let 'signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
  let result : bits('N) = __GetSlice_int('N, unsigned_sum, 0);
  let n : bits(1) = [result['N - 1]];
  let z : bits(1) = if IsZero(result) then 0b1 else 0b0;
  let c : bits(1) = if UInt(result)==unsigned_sum then 0b0 else 0b1;
  let v : bits(1) = if SInt(result)==signed_sum then 0b0 else 0b1;
  return((result, ((n @ z) @ c) @ v))
}
```

```
let iseq = [W]; (wco&scl); [DC]; (wco&scl); [IC]  (*1*)

(* Observed-by *)
let obs = rfe
        | fr
        | wco                                     (*2*)
        | irf                                     (*3*)
        | ifr; iseq                               (*4*)

(* Fetch-ordered-before *)
let fob = [IF]; fpo; [IF]                         (*5*)
        | [IF]; fe                                (*6*)
        | [ISB]; fe^-1; fpo                       (*7*)

(* Dependency-ordered-before *)
let dob = addr | data
        | ctrl; [W]
        | (ctrl | (addr; po)); [ISB]
     (* | [ISB]; po; [R] *)                       (*8*)
        | addr; po; [W]
        | (addr | data); rfi

(* Atomic-ordered-before *)
let aob = rmw
        | [range(rmw)]; rfi; [A|Q]

(* Barrier-ordered-before *)
let bob = [R|W]; po; [dmb.sy]
        | [dmb.sy]; po; [R|W]
        | [L]; po; [A]
        | [R]; po; [dmb.ld]
        | [dmb.ld]; po; [R|W]
        | [A|Q]; po; [R|W]
        | [W]; po; [dmb.st]
        | [dmb.st]; po; [W]
        | [R|W]; po; [L]
        | [R|W|F|DC|IC]; po; [dsb.ish]            (*9*)
        | [dsb.ish]; po; [R|W|F|DC|IC]            (*10*)
        | [dmb.sy]; po; [DC]                      (*11*)

(* Cache-op-ordered-before *)
let cob = [R|W]; (po&scl); [DC]                   (*12*)
        | [DC]; (po&scl); [DC]                    (*13*)

(* Ordered-before *)
let ob = (obs|fob|dob|aob|bob|cob)+

(* Internal visibility requirement *)
acyclic (po-loc|fr|co|rf) as internal

(* External visibility requirement *)
irreflexive ob as external

(* Atomic *)
empty rmw & (fre; coe) as atomic

(* Constrained unpredictable *)
let cff = ([W];loc;[IF]) \ ob^-1 \ (co;iseq;ob)   (*14*)
(cff_bad cff) = CU                                (*15*)
```

Fig. 4. Axiomatic model, as extended by Simner et al. [26]

B The Armv8-A axiomatic concurrency model

In this appendix we recall the Armv8-A axiomatic concurrency model, which combines the official Arm specification for user concurrency (from the Armv8-A manual Issue B.a for Armv8.2-A, ARM DDI 0487B.a [9,12]) and the additions for instruction fetch semantics by Simner et al [26]. Figure 4 gives the full definition as presented there.

The model is expressed in terms of axioms of candidate executions, complete hypothetical executions of the input program, abstracted in terms of memory events and relations over them. The usual fundamental relations are program order (`po`), relating same-thread events in the order of the execution's control-flow unfolding; reads-from (`rf`), relating write events to the read events that read from them; coherence (`co`), a total order on memory writes corresponding to the sequence they propagate to memory; and the derived from-reads relation (`fr = rf`$^{-1}$`; co`), relating reads to same-address writes that are coherence-after the write they read from.

The extended Arm model has four axioms. The `internal` axiom is a standard per-location-SC/coherence axiom; `atomic` specifies the atomicity guarantees given by load/store exclusive pairs and atomic memory operations. The axiom `external` is the "main" axiom. It essentially requires that the ordering induced by the interaction across threads, captured by the `obs` ("observed by") relation, is compatible with the thread-internal ordering `fob|dob|aob|bob|cob`. Here:

- `fob` is instruction-fetch-related ordering,
- `dob` ordering resulting from dataflow and control-flow dependencies,
- `aob` ordering around exclusive instructions and atomic memory operations,
- `bob` barrier ordering, and
- `cob` ordering due to cache maintenance operations.

Finally, the fourth axiom (15), related to instruction fetching, is explained below. The main additions for instruction fetching were:

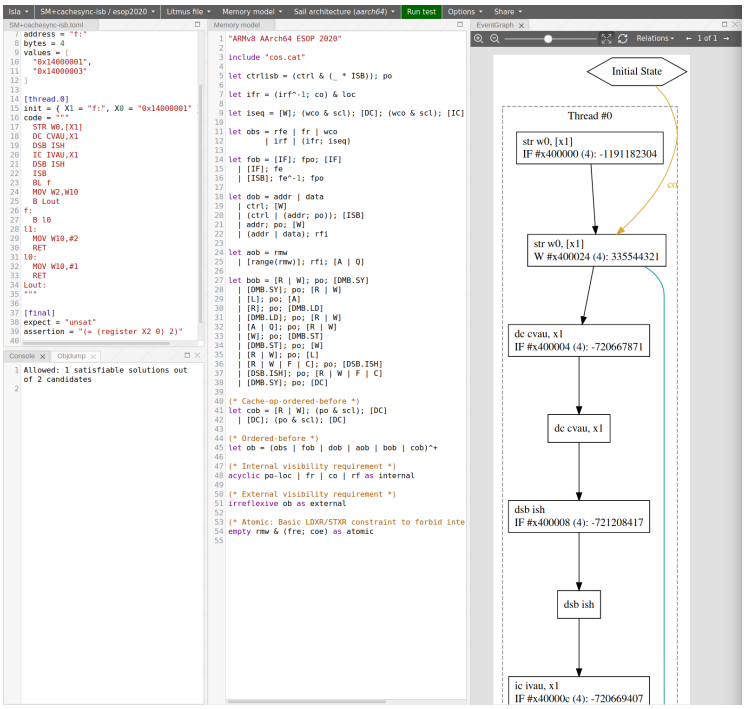
- The candidate execution has additional data:
  - events for instruction fetches (`IF`) and cache maintenance operations (`DC` and `IC`);
  - the `CU` bit, indicating constrained unpredictable executions; and
  - the relations `irf`, relating a write with instruction fetches that read from it; `wco`, extending the `co` coherence relation to include cache maintenance operations; `fpo`, the program order relation between instruction fetch events; `fe`, relating instruction fetches with any event originating from the execution of the fetched instruction; and `scl`, relating same-cache-line events.
- `obs` includes the extended coherence order `wco` (2), and orders any instruction fetch to be after the write it read from (3) and before any from-reads-related write that is sufficiently synchronised (4 and 1);
- `fob` orders fetches in fetch-program-order, (5), fetches before the instruction's execute event (6), and instruction fetches after program-order earlier ISBs (7);

– `bob` contains ordering created by `dsb.ish` (9 and 10) and ordering of `DC` instructions with respect to `dmb.sy` barriers (11);
– the model defines `cff`, the could-fetch-from relation, that for a given instruction fetch captures the writes the fetch could have read from, including the one it did read from (14); and
– specifies that certain executions have constrained unpredictable behaviour (15): if this set contains more than one write and if one of these is the write of an instruction that is considered to be not concurrently modifiable:

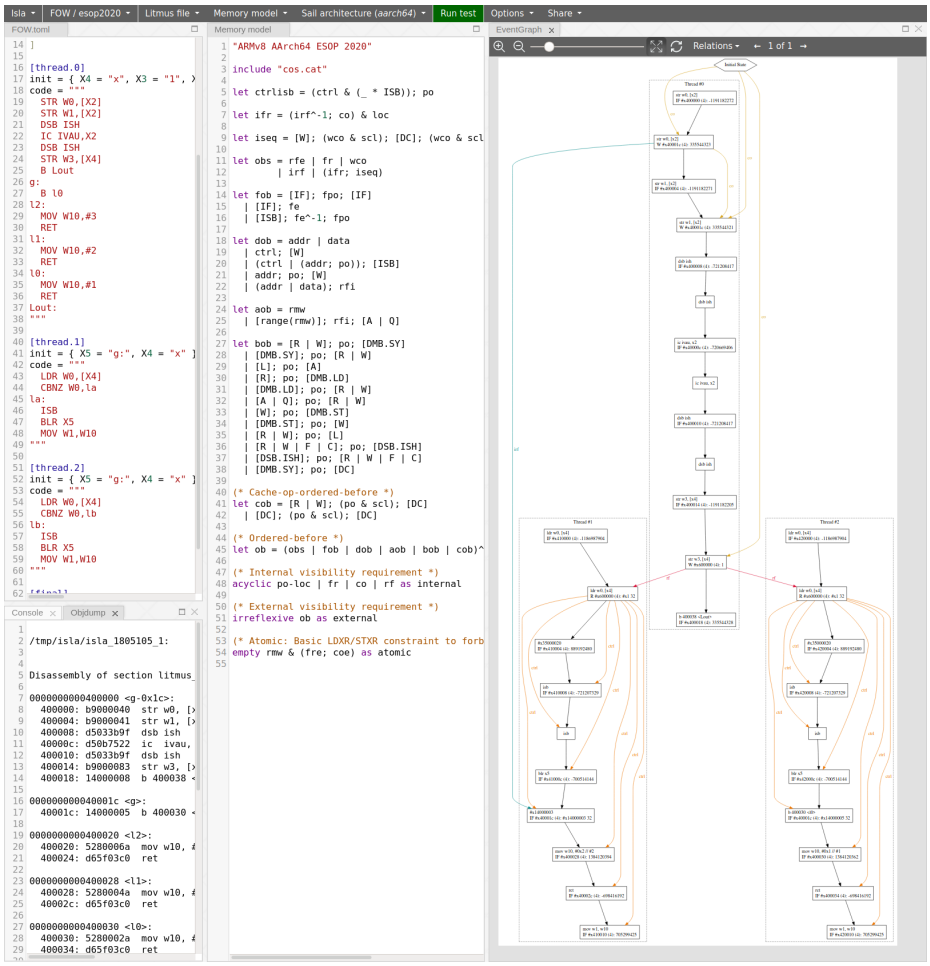
$$\texttt{cff\_bad cff} = \exists i \in \texttt{IF}.\ |\{w|(w,i) \in \texttt{cff}\}| > 1 \wedge \exists w.(w,i) \in \texttt{cff} \wedge \neg\textsf{concurrently-modifiable}\ (\textsf{val}\ w).$$

C Instruction-fetch litmus test examples

The screenshot below shows our tool evaluating the SM+cachesync-isb test from [26], showing the allowed execution where the instruction fetch at `f` observes the earlier write—the blue line in the is an `irf` edge from the write to the instruction fetch at `f`.



The screenshot below shows our tool evaluating the FOW test from [26], which demonstrates how each thread can observe different values for the same instruction, despite cache invalidation. This is one of the tests Simner et al's axiomatic tool was unable to handle, as it features one thread jumping into another thread's code where an instruction can be modified.



References

1. The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Document Version 20191214-draft. https://riscv.org/technical/specifications/ (Jul 2020), accessed 2020-09-23. 238 pages
2. Alglave, J., Cousot, P., Maranget, L.: Syntax and semantics of the weak consistency model specification language cat. CoRR **abs/1608.07531** (2016), http://arxiv.org/abs/1608.07531
3. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Computer Aided Verification - 25th International Conference, CAV. pp. 141–157 (2013). https://doi.org/10.1007/978-3-642-39799-8_9, https://doi.org/10.1007/978-3-642-39799-8_9
4. Alglave, J., Maranget, L.: The `diy7` tool. http://diy.inria.fr/, accessed 2021-01-28
5. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Litmus: running tests against hardware. In: Proceedings of TACAS 2011: the 17th international conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 41–44. Springer-Verlag, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_5
6. Alglave, J., Maranget, L., Tautschnig, M.: Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. ACM TOPLAS **36**(2), 7:1–7:74 (Jul 2014). https://doi.org/10.1145/2627752
7. Arm: Arm Architecture Reference Manual: Armv8, for Armv8-A architecture profile. https://developer.arm.com/documentation/ddi0487/fc (Jul 2020), accessed 2020-09-23. 8248 pages.
8. Arm: Memory model tool. https://developer.arm.com/architectures/cpu-architecture/a-profile/memory-model-tool (2020), accessed 2021-01-26
9. ARM Ltd.: ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile) (2017), ARM DDI 0487B.a (ID033117), https://developer.arm.com/documentation/ddi0487/b/?lang=en
10. Armstrong, A., Bauereiss, T., Campbell, B., Reid, A., Gray, K.E., Norton, R.M., Mundkur, P., Wassell, M., French, J., Pulte, C., Flur, S., Stark, I., Krishnaswami, N., Sewell, P.: ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS (Jan 2019), http://www.cl.cam.ac.uk/~pes20/sail/
11. Bornholt, J., Torlak, E.: Synthesizing memory models from framework sketches and litmus tests. In: Cohen, A., Vechev, M.T. (eds.) Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 467–481. ACM (2017). https://doi.org/10.1145/3062341.3062353, https://doi.org/10.1145/3062341.3062353
12. Deacon, W.: The ARMv8 Application Level Memory Model. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat (2016)
13. Flur, S., French, J., Gray, K., Pulte, C., Sarkar, S., Sewell, P.: RMEM. www.cl.cam.ac.uk/~pes20/rmem/ (2020), accessed 2021-01-28
14. Gray, K.E., Kerneis, G., Mulligan, D., Pulte, C., Sarkar, S., Sewell, P.: An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In: Proc. MICRO-48, the 48th Annual IEEE/ACM International Symposium on Microarchitecture (Dec 2015). https://doi.org/10.1145/2830772.2830775
15. Lau, S., Gomes, V.B.F., Memarian, K., Pichon-Pharabod, J., Sewell, P.: Cerberus-BMC: A Principled Reference Semantics and Exploration Tool for Concurrent and Sequential C. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification. pp. 387–397. Springer International Publishing (2019)

16. Mador-Haim, S., Maranget, L., Sarkar, S., Memarian, K., Alglave, J., Owens, S., Alur, R., Martin, M.M.K., Sewell, P., Williams, D.: An axiomatic memory model for POWER multiprocessors. In: Proceedings of the 24th International Conference on Computer Aided Verification. pp. 495–512 (2012). https://doi.org/10.1007/978-3-642-31424-7_36
17. Martonosi Research Group: Check research tools and papers. https://check.cs.princeton.edu/, accessed 2021-01-28
18. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Proceedings of TPHOLs 2009: Theorem Proving in Higher Order Logics, LNCS 5674. pp. 391–407 (2009). https://doi.org/10.1007/978-3-642-03359-9_27, 10.1007/978-3-642-03359-9_27
19. Park, S., Dill, D.L.: An executable specification and verifier for relaxed memory order. IEEE Trans. Computers **48**(2), 227–235 (1999). https://doi.org/10.1109/12.752664, https://doi.org/10.1109/12.752664
20. Pulte, C.: The Semantics of Multicopy Atomic ARMv8 and RISC-V. Ph.D. thesis, University of Cambridge (2018), https://www.repository.cam.ac.uk/handle/1810/292229
21. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. In: POPL 2018: Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (Jan 2018). https://doi.org/10.1145/3158107
22. Pulte, C., Pichon-Pharabod, J., Kang, J., Lee, S.H., Hur, C.K.: Promising-ARM/RISC-V: A simpler and faster operational concurrency model. In: PLDI 2019: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (Jun 2019). https://doi.org/10.1145/3314221.3314624
23. Reid, A.: Trustworthy specifications of ARM v8-A and v8-M system level architecture. In: FMCAD 2016. pp. 161–168 (October 2016), https://alastairreid.github.io/papers/fmcad2016-trustworthy.pdf
24. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: Proceedings of PLDI 2011: the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation. pp. 175–186 (2011). https://doi.org/10.1145/1993498.1993520, http://doi.acm.org/10.1145/1993498.1993520
25. Sarkar, S., Sewell, P., Zappa Nardelli, F., Owens, S., Ridge, T., Braibant, T., Myreen, M., Alglave, J.: The semantics of x86-CC multiprocessor machine code. In: Proceedings of POPL 2009: the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages. pp. 379–391 (Jan 2009). https://doi.org/10.1145/1594834.1480929, http://doi.acm.org/10.1145/1594834.1480929
26. Simner, B., Flur, S., Pulte, C., Armstrong, A., Pichon-Pharabod, J., Maranget, L., Sewell, P.: Armv8-a system semantics: instruction fetch in relaxed architectures. In: ESOP 2020: Proceedings of the 29th European Symposium on Programming (Apr 2020), http://www.cl.cam.ac.uk/~pes20/iflat/top-extended.pdf
27. Trippel, C., Manerkar, Y.A., Lustig, D., Pellauer, M., Martonosi, M.: Full-stack memory model verification with tricheck. IEEE Micro **38**(3), 58–68 (2018). https://doi.org/10.1109/MM.2018.032271062, https://doi.org/10.1109/MM.2018.032271062
28. Wickerson, J., Batty, M., Sorensen, T., Constantinides, G.A.: Automatically comparing memory consistency models. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming

Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 190–204. ACM (2017), http://dl.acm.org/citation.cfm?id=3009838

29. Yang, Y., Gopalakrishnan, G., Lindstrom, G., Slind, K.: Analyzing the intel itanium memory ordering rules using logic programming and SAT. In: Geist, D., Tronci, E. (eds.) Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2860, pp. 81–95. Springer (2003). https://doi.org/10.1007/978-3-540-39724-3_9, https://doi.org/10.1007/978-3-540-39724-3_9
30. Yang, Y., Gopalakrishnan, G., Lindstrom, G., Slind, K.: Nemos: A framework for axiomatic and executable specifications of memory consistency models. In: 18th International Parallel and Distributed Processing Symposium (IPDPS 2004),Santa Fe, New Mexico, USA (2004). https://doi.org/10.1109/IPDPS.2004.1302944, http://dx.doi.org/10.1109/IPDPS.2004.1302944
31. Zhang, H., Trippel, C., Manerkar, Y.A., Gupta, A., Martonosi, M., Malik, S.: ILA-MCM: Integrating memory consistency models with instruction-level abstractions for heterogeneous system-on-chip verification. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–10 (2018). https://doi.org/10.23919/FMCAD.2018.8603015