Arm systems semantics

Ben Simner
University of Cambridge

November 28, 2022

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text.

This dissertation contains 55200 total words as counted by `detex | wc -w`.

This dissertation does not exceed the regulation length of 60 000 words, including tables and footnotes.

10 # Contents

11 **1 Introduction** **6**
12 1.1 Thesis Introduction 7
13 1.2 Modern microprocessors and Arm 7
14 1.3 Processor architecture and Armv8-A 7
15 1.4 Semantics of these architectures 7
16 1.5 Systems software 7
17 1.6 Hypervisors 8

18 **Preface to relaxed memory** **9**

19 **2 Modelling Armv8-A: background** **10**
20 2.1 Litmus tests 10
21 2.1.1 The litmus zoo 11
22 2.2 Operational models 13
23 2.3 Promising 13
24 2.4 Axiomatic-style models 15
25 2.4.1 Informal description and the Cat language 15
26 2.4.2 The Armv8-A axiomatic model 15

27 **3 Instruction fetch: an introduction** **17**
28 3.1 Introduction 17
29 3.2 Industry Practice and the Existing ARMv8-A Prose 19
30 3.3 Instruction Fetch Phenomena and Examples 22
31 3.3.1 Instruction-Fetch Atomicity 23
32 3.3.2 Coherence 23
33 3.3.3 Instruction Synchronisation 25
34 3.3.4 Multi-Copy Atomicity 27
35 3.3.5 Strength of the `IC` Instruction 28
36 3.3.6 Strength of the `DC` Instruction 29
37 3.4 An Operational Semantics for Instruction Fetch 29
38 3.5 An Axiomatic Semantics for Instruction Fetch 31
39 3.6 Validation 33
40 3.7 Related Work 35
41 3.8 Conclusion 36

42 **4 Instruction fetch: operationally** **37**
43 4.1 Shape of the model 37
44 4.2 Extending flat 37

45 **5 Instruction fetch: axiomatically** **38**
46 5.1 Candidate model 38
47 5.2 The axioms 38
48 5.3 Executing the model in Isla 38

49 **6 Instruction fetch: validation** **39**
50 6.1 Executable operational semantics 39
51 6.1.1 Making the model executable 39

6.2 Extension to isla-axiomatic . . . 39
6.3 Hardware testing . . . 39
6.3.1 Custom harness . . . 39
6.3.2 Extending herdtools . . . 39
6.3.3 Results from hardware . . . 39
6.4 Correspondence between the models . . . 39

7 Pagetables and the VMSA **40**
7.1 Introduction . . . 40
7.2 Virtual Memory . . . 40
7.3 Arm Translation Tables . . . 42
7.3.1 Translation table format . . . 42
7.3.2 The Arm translation table walk . . . 43
7.4 Virtualisation and a second stage of translation . . . 47
7.5 Translation regimes . . . 49
7.6 Arm pseudocode . . . 50
7.6.1 The lifecycle of a store . . . 50
7.6.2 Writes to memory . . . 51
7.6.3 Translation table walks . . . 55
7.7 Caching in TLBs . . . 55
7.8 Arm ASL Reference . . . 58
7.8.1 `AArch64.TranslateAddress` . . . 58
7.8.2 `AArch64.FullTranslate` . . . 59
7.8.3 `AArch64.S1Translate` . . . 60
7.8.4 `AArch64.S1Walk` . . . 63
7.8.5 `AArch64.S2Translate` . . . 65
7.8.6 `AArch64.S2Walk` . . . 67
7.8.7 `AArch64.FetchDescriptor` . . . 69

8 Relaxed virtual memory **70**
8.1 Virtual memory litmus tests . . . 72
8.2 Aliased data memory . . . 74
8.2.1 Virtual coherence . . . 74
8.2.2 Aliasing different locations . . . 78
8.2.3 Might be same (physical) address . . . 79
8.3 What can be cached in TLBs . . . 79
8.3.1 Microarchitectural TLBs . . . 79
8.3.2 Model MMU . . . 80
8.3.3 Invalid entries . . . 81
8.4 Reads not from TLB . . . 82
8.4.1 Out-of-order execution . . . 82
8.4.2 Enforcing thread-local ordering . . . 84
8.4.3 Enhanced Translation Synchronization . . . 90
8.4.4 Forwarding to the translation table walker . . . 91
8.4.5 Speculative execution . . . 93
8.4.6 Single-copy atomicity . . . 94
8.4.7 Multi-copy atomicity . . . 94
8.4.8 Translation-table-walk intra-walk ordering . . . 96
8.4.9 Multiple translations within a single instruction . . . 96
8.5 Caching of translations in TLBs . . . 97
8.5.1 Cached translations . . . 97
8.5.2 TLB fills . . . 98
8.5.3 μTLBs . . . 98
8.5.4 Partial caching of walks . . . 100
8.5.5 Reachability . . . 101
8.6 TLB maintenance . . . 101
8.6.1 Recovering coherence . . . 101
8.6.2 Thread-local ordering and `TLBI` . . . 105
8.6.3 Broadcast . . . 106

109 8.6.4 Virtualization . . . 108
110 8.6.5 Break-before-make . . . 111
111 8.6.6 ASIDs and VMIDs . . . 111
112 8.6.7 Access permissions . . . 113
113 8.7 Context synchronisation . . . 117
114 8.7.1 Relaxed system registers . . . 117
115 8.8 Contributions . . . 118

116 **9 An axiomatic VMSA model** **119**
117 9.1 Extended candidate executions . . . 119
118 9.1.1 Candidate events . . . 119
119 9.1.2 Candidate relations . . . 121
120 9.2 Cat model . . . 122
121 9.3 Axioms . . . 122
122 9.4 Relations . . . 124
123 9.4.1 `obs` . . . 124
124 9.4.2 `dob` . . . 125
125 9.4.3 `bob` . . . 125
126 9.4.4 `tob` . . . 125
127 9.4.5 `ctxob` . . . 126
128 9.4.6 `obfault` and `obETS` . . . 127
129 9.4.7 `obtlbi` . . . 127

130 **10 Validating the VMSA model** **130**
131 10.1 Extending isla-axiomatic . . . 130
132 10.2 Running on hardware: system-litmus-harness . . . 130
133 10.2.1 Harness overview . . . 130
134 10.2.2 Results from hardware . . . 130

135 **11 An operational VMSA model** **131**
136 11.1 Introduction . . . 131
137 11.2 Structure of the state . . . 131
138 11.2.1 The MMU . . . 131
139 11.2.2 Its TLB . . . 131
140 11.3 Virtual memory axioms . . . 131
141 11.4 Break-before-make violation detection . . . 132
142 11.5 A weaker VMSA model . . . 132
143 11.6 Executing the models with Isla . . . 132

144 **12 Limitations** **133**
145 12.1 Exceptions and Interrupts . . . 133
146 12.2 Cross-interaction of instruction fetch and virtual memory . . . 133
147 12.3 Other architectures . . . 133

148 **13 Conclusion** **134**

149 **Glossary** **135**

Chapter 1

Introduction

Over the previous years and decades, much work has gone into writing down what it is the computers we use every day actually do. To define, mathematically and precisely, the *architecture* that the processors in our computers implement: Intel/AMD's x86, Arm's Armv8-A, IBM's Power, RISC-V, and so on.

Architectures can be thought of as abstractions of the underlying hardware. As programming languages whose syntax is defined by the *ISA* (or Instruction Set Architecture), and whose semantics is the composition of the sequential behaviours of the individual instructions and registers from the ISA, with the machine execution model: the thread and storage subsystems. Architecture therefore can be thought of as the *interface* between hardware and software: defining the guarantees hardware must give and that software may rely upon. In theory, this interface is straight-forward to define. One can give precise formal semantics to the individual instructions, as Arm does with its *Architecture Specification Language* (ASL), and then tie instructions together with a fetch-decode-execute loop. In practice, however, modern industrial architectures accumulate great complexity and subtlety. The Armv8-A and Intel reference manuals have 11,500 [1], and 4922 [2] pages respectively. Covering everything from the ISA to the interactions between the ISA and the thread and storage subsystems.

The complexity of these interfaces becomes most apparent with the interaction with *multicore* systems. When multiple processors are executing concurrently, and communicating through shared memory, then various hardware optimisations, which are usually invisible to the programmer outside of timing effects, can become *architecturally visible*, that is they affect the semantics of the machine code. Over the years, these effects have been studied as part of the field of 'relaxed memory' research, resulting in numerous formal models for a variety of microprocessor architectures giving a precise mathematical semantics to the concurrent behaviours of 'userland' machine code programs. We now seek to expand this body of work, to cover not just those parts of the architectures used by userland processes, but the features required by systems software to function.

In this work we will focus on the Armv8-A architecture: the *application*-class processors that we find powering a large proportion of modern mobile devices. There are a few reasons why we shall focus on Arm: (1) they are ubiquitous and millions of people rely on software running on them every day, (2) Arm has a diverse ecosystem of implementations, meaning software must program to this abstract interface much more tightly than one might for other architectures, and (3) Arm have put a large amount of effort into precisely and formally defining their ISA.

Specifically, we will focus on key architectural features required by operating systems and hypervisors, which are not or only partially accessible to userland processes: instruction fetching, cache maintenance, virtual memory and TLB maintenance, and exceptions.

Armv8-A Armv8-A is a modern industrial reduced-instruction-set architecture. Execution of an Armv8-A processor is split into two modes: AArch64 (for 64-bit execution) or AArch32 (for 32-bit execution). AArch64 mode uses the A64 instruction set. AArch32 mode can be using either the T32 or A32 instruction sets. This is illustrated in Figure 1.1.

A64, currently, has 402 'base' instructions and another 1,205 vector, matrix and SIMD instructions. It has 31 general-purpose registers, accessible through 32-bit views as `w0-w30`, or as 64-bit views as `x0-x30`. It has a dedicated zero register (`wzr`/`xzr`), and stack pointer register (`sp`). Instructions are fixed-width and in the typical RISC style, with instructions reading operands from registers, and writing results back to registers, with only limited immediate values. Execution in AArch64 is split into 4 'exception levels', which demark the levels of privilege that a process may have, ranging from EL0 (least privileged) to EL3 (most privileged). Typically userland

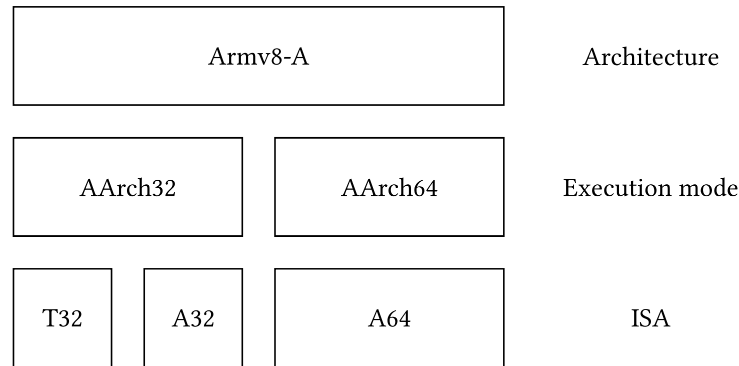


Figure 1.1: Armv8-A structure.

processes execute at EL0, with very limited access to hardware features; with operating systems running at EL1, hypervisors running at EL2, and any firmware and secure monitor running at EL3.

So, each CPU: has its own bank of registers; is executing in either AArch64 or AArch32 execution mode; fetching, decoding and executing instructions from either the A64, A32 or T32 ISAs; at one of EL0,EL1,EL2 or EL3. For this work, we will focus on AArch64 and its A64 ISA, and hypervisors and below (EL2, EL1 and EL0).

Systems software When we use our computers on a daily basis, we are typically interacting with *userland*: unprivileged programs, with restricted access to hardware. These userland programs make up the bulk of the applications we use every day, from spreadsheets, to web browsers, text editors and so on. They typically execute with the least privilege (at EL0), and with the operating systems and hypervisors below them restricting the access to memory they have through the use of *virtual memory*.

Operating systems split userland execution into *processes*: instances of programs, with some associated dedicated virtual memory. It is the operating system, executing with more privilege (at EL1), that configures and schedules these processes.

Thesis overview

1.1 Thesis Introduction

What this thesis is trying to do

1.2 Modern microprocessors and Arm

What/who are Arm and what do they do? how do they relate to others?

1.3 Processor architecture and Armv8-A

Arch vs uarch and why do we care about the A profile.

1.4 Semantics of these architectures

Briefly how do we how and why we write semantics and what can we do with them

1.5 Systems software

What is a systems software and where/why is the semantics of them lacking

215 ## 1.6 Hypervisors

216 Narrow in on hypervisors and their importance and mention pKVM.

Preface to Relaxed Memory

In days gone by, the implementations of our programming languages, the compilers and interpreters, implemented in software and hardware, were straight-forward and direct encodings of the desired semantics. As time progressed these implementations became legacy; acquiring multiple layers of abstraction, with implementations made with ever increasing complexity being built upon other implementations with their own abstractions and complexity. Our compilers and our hardware re-write our programs to be faster, use less space, and be more compact. They propagate and duplicate reads, subsume or outright delete writes, change the order we wrote the operations in, replace one computation with another, or even just eliminate whole parts of the program entirely.

It is, perhaps, believed that such optimisations are *semantics preserving*, that they, aside from the timing effects they are designed to cause, are invisible to the programmer. This is, alas, untrue. Many of these optimisations are highly desireable, yet, seem fundamentally incompatible with concurrency. Our multithreaded programs, and our multicore processors, make these incompatibilities impossible to hide.

Take, as an example, Intel's x86 microprocessor architecture. Of which, we are all familiar. It, quite sensibly, wants to allow its implementations to perform a common optimisation: to batch many smaller writes together. This *store buffering* optimisation is ubiquitous in the hardware world, it is, however, not semantics preserving. Multiple threads of execution, running on multiple cores, may have mutually inconsistent views of memory; where, at the same point in time, different cores have differing opinions on the value at a particular memory address. This disagreement poisons the program, as, if the programmer reads from that memory location, they shall get different answers on different cores. This can break key invariants of our software, leading to critical bugs in our synchronisation primitives, our data structures and our software more generally, if the programmer was unaware of these behaviours and their mitigations.

Intel is not alone, and store buffering not the only behaviour our hardware exhibits. Arm, RISC-V and IBM's Power architectures all exhibit their own behaviours, with their own mitigations. Each microprocessor architecture comes with a reference manual, comprised of thousands, or tens of thousands, of pages of a mix of prose and pseudocode, attempting to describe these, and other, behaviours. We find that these architectures are incomparable, they have different sets of allowable behaviours which break the sequential consistency fantasy of the world. Reordering of instructions, prefetching and caching of data, buffering of writes and loads, hierarchical cache layouts, branch prediction, and speculation, are, on some of those architectures, but not others, allowed to become visible to the programmer. We find, that it is not that some implementations perform these optimisations while others do not, but that architectures do not always require implementations hide their consequences from the programmer: instead allowing the hardware to be more loose with hazard checking and cache invalidation and so on, where the performance gains are considered a greater benefit than the semantic loss.

It is not just our hardware that has these concerns. A variety of software languages, including C, C++, Java, Rust, and Haskell, are all known to have comparable behaviours, derived both from similar optimisations done by their compilers and interpreters, but also inherited from the hardware they run upon.

It is, therefore, imperative, that we, as a community, endeavour to understand the what, why, when and where of these behaviours, to precisely write down how they affect our programming languages and build tools and techniques to help identify, explore, test, check and verify that our programs are correct. This is, in a nutshell, the field of relaxed memory.

Chapter 2

Modelling Armv8-A: background

Now we turn our attention to the current, industry standard, methods of precisely and formally modelling the relaxed memory behaviours. There are principly three ways this is done: through operational models that mimic the mechanisms we see on hardware, with axiomatic-style models which filter out whole-program executions based on some predicate, and with the promising model.

We shall see that the idea of *litmus testing* is central. Litmus tests provide a way of succinctly, and efficiently, describing, and enumerating, the behaviours the various models should allow. We will start by looking at litmus testing in general, and some specific litmus tests of interest to the Armv8-A models, before looking at the models in detail.

2.1 Litmus tests

The foundation of much of the relaxed memory work has been focused on *litmus tests*, small, self-contained, executable, snippets of code. They each capture a simple pattern or shape one may find in software.

Take the classic MP ("Message passing") litmus test, as an example. The code listing for the Armv8-A (AArch64) variant can be found in Figure 2.1. The 'MP' portion of the name captures the *shape*. The MP shape implies a two-threaded test with two locations with one thread (usually written first) writing to the locations, and another thread reading them in the converse order. The second half of the name ('+pos') designates the variation on the shape. Typically the variations are defined as the sequence of orderings between events (separated by -) for each thread (separated by +). In this case, it is the variation with just program order (po) between each event, on both threads.

MP+pos AArch64

Initial state: `0:X1=x, 0:X3=y, 1:X1=y, 1:X3=x, *x=0, *y=0`	
Thread 0	**Thread 1**

`MOV X0,#1`
`STR X0,[X1]`
`MOV X2,#1`
`STR X2,[X3]`
	`LDR X0,[X1]`
	`LDR X2,[X3]`

Allowed: `1:X0=1, 1:X2=0`	

Figure 2.1: MP test code listing.

The code listing given is totally standard. The top line contains the name of the litmus test (MP+pos) and architecture this variant is for (AArch64), the second section contains the initial register and memory state, the next section contains the literal code listing for each thread, with the final state at the bottom being the interesting outcome we wish to explore.

On Arm, this outcome is allowed. We can imagine that there may be many executions of the listed code, where the instructions of the two threads are interleaved in different ways. To see the highlighted outcome, with Thread 1 reading 1 for y but 0 for x, there is only one possible combination of reads: that the read of y reads from the write

to `y`, and the read of `x` reads from the initial memory state. This execution can be represented as a graph of the key events (reads and writes) of the program, and their implicit orderings. The execution graph that corresponds with the allowed outcome can be found in Figure 2.2.

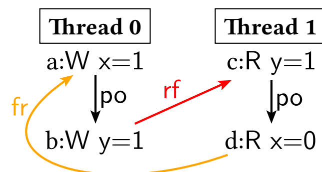


Figure 2.2: MP test execution diagram.

The nodes on the left, under the Thread 0 label, correspond to events from executing Thread 0 of the program, where the '`a: W x = 1`' event (that is, event labelled `a` is a write to `x` of the value `1`), corresponds to the propagation of the first store in Thread 0 to memory, and event `b` corresponds to the second store being propagated. They are related by the `po` edge saying that event `a`'s instruction occured before event `b`'s instruction in the program. Similarly under Thread 1 we see the event '`c: R y = 1`' for the first load reading `y` and seeing the value `1`. This value was read from the write event `b`, and so the event `b` is related to the read event `c` with the reads-from edge `rf`. Finally, the load of `x` reads from the initial value in memory, so we have another read event, labelled `d`, which reads 0. The read of `x` must be ordered before the write of `x`, so the read and write are related by `fr` (from reads).

On Arm, the writes and reads need not execute in the order they appear in the program. So, while this execution appears to have a cyclic dependency in the order events must have happened in, the cycle can be broken by re-ordering the execution of the reads or writes. The execution is, therefore, allowed, and we observe this outcome on hardware.

2.1.1 The litmus zoo

We use litmus tests to, generally, capture some *behaviour*: one particular pattern in code, or a specific hardware mechanism that is responsible for allowing or forbidding the test. Many litmus tests may exercise many microarchitectural mechanisms whose confluence leads to the final result, or where there may be multiple different mechanisms that could independently lead to the final result. For example, in the MP+pos test we just saw, there are two well-understood microarchitectual explanations: that the stores propagate out-of-order, or that the loads satisfy out-of-order. Either explanation is sufficient, and one needs to prevent both to forbid the outcome.

Previous work has enumerated these various patterns to produce a large collection of litmus tests, for a range of architectures, each with an assortment of variations for different intra-thread orderings (for barriers, dependencies and so on). We will not do an exhausitve review of all the behaviours that are allowed and forbidden in Arm, instead refer to test7 **TODO: ?CITE?**, Pulte et al **TODO: ?CITE?**, Flur **TODO: ?CITE?**and Alglave et al **TODO: ?CITE?**, but we will briefly look at some of the behaviours that the reader should understand before progressing to future chapters. Namely, coherence, barriers and dependencies, and multi-copy atomicity.

Thread-local ordering On Arm, instructions need not execute in the order they appear in the program. Reads and writes are free to be re-ordered with respect to each other, with few restrictions. This is in contrast to other architectures such as Intel's x86, where only writes can be re-ordered with respect to program-order later reads (through store buffering).

Not all re-orderings are permissible; Arm require that single-threaded programs should behave as if executed sequentially. This means that non-SC executions only come about through the interaction between multiple threads. We have already seen this with the MP test mentioned earlier. To forbid the outcome of that test, we can add barriers or dependencies to enforce thread-local ordering preventing the events from being reordered. Two (forbidden) variations of MP can be found in Figure 2.3.

Control dependencies, however, do not necessarily enforce order on Arm. Speculation allows reads to happen 'early', but not writes, giving us the asymmetry in the outcomes in Figure 2.4.

Coherence A fundamental guarantee provided by most modern microprossor architectures is *coherence*: that there is a total order that writes happen in for each location that all threads agree on. Microarchitecturally this means that the storage subsystem (with all its caches) must remember which write is the 'most recent' (coherence latest) write and reads should always read from that.

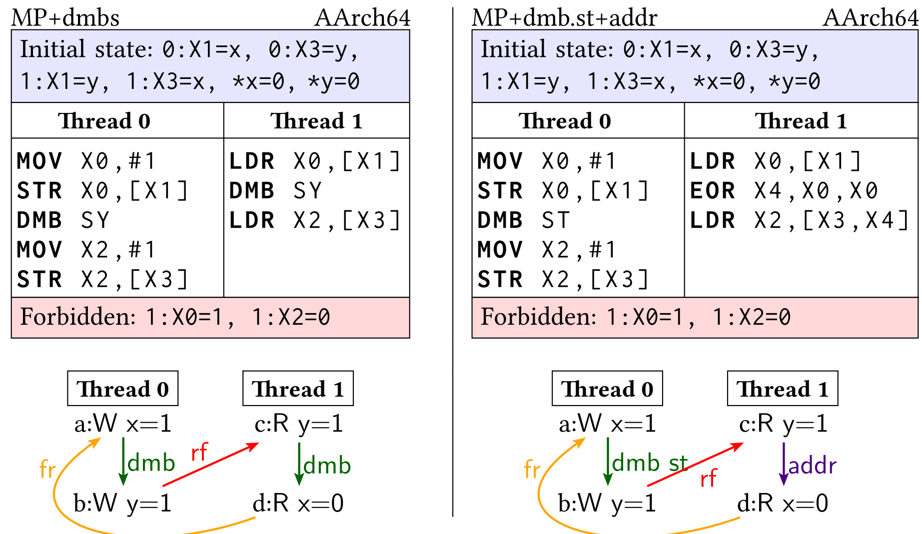


Figure 2.3: Two variants of MP with thread-local ordering.
On the left: MP+dmbs with Arm DMB barrier between instructions.
On the right: MP+dmb.st+addr with an address dependency between the reads.

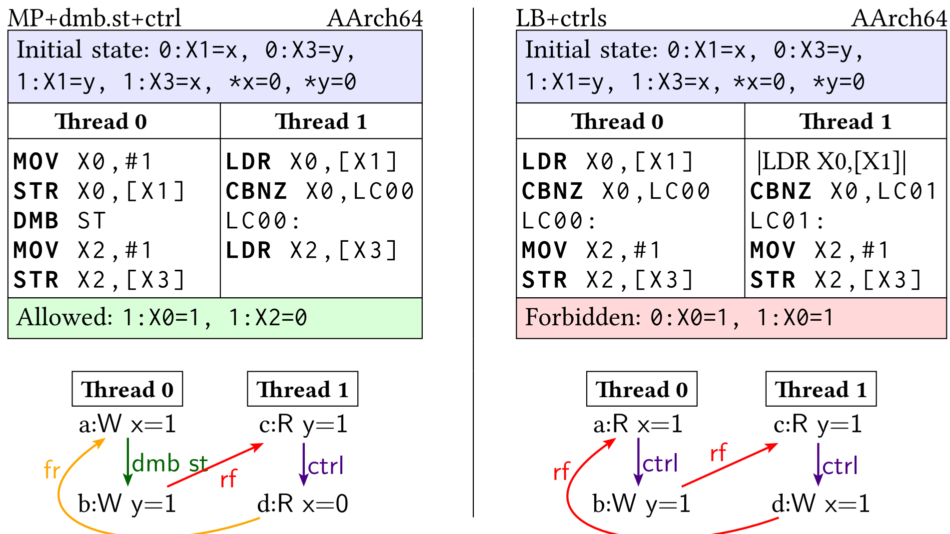


Figure 2.4: Two litmus tests with speculation.
On the left: MP+dmb.st+ctrl with Arm DMB barrier between the writes, but a control dependency between the reads.
On the right: LB+ctrls, a variant of the classic ‘load buffering’ litmus test, with control dependencies to both writes.

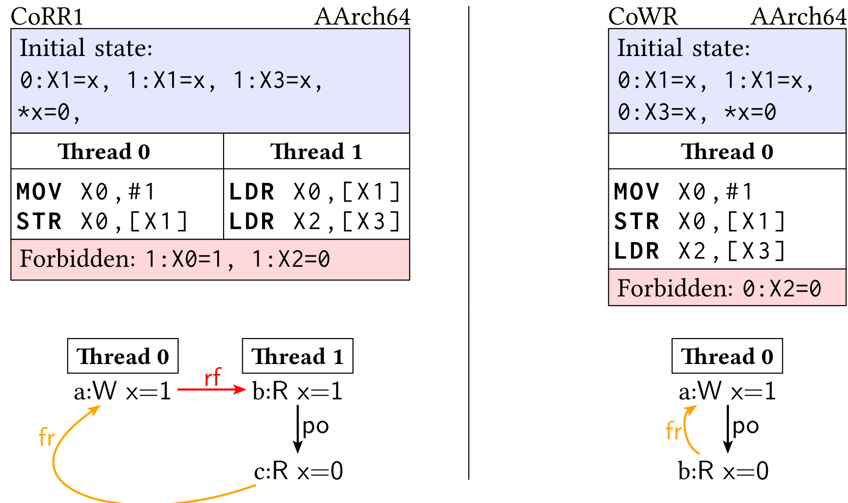


Figure 2.5: Two coherence litmus tests.
On the left: CoRR1, that two subsequent reads of the same location in the same thread should be consistent with the coherence order. On the right: CoWR, that a read of a location cannot skip over a newer program-order earlier write from the same thread.

The key litmus tests for coherence can be found in Figure 2.5.

Multi-copy atomicity and write-forwarding When combining more than two threads, coherence alone does not ensure that writes are propagated to all threads simultaneously. This behaviour is known as *multi-copy atomicity*.

Arm has a kind of partial multi-copy atomicity, known as *other*-multi-copy atomicity, where writes can be observed by writer thread earlier than they can be seen by other threads, but once a write has propagated to another thread then all threads must see that write. Figure 2.6 shows the former, that writes can be observed locally (through *write forwarding*) before being propagated to other threads, even down speculative branches, and Figure 2.7 shows the classic independent-reads independent-writes (IRIW) litmus test, which demonstrates the latter point, that writes propagate to all threads simulatenously.

2.2 Operational models

Microarchitectural style and how they aren't "how the hardware works" But don't actually explain flat ...

2.3 Promising

TODO: talk about after axiomatic?

MP+dmb.st+addr-rfi-addr	AArch64
Initial state: `0:X1=x, 0:X3=y, 1:X1=y, 1:X3=x, 1:X3=z, 1:X5=z, *x=0, *y=0, *z=0`	
Thread 0	**Thread 1**

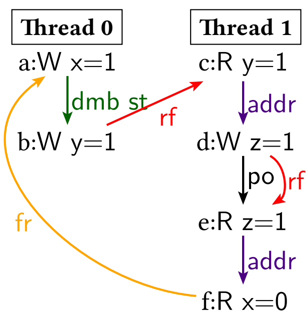
`MOV X0,#1`
`STR X0,[X1]`
`DMB ST`
`MOV X2,#1`
`STR X2,[X3]`
	`LDR X0,[X1]`
	`EOR X8,X0,X0`
	`MOV X2,#1`
	`STR X2,[X3,X8]`
	`LDR X4,[X5]`
	`EOR X9,X4,X4`
	`LDR X6,[X7,X9]`

Allowed: `1:X0=1, 1:X4=1, 1:X6=0`	

PPOCA	AArch64
Initial state: `0:X1=x, 0:X3=y, 1:X1=y, 1:X3=z, 1:X5=z 1:X8=x, *x=0, *y=0`	
Thread 0	**Thread 1**

`MOV X0,#1`
`STR X0,[X1]`
`MOV X2,#1`
`STR X2,[X3]`
	`LDR X0,[X1]`
	`CBNZ X0,LC00`
	`LC00:`
	`MOV X2,#1`
	`STR X2,[X3]`
	`LDR X4,[X5]`
	`EOR X6,X4,X4`
	`LDR X7,[X8]`

Allowed: `1:X0=1, 1:X4=1 1:X7=0`	



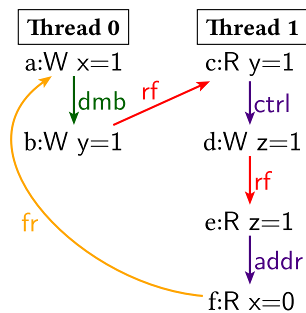


Figure 2.6: Two litmus tests with write forwarding.
On the left: MP+dmb.st+addr-rfi-addr with write-forwarding down a non-speculative branch.
On the right: PPOCA, with write-forwarding down a speculative branch.

IRIW+dmbs			AArch64
Initial state: `0:X1=x, 1:X1=x, 1:X3=y, 2:X1=y, 3:X1=y, 3:X3=x, *x=0, *y=0`			
Thread 0	**Thread 1**	**Thread 2**	**Thread 3**

`MOV X0,#1`
`STR X0,[X1]`
	`LDR X0,[X1]`
	`MOV X2,#1`
	`DMB SY`
	`LDR X2,[X3]`
		`MOV X0,#1`
		`STR X0,[X1]`
			`LDR X0,[X1]`
			`MOV X2,#1`
			`DMB SY`
			`LDR X2,[X3]`

Forbidden: `1:X0=1, 1:X2=0, 3:X0=1, 3:X2=0`			

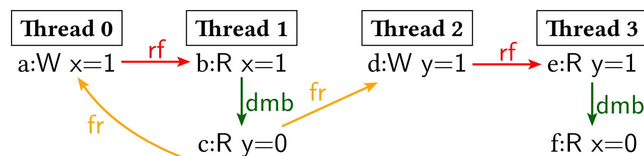


Figure 2.7: IRIW+dmbs: a classic multi-copy atomicity litmus test.

340 ## 2.4 Axiomatic-style models

341 A newer approach, devised by Alglave et al **TODO: ?CITE?**, describes the allowed behaviour with a set of *axioms*
342 constraining *candidate* executions.

343 The candidates are whole-program executions, consistent with the intra-instruction semantics, but with no
344 constraints on the values of reads or writes. The axioms then discard some of these executions as inconsistent,
345 if they violate a semantic property the architecture enforces. Then, if for a given program and final state, there
346 are any candidate executions, which gives rise to the aforementioned final state, which are consistent with the
347 axioms of the model, then the model is said to *allow* that execution.

348 ### 2.4.1 Informal description and the Cat language

349 We can think of a candidate execution as a graph of the events of the program, with some intrinsic relations
350 capturing the sequencing of those events.

351 Revisiting the MP example we saw earlier, Figure 2.2 illustrates a typical candidate execution.

352 ### 2.4.2 The Armv8-A axiomatic model

353 We define the Armv8-A axiomatic-style model in two parts: (1) the generation of candidates given a program, and
354 (2) the relations and axioms.

355 **Candidate generation** Candidates are generated using the intra-instruction semantics, defined by Arm's ASL
356 pseudocode. Formally, we assign a denotation function $[\![-]\!]$ which relates an Arm machine code program with a
357 set of candidate executions. We do not, here in this work, try to give a precise definition of this function. Instead,
358 we delegate to the Arm pseudocode for the intra-instruction semantics, and to software to compose instructions
359 together, work out dependencies, and glue multiple threads together, into a whole program candidate execution.
360 **TODO: Why not do that here? I could probably just write down.**

361 $[\![-]\!] \ : \ \mathcal{P}(\mathtt{Candidate})$
362 $\mathtt{Candidate} \ : \ \mathcal{P}(\mathtt{N}) \times \mathtt{N} \rightarrow \mathtt{Event} \times \mathcal{C}_R$
363 $\mathcal{C}_R \ : \ \langle \mathbf{po}, \mathbf{loc}, \mathbf{addr}, \mathbf{ctrl}, \mathbf{data}, \mathbf{rmw}, \mathbf{ext} \ : \ \mathtt{N} \leftrightarrow \mathtt{N} \rangle$

364 **TODO: preexecution v candidate**

365 A candidate execution is made up of a set of event names, a labelling function, and a collection of candidate
366 relations, where the event labels are defined as follows:

367 $\mathtt{Event} \equiv \mathtt{Reads} \cup \mathtt{Writes} \cup \mathtt{Barriers}$
368 $\mathtt{Reads} \equiv \{\mathbf{R}, \mathbf{A}, \mathbf{Q}\} \times \mathtt{Loc} \times \mathtt{Val}$
369 $\mathtt{Writes} \equiv \{\mathbf{W}, \mathbf{L}\} \times \mathtt{Loc} \times \mathtt{Val}$
370 $\mathtt{Barriers} \equiv \{\mathbf{DMB.LD}, \mathbf{DMB.ST}, \mathbf{DMB.SY}, \mathbf{ISB}\}$
371 $\mathtt{Loc} \equiv \mathbf{BV}_{48}$
372 $\mathtt{Val} \equiv \mathbf{BV}_{64}$

373 The candidate relations are:

374 - ▷ program order: E_1 po E_2 iff the instruction for E_1 occurs before E_2 in the source program.
375 - ▷ same-location: M_1 loc M_2 iff the address of M_1 is the same location as used by M_2.
376 - ▷ address dependent: R_1 addr RW_2 iff the value read by R_1 is used in the calculation of the address RW_2.
377 - ▷ data dependent: R_1 data W_2 iff the value read by R_1 is used in the calculation of the value to write in W_2.
378 - ▷ control dependent: R_1 ctrl RW_2 iff the value read by R_1 is used to determine whether or not the instruction
379 RW_2 originates from would have executed at all.
380 - ▷ read-modify-write: R_1 rmw W_2 for the separate read and write events of an atomic update.
381 - ▷ external: E_1 ext E_2 iff E_1 and E_2 originate from different threads.

```
(* observed by *)
let obs = rfe | fr | co
(* dependency-ordered-before *)
let dob =
    addr | data
  | (ctrl | (addr ; po)) ; [W]
  | (addr | data); rfi
(* atomic-ordered-before *)
let aob = rmw
  | [range(rmw)]; rfi; [A | Q]
(* barrier-ordered-before *)
let bob = [R] ; po ; [dmbld]
  | [W] ; po ; [dmbst]
  | [dmbst]; po; [W]
  | [dmbld]; po; [R|W]
  | [L]; po; [A]
  | [A | Q]; po; [R | W]
  | [R | W]; po; [L]
(* Ordered-before *)
let ob1 = obs | dob | aob | bob
let ob = ob1^+
(* Internal visibility requirement *)
acyclic po-loc | fr | co | rf as internal
(* External visibility requirement *)
irreflexive ob as external
(* Atomic: Basic LDXR/STXR constraint to
   forbid intervening writes. *)
empty rmw & (fre; coe) as atomic
```

Figure 2.8: Deacon et al's Armv8-A MCA Axiomatic model.

382 **Axioms** The axiomatic model is then defined by its *axioms*. An axiom in the model is an assertion of the
383 acyclicity of a relation over L. These relations are constructed using a relation algebra $\mathcal{A}$: composing the relations
384 of $\mathcal{C}_R$ and the existentially quantified relations **co** (coherence-order) and **rf** (reads-from) and the restricted identity
385 relation ($\mathbf{id}_E$, for identity over events with label E), with the standard relation operators: union (|), intersection
386 (&), relation composition (using the flipped operator ;), transitive closure (*) and relation inverse ($^{-1}$). The model
387 is then, formally, a set of terms over $\mathcal{A}$.

388 $\mathcal{A}$: $(\mathcal{C}_R \cup \langle \mathbf{co}, \mathbf{rf}, \mathbf{id}_E \rangle,\ \langle \&, |, ;, +, *, {}^{-1} \rangle)$
389 Model $\subseteq \mathcal{T}(\mathcal{A})$

390 Such models are usually defined in the Cat language, and the canonical Cat Armv8-A multi-copy atomic axiomatic
391 model is given by Deacon et al **TODO: ?CITE?**and can be found in Figure 2.8.

392

Chapter 3

Instruction fetch: an introduction

3.1 Introduction

Computing relies on the *architectural abstraction*: the specification of an envelope of allowed hardware behaviour that hardware implementations should lie within, and that software should assume. These interfaces, defined by hardware vendors and relatively stable over time, notionally decouple hardware and software development; they are also, in principle, the foundation for software verification. In practice, however, industrial architectures have accumulated great complexity and subtlety: the ARMv8-A and Intel architecture reference manuals are now 7476 and 4922 pages [3, 2], and hardware optimisations, including out-of-order and speculative execution, result in surprising and poorly-understood programmer-observable behaviour. Architecture specifications have historically also been entirely informal, describing these complex envelopes of allowed behaviour solely in prose and pseudocode. This is problematic in many ways: do not serve as clear documentation, with the inevitable ambiguity and incompleteness of informal prose leaving major questions unanswered; without a specification that is executable as a test oracle (that can decide whether some observed behaviour is allowed or not), hardware validation relies on test suites that must be manually curated; without an architecturally-complete emulator (that can exhibit all allowed behaviour), it is very hard for software developers to "program to the specification" – they rely on test-and-debug development, and can only test above the hardware implementation(s) they have; and without a mathematically rigorous semantics, formal verification of hardware or software is impossible.

Over the last 10 years, much has been done to put architecture specifications on a more rigorous footing, so that a single specification can serve all those purposes. There are three main problems, two of which are now largely solved.

The first is the instruction-set architecture (ISA): the specification of the sequential behaviour of individual instructions. This is chiefly a problem of scale: modern industrial architectures such as Arm or x86 have large instruction sets, and each instruction involves many details, including its behaviour at different privilege levels, virtual-to-physical address translation, and so on – a single Arm instruction might involve hundreds of auxiliary functions. Recent work by Reid et al. within Arm [4, 5, 6] transitioned their internal ISA description into a mechanised form, used both for documentation and testing, and with him we automatically translated this into publicly available Sail definitions and thence into theorem-prover definitions [7, 8]. Other related work is in §3.7.

The second is the relaxed-memory concurrent behaviour of "user-mode" operations: memory writes and reads, and the mechanisms that architectures provide to enforce ordering and atomicity (dependencies, memory barriers, load-linked/store-conditional operations, etc.). In 2008, for ARMv7, IBM POWER, and x86, this was poorly understood, and the architects regarded even their own prose specifications as inscrutable. Now, following extensive work by many people [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25], ARMv8-A has a well-defined and simplified model as part of its specification [3, B2.3], including a prose transcription of a mathematical model [26], and an equivalence proof between operational and axiomatic presentations [9, 10]; RISC-V has adopted a similar model [27]; and IBM POWER and x86 have well-established de-facto-standard models. All of these are experimentally validated against hardware, and supported by tools for exhaustively running tests [28, 29]. The combination of these models and the ISA semantics above is enough to let one reason about or model-check concurrent algorithms.

That leaves the third part of the problem: the "system" semantics, of instruction-fetch and cache maintenance, exceptions and interrupts, and address translation and TLB (translation lookaside buffer) maintenance. Just as

434 for "user-mode" relaxed memory, these are all areas where microarchitectural optimisations can have surprising
435 programmer-visible effects, especially in the concurrent context. The mechanisms are relied on by all code, but they
436 are explicitly managed only by systems code, in just-in-time (JIT) compilers, dynamic loaders, operating-system
437 (OS) kernels, and hypervisors. This is, of course, exactly the security-critical computing base, currently trusted but
438 not trustworthy, that is especially in need of verification – which requires a precise and well-validated definition
439 of the architectural abstraction. Previous work has scarcely touched on this: none of seL4 [30], CertiKOS [31, 32],
440 Komodo [33], or [34, 35], address realistic architecture concurrency, and they use (at best) idealised models of the
441 sequential systems architecture. The CakeML [36, 37] and CompCert [38] verified compilers target only sequential
442 user-mode ISA fragments.

443 In this paper we focus on one aspect of system semantics: instruction fetch and cache maintenance, for ARMv8-A.
444 The ability to execute code that has previously been written to data memory is fundamental to computing:
445 fine-grained self-modifying code is now rare, and (rightly) deprecated, but program loading, dynamic linking, JIT
446 compilation, debugging, and OS configuration all rely on executing code from data writes. However, because
447 these are relatively infrequent operations, hardware designers have been able to optimise by partially separating
448 the instruction and data paths, e.g. with distinct instruction caching, which by default may not be coherent with
449 data accesses. This can introduce programmer-visible behaviour analogous to that of user-mode relaxed-memory
450 concurrency, and require specific additional synchronisation to correctly pick up code modifications. Exactly what
451 these are is not entirely clear in the current ARMv8-A architecture text, just as pre-2018 user-mode concurrency
452 was not.

453 Our main contribution is to clarify this situation, developing precise abstractions that bring the instruction-
454 fetch part of ARMv8-A system behaviour into the domain of rigorous semantics. Arm have stated [private
455 communication] that they intend to incorporate a version of this into their architecture. We aim thereby to enable
456 future work on system software verification using the techniques of programming languages research: program
457 analysis, model-checking, program logics, etc. We begin (§3.2) by recalling the informal architectural guarantees
458 that Arm provide, and the ways in which real-world software systems such as Linux, JavaScript, and WebAssembly
459 change instruction memory. Then:

460 (1) **We explore the fundamental phenomena and architecture design questions with a series of exam-**
461 **ples** (§3.3). We explore the interactions between instruction fetching, cache maintenance and the 'usual' relaxed
462 memory stores and loads, showing that instruction fetches are more relaxed, and how even fundamental coher-
463 ence guarantees for data memory do not apply to instruction fetches. Most of these questions arose during the
464 development of our models, in detailed ongoing discussion with the Arm Chief Architect and other Arm staff.
465 They include questions of several different kinds. Six are clear from the Arm prose specification. Of the others:
466 two are not implied by the prose but are natural choices; five involved substantive new choices by Arm that had
467 not previously been considered and/or documented; for two, either choice could be reasonable, and Arm chose
468 the simpler (and weaker) option; and for one, Arm were independently already strengthening the architecture to
469 accommodate existing software.

470 (2) **We give an operational semantics for Arm instruction fetch and icache maintenance** (§3.4). This is
471 in an abstract-microarchitectural style that supports an operational intuition for how hardware actually works,
472 while abstracting from the mass of detail and the microarchitectural variation of actual hardware implementations.
473 We do so by extending the Flat model [10] with simple abstractions of instruction caches and the coherent data
474 cache network, in a way that captures the architectural intent, defining the entire envelope of behaviours that
475 implementations should be allowed to exhibit.

476 (3) **We give a more concise presentation of the model in an axiomatic style** (§3.5), extending the "user-
477 mode" axiomatic model from previous work [10, 9, 26, 3], and intended to be functionally equivalent. We discuss
478 how this too matches the architectural intent.

479 (4) **We validate all this** in two ways: by the extensive discussion with Arm staff mentioned above, and by
480 experimental testing of hardware behaviour, on a selection of ARMv8-A cores designed by multiple vendors
481 (§3.6). We run tests on hardware with a mild extension of the Litmus tool [39, 17]. We make the operational
482 model executable as a test oracle by integrating it into the RMEM tool and its web interface [28], introducing
483 optimisations that make it possible to exhaustively execute the examples. We make the axiomatic model executable
484 as a test oracle with a new tool that takes litmus tests and uses a Sail [7] definition of a fragment of the ARMv8-A
485 ISA to generate SMT problems for the model. We then compare hardware and the two models for the handwritten
486 tests (modulo two tests not supported by the axiomatic checker), compare hardware and the operational model on
487 a suite of 1456 tests, automatically generated with an extension of the diy tool [40], and check the operational and
488 axiomatic models against sets of previous non-ifetch tests. In all this data our models are equivalent to each other

and consistent with hardware observations, except for one case where our testing uncovered a hardware bug on a Qualcomm device.

Finally, we discuss other related work (§3.7) and conclude (§3.8). We do all this for ARMv8-A, but other relaxed architectures, e.g. IBM POWER and RISC-V, face similar issues; our tests and tooling should enable corresponding work there.

The models are too large to include or explain in full here, so we focus on explaining the motivating examples, the main intuition and style of the operational model, in a prose rendering of its executable mathematics, and the definition of the axiomatic model. Appendices provide additional examples, a complete prose description of the operational model, and additional explanation of the axiomatic model. The complete executable mathematics version, the web-interface tool for running it, and our test results are at https://www.cl.cam.ac.uk/~pes20/iflat/.

Caveats and Limitations Our executable models are integrated with a substantial fragment of the Sail ARMv8-A ISA (similar to that used for CakeML), but not yet with the full ISA model [7, 4, 5, 6]; this is just a matter of additional engineering. We only handle the 64-bit AArch64 part of ARMv8-A, not AArch32. We do not handle the interaction between instruction fetch and mixed-size accesses, or other variants of the cache maintenance instructions, e.g. those used for interaction with DMA engines, and variants by set or way instead of by virtual address. Finally, the equivalence between our operational and axiomatic models is validated experimentally. A proof of this equivalence is essential in the long term, but would be a major work in itself: the complexity makes mechanisation essential, but the operational model (in all its scale and complexity) has not yet been subject to mechanised proof. Without instruction fetch, a non-mechanised proof was the main result of an entire PhD thesis [9], and we expect the addition of instruction fetch to require global changes to the argument.

3.2 Industry Practice and the Existing ARMv8-A Prose

Computer architecture relies on a host of sophisticated techniques, including buffering, caching, prediction, and pipelining, for performance. For the normal memory reads and writes of "user-mode" concurrency, the programmer-visible relaxed-memory effects largely arise from store buffering and from out-of-order and speculative pipeline behaviour, not from the cache hierarchy (though some IBM POWER phenomena do arise from the interconnect, and from late processing of cache invalidates). All major architectures provide a strong per-location guarantee of *coherence*: for each memory location, different threads cannot observe the writes to that location in different orders. This is implemented in hardware by coherent cache protocols, ensuring (roughly) that each cache line is writable by at most one hardware thread at a time, and by additional machinery restricting store buffer and pipeline behaviour. Then each architecture provides additional synchronisation mechanisms to let the programmer enforce ordering properties involving multiple locations.

At first sight, one might expect instruction fetches to act like other memory reads but, because writes to instruction memory are relatively rare, hardware designers have adopted different caching mechanisms. The Arm architecture carefully does not mandate exactly what these must be, to allow a wide range of possible hardware implementations, but, for example, a high-performance Arm processor might have per-core separate L1 instruction and data caches, above a unified per-core L2 cache and an L3 cache shared between cores. There may also be additional structures, e.g. per-core fetch queues, and caching of decoded micro-operations. This instruction caching is not necessarily coherent with data memory accesses: *"the architecture does not require the hardware to ensure coherency between instruction caches and memory"* [3, B2.4.4 (B2-114)]; instead, programmers must use explicit cache maintenance instructions. The documentation gives a particular sequence of these: *"If software requires coherency between instruction execution and memory, it must manage this coherency using Context synchronization events and cache maintenance instructions. The following code sequence can be used to allow a processing element (PE) to execute code that the same PE has written."*

```
; Coherency example for data and instruction accesses [...]
; Enter this code with <Wt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Xn.
STR Wt, [Xn]; Store new instruction
DC CVAU, Xn ; Clean data cache by virtual address (VA) to PoU
DSB ISH     ; Ensure visibility of the data cleaned from cache
IC IVAU, Xn ; Invalidate instruction cache by VA to PoU
DSB ISH     ; Ensure completion of the invalidations
ISB         ; Synchronize the fetched instruction stream
```

At first sight, this may be entirely mysterious. The remainder of the paper establishes precise semantics for each instruction, explaining why each is required, but as a rough intuition:

1. The `DC CVAU,Xn` cleans this core's data cache for address `Xn`, pushing the new write far enough down the hierarchy for an instruction fetch that misses in the instruction cache to be guaranteed to see the new value. This point is the *Point of Unification* (PoU) and is usually the point where the instruction and data caches become unified (L2 for most modern devices).
2. The `DSB ISH` waits for the clean to have happened before letting the later instructions execute (without this, the sequence itself can execute out-of-order, and the clean might not have pushed the write down far enough before the instruction cache is updated). The `ISH` makes this specific to the *Inner Shareable Domain*: the processor itself, not the system-on-chip. We do not model shareability domains in this paper, so this is equivalent to a `DSB SY`.
3. The `IC IVAU,Xn` invalidates any entry for that address in the instruction caches for all cores, forcing any future fetch to miss in the instruction cache, and instead read the new value from the data memory hierarchy; it also touches some fetch queue machinery.
4. The second `DSB ISH` ensures the invalidation completes.
5. The final `ISB` flushes this core's pipeline, forcing a re-fetch of all program-order-later instructions.

Some hardware implementations provide extra guarantees, rendering the `DC` or `IC` instructions unnecessary. Arm allow software to discover this in an architectural way, by reading the `CTR_EL0` register's `DIC` and `IDC` bits. Our modelling handles this, but for brevity we only discuss the weakest case, with `CTR_EL0.DIC=CTR_EL0.IDC=0`, that requires full cache maintenance.

Arm make clear that instructions can be prefetched (perhaps speculatively): *"How far ahead of the current point of execution instructions are fetched from is IMPLEMENTATION DEFINED. Such prefetching can be either a fixed or a dynamically varying number of instructions, and can follow any or all possible future execution paths. For all types of memory, the PE might have fetched the instructions from memory at any time since the last Context synchronization event on that PE."*

Concurrent modification and instruction fetch require the same sequence, with an `ISB` on each thread that executes the new instructions, and the rest of the sequence on the modifying thread [3, B2.2.5 (B2-94)]. Concurrent modification without synchronisation is restricted to particular instructions (`B` (branch), `BL` (branch-and-link), `BRK` (break), `SMC`, `HVC`, `SVC` (secure monitor, hypervisor, and supervisor calls), `ISB`, and `NOP`), otherwise there could be *constrained unpredictable behaviour*: *"any behavior that can be achieved by executing any sequence of instructions that can be executed from the same Exception level"*. Concurrent modification of conditional branches is allowed but can result in the old condition with the new target address or vice versa.

All this gives some guidance for programmers, but it leaves the exact semantics of instruction fetch and those cache maintenance instructions unclear, and in practice software typically does not use the above sequence verbatim. For example, it may synchronise a range of addresses at once, looping the `DC` and `IC` parts, or the final `ISB` may be subsumed by instruction synchronisation from exception entry or return. Linux has many places where it modifies code at runtime: in boot-time patching of *alternatives*, modifying kernel code to specialise it to the particular hardware being run on; when the kernel loads code (e.g. when the user calls `dl_open`); and in the `ptrace` system call, used e.g. by the GDB debugger to patch arbitrary instructions with breakpoints at runtime. In Google's *Chrome* web browser, its WebAssembly and JavaScript just-in-time (JIT) compilers are required to both write new code during execution and modify existing code at runtime. In JavaScript, this modification happens inside a single thread and so is quite straightforward. The WebAssembly case is more complex, as one thread is modifying the code of another. A software thread can also be moved (by the OS or hypervisor) from one hardware thread to another, perhaps while it is in the middle of some instruction cache maintenance. Moreover, for security reasoning, we have to be able to bound the possible behaviour of arbitrary code.

All this means that we cannot treat the above sequence as a whole, as an opaque black box. Instead, we need a precise semantics for each individual instruction, but the existing prose documentation does not provide that.

The problem we face is to give such a semantics, that correctly defines behaviour in arbitrary concurrent contexts, that captures the Arm architectural intent, that is strong enough for software, and that abstracts from the variety of hardware implementations (e.g. with differing cache structures) that the architecture intends to allow – but which programmers should not have to think about.

[3, B2.4.4 (B2-114)] *Synchronization and coherency issues between data and instruction accesses*

How far ahead of the current point of execution instructions are fetched from is IMPLEMENTATION DEFINED. Such prefetching can be either a fixed or a dynamically varying number of instructions, and can follow any or all possible future execution paths. For all types of memory:

- ▷ The PE might have fetched the instructions from memory at any time since the last Context synchronization event on that PE.
- ▷ Any instructions fetched in this way might be executed multiple times, if this is required by the execution of the program, without being refetched from memory. In the absence of a Context synchronization event, there is no limit on the number of times such an instruction might be executed without being refetched from memory.

The Arm architecture does not require the hardware to ensure coherency between instruction caches and memory, even for locations of shared memory.

If software requires coherency between instruction execution and memory, it must manage this coherency using Context synchronization events and cache maintenance instructions. The following code sequence can be used to allow a PE to execute code that the same PE has written.

```
; Coherency example for data and instruction accesses within the same Inner Shareable domain.
; Enter this code with <Wt> containing a new 32-bit instruction,
; to be held in Cacheable space at a location pointed to by Xn.
STR Wt, [Xn]
DC CVAU, Xn ; Clean data cache by VA to point of unification (PoU)
DSB ISH ; Ensure visibility of the data cleaned from cache
IC IVAU, Xn ; Invalidate instruction cache by VA to PoU
DSB ISH ; Ensure completion of the invalidations
ISB ; Synchronize the fetched instruction stream

```

Note:

- ▷ For Non-cacheable or Write-Through accesses, the clean data cache by VA instruction is not required. However, the invalidate instruction cache instruction is required because the ARMv8-A AArch64 architecture allows Non-cacheable accesses to be held in an instruction cache. See Non-cacheable accesses and instruction caches on page D4-2359.
- ▷ This code can be used when the thread of execution modifying the code is the same thread of execution that is executing the code. The Armv8 architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization. See Concurrent modification and execution of instructions on page B2-94.
- ▷ The system software controls whether these cache maintenance instructions are available to the application level by setting `SCTLR_EL1.UCI`.

Note: If this sequence is not executed between writing data to a location and executing the instruction at that location, the lack of coherency between instruction caches and memory means that the instructions that are executed might be the old instruction or the updated instruction, and which is used can arbitrarily vary during execution. It must not be assumed by software, before the synchronization sequence is executed, that when the updated instruction has been seen, the old instruction will not be seen again.

———————

[3, B2.2.5 (B2-94)] *Concurrent modification and execution of instructions*

The Armv8 architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization.

Concurrent modification and execution of instructions can lead to the resulting instruction performing any behavior that can be achieved by executing any sequence of instructions that can be executed from the same Exception level, except where each of the instruction before modification and the instruction after modification is one of a B , BL , BRK , HVC , ISB , NOP , SMC , or SVC instruction.

For the B , BL , BRK , HVC , ISB , NOP , SMC , and SVC instructions the architecture guarantees that, after modification of the instruction, behavior is consistent with execution of either:

- ▷ The instruction originally fetched.
- ▷ A fetch of the modified instruction.

If one thread of execution changes a conditional branch instruction, such as B or BL , to another conditional instruction and the change affects both the condition field and the branch target, execution of the changed instruction by another thread of execution before the change is synchronized can lead to either:

- ▷ The old condition being associated with the new target address.
- ▷ The new condition being associated with the old target address.

These possibilities apply regardless of whether the condition, either before or after the change to the branch instruction, is the always condition.

For all other instructions, to avoid UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior, instruction modifications must be explicitly synchronized before they are executed. The required synchronization is as follows:

1. 1. No PE must be executing an instruction when another PE is modifying that instruction.
2. 2. To ensure that the modified instructions are observable, a PE that is writing the instructions must issue the following sequence of instructions and operations:

   ```
   ; Coherency example for data and instruction accesses within the same Inner Shareable domain.
   ; Enter this code with <Wt> containing a new 32-bit instruction,
   ; to be held in Cacheable space at a location pointed to by Xn.
   STR Wt, [Xn]
   DC CVAU, Xn ; Clean data cache by VA to point of unification (PoU)
   DSB ISH ; Ensure visibility of the data cleaned from cache
   IC IVAU, Xn ; Invalidate instruction cache by VA to PoU
   DSB ISH ; Ensure completion of the invalidations

   ```

   Note:

   - ▷ The DC CVAU operation is not required if the area of memory is either Non-cacheable or Write-Through Cacheable.
   - ▷ If the contents of physical memory differ between the mappings, changing the mapping of VAs to PAs can cause the instructions to be concurrently modified by one PE and executed by another PE. If the modifications affect instructions other than those listed as being acceptable for modification, synchronization must be used to avoid UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior.
3. 3. In a multiprocessor system, the IC IVAU is broadcast to all PEs within the Inner Shareable domain of the PE running this sequence. However, when the modified instructions are observable, each PE that is executing the modified instructions must issue the following instruction to ensure execution of the modified instructions:

   ```
   ISB ; Synchronize fetched instruction stream

   ```

For more information about the required synchronization operation, see Synchronization and coherency issues between data and instruction accesses on page B2-114.

Note: For information about memory accesses caused by instruction fetches, see Ordering relations on page B2-100.

3.3 Instruction Fetch Phenomena and Examples

We now describe the main instruction-fetch phenomena and architecture design questions for ARMv8-A, illustrated by handwritten litmus tests, to guide the following model design.

689 ### 3.3.1 Instruction-Fetch Atomicity

690 The first point, as mentioned in §3.2, is that concurrent modification and fetch is only permitted if the original and
691 modified instructions are in a particular set: various branches, supervisor/hypervisor/secure-monitor calls, the
692 ISB instruction synchronisation barrier, and NOP. Otherwise, the architecture permits *constrained unpredictable*
693 behaviour, meaning that the resulting machine state could be anything that would be reachable by arbitrary
694 instructions at the same exception level. The following W+F test illustrates this.

695

W+F — AArch64

Initial state: 0:W0="SUB X0,X0,#1", 0:X1=l	
Thread 0	**Thread 1**
STR W0,[X1] // modify Thread 1 at l	l: **ADD** X0,X0,#1 // initial code
Allowed: constrained-unpredictable final state	

696 In this test Thread 0 performs a memory store (with the STR instruction) to the code that Thread 1 is executing;
697 overwriting the ADD X0,X0,#1 instruction with the 32-bit encoding of the SUB X0,X0,#1 instruction. If the fetch
698 were atomic, the outcome of this test would be the result of executing either the ADD or the SUB instruction, but,
699 since at least one of those is not in the set of the 8 atomically-fetchable instructions given previously, Thread 1
700 has constrained-unpredictable behaviour and the final state is very loosely constrained. Note, however, that this
701 is nonetheless much stronger than the C/C++ whole-program undefined behaviour in the presence of a data race:
702 unlike C/C++, a hardware architecture has to define a useful envelope of behaviour for arbitrary code, to provide
703 guarantees for the rest of the system when one user thread has a race.

704 #### Conditional Branches

705 For conditional branches, the Arm architecture provides a specific non-single-copy-atomic fetch guarantee: the exe-
706 cution will be consistent with either the old or new target, and either the old or new condition.
707 For example, this W+F+branches test can overwrite
708 a B.EQ g with a B.NE h, and end up executing
709 B.NE g or B.EQ h instead of one of those. Our
710 future examples will only modify NOPs and uncon-
711 ditional branch instructions.

W+F+branches — AArch64

Initial state: 0:W0="B.NE h", 0:X1=l	
Thread 0	**Thread 1**
STR W0,[X1]	l: B.EQ g
Allowed: execute "B.NE g"	

712 ### 3.3.2 Coherence

713 Data writes and reads are coherent, in Arm and
714 in other major architectures: in any execution, for
715 each address, the reads of each hardware thread must see a subsequence of the total *coherence order* of all writes
716 to that address. The plain-data CoRR test [18] illustrates one case of this: it is forbidden for a thread to read a
717 new write of x and then the initial state for x. However, instruction fetches are not necessarily coherent: one
718 instruction fetch may be inconsistent with a program-order-previous fetch, and the data and instruction streams
719 can become out-of-sync with each other. We explore three kinds of coherence:

720 – Instruction-to-Instruction Coherence: whether fetches of the same location must observe writes to the
721 same location coherently.

722 – Data-to-Instruction Coherence: whether fetches and then reads to the same location must observe writes to
723 the same location coherently.

724 – Instruction-to-Data Coherence: whether reads and then fetches of the same location must observe writes to
725 the same location coherently.

726 #### Instruction-to-Instruction Coherence

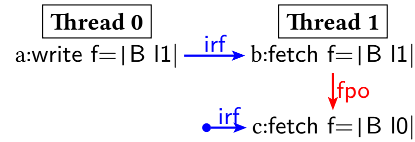
727 Arm explicitly do not guarantee any consistency between fetches of the same location: fetching an instruction
728 does not mean that a later fetch of that location will not see an older instruction [3, B2.4.4]. This is illustrated by
729 CoFF, like CoRR but with fetches instead of reads.

730 Here Thread 1 makes two calls to address f (BL is branch-and-link), while Thread 0 overwrites the instruction
731 at that address. The interesting potential execution is that in which the first call to f fetches and executes the

CoFF		AArch64
Initial state: `0:W0="B l1", 0:X1=f`		
Thread 0	**Thread 1**	**Common**

`STR W0,[X1] //a`	`BL f`
	`MOV X0,X10`
	`BL f`
	`MOV X1,X10`
		`f:  B l0`
		`l1: MOV X10,#2`
		`    RET`
		`l0: MOV X10,#1`
		`    RET`

Allowed: `1:X0=2, 1:X1=1`		



newly-written `B l1`, but the second call fetches and executes the original `B l0`. We can view such executions as graphs, similar to previous axiomatic-model candidate executions but with new fetch events, one per instruction, and new edges. As usual, we use `po` and `rf` edges for the program-order and reads-from relations, together with:

- `fe` (fetch-to-execute), which relates the fetch event of an instruction to all the execution events (memory writes, reads or barriers) of the instruction;
- `irf` (instruction-read-from), relating a write to all fetches that read from it (analogous to reads-from, `rf`); and
- `fpo` (fetch-program-order), relating fetches of instructions that are in program order (analogous to program order, `po`).

Edges from the initial state are drawn from a small circle. Since we do not modify the code of most locations, we usually omit the fetch events for those instructions, showing only a subgraph of the interesting events, e.g. as on the right above. For Arm, this execution is both architecturally allowed and experimentally observed.

Here, and in future tests, we assume some common code consisting of a function at address `f` which always has the same shape: a branch that might be overwritten, which selects a block that writes a value to register `X10` before returning. This is sometimes duplicated at different addresses (`f1`, `f2`, ...) or extended to g, with three cases. We sometimes elide the common code.

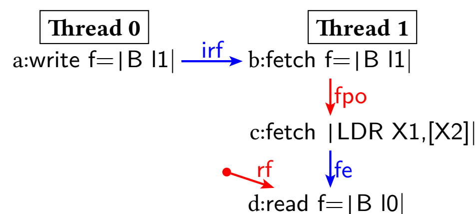
Data-to-Instruction Coherence

Fetching from a particular write does imply that program-order-later reads from the same address will see that write (or a coherence successor thereof). This is a *data-to-instruction* coherence property, illustrated by CoFR below. Here Thread 1 fetches the newly-written `B l1` at `f` and then, when reading from `f` with its `LDR` load instruction, cannot read the original `B l0` instruction (it can only read the new `B l1`).

CoFR		AArch64
Initial state: `0:W0="B l1", 0:X1=f, 1:X2=f`		
Thread 0	**Thread 1**	**Common**

`STR W0,[X1]`	`BL f`
	`MOV X0,X10`
	`LDR X1,[X2]`
		`f: B l0`
		`l1: MOV X10,#2`
		`RET`
		`l0: MOV X10,#1`
		`RET`

Forbidden: `1:X0=2, 1:X1="B l0"`		



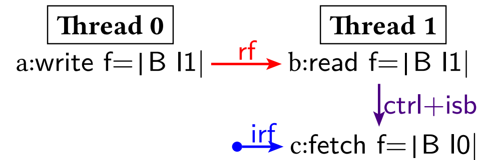
This is not clear in the existing prose specification, but the architectural intent that emerged during discussion with Arm is that the given execution should be forbidden, reflecting microarchitectural choices that (1) instructions decode in order, so the fetch b must occur before the read d, and (2) fetches that miss in the instruction cache must read from data storage, so the instruction cache cannot be ahead of the available data. This ensures that fetching from a write means that all threads are now guaranteed to read from that write (or another coherence-after it).

Instruction-to-Data Coherence

In the other direction, reading from a particular write to some location does *not* imply that later fetches of that location will see that write (or a coherence successor), as in the following CoRF+ctrl-isb.

CoRF+ctrl-isb AArch64

Initial state: `0:W0="B l1", 0:X1=f, 1:X2=f`		
Thread 0	**Thread 1**	**Common**
STR W0,[X1]	**LDR** X0,[X2] **CBNZ** X0,l l: **ISB** BL f **MOV** X1,X10	f: B l0 l1: **MOV** X10,#2 RET l0: **MOV** X10,#1 RET
Allowed: `1:X0="B l1", 1:X1=1`		



Here Thread 1 has a control dependency and an instruction synchronisation barrier (the `CBNZ` conditional branch, dependent on the value read by its `LDR` load, and `ISB`), abbreviated to ctrl+isb, between its load and the fetch from `f`. If the latter were a data load, this would ensure the two loads are satisfied in order. This is not explicit in the existing prose, but it is what one would expect, and it is observed in practice. Microarchitecturally, it is easily explained by an out-of-date entry for `f` in the instruction cache of Thread 1: if Thread 1 had previously fetched `f` (perhaps speculatively), and that instruction cache entry has not been evicted or explicitly invalidated since, then this fetch of `f` will simply read the old value from the instruction cache without going out to data memory. The `ISB` ensures that `f` is freshly fetched, but does not ensure that Thread 1's instruction cache is up-to-date with respect to data memory.

3.3.3 Instruction Synchronisation

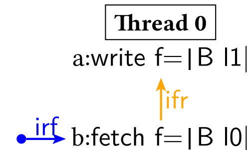
Instruction fetches satisfy few guarantees, so explicit synchronisation must be performed when modifying the instruction stream.

Same-Thread Synchronisation

Test SM below shows the simplest self-modifying code case: without additional synchronisation, a write to program memory can be ignored by a program-order-later fetch.

SM AArch64

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



In this execution, the fetch `b`, fetching the instruction at `f`, fetches a value from a write coherence-before `a`, even though `b` is the fetch of an instruction program-order after `a`. We illustrate this with an *instruction from-reads* (`ifr`) edge. This is a derived relation, analogous to the usual *from-reads* (`fr`) relation, that relates each fetch to all writes that are coherence-after the write it read from; it is defined as $\texttt{ifr} = \texttt{irf}^{INVERSE}; \texttt{co}$. If the fetch were a data read, this would be a forbidden coherence shape (COWR). As it is, it is architecturally allowed, as described explicitly by Arm [3, B2.4.4], and it is experimentally observed on all devices we have tested. Microarchitecturally, this too is simply due to fetches from old instruction cache entries.

Cache Maintenance

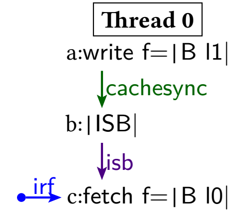
As we saw in §3.2, the Arm architecture provides cache maintenance instructions to synchronise the instruction and data streams: the `DC` data-cache clean and `IC` instruction-cache invalidate instructions. To forbid the relaxed outcome of SM, by forcing a fetch of the modified code, the specified sequence of cache maintenance instructions must be inserted, with an `ISB`.

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



Now the outcome is forbidden. The cache synchronisation sequence `DC CVAU; DSB ISH; IC IVAU; DSB ISH` (which we abbreviate to a single `cachesync` edge) ensures that by the time the `ISB` executes, the instruction and data memory have been made coherent with each other for `f`. The `ISB` then ensures the final fetch of `f` is ordered after this sequence. The microarchitectural intuition for this was in §3.2; our §3.4 operational model will describe the semantics of each instruction.

Cross-Thread Synchronisation

We now consider modifying code that can be fetched by other threads, using variants of the standard message-passing shape MP. That checks whether two writes (to different locations) on one thread can be seen out-of-order by two reads on another thread; here we replace one or both of those reads by fetches, and ask what synchronisation is required to ensure that the relaxed outcome is forbidden. Consider first an MP variant where the first write is of a new instruction, and the second is just a simple data memory flag:

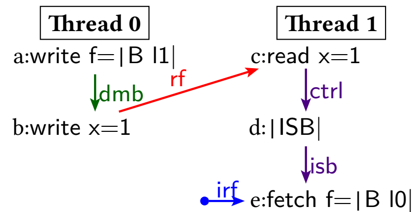
MP.RF+dmb+ctrl-isb AArch64

Initial state: 0:W0="B l1", 0:X1=f, 0:X2=1, 0:X3=x, 1:X2=x, [x]=0

Thread 0	Thread 1

STR W0,[X1]
DMB ISH
STR X2,[X3]
	LDR X0,[X2]
	CBNZ X0,l
	l: ISB
	BL f
	MOV X1,X10

Allowed: 1:X0=1, 1:X1=1



This test includes sufficient synchronisation on each thread to enforce thread-local ordering of data accesses: the `DMB` in Thread 0 ensures the writes `a` and `b` propagate to memory in program order, and the control-dependency into an `ISB` on Thread 1 ensures the read `c` and the fetch `e` happen in program order. However, as we saw in §3.2, this is not enough to synchronise concurrent modification and execution of code in ARMv8-A. Thread 0 needs the entire cache synchronization sequence (giving test MP.RF+cachesync+ctrl-isb, not shown), not just a `DMB`, to forbid this outcome.

Another variant of this MP-shape test where the message passing itself is done using modification of code gives a much stronger guarantee, as can be seen from the following MP.FR+dmb+fpo-fe test. This is not clear from

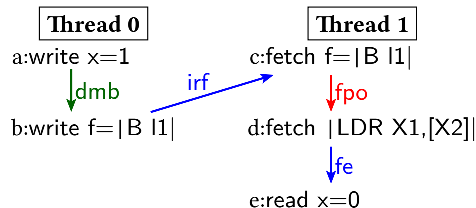
MP.FR+dmb+fpo-fe AArch64

Initial state: 0:X0=1, 0:X1=x, 1:X2=x, [x]=0, 0:W2="B l1", 0:X3=f

Thread 0	Thread 1

STR X0,[X1]
DMB ISH
STR W2,[X3]
	BL f
	MOV X0,X10
	LDR X1,[X2]

Forbidden: 1:X0=2, 1:X1=0



the architecture manual, but this outcome is already forbidden with only the `DMB`. This is for similar reasons to the above CoFR test: since Thread 1 fetched the updated value for `f`, we know that value must have reached at least the data caches (since that is where the instruction cache reads from) and therefore multi-copy atomicity guarantees that a normal load instruction will observe it.

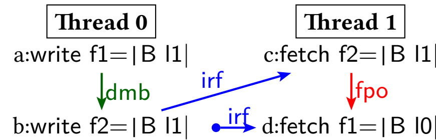
The final variant of these MP-shaped tests has both Thread 0 writes be of new instructions. This idiom is very common in practice; it is currently how Chrome's WebAssembly JIT synchronises the modified thread with the new code.

MP.FF+dmb+fpo AArch64

Initial state: `0:W0="B l1", 0:X1=f1, 0:W2="B l1", 0:X3=f2`	
Thread 0	**Thread 1**

`STR W0,[X1]`
`DMB ISH`
`STR W2,[X3]`
	`BL f2`
	`MOV X0,X10`
	`BL f1`
	`MOV X1,X10`

Allowed: `1:X0=2, 1:X1=1`	



Without the full cachesync sequence on Thread 0, this is an allowed outcome. Interestingly, adding the cachesync sequence to Thread 0 (Test MP.FF+cachesync+fpo, not shown) is sufficient to make the outcome forbidden, without an `ISB` in Thread 1, as the cachesync sequence is intended to make it appear that fetches occur in program order. Microarchitecturally, that could be ensured in two ways: either by actually fetching in-order, or by making the `IC` instruction not only invalidate all the instruction caches (for this address) but also clean any core's pre-fetch buffer stale entries (for this address). Architecturally, this is not clear in the current prose, but, concurrent with this work, Arm were independently strengthening their definition to make it so.

Incremental Synchronisation

The cache synchronisation sequence need not be contiguous, or even all in the same thread. So long as the sequence in its entirety has been performed by the time the fetch happens, then the instruction stream will have been made consistent with the data stream for that address.

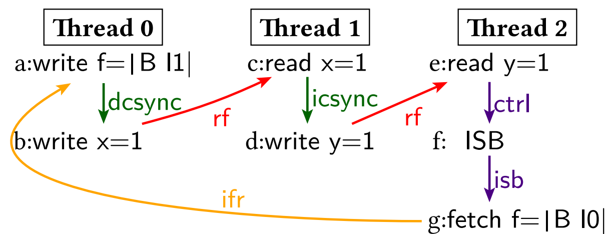
This is demonstrated by the following test, where Thread 0 performs a write to `f` and then only a `DC` before synchronizing with Thread 1, which performs the `IC`, while Thread 2 observes the modified code. This can happen in practice when a software thread is migrated between hardware threads at runtime, by a hypervisor or OS. Thread 0 and Thread 1 may just represent the runtime scheduling of a single process, beginning execution on hardware Thread 0 but migrated to hardware Thread 1 between the `DC` and `IC` instructions. In the graph, the dcsync and icsync represent the `DC;DSB ISH` and `DSB ISH;IC;DSB ISH` combinations. The `DC` does not need a preceding `DSB ISH` because it is ordered w.r.t. the preceding store to the same cache line.

ISA2.F+dc+ic+ctrl-isb AArch64

Initial state: `0:W0="B l1", 0:X1=f, 0:X2=1, 0:X3=x, [x]=0, 1:X4=f, 1:X1=x, 1:X2=1, 1:X3=y, [y]=0, 2:X2=y`		
Thread 0	**Thread 1**	**Thread 2**

`STR W0,[X1]`
`DC CVAU, X1`
`DSB ISH`
`STR X2,[X3]`
	`LDR X0,[X1]`
	`DSB ISH`
	`IC IVAU, X4`
	`DSB ISH`
	`STR X2,[X3]`
		`LDR X0,[X2]`
		`CBZ X0,l`
		`l:ISB`
		`BL f`
		`MOV X1,X10`

Forbidden: `1:X0=1, 1:X1=1`		



Here the `IC` gets broadcast to all threads [3, B2.2.5p3], and so the fact that it happens on a different thread to the `DC` does not affect the outcome. Similarly, if the `DC` were to happen on another thread first (to get the test MP.RF+[dc]-ic+ctrl-isb, not shown), then it would have the effect of ensuring consistency globally, for all threads.

3.3.4 Multi-Copy Atomicity

For data accesses, the question of whether they are *multi-copy atomic* is a crucial one for relaxed architectures. IBM POWER, ARMv7, and pre-2018 ARMv8-A are/were non-multi-copy atomic: two writes to different addresses could become visible to distinct other threads in different orders. Post-2018 ARMv8-A and RISC-V are multi-copy atomic (or "other multi-copy-atomic" in Arm terminology) [10, 9, 3]: the programmer can assume there is a single shared memory, with all relaxed-memory effects due to thread-local out-of-order execution.

However, for fetches, due to the lack of any fetch atomicity guarantee for most instructions (§3.3.1), and the lack of coherent fetches for the others (§3.3.2), the question of multi-copy atomicity is not particularly interesting. Tests are either trivially forbidden (by data-to-instruction coherence) or are allowed but only the full cache synchronisation sequence provides enough guarantees to forbid it, and (§3.3.3) this ensures all cores will share the same consistent view of memory.

3.3.5 Strength of the `IC` Instruction

Multiple Points of Unification

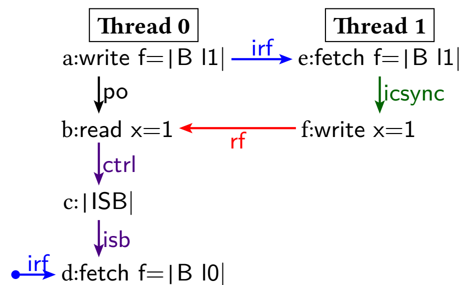
Cleaning the data cache, using the `DC` instruction, makes a write visible to instruction memory. It does this by pushing the write past the Point of Unification. However, there may be multiple Points of Unification: one for each core, where its own instruction and data memory become unified, and one for the entire system (or shareability domain) where all the caches unify. Fetching from a write implies that it has reached the closest PoU, but does not imply it has reached any others, even if the write originated from a distant core. Consider: Here Thread 0

SM.F+ic AArch64

Initial state: `0:W0="B l1", 0:X4=f, 0:X3=x, [x]=0, 1:X4=f, 1:X2=1, 1:X3=x`	
Thread 0	**Thread 1**

`STR W0,[X4]`
`LDR X2,[X3]`
`CBZ X2,l`
`l: ISB`
`BL f`
`MOV X1,X10`
	`BL f`
	`MOV X0,X10`
	`IC IVAU, X4`
	`DSB ISH`
	`STR X2,[X3]`

Allowed: `1:X0=2, 0:X2=1, 0:X1=1`	



modifies `f`, Thread 1 fetches the new value and performs just an `IC` and `DSB`, before signalling Thread 0 which also fetches `f`. That `IC` is not strong enough to ensure that the write is pulled into the instruction cache of Thread 0.

This is not clear in the existing prose, but the architectural intent is that it be allowed (i.e., that `IC` is weak in this respect). We have not so far observed it in practice. The write may have passed the Point of Unification for Thread 1, but not the shared Point of Unification for both threads. In other words, the write might reach Thread 1's instruction cache without being pushed down from Thread 0's data cache. Microarchitecturally this can be explained by *direct data intervention* (DDI), an optimisation allowing cache lines to be migrated directly from one thread's (data) cache to another. The line could be migrated from Thread 0 to Thread 1, then pushed past Thread 1's Point of Unification, making it visible to Thread 1's instruction memory without ever making it visible to Thread 0's own instruction memory. The lack of coherence between instruction and data caches would make this observable, even in multi-copy atomic machines.

Stale Fetches

So far, we have only talked about fetching from two distinct writes. But theoretically there is no limit to how far back we can fetch from, with insufficient synchronization. The MP.RF+dmb+ctrl-isb test (§3.3.3) required the full `cachesync` sequence to forbid the given behaviour. Below we give a test, FOW, similar to that MP-shaped test but allowing many consumer threads to independently and simultaneously see different values in their instruction memory, even after invalidating their caches.

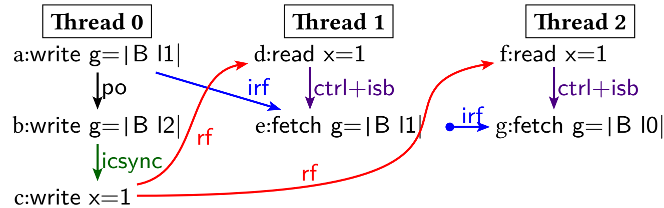
This is not clear in the existing architecture text. It is a case where the architecture design is not very constrained. On the one hand, it has not been observed, and it is thought unlikely that hardware will ever exhibit this behaviour: it would require keeping multiple writes in the coherent part of the data caches, rather than a single dirty line, which would require more complex cache coherence protocols. On the other hand, there does not seem to be any benefit to software from forbidding it. Arm therefore prefer the choice that gives a simpler and weaker model (here the two happen to coincide), to make it easier to understand and to provide more flexibility for future microarchitectural optimisations. We therefore design our models to allow the above behaviour.

FOW AArch64

Initial state: `0:W0="B l1", 0:X2=g, 0:W1="B l2", 0:X3=1, 0:X4=x, [x]=0, 1:X4=x, 2:X4=x`			
Thread 0	**Thread 1**	**Thread 2**	**Common**

`STR W0,[X2]`
`STR W1,[X2]`
`DSB ISH`
`IC IVAU, X2`
`DSB ISH`
`STR X3,[X4]`
	`LDR X0, [X4]`
	`CBNZ X0, la`
	`la: ISB`
	`BL g`
	`MOV X1,X10`
		`LDR X0, [X4]`
		`CBNZ X0, lb`
		`lb: ISB`
		`BL g`
		`MOV X1,X10`
			`g: B l0`
			`l2: MOV X10, #3`
			`RET`
			`l1: MOV X10, #2`
			`RET`
			`l0: MOV X10, #1`
			`RET`

Allowed: `1:X0=1, 1:X1=2, 2:X0=1, 2:X1=1`			



3.3.6 Strength of the DC Instruction

Instruction Cache depth

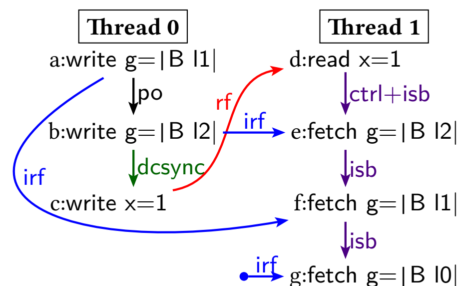
Test CoFF (§3.3.2) showed that fetches can see "old" writes. In principle, there is no limit to the depth of the instruction-cache hierarchy: there could be many values for a single location cached in the instruction memory for each core, even if the data cache has been cleaned. The test below illustrates this, with Thread 1 able to see all three values for g.

MP.RF+dc+ctrl-isb-isb AArch64

Initial state: `0:W0="B l1", 0:X2=g, 0:W1="B l2", 0:X3=1, 0:X4=x, [x]=0, 1:X4=x`		
Thread 0	**Thread 1**	**Common**

`STR W0,[X2]`
`STR W1,[X2]`
`DSB ISH`
`DC CVAU,X2`
`DSB ISH`
`STR X3,[X4]`
	`LDR X0, [X4]`
	`CBNZ X0, l`
	`l:ISB`
	`BL g`
	`MOV X1,X10`
	`ISB`
	`BL g`
	`MOV X2,X10`
	`ISB`
	`BL g`
	`MOV X3,X10`
		`g: B l0`
		`l2:MOV X10,#3`
		`RET`
		`l1:MOV X10,#2`
		`RET`
		`l0:MOV X10,#1`
		`RET`

Allowed: `1:X0=1, 1:X1=3, 1:X2=2, 1:X3=1`		



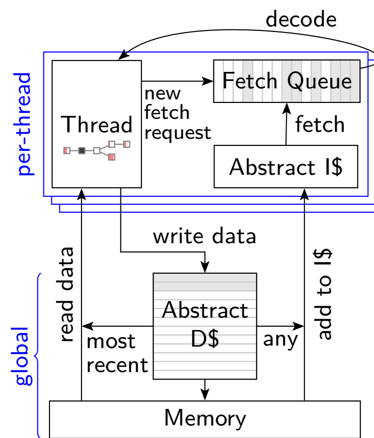
This is similar to the preceding FOW case: it is thought unlikely that hardware will exhibit this in practice, but the desire for the simpler and weaker option means the architectural intent is to allow it, and we follow that in our models.

3.4 An Operational Semantics for Instruction Fetch

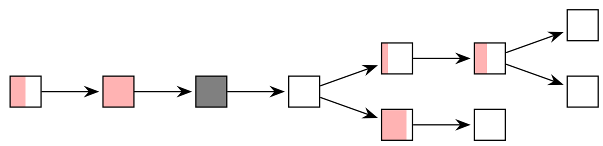
Previous work on operational models for IBM POWER and Arm "user-mode" concurrency [18, 16, 13, 12, 11, 10] has shown, surprisingly, that as far as programmer-visible behaviour is concerned, one can abstract from almost all hardware implementation details of data memory (store queues, the cache hierarchy, the cache protocol, etc.). For ARMv8-A, following their 2018 shift to a multicopy-atomic architecture, one can do so completely: the *Flat* model of [10] has a shared flat memory, with a per-thread out-of-order thread subsystem, modelling pipeline effects, responsible for all observable relaxed behaviour. For instruction-fetch, it is no longer possible to abstract completely from the data and instruction cache hierarchy, but we can still abstract from much of it.

The Flat Model

is a small-step operational semantics for multi-copy atomic ARMv8-A, including the relaxed behaviours of loads and stores [10]. Its states are abstract machine states consisting of a tree of instructions for each thread, and a flat mem-



ory subsystem shared by all threads. Each instruction in each thread corresponds to a sequence of transitions, with some guards and a potential effect on the shared memory state. The Flat model is made executable in our RMEM tool, which can exhaustively interleave transitions to enumerate all the possible behaviours. The tree of instructions for each thread models out-of-order and speculative execution explicitly. Below we show an example for a thread that is executing 10 instruction instances. Some (grey) are finished, no longer subject to restart; others (pink) have run some but perhaps not all of their instruction semantics; instructions are not necessarily atomic. Those with multiple children are branch instructions with multiple potential successors speculated simultaneously.



For each state, the model defines the set of allowed transitions, each of which steps to a new machine state. Transitions correspond to steps of single instructions, and individual instructions may give rise to many. Example transitions include Register Write, Propagate Write to Memory, etc.

iFlat Extension

Originally, Flat had a fixed instruction memory, with a single transition that can speculate the address of any program-order successor of any instruction in flight, fetch it from the fixed instruction memory, and decode it. We now remove that fixed instruction memory, so that instructions can be fetched from data writes, and add the additional structures as shown on the right. These are all of unbounded size, as is appropriate for an architecture definition.

Fetch Queues (per-thread)

These are ordered buffers of pre-fetched entries, waiting to be decoded and begin execution. Entries are either a fetched 32-bit opcode, or an unfetched request. The fetch queues allow the model to speculate and pre-fetch many instructions ahead of where the thread is currently executing. The model's fetch queues abstract from multiple real-hardware structures: instruction queues, line-fill buffers, loop buffers, and slots objects. We keep a close relation to this underlying microarchitecture by allowing out-of-order fetches, but we believe this is not experimentally observable on real hardware.

Abstract Instruction Cches (per-thread)

These are just sets of writes. When the fetch queue requests a new entry, it gets satisfied from the instruction cache, either immediately (a *hit*) or at some later point in time (a *miss*). The instruction cache can contain many possible writes for each location (§3.3.6), and it can be spontaneously updated with new writes in the system at any time ([3, B2.4.4]). To manage `IC` instructions, each thread keeps a list of addresses yet to be invalidated by in-flight `IC`s.

Data Cache (global)

Above the single shared flat memory for the entire system, which sufficed for the multi-copy-atomic ARMv8-A data memory, we insert a shared buffer which is just a list of writes; abstracting from the many possible coherent data cache hierarchies. Data reads must be coherent, reading from the most recent write to the same address in the buffer, but instruction fetches are allowed to read from any such write in the buffer (§3.3.2).

Transitions

To accommodate instruction fetch and cache maintenance, we introduce new transitions: Fetch Request, Fetch Instruction, Fetch Instruction (Unpredictable), Fetch Instruction (B.cond), Decode Instruction, Begin IC, Propagate IC to Thread, Complete IC, Perform DC, and Update Instruction Cache. We also have to modify some Flat transitions: Commit ISB, Wait for DSB, Commit DSB, Propagate Memory Write, and Satisfy Read from Memory. These transitions define the lifecycle of each instruction: a request gets issued for the fetch, then at some later point the fetch gets satisfied from the instruction cache, the instruction is then decoded (in program-order) and then handed to the existing semantics to be executed. To give a flavour, we show just one, the *Propagate IC to Thread* transition, which is responsible for invalidation of the abstract instruction caches. This is a prose rendering of the rule in our executable mathematical model, which is expressed in the typed functional subset of Lem [41].

Propagate IC to Thread

An instruction *i* (with ID *iiid*) in state WAIT_IC*(address, state_cont)* can do the relevant invalidate for any thread *tid'*, modifying that thread's instruction cache and fetch queue, if there exists a pending entry *(iiid, address)* in that thread's *ic_writes*. Action:

1. for any entry in the fetch queue for thread *tid*, whose *program_loc* is in the same minimum-size instruction cache line as *address*, and is in FETCHED(_) state, set it to the UNFETCHED state;
2. for the instruction cache of thread *tid*, remove any write-slices which are in the same instruction cache line of minimum size as *address*.

This rule can be found under the same name in the full prose description, and in the `handle_ic_ivau` and `flat_propagate_cache_maintenance` functions in `machineDefThreadSubsystem.lem` and `machineDefFlatStorageSubsyst` in the executable mathematics. Cache maintenance operations work over entire cache lines, not individual addresses. Each address is associated with at least one cache line for the data (and unified) caches, and one for the instruction caches. The cache line of minimum size is the (architected) smallest possible cache line for each of these.

Example

This model correctly explains all the behaviours of §3.3. We illustrate this by revisiting the cache synchronization explanation of §3.2, which can now be re-interpreted w.r.t. our precise model, and using this to explain the thread migration case of §3.3.3. Given `DC Xn; DSB; IC Xn; DSB` we can use this model to give meaning to it (omitting uninteresting transitions): First the `DC CVAU` causes a **Perform DC** transition. This pushes any write that might have been in the abstract data cache into memory. Now the first `DSB`'s **Commit DSB** can be taken, allowing **Begin IC** to happen. This creates entries for each thread, which are discharged by each **Propagate IC to Thread** (see above). Once all entries are invalidated, a **Complete IC** can happen. Now, if any thread decodes an instruction for that address, it must have been fetched from the write the `DC` pushed, or something coherence-after it. If the software thread performing this sequence is interrupted and migrated (by the OS) to a different hardware thread, then, so long as the OS includes the `DSB` to maintain the thread-local DC ordering, the `DC` will push the write in an identical way, since it only affects the global abstract data cache. The `IC` transitions can all be taken, and the sequence continues as before, just on a new hardware thread. So when the second `DSB` finishes, and the final **Commit DSB** transitions is taken, the effect of the full sequence will be seen system-wide even if the thread was migrated.

3.5 An Axiomatic Semantics for Instruction Fetch

Based on the operational model, we develop an axiomatic semantics, as an extension of the ARMv8 axiomatic reference model [26, 10]. Since that does not have mixed-size support, we do not model the concurrent modification of conditional branches (§3.3.1), as this would require mixed-size machinery. The existing axiomatic model is

974 a predicate on *candidate executions*, hypothetical complete executions of the given program that satisfy some
975 basic well-formedness conditions, defining the set of *valid* executions to be those satisfying its axioms. Each
976 candidate execution abstractly captures a particular concrete execution of the program in terms of events and
977 relations over them. This model is expressed in the herd language [14, 21, 29]. The events of these executions are
978 memory reads (the set R), memory writes (W), and memory barrier/fence events (F). The relations are: *program*
979 *order* (po), capturing the sequencing of events by the same thread in the execution's control-flow unfolding;
980 *reads-from* (rf), relating a write event w with any read event r that reads from it; the *coherence order* (co), recording
981 the execution's sequencing of same-address writes in memory; and *read-modify-write* (rmw), capturing which
982 load/store exclusive instructions form a successful exclusive *pair* in the execution. The derived relation *from-reads*
983 $\texttt{fr} = \texttt{rf}INVERSE$;co relates a read r with a write w' if r reads from a write w coherence before w'. In addition,
984 candidate executions also have relations capturing dependencies between events: address (addr), data (data), and
985 control dependencies (ctrl). The relation loc relates any two read/write events that are to the same memory
986 address. The model also has relations suffixed "i" and "e": rfi/rfe, coi/coe, fri/fre. These are the restrictions
987 of the relations rf, co, and fr, to same-thread/"internal" event pairs or different-thread/"external" event pairs.
988 The model is defined in relational algebra. In herd, R;S stands for sequential composition of relations R and S,
989 RINVERSE for the inverse of relation R, R|S and R&S for the union and intersection of R and S, and [A];R;[B] for
990 the restriction of R to the domain A and range B.

991 Handling instruction fetch requires extending the notion of candidate execution. We add new events: an *instruction-*
992 *fetch* (IF) event for each executed instruction; a DC event for each DC CVAU instruction; an IC event for each
993 IC IVAU and IC IALLU instruction. We replace po with *fetch-program-order* (fpo) which orders the IF event
994 of an instruction before any program-order later IF events. We add a relation *same-cache-line* (scl), relating
995 reads, writes, fetches, DC and IC events to addresses in the same cache line. We add an acyclic transitively closed
996 relation wco, which extends co with orderings for cache maintenance (DC or IC) events: it includes an ordering
997 (e, e') or (e', e) for any cache maintenance event e and same-cache-line event e' if e' is a write or another cache
998 maintenance event; where co = ([W];wco;[W]) & loc. The loc, addr, and ctrl are all extended to include
999 DC and IC events. We add a *fetch-to-execute* relation (fe), relating an IF event to any event generated by the
1000 execution of that instruction; and an *instruction-read-from* relation (irf), which relates a write to any IF event
1001 that fetches from it. Finally, we add a boolean *constrained-unpredictable* (CU) to detect badly behaved programs.
1002 Now we derive the following relations: the standard po relation, as po = feINVERSE;fpo;fe (two events e and
1003 e' are po-related if their fetch-events are fpo-related); and *instruction-from-reads* (ifr), the analogue of fr for
1004 instruction fetches, relating a fetch to all writes coherence-after the one it fetched from: ifr = irfINVERSE;co.

1005 We then make two semantics-preserving rewrites of the existing model to make adding instruction fetches easier
1006 (described in the appendix); and make the following changes and additions to the model. The full model is shown
1007 in Figure 3.1, with comments pointing to the relevant locations in the model definition. For lack of space we only
1008 describe the main addition, the iseq relation, in detail (including its correspondence with the operational model
1009 of §3.4); for the others we give an overview and refer to the appendix for the full description.

1010 We define the relation iseq, relating some write w to address x to an IC event completing a cache synchronisation
1011 sequence (not necessarily on a single thread): w is followed by a same-cache line DC event, which is in turn
1012 followed by a same-cache line IC event. In operational model terms, this captures traces that propagated w to
1013 memory, subsequently performed a same-cache-line DC, and then began an IC (and eagerly propagated the IC to
1014 all threads). In any state after this sequence it is guaranteed that w, or a coherence-newer same-address write, is
1015 in the instruction cache of all threads: performing the DC has cleared the abstract data cache of writes to x, and
1016 the subsequent IC has removed old instructions for location x from the instruction caches, so that any subsequent
1017 updates to the instruction caches have been with w, or co-newer writes. Adding ifr;iseq to the *observed-by*
1018 relation (obs) (4) relates an instruction fetch *i* to location x to an IC *ic* if: i fetched from a write w to x, some
1019 write w' to x is coherence-after w, and *ic* completes a cache synchronisation sequence (iseq) starting from w'.
1020 Then the irreflexive ob axiom requires that i must be ordered-before *ic* (because it would otherwise have
1021 fetched w').We now briefly overview other changes made to the axiomatic model and their intuition. We include
1022 irf in obs (3): for an instruction to be fetched from a write, the write has to have been done before. We add a
1023 relation *fetch-ordered-before* (fob) (5-7), which is included in *ordered-before*. The relation fob includes fpo and
1024 fe; including fpo (5) requires fetches to be ordered according to their position in the control-flow unfolding
1025 of the execution. and including the fe (*fetch-to-execute*) relation (6) captures the idea that an instruction must
1026 be fetched before it can execute; fetches program-order-after an ISB happen after the ISB (or else are restarted)
1027 (7). For DSB ISH instructions the edge [R|W|F|DC|IC];po;[dsb.ish] is included in ob (9): DSB ISHs are
1028 ordered with all program-order-preceding non-fetch events. Symmetrically, all non-IF events are ordered after
1029 program-order-preceding dsb.ish events (10). DCs wait for preceding dmb.sy events (11). We include the
1030 relation *cache-op-ordered-before* (cob) in ob. This relation orders DC instructions with program-order previous

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

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

(* Fetch-ordered-before *)
let fob = [IF]; fpo; [IF]       (*5*)
  | [IF]; fe                    (*6*)
  | [ISB]; feINVERSE; 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]) \
            obINVERSE \ (co;iseq;ob)
cff_bad cff ≡ CU                  (*15*)
```

Figure 3.1: Axiomatic model

reads/writes and other DCs to the same cache line (12,13).

Finally, *could-fetch-from* (cff) (14) captures, for each fetch i, the writes it could have fetched from (including the one it did fetch from), which we use to define the *constrained unpredictable* axiom cff_bad (not given) (15).

3.6 Validation

To gain confidence in the presented models we validated the models against the Arm architectural intent, against each other, and against real hardware.

Validation against the Architecture

To ensure our models correctly captured the architectural intent we engaged in detailed discussions with Arm, including the Arm chief architect. These involved inventing litmus tests (including, those described in §3.3 and many others) and discussing what the architecture should allow in each case.

Validating against hardware

To run instruction-fetch tests on hardware, we extended the litmus tool [17]. The most significant extension consists in handling code that can be modified, and thus has to be restored between experiments. To that end, code copies are executed, those copies reside in mmap'd memory with (execute permission granted. Copies are made from "master" copies, in effect C functions whose contents basically consist of gcc extended inline assembly. Of course, such code has to be position independent, and explicit code addresses in test initialisation sections (such as in 0:X1=l in the test of §3.3.1) are specific to each copy. All the cache handling instructions used in our experiments are all allowed to execute at exception level 0 (user-mode), and therefore no additional privilege is needed to run the tests.

To automatically generate families of interesting instruction-fetch tests, we extended the diy test generation tool [40] to support instruction-fetch reads-from (irf) and instruction-fetch from-reads (ifr) edges, in both internal (same-thread) and external (inter-thread) forms, and the cachesync edge. We used this to generate 1456 tests involving those edges together with po, rf, fr, addr, ctrl, ctrlisb, and dmb.sy. diy does not currently support bare DC or IC instructions, locations which are both fetched and read from, or repeated fetches from the same location.

We then ran the diy-generated test suite on a range of hardware implementations, to collect a substantial sample of actual hardware behaviour.

Correspondence between the models

We experimentally test the equivalence of the operational and axiomatic models on the above hand-written and diy-generated tests, checking that the models give the same sets of allowed final states, and that these are consistent with the hardware observations.

Making the models executable as a test oracle

To make the operational model executable as a test oracle, capable of computing the set of all allowed executions of a litmus test, we must be able to *exhaustively enumerate* all possible traces. For the model as presented, doing this naively is infeasible: for each instruction it is theoretically possible to speculate any of the 2^{64} addresses as potential next address, and the interleaving of the new fetch transitions with others leads to an additional combinatorial explosion.

We address these with two new optimisations. First, we extend the fixed-point optimisation in RMEM (incrementally computing the set of possible branch targets) [10] to keep track not only of indirect branches but also the successors of every program location, and only allow speculating from this set of successors. Additionally, we track during a test which locations were both fetched and modified during the test, and eagerly take fetch and decode transitions for all other locations. As before, the search then runs until the set of branch targets *and* the set of modified program-locations reaches a fixed point. We also take some of the transitions eagerly to reduce the search space, in cases where this cannot remove behaviour: **Wait for IC**, **Complete IC**, **Fetch Request**, and **Update Instruction Cache**.

Making the axiomatic model executable as a test oracle

The axiomatic model is expressed in a herd-like form, but the herd tool does not support instruction fetch and cache maintenance instructions. To make the model executable as a test oracle, we built a new tool that takes litmus tests and uses a Sail [7] definition of a fragment of the ARMv8-A ISA to generate SMT problems for the model. Using the Sail instruction semantics, we generate a Sail program that corresponds to each thread within a litmus test. The tool then partially evaluates these programs using the concrete values for addresses and registers specified in the litmus file, while allowing memory values and arbitrary addresses to remain symbolic. Using a Sail to SMT-LIB backend, these are translated into SMT definitions that include all possible behaviours of each thread as satisfiable solutions. The rules for the axiomatic model are then applied as assertions restricting the possible behaviours to just those allowed by the axiomatic model. The tool also derives the `addr` and `data` relations, using the syntactic dependencies within the instruction semantics to derive the syntactic dependencies between instructions.

For litmus tests, where we can know up-front which instructions may be modified, we would like to avoid generating `IF` events for instructions that cannot be modified. If we naively removed certain `IF` events, however, we would break the correspondence between `po` and `fe`$INVERSE$`;fpo;fe`. This can be worked around by ensuring that every modifiable instruction generates an event which appears in `po`, allowing `fpo` between the modifiable instructions to instead be derived as `fe;po;fe`$INVERSE$. Branches emit a special branch address announce event for this purpose, which is also used to derive the `ctrl` relation. The `fpo` relation can then be modified, replacing `[ISB];fe`$INVERSE$`;fpo` with `[ISB];po;fe`$INVERSE$ and adding `[ISB];po`. The second change ensures that all the transitive edges generated by `[ISB];fe`$INVERSE$`;fpo` followed by `[IF];fe` remain with `fob` and hence `ob`.

A limitation of this approach is it cannot support cases where two threads both attempt to execute the same possibly-modified instruction, as in the SM.F+ic and FOW tests.

Validation results

First, to check for regressions, we ran the operational model on all the 8950 non-mixed-size tests used for developing the original Flat model (without instruction fetch or cache maintenance). The results are identical, except for 23 tests which did not terminate within two hours. We used a 160 hardware-thread POWER9 server to run the tests.

1104 We have also run the axiomatic model on the 90 basic two-thread tests that do not use Arm release/acquire
1105 instructions (not supported by the ISA semantics used for this); the results are all as they should be. This takes
1106 around 30 minutes on 8 cores of a Xeon Gold 6140.

1107 Then, for the key handwritten tests mentioned in this paper, together with some others (that have also been
1108 discussed with Arm), we ran them on various hardware implementations and in the operational and axiomatic
1109 models. The models' results are identical to the Arm architectural intent in all cases, except for two tests which
are not currently supported by the axiomatic checker.

Test	Arm intent	op. model	ax. model	hardware obs.
CoFF	allow	=	=	42.6k/13G
CoFR	forbid	=	=	0/13G
CoRF+ctrl-isb	allow	=	=	3.02G/13G
SM	allow	=	=	25.8G/25.9G
SM+cachesync-isb	forbid	=	=	0/25.9G
MP.RF+dmb+ctrl-isb	allow	=	=	480M/6.36G
MP.RF+cachesync+ctrl-isb	forbid	=	=	0/13G
MP.FR+dmb+fpo-fe	forbid	=	=	0/13G
MP.FF+dmb+fpo	allow	=	=	447M/13G
MP.FF+cachesync+fpo	forbid	=	=	[F]2.3k/13G
ISA2.F+dc+ic+ctrl-isb	forbid	=	=	0/6.98G
SM.F+ic	allow	=	unsupported	[U]0/12.9G
FOW	allow	=	unsupported	[U]0/7G
MP.RF+dc+ctrl-isb-isb	allow	=	=	[U]0/12.94G
MP.R.RF+addr-cachesync+dmb+ctrl-isb	forbid	=	=	0/6.97G
MP.RF+dmb+addr-cachesync	allow	=	=	[U]0/6.34G

[The hardware observations are the sum of testing seven devices: a Snapdragon 810 (4x Arm A53 + 4x Arm A57 cores), Tegra K1 (2x NVIDIA Denver cores), Snapdragon 820 (4x Qualcomm Kryo cores), Exynos 8895 (4x Arm A53 + 4x Samsung Mongoose 2 cores), Snapdragon 425 (4x Arm A53), Amlogic 905 (4x Arm A53 cores), and Amlogic 922X (4x Arm A73 + 2x Arm A53 cores). U: allowed but unobserved. F: forbidden but observed.]
1110

1111 Our testing revealed a hardware bug in a Snapdragon 820 (4 Qualcomm Kryo cores). A version of the first
1112 cross-thread synchronisation test of §3.3.3 but with the full cache synchronisation (MP.RF+cachesync+ctrl-isb)
1113 exhibited an illegal outcome in 84/1.1G runs (not shown in the table), which we have reported. We have also seen
1114 an anomaly for MP.FF+cachesync+fpo, currently under investigation by Arm. Apart from these, the hardware
1115 observations are all allowed by the models. As usual, specific hardware implementations are sometimes stronger.

1116 Finally, we ran the 1456 new instruction-fetch diy tests on a variety of hardware, for around 10M iterations each,
1117 and in the operational model. The model is sound with respect to the observed hardware behaviour except for
1118 that same Snapdragon 820 device.

1119 3.7 Related Work

1120 To the best of our knowledge, no previous work establishes well-validated rigorous semantics for any systems
1121 aspects, of any current production architecture, in a realistic concurrent setting.

1122 The closest is Raad et al.'s work on non-volatile memory, which models the required cache maintenance for
1123 persistent storage in ARMv8-A [42], as an extension to the ARMv8-A axiomatic model, and for Intel x86 [43] as
1124 an operational model, but neither are validated against hardware. In the sequential case, Myreen's JIT compiler
1125 verification [44] models x86 icache behaviour with an abstract cache that can be arbitrarily updated, cleared
1126 on a `jmp`. For address translation, the authoritative Arm-internal ASL model [4, 5, 6], and Sail model derived
1127 from it [7] cover this, and other features sufficient to boot an OS (Linux), as do the handwritten Sail models for
1128 RISC-V (Linux and FreeBSD) and MIPS/CHERI-MIPS (FreeBSD, CheriBSD), but without any cache effects. Goel
1129 et al. [45, 46] describe an ACL2 model for much of x86 that covers address translation; and the Forvis [47] and
1130 RISCV-PLV [48] Haskell RISC-V ISA models are also complete enough to boot Linux. Syeda and Klein [49, 50]
1131 provide an somewhat idealised model for ARMv7 address translation and TLB maintenance. Komodo [33] uses a
1132 handwritten model for a small part of ARMv7, as do Guanciale et al. [34, 35]. Romanescu et al. [51, 52] do discuss

1133 address translation in the concurrent setting, but with respect to idealised models. Lustig et al. [53] describe a
1134 concurrent model for address translation based on the Intel Sandy Bridge microarchitecture, combined with a
1135 synopsis of some of the relevant Linux code, but not an architectural semantics for machine-code programs.

1136 3.8 Conclusion

1137 The mainstream architectures are the most important programming languages used in practice, and their systems
1138 aspects are fundamental to the security (or lack thereof) of our computing infrastructure. We have established a
1139 robust semantics for one of those systems aspects, soundly abstracting the hardware complexities to a manageable
1140 model that captures the architectural intent. This enables future work on reasoning, model-checking, and
1141 verification for real systems code.

1142 Acknowledgements

1143 This work would not have been possible without generous technical assistance from Arm. We thank Richard
1144 Grisenthwaite, Will Deacon, Ian Caulfield, and Dave Martin for this. We also thank Hans Boehm, Stephen
1145 Kell, Jaroslav Ševčík, Ben Titzer, and Andrew Turner, for discussions of how instruction cache maintenance is
1146 used in practice, and Alastair Reid for comments on a draft. This work was partially supported by EPSRC grant
1147 EP/K008528/1 (REMS), ERC Advanced Grant 789108 (ELVER), an ARM iCASE award, and ARM donation funding.
1148 This work is part of the CIFV project sponsored by the Defense Advanced Research Projects Agency (DARPA)
1149 and the Air Force Research Laboratory (AFRL), under contract FA8650-18-C-7809. The views, opinions, and/or
1150 findings contained in this paper are those of the authors and should not be interpreted as representing the official
1151 views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

1152 Chapter 4

1153 Instruction fetch: operationally

1154 4.1 Shape of the model

1155 Need for DC+IC to be separate components that compose together nicely Modelling caches and buffers explicitly
1156 (e.g. a uarch-style model)

1157 4.2 Extending flat

1158 Draw out the actual state List all the new transitions Justify each

1159 Chapter 5

1160
Instruction fetch: axiomatically

1161
5.1 Candidate model

1162 New events and built-in relations

1163
5.2 The axioms

1164 Describe, in detail, the diff from Will's axiomatic model including Christopher's semantics-preserving transforma-
1165 tion of the model

1166
5.3 Executing the model in Isla

1167 (Probably just cite the cav isla paper here)

1168 Chapter 6

1169
Instruction fetch: validation

1170
6.1 Executable operational semantics

1171
6.1.1 Making the model executable

1172
6.2 Extension to isla-axiomatic

1173 Mostly a re-cap of Alasdair's CAV'21 paper.

1174
6.3 Hardware testing

1175
6.3.1 Custom harness

1176 A brief re-cap of the original harness I built, and how it worked.

1177 https://github.com/bensimner/rems-ifetch-harness

1178
6.3.2 Extending herdtools

1179 Mostly a re-cap of work done by Luc Maranget to get diy7 and litmus7 running.

1180
6.3.3 Results from hardware

1181
6.4 Correspondence between the models

Chapter 7

Pagetables and the VMSA

This chapter is based, in part, on: Chapter D5 of the Arm Architecture Reference Manual DDI 0487H.a; and, Relaxed virtual memory in Armv8-A [54] by Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell, published in the proceedings of the 31st European Symposium on Programming (ESOP, 2022).

7.1 Introduction

Modern computers heavily rely on *virtual memory* to enforce security boundaries: hypervisors and operating systems manage mappings from virtual to physical addresses in order to restrict the access individual processes and guest operating systems have to the underlying physical memory, and to memory-mapped devices. With the endemic use of memory-unsafe languages, even for critical infrastructure, understanding and verifying the programs which manage virtual memory mappings is more vital than ever, driving current interests in hypervisors. The virtual machines those hypervisors enable are the key pieces of software which have become solely responsible for implementing such critical security properties.

The following chapters focus on these aspects of the architecture, on virtual memory and virtualisation and the software they enable, with the aim of giving a precise formal semantics for the purpose of verifying real systems software which use those features.

I first give a description of the sequential behaviour of Arm's virtual memory (this chapter); then describe the *relaxed* behaviours and any open questions about Arm's virtual memory (§8); give our precise axiomatic semantics that capture these behaviours (§9); give an overview of the tooling and validation of the given model(s) (§10); and, finally, a sketch of an equivalent operational semantics (§11).

This chapter overview The remainder of this chapter will give: a brief overview of Arm's virtual memory systems architecture (§7.2); a detailed description of the Arm translation table format (§7.3); an overview of the multiple stages of translation (§7.4), and the different translation regimes (§7.5); a detailed explanation of the official Arm translation table walk pseudocode (§7.6); and finally a discussion on the existence and purpose of translation lookaside buffers (§7.7). This chapter does not present any new contributions or novel research, instead, it is a brief but necessary overview of the required architectural features.

7.2 Virtual Memory

Armv8-A8's *virtual memory system architecture* (or *VMSA*) defines the virtual memory and virtulisation features of the Arm architecture. Its structure is described, in detail, in Chapter D5 of the Arm Architecture Reference Manual [1].

Conventionally, we think of memory as being a flat array of bytes, indexed by *physical addresses*. For smaller trusted devices, such as microcontrollers, this may be the end of the story. However larger 'application' class processors rely heavily on virtual addressing: interposing one or more layers of indirection between the accesses using the *virtual* addresses of the program and the 'true' physical addresses of memory. This indirection allows systems running on those processors to:

1. partition the physical resources between different programs, giving access to only those resources that each program needs, and protecting those resources from other programs that do not need to access them;
2. indirect accesses through specific ranges of addresses with convenient numeric values; and
3. update those indirections at runtime to add, remove, or otherwise modify, the mappings to physical memory, to support techniques such as copy-on-write and paging.

To manage all this, typical operating systems splits the programs into distinct *processes* and associates each process with its own virtual to physical mapping. These mappings take the form of partial functions from the process' own (virtual) addresses to the real hardware physical addresses along with some permissions:

$$\texttt{translate} : \texttt{VirtualAddress} \rightharpoonup \texttt{PhysicalAddress} \times 2^{\{\texttt{Read},\texttt{Write},\texttt{Execute}\}}$$

Note that this is a simplification. A more accurate `translate` function is given later on **TODO: ?REF?**.

Typically an operating system would create one such mapping for every process, partitioning the physical memory into disjoint subsets of physical addresses (the *range* of the translate function), and would allocate some convenient numeric values to be the virtual addresses the process interacts with (the *domain* of the translate function). Having this separation allows the processes to be given convienently aligned contiguous chunks of virtual address space even if the underlying physical resources are highly fragmented, or, in the case of paging, potentially not present in memory at all. Additionally, operating systems can provide many processes with mappings to the same physical resource (such as memory-mapped devices) and control which processes have access to such devices at any point in time.

These mappings give rise to separate *address spaces* for each process. The diagram in Figure 7.1 illustrates an example, with two processes named `P0` and `P1` each with their own virtual address space. The left-hand side shows a representation of the 'memory' as the processes see it, with the memory split into *pages* (fixed-size blocks of contiguous addresses). The right-hand side is the equivalent representation of the actual physical memory, with each *physical* page of the available RAM. Note that this diagram shows the virtual address space as being smaller than the physical one, but in general, they may be the same size, or the virtual address space may be even larger than the physical space.

If we assume each page has size `0x1000` then page `1` contains addresses `0x1000` to `0x1FFF` inclusive, and we can interpret the diagram like so:

- ▷ For `P0`:
  - virtual addresses in pages `1`, and `3` are unmapped.
  - virtual addresses in pages `0` and `2` map to physical addresses in physical page `1`.
  - virtual addresses in page `4` map to physical addresses in physical page `5`.
- ▷ For `P1`:
  - virtual addresses in pages `0` and `4` are unmapped.
  - virtual addresses in page `1` map to physical addresses in physical page `5`.
  - virtual addresses in page `2` map to physical addresses in physical page `7`.
  - virtual addresses in page `3` map to physical addresses in physical page `8`.

For example, if process `P0` reads the address `0x2305`, it will actually read from the physical location `0x1305`, since virtual page `2` was mapped to physical page `1` in `P0`'s address space.

Each address space corresponds to a distinct `translate` function. Note that these mappings may be: non-injective (contain *aliasing* of multiple virtual addresses to the same physical address); partial (where some virtual addresses do not map to a physical address at all); or overlapping with other processes' address spaces, in either the domain (for example, the physical page `5` is mapped in both `P0` and `P1`), or range (for example, the virtual page `2` is mapped in both `P0` and `P1` but to different physical pages), or both.

Large application-class processor architectures, such as Armv8-A, often provide hardware support in the form of the *memory management unit* (the MMU), which, once configured by software, will automatically perform the translation from virtual to physical addresses. Software is then required to manage a set of translation functions, and is responsible for ensuring the correct translation function is being used by the MMU whenever a context switch occurs, and handle any *faults* that the MMU generates.

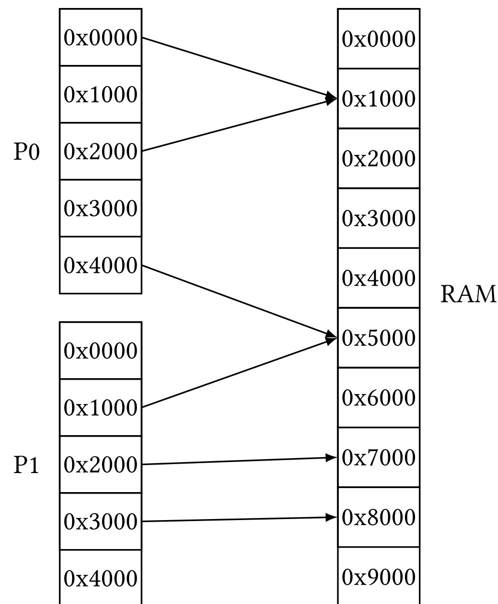


Figure 7.1: Example virtual and physical address spaces for two processes.

1265 ## 7.3 Arm Translation Tables

1266 Software configures the MMU through the creation and modification of sets of *translation tables* (also referred to
1267 as *page tables*) for each of the translation functions.

1268 The translation tables form an in-memory tree data structure which encode the (partial) `translate` function.
1269 Software creates and maintains these trees, and tells the MMU which tree (and so which translation function) to
1270 use at runtime. The hardware then reads from this tree structure to perform the translation, or from one of the
1271 various caching structures described in **TODO: ?REF?**, whenever the process reads from, or writes to, memory.

1272 A pointer to the root of the tree is stored in the TTBR ("Translation table base register") register (or rather, one
1273 of the various base registers described in more detail in **TODO: ?REF?**), and this determines which translation
1274 function is currently in use by that processor.

1275 Each node in the tree is a page-aligned chunk of memory which is treated as an array of 64-bit entries. Each entry
1276 is responsible for mapping some fixed part of the domain of the translation function, with the root table mapping
1277 the entire address space.

1278 The tree is separated into different *levels.* with a root table pointed to by the base register and each subsequent
1279 child tree increases in level going deeper into the tree. Typically the root is at level 0 with a maximum depth of 4
1280 (down to level 3), but the various configurations are discussed in the next section.

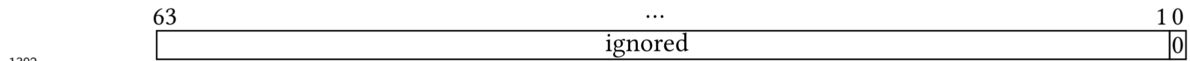
1281 ### 7.3.1 Translation table format

1282 Arm's virtual memory system architecture is highly configurable. Writing to the SCTLR ("System control register")
1283 and TCR ("Translation control register") system registers allow the software developer to choose a configuration
1284 from a whole host of various options. To give a flavour of this configurability I list some of the configuration bits,
1285 some of which will be discussed in more detail in the next chapter; these include: the size of virtual addresses; the
1286 number of levels in the tree; the starting level; the size of a single page (or in Arm terminology, the size of the
1287 *translation granule*); the number of ASIDs and VMIDs; alignment requirements; memory attributes for hardware
1288 walks; enabling hardware management of access flags and dirty bits; write-execute-never permissions; and so on.
1289 To simplify things, in this dissertation, we consider just one common configuration, the one currently used by the
1290 Linux kernel: a tree of translation tables with maximum depth 4, with 4KiB pages with 48-bit addresses, unless
1291 explicitly stated otherwise.

1292 Each node is a table of 512 64-bit entries, bound as one 4096-byte block of memory. Each table controls the
1293 mapping of a fixed range of the virtual address space. This range is split into 512 equal slices, with each entry
1294 responsible for its slice. Each of those entries can be one of:

1. An *invalid* entry, which indicates that this slice of the domain is unmapped;
2. A *table* entry, pointing to a next-level table (a child tree) which recursively maps this slice of the domain; or
3. A *page* (last-level) or *block* (non-last-level) entry which defines a single fixed-size mapping for this slice of the domain.

Invalid entries An invalid entry is defined by `bit[0]` of the entry being `0`. The top 63 bits are ignored by hardware, and software is free to use those bits to store any metadata it wishes. Invalid entries may exist at any level in the tree.



Block or page entries Block and page entries are similar to each other; both create a mapping for a contiguous slice of the domain mapped by the entry, encoded as an output address (OA) with some metadata (including access permissions, memory type, and some software-defined bits).

The OA is aligned to the size of the slice of the domain being mapped. For page entries, the OA is aligned on a page boundary. A block entry's OA at level 2 would be 2MiB aligned, and a block entry's OA at level 1 would be GiB aligned. This corresponds to the hardware reserving `bits[n:12]` of the entry to be `0` depending on how deep the entry is: at level 1 `n==30`; at level 2 `n==21`; and at level 3 `n==12`.

Block entries can exist at levels 1 and 2. Page entries can only exist at level 3.

For block entries `bit[1]` is `0`, for page entries `bit[1]` is 1.

Metadata (access permissions, shareability, memory type) are encoded into the `attrs` bits.

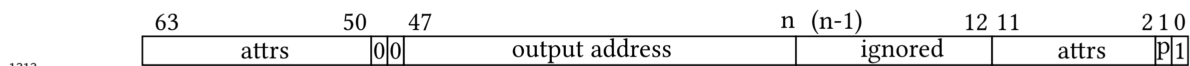
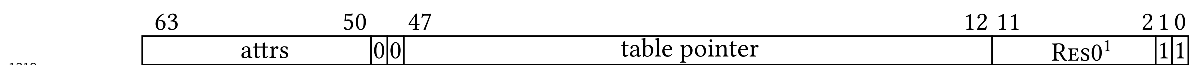


Table entries A table entry contains a page-aligned pointer to a child table, but can also contain similar metadata as the block or page entry, including access permissions (read/write/execute), which are combined with any permissions from the child table.

Table entries are allowed only at levels 0–2.



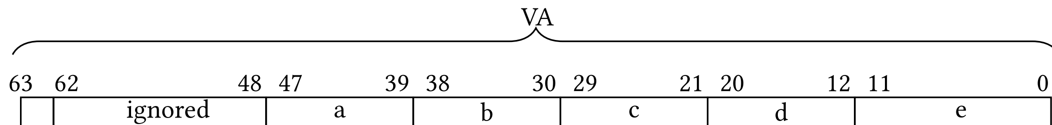
7.3.2 The Arm translation table walk

When the processor executes an instruction which takes an address, such as the Arm `LDR` or `STR` instructions, those addresses are virtual (addresses used by instructions are always virtual addresses). The hardware converts each virtual address to a physical address, and the MMU performs this conversion.

To do this, the MMU reads the `TTBR` to get the currently in-use tree of translation tables. Then the MMU itself reads memory and walks the tree (except when it can read from a previously cached translation, as described in the next chapter) effectively computing the partial `translate` function the tree encodes, producing the physical address and any permissions, or reporting a fault back to the processor if the virtual address was unmapped, or if the permissions forbid the requested operation.

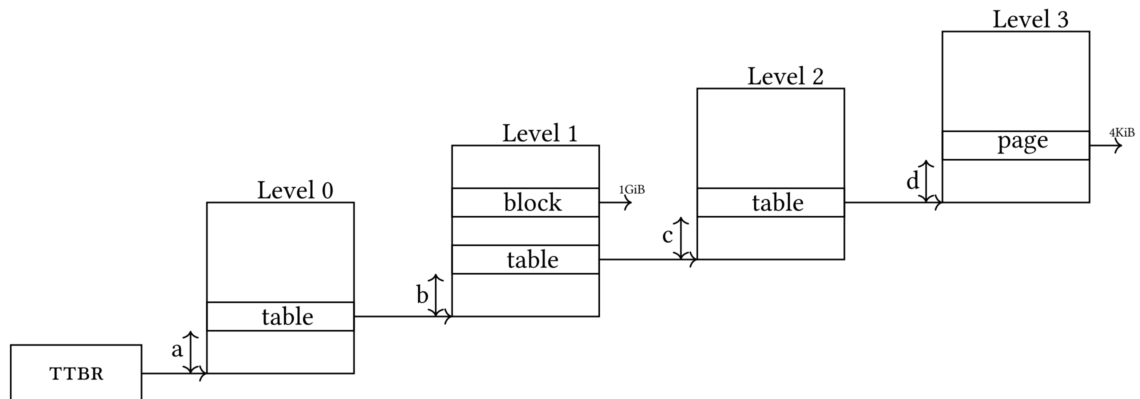
Walk overview The hardware walker first slices up the input virtual address into chunks: the most-significant bit is used to determine which base register to use (see §7.5); the next 15 bits are typically ignored by hardware; the rest of the address is split into 9–bit fields which we refer to as fields *a*–*d*, with the remaining bits as field *e*. Fields *a*–*d* are used for indexing into the tables; and field *e* is the offset in the page, which is added to the final output address.

[1] The Arm architecture requires these bits are 0 and are reserved for future use.



The walk then proceeds, with the MMU taking the following steps:

1 Read the `TTBR` register.
2 Perform a 64-bit single-copy atomic read of `Mem[TTBR+8*a]` to read the entry in the Level 0 table. Call the result `L0entry`.
   a If `L0entry[0]` is 0 (that is, it's an invalid entry) then report a fault back to the processor.
   b Otherwise if `L0entry[1]` is 0 then report a fault back to the processor (top-level tables cannot have block mappings).
3 Perform a 64-bit single-copy atomic read of `Mem[L0entry.table_pointer+8*b]` to read the entry in the Level 1 table, which we will call `L1entry`.
   a If `L1entry[0]` is 0 then report a fault back to the processor.
   b If `L1entry[1]` is 0 (it's a block entry):
      i If the access is not permitted (See §7.3.2 "**Access permissions**"), report a fault to the processor.
      ii Otherwise, return the output address (See §7.3.2 "**Computing the final output address**") back to the processor.
4 Perform a 64-bit single-copy atomic read of `Mem[L1entry.table_pointer+8*c]` to read the entry in the Level 1 table, which we will call `L2entry`.
   a If `L2entry[0]` is 0 then report a fault back to the processor.
   b If `L2entry[1]` is 0 (it's a block entry):
      i If the access is not permitted, report a fault to the processor.
      ii Otherwise, return the output address back to the processor.
5 Perform a 64-bit single-copy atomic read of `Mem[L2entry.table_pointer+8*d]` to read the entry in the Level 3 table, which we will call `L3entry`.
   a If `L3entry[0]` is 0 then report a fault back to the processor.
   b Else if `L3entry[1]` is 0, report a fault back to the processor (this encoding is reserved and is treated as invalid).
   c `L3entry[1]` is 1 (it's a page entry):
      i If the access is not permitted, report a fault to the processor.
      ii Otherwise, return the output address back to the processor.



Computing the final output address The output address (OA) of the final descriptor is the start of the range mapped by the entry. The low order bits are all 0 in the output address, and need to be added on to compute the final output address of the translation.

To compute this final output address the MMU takes the OA from the entry, and the level in the tree the entry is at, and 'completes' the address by bitwise appending the remaining fields to create the complete 48-bit output address. Recall that the `OA` field of the block mappings gets wider the deeper in the tree you are, and so for a 1GiB entry the `OA` field is only 18 bits wide but for a 4KiB page entry its `OA` field is the full 36 bits.

▷ For a 1GiB (level 1) block entry; `PA = OA::c::d::e`

▷ For a 2MiB (level 2) block entry; `PA = OA::d::e`

▷ For a 4KiB (level 3) page entry; `PA = OA::e`

Note that this process means that the least-significant 12 bits of the input VA are unchanged and remain the same in the final output PA, regardless of how the translation function is configured.

Access permissions Once the walk is complete, and the final output address calculated, the MMU checks to see whether the requested access is permitted. Each level of the table can contain some access permissions and those permissions get combined at the end to calculate the final permissions.

For data accesses (reading and writing), table entries have an `APTable` field (`bits[62:61]`), and block/page entries have `AP[2:1]`[1] field (`bits[7:6]`). These fields can be decoded using the following table:

Field	When set (1)	When unset (0)
`AP[2]`	Read-only	Read&Write
`AP[1]`	Allow at EL1&0	Allow at EL1 only
`APTable[1]`	Force read-only	No effect on permissions.
`APTable[0]`	Force forbid access at EL0	No effect on EL0 permissions.

For executable permissions, which permit or forbid instruction fetching from some region of memory, there are no dedicated encodings of the access permission bits. Instead, all mappings are executable by default, unless one of the following applies: the region is mapped writeable at EL0, as writable EL0 regions are never executable at EL1; a global `WXN` ("Write-execute-never") configuration bit is set, and the entry was writeable; or, when one of the various translation table entry `XN` ("Execute-never") bits are set. For simplicity, this chapter assumes that execute-never bits are always disabled; see the full description in the Arm ARM **TODO: ?REF?**for more information.

To combine access permissions from the whole walk, the MMU takes the bitwise union of each of the `APTable` fields from each table entry, and then intersects the result with the final `AP[2:1]` field to produce a final set of permissions. Figure 7.2 contains a decoding table for a given table and leaf access permissions, for testing whether a requested access is permitted. If the requested access is not permitted, then the MMU generates a permission fault, which is reported back to the processor.

Faults The MMU may emit one of several fault types during a translation table walk (these are referred to by Arm as the *MMU fault* types):

- ▷ Translation fault.
  These are caused by the mapping being invalid, either because `bit[0]` was 0, or because the descriptor encoding was reserved-as-invalid. Translation faults also result from trying to translate an address that is outside the 48-bit input address range.
- ▷ Permission fault.
  For when the mapping was valid, but the access permissions do not permit the requested access (for example, trying to write to a read-only address).
- ▷ Access flag fault.
  These are generated when hardware management of access flags is disabled and the access flag bit is set.
- ▷ TLB Conflict aborts (see **TODO: ?REF?**).
- ▷ Alignment fault.
  Generated when an operation expects an aligned memory address, but is given a misaligned one, and alignment checking is enabled in the `SCTLR`.
- ▷ Address size fault.
  For when the OA, or `TTBR`, has a value that is out of the physical address range.
- ▷ Synchronous external abort on a translation table walk.
  These are *external aborts* (that come from the system not from the MMU) that happen due to accesses that the MMU generated. For example, if the next-level table field pointed to an address for which there was no memory or device, the system-on-chip would return a fault to the processor.

These faults lead to processor exceptions. The fault type is stored in the `ESR_ELn` ("exception syndrome register") register's `EC` ("exception class") field, and any supplementary information is stored in its `ISS` ("instruction specific syndrome") field (such as which level in the tree the fault came from, whether the originating instruction was a read or a write, and). Exception handling code can read the `ESR` register to determine the fault type and cause,

[1] Block/page entries do not store the entire `AP` field but only `AP[2:1]`. `AP[0]` is not present in AArch64.

APTable[1]	APTable[0]	AP[2]	AP[1]	EL1			EL0		
				R	W	X	R	W	X
0	0	0	0	✓	✓	✓	×	×	✓
0	0	0	1	✓	✓	×	✓	✓	✓
0	0	1	0	✓	×	×	×	×	✓
0	0	1	1	✓	×	✓	✓	×	×
0	1	0	0	✓	✓	✓	×	×	✓
0	1	0	1	✓	✓	×	×	×	✓
0	1	1	0	✓	×	×	×	×	✓
0	1	1	1	✓	×	✓	×	×	×
1	0	0	0	✓	×	✓	×	×	✓
1	0	0	1	✓	×	×	✓	×	✓
1	0	1	0	✓	×	×	×	×	✓
1	0	1	1	✓	×	✓	✓	×	×
1	1	0	0	✓	×	✓	×	×	✓
1	1	0	1	✓	×	×	×	×	✓
1	1	1	0	✓	×	×	×	×	✓
1	1	1	1	✓	×	✓	×	×	×

Figure 7.2: Merging Access Permissions (Stage 1, EL1&0).
Entries in **red** highlight differences from the `APTable=00`.

and can read the `FAR_ELn` ("fault address register") to determine the virtual address which triggered the fault, and handle the fault appropriately.

Memory Attributes The processor does not necessarily know what is located at any physical address. There may be some dynamic random-access memory (DRAM, what one would generally consider 'memory'), but there may also be other memory-mapped devices, or non-volatile memory, or other peripherals, or possibly nothing at all.

To help accommodate this, hardware allows software to mark regions of memory as one of either *device* memory, *normal cacheable* memory, or normal *non-cacheable* memory, using the translation tables.

The desired memory type is determined from the `AttrIndx` field (`bits[4:2]`) in block and page entries. Instead of being directly encoded into this field, Arm chose to have the actual attributes stored in a separate register: the `MAIR` ("Memory attribute indirection register") register. The `MAIR` stores an array of eight 8-bit fields each of which contains an encoding of a memory type. The `AttrIndx` field in the entry is an integer in the range 0–7, which is the index of the field in the `MAIR` register to use.

This indirection means that the final result of translation depends not only on the value of the final leaf entry in memory, but on the value of certain system registers, such as the `MAIR`, at that time of the translation table walk.

Below are the three most common encodings for a `MAIR` field, and the ones that will be useful later when discussing tests:

- ▷ `0b0000_0000`: device memory.
- ▷ `0b0100_0100`: normal non-cacheable memory.
- ▷ `0b1111_1111`: normal cacheable memory, inner&outer write-back non-transient, read&write-allocating.

Memory locations marked as device tell the hardware that reads or writes to those locations may have side-effects. This means hardware treats those locations differently: there will be no speculative instruction fetches, reads, or writes to those locations; writes to those locations will not *gather* into larger writes; reads and writes to those locations will not re-order with respect to others; those locations generally will not get cached; and other thread-local optimizations get disabled. Note that Arm define a wide range of device memory types, allowing

the systems programmer to selectively re-enable some of the previously described behaviours to enable better performance where they deem it safe to do so.

For normal memory the software can choose between *cacheable* or non-*cacheable* memory. Arm provide a range of different options for the cacheability:

- ▷ non-cacheable
- ▷ write-back cacheable
- ▷ write-through cacheable

As with other features, there is a wide scope for configuration: separately configuring inner (L1,L2) and outer (L3) caches, and adding cache allocation hints (allocating on reads, writes or both).

As we will see later (**TODO: ?REF?**), the ability to change cacheability, or even have multiple aliases with different cacheability attributes, give rise to interesting behaviours and security considerations.

7.4 Virtualisation and a second stage of translation

So far this chapter has focused on operating systems and processes. However, modern systems isolate not just processes within an operating system but entire operating systems from one another, within a hypervisor.

To do this, software uses the virtual memory abstraction again, adding an extra layer. This layer, like the previous one, is supported by hardware. Processes use virtual addresses which are converted to *intermediate physical* (also sometimes known as *guest physical*) addresses using the operating system's configured translation tables but then these intermediate physical addresses (IPAs) go through another round of translation to convert those IPAs into the final physical address.

Arm calls these *stages* of translation, and the MMU supports both stages and can perform the full translation from virtual to physical (via the intermediate physical) address.

This means software must manage two sets of translation tables: operating systems manage the *stage 1* tables to convert VAs to IPAs; and hypervisors manage *stage 2* tables to convert those IPAs to PAs; this gives two separate `translate` functions, which the MMU composes together at runtime:

$$\begin{aligned}\texttt{translate_stage1} &: \texttt{VirtualAddress} \rightharpoonup \texttt{IPA} \times \texttt{Permissions} \times \texttt{MemoryType}\\ \texttt{translate_stage2} &: \texttt{IPA} \rightharpoonup \texttt{PhysicalAddress} \times \texttt{Permissions} \times \texttt{MemoryType}\end{aligned}$$

Hypervisors (running at EL2) can configure the stage 2 translate function by creating translation tables with a similar format as before and then storing a pointer to the root of this tree in the `VTTBR` ("Virtualization translation table base register") register. The MMU will read the `VTTBR` whenever it needs to perform a second-stage translation to convert an IPA to a PA, and will do the translation table walk over that tree in much the same way as described earlier for (what we can now call) the first-stage translation.

This results in two address spaces, a virtual address space and an intermediate-physical address space. Figure 7.3 contains an example layout of these address spaces for a machine running three processes (`P0`,`P1`,`P2`) in two operating systems (`OS0`,`OS1`). As with the earlier diagram in Figure 7.1, each column is a (set of) address spaces, with transformations between them defined by their respective translation functions. On the left-hand side are the virtual address spaces of the various processes, whose virtual addresses are translated (using the translation tables pointed to by the `TTBR` register) into intermediate-physical addresses in the central address spaces (for the respective OS). Those IPAs are then translated (using the `VTTBR`) into the final physical address.

Concretely, if `P1` reads from address `0x1001`, it will be translated into the IPA `0x3001` in `OS0`'s address space, which then gets translated again, and the processor will actually read from RAM at location `0x6001`.

Differences in the translation table format from stage 1 Stage 2 translation tables are similar to their stage 1 counterparts, but there are some minor differences:

- ▷ Stage 2 table entries do not have any additional attributes, and so do not have an `APTable` field.
- ▷ Stage 2 `AP` field (called `S2AP`) has a slightly different (and simpler) format, see Figure 7.4.
- ▷ Stage 2 block and page entries do not have a `MemAttrIndx` field but rather encode the memory type directly into the `MemAttr` field `bits[5:2]` (see the full description in the Arm ARM [1, D5-4874] for all possible encodings):

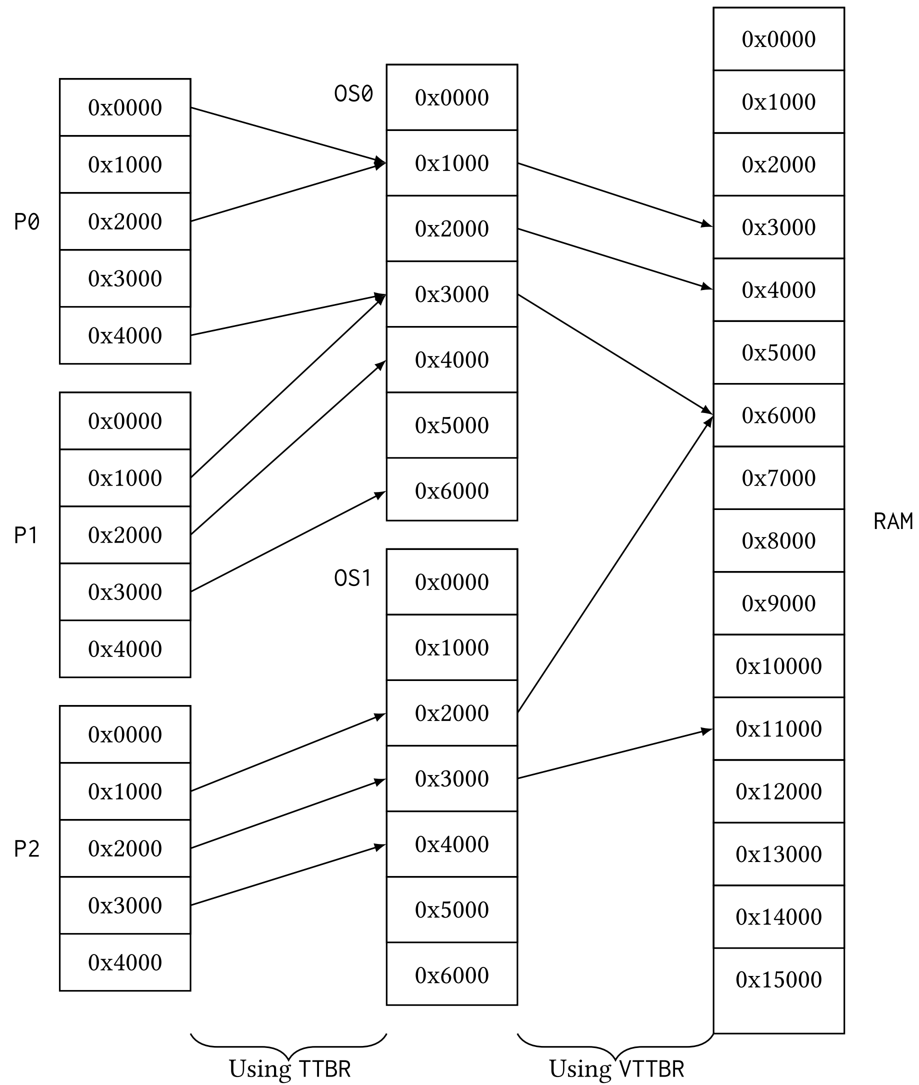


Figure 7.3: Example virtual, intermediate physical, and physical address spaces for three processes running on two operating systems.

Field	When set (1)	When unset (0)
S2AP[1]	Writeable	not Writeable
S2AP[0]	Readable	not Readable

Figure 7.4: **S2AP** field encoding.

1488 – 0b0000: Device memory.
1489 – 0b0101: Normal non-cacheable.
1490 – 0b1111: Normal write-back inner&outer cacheable.

1491 These are interesting as they mean that the stage 1 and stage 2 attributes (permissions and memory types) must
1492 be *combined* in order to produce the final output. This combination is not just a case of letting stage 2 overrule the
1493 stage 1 settings but rather that both stages get a veto: if stage 1 sets the memory type to be device or non-cacheable
1494 then it overrules what stage 2 sets. Similarly, if stage 1 permissions forbid an access then the stage 2 permissions
1495 cannot overrule that.

1496 **Second-stage translations during a first-stage walk** There is a complication with the story so far. The
1497 stage 1 tables are created by the operating system, which is using an intermediate physical address space, not a
1498 physical one. The writes the OS does to the tables will be translated, as they are normal data writes. But, the
1499 tables themselves contain references to other tables, and those entries will be intermediate physical addresses,
1500 and so, they must also be translated, including the value of the TTBR itself.

1501 In our assumed configuration of 4KiB pages and 4 levels of translation, this leads to a maximum of 24 memory
1502 accesses to perform the translation: 4 reads of stage 1 translation tables, 16 reads of stage 2 translation tables
1503 during those stage 1 walks, and a final 4 reads of the stage 2 translation tables to translate the output IPA into the
1504 final PA.

1505 ## 7.5 Translation regimes

1506 As mentioned earlier, there are multiple translation table base registers. Each of them defines a translation
1507 function, pointing to the root of the tree of translation tables which define it. These translation functions are then
1508 composed together into various translation *regimes*, each defining the set of translation functions (and therefore
1509 which translation table base registers) which will be used for translations done by the processor.

1510 Arm define a set of these translation regimes. Figure 7.5 gives an overview of three of the most common regimes,
1511 which are:

1512 ▷ EL1&0 (two-stage)
1513 – For programs executing at EL0 or EL1 when virtualisation (at EL2) is enabled.
1514 – VAs with the high bit set are translated into IPAs using the EL1-configured register, TTBR1_EL1.
1515 VAs are typically split into 'high' and 'low' regions with different translations, primarily used for
1516 separate kernel and user address spaces.
1517 – VAs without the high bit set are translated into IPAs using the EL1-configured register, TTBR0_EL1.
1518 – IPAs are translated to PAs using the EL2-configured VTTBR_EL2 register.
1519 ▷ EL1&0 (single-stage)
1520 – For programs executing at EL0 or EL1 when virtualisation (at EL2) is disabled.
1521 – VAs with the high bit set are translated into PAs using the EL1-configured register, TTBR1_EL1.
1522 – VAs without the high bit set are translated into PAs using the EL1-configured register, TTBR0_EL1.
1523 ▷ EL2
1524 – For programs executing at EL2.
1525 – VAs without the high bit set are translated into PAs using the EL2-configured register, TTBR0_EL2.
1526 – VAs with the high bit set are always unmapped.

1527 Which translation regime is being used is defined by various system registers and the current system state.

1528 ▷ Translations at EL1 or EL0 use one of the EL1&0 regimes.
1529 ▷ Translations at EL2 use the EL2 regime.
1530 ▷ TCR_EL2 (set at EL2) determines whether the EL1&0 is a single-stage or two-stage regime.
1531 ▷ TTBR0_EL1, TTBR1_EL1 determine the stage 1 of the EL1&0 regimes, and can be set at EL1 or higher.
1532 ▷ TTBR0_EL2 determines the stage 1 of the EL2 regime, and can only be set at EL2 or higher.

1533 ▷ VTTBR_EL2 determines the stage 2 of the EL1&0 regime, and can only be set at EL2 or higher.

1534 Arm define a wide range of other regimes, see the Arm ARM **TODO: ?REF?**. For simplicity, we ignore secure
1535 modes, including all of EL3.

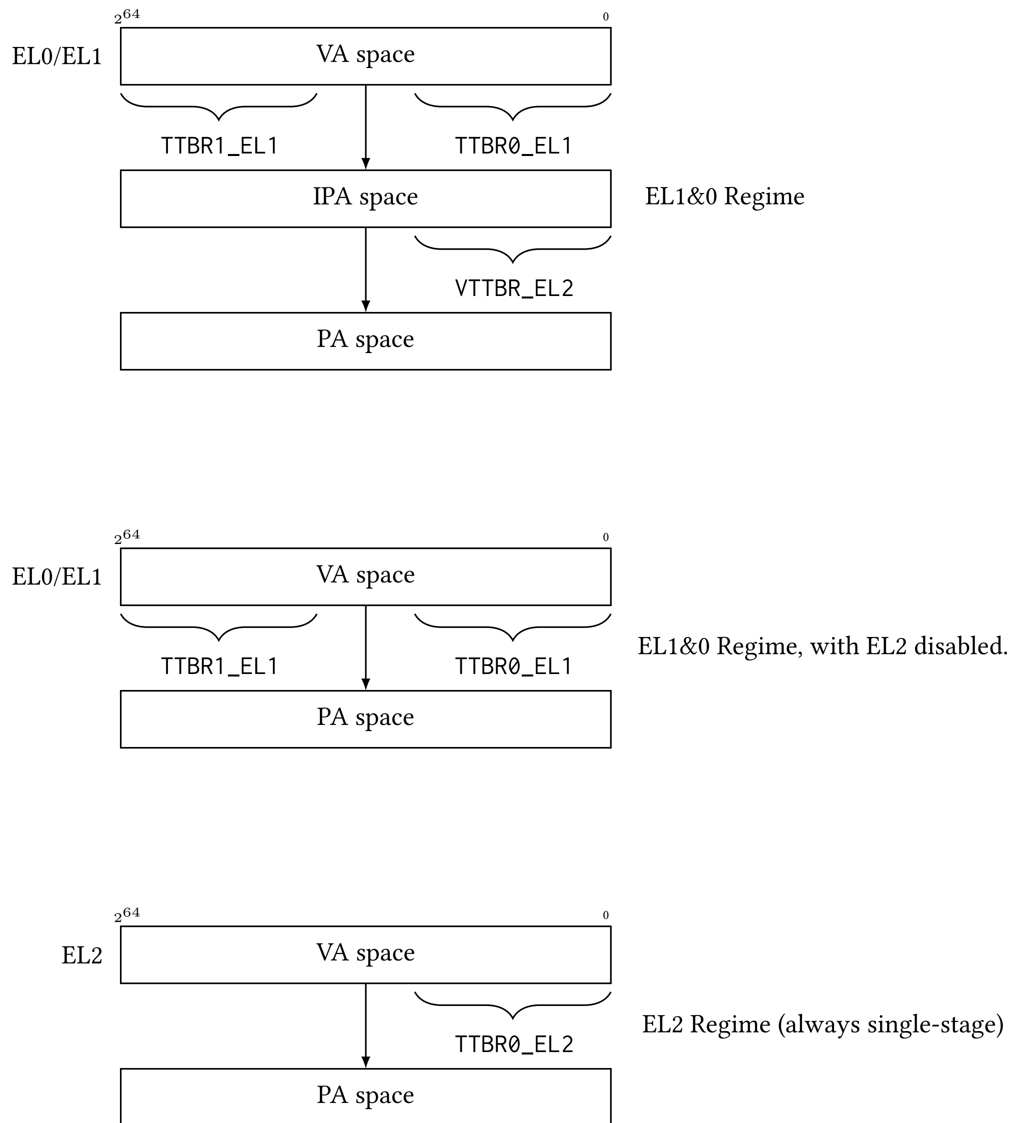


Figure 7.5: Translation regimes that apply to EL0,EL1, and EL2.

1536
7.6 Arm pseudocode

1537 It is now useful to examine the official Arm pseudocode, especially those parts that relate to memory events.

1538 We will do this in three steps: first, by looking at the pseudocode that is executed for an Arm store instruction;
1539 following the memory accesses that it performs down to any translations it performs; finally looking at the Arm
1540 translation table walker in full. There is a lot of detail infused throughout the Arm psueocode, so in this section
1541 we shall focus on the most pertient parts, and give some idea of what detail is omitted.

1542
7.6.1 The lifecycle of a store

1543 Arm give precise executable semantics for every instruction in their domain-specific Architecture Specification
1544 Language (ASL). This ASL code defines the sequential intra-instruction behaviour of each instruction, including
1545 memory accesses, and any translation table walks they perform.

```
1        bits(64) address;
2        bits(datasize) data;
3
4        if HaveMTE2Ext() then
5            SetTagCheckedInstruction(tag_checked);
6
7        if n == 31 then
8            if memop != MemOp_PREFETCH then CheckSPAlignment();
9            address = SP[];
10       else
11           address = X[n];
12
13       if ! postindex then
14           address = address + offset;
15
16       case memop of
17           when MemOp_STORE
18               if rt_unknown then
19                   data = bits(datasize) UNKNOWN;
20               else
21                   data = X[t];
22               Mem[address, datasize DIV 8, acctype] = data;
23
24           when MemOp_LOAD
25               data = Mem[address, datasize DIV 8, acctype];
26               if signed then
27                   X[t] = SignExtend(data, regsize);
28               else
29                   X[t] = ZeroExtend(data, regsize);
30
31           when MemOp_PREFETCH
32               Prefetch(address, t<4:0>);
33
34       if wback then
35           if wb_unknown then
36               address = bits(64) UNKNOWN;
37           elsif postindex then
38               address = address + offset;
39           if n == 31 then
40               SP[] = address;
41           else
42               X[n] = address;
43
```

Figure 7.6: Arm “STR (immediate)” ASL code.

1546 **TODO: the importance of the ASL, and of sequential v concurrent behaviour will already be explained,**
1547 **but recap here anyway?**

1548 Figure 7.6 shows the Arm ASL for the “STR (Immediate)” instruction: `STR Xt,[Xn]`. This instruction writes the
1549 value contained in register `Xt` into the memory location stored in register `Xn`. The figure has some uninteresting
1550 (for this thesis) parts greyed out: those parts that deal with optional extensions such as memory tagging; unknown
1551 register values; register writeback; and, the load and prefetch instructions which use the same ASL code.

1552 The ASL code first reads the virtual address either from the stack pointer (line 9) or by reading register `Xn` (line 11).
1553 It then reads the data from the register `Xt` (line 21), which will be written to memory. Finally, it performs the
1554 store itself using the `Mem[]` function (line 22).

1555 7.6.2 Writes to memory

1556 The `Mem[]` function is responsible for checking alignment and performing each memory access the instruction
1557 does. The ASL for `Mem[]` can be found in Figure 7.7.

1558 It does some alignment checks, and then calls `MemSingle[]` once for each single copy atomic write the access
1559 performs.

1560 For example, for a fully aligned store, it calls `MemSingle[]` just once (lines 37 or 57), and, for a misaligned store, it
1561 will call `MemSingle[]` once for each byte (line 51).

1562 The `MemSingle[]` call then performs the translation, and (if successful), the actual write to memory. Its ASL can be
1563 found in Figure 7.8, with parts for extensions and store pair greyed out. On line 12, it calls `AArch64.TranslateAddress`
1564 to do the translation table walk. If the translation succeeds, then the code calls `PhysMemWrite` (on line 40), an
1565 uninterpreted function with no behaviour in ASL, which represents the actual write to memory. After perhaps
1566 handling any external aborts from the write, the function returns.

```
1  Mem[bits(64) address, integer size, AccType acctype, boolean ispair] = bits(size*8)
     value_in
2      boolean iswrite = TRUE;
3      constant halfsize = size DIV 2;
4      bits(size*8) value = value_in;
5      bits(halfsize*8) lowhalf, highhalf;
6      boolean atomic;
7      boolean aligned;
8      if BigEndian(acctype) then
9          value = BigEndianReverse(value);
10
11     if ispair then
12         // check alignment on size of element accessed, not overall access size
13         aligned = AArch64.CheckAlignment(address, halfsize, acctype, iswrite);
14     else
15         aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
16     if ispair then
17         atomic = CheckAllInAlignedQuantity(address, size, 16);
18     elsif size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
19         if !HaveLSE2Ext() then
20             atomic = aligned;
21         else
22             atomic = CheckAllInAlignedQuantity(address, size, 16);
23     elsif (acctype IN {AccType_VEC, AccType_VECSTREAM}) then
24         // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic
      accesses
25         // 64-bit aligned.
26         atomic = address == Align(address, 8);
27     else
28         // 16-byte integer access
29         atomic = address == Align(address, 16);
30
31     if !atomic && ispair && address == Align(address, halfsize) then
32         single_is_aligned = TRUE;
33         <highhalf, lowhalf> = value;
34         AArch64.MemSingle[address, halfsize, acctype, single_is_aligned, ispair] =
     lowhalf;
35         AArch64.MemSingle[address + halfsize, halfsize, acctype, single_is_aligned,
     ispair] = highhalf;
36     elsif atomic && ispair then
37         AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
38     elsif !atomic then
39         assert size > 1;
40         AArch64.MemSingle[address, 1, acctype, aligned] = value<7:0>;
41
42         // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned
      Device memory
43         // access will generate an Alignment Fault, as to get this far means the
     first byte did
44         // not, so we must be changing to a new translation page.
45         if !aligned then
46             c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
47             assert c IN {Constraint_FAULT, Constraint_NONE};
48             if c == Constraint_NONE then aligned = TRUE;
49
50         for i = 1 to size-1
51             AArch64.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
52     elsif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
53         <highhalf, lowhalf> = value;
54         AArch64.MemSingle[address, halfsize, acctype, aligned, ispair] = lowhalf;
55         AArch64.MemSingle[address + halfsize, halfsize, acctype, aligned, ispair] =
     highhalf;
56     else
57         AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
58     return;
59
```


Figure 7.7: Arm `Mem[]` write function call ASL code.

```
1  AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned
     , boolean ispair] = bits(size*8) value
2    assert size IN {1, 2, 4, 8, 16};
3    constant halfsize = size DIV 2;
4    if HaveLSE2Ext() then
5        assert CheckAllInAlignedQuantity(address, size, 16);
6    else
7        assert address == Align(address, size);
8
9    AddressDescriptor memaddrdesc;
10   iswrite = TRUE;
11
12   memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size)
     ;
13   // Check for aborts or debug exceptions
14   if IsFault(memaddrdesc) then
15       AArch64.Abort(address, memaddrdesc.fault);
16
17   // Effect on exclusives
18   if memaddrdesc.memattrs.shareability != Shareability_NSH then
19       ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);
20
21   // Memory array access
22   AccessDescriptor accdesc;
23   if HaveTME() then
24       accdesc = CreateAccessDescriptor(acctype);
25       accdesc.transactional = TSTATE.depth > 0;
26       if accdesc.transactional && !MemHasTransactionalAccess(memaddrdesc.memattrs)
      then
27           FailTransaction(TMFailure_IMP, FALSE);
28   else
29       accdesc = CreateAccessDescriptor(acctype);
30
31   if HaveMTE2Ext() then
32       if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
33           bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
34           if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
35               AArch64.TagCheckFault(ZeroExtend(address, 64), acctype, iswrite);
36
37   PhysMemRetStatus memstatus;
38   (atomic, splitpair) = CheckSingleAccessAttributes(address, memaddrdesc.memattrs,
      size, acctype, iswrite, aligned, ispair);
39   if atomic then
40       memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
41       if IsFault(memstatus) then
42           HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
43   elsif splitpair then
44       assert ispair;
45       bits(halfsize*8) lowhalf, highhalf;
46       <highhalf, lowhalf> = value;
47
48       memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, lowhalf);
49       if IsFault(memstatus) then
50           HandleExternalWriteAbort(memstatus, memaddrdesc, halfsize, accdesc);
51       memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
52       memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, highhalf);
53       if IsFault(memstatus) then
54           HandleExternalWriteAbort(memstatus, memaddrdesc, halfsize, accdesc);
55   else
56       for i = 0 to size-1
57           memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, value<8*i+7:8*i>);
58           if IsFault(memstatus) then
59               HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
60           memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
61   return;
62
```

Figure 7.8: Arm `MemSingle[]` write function call ASL code.

1567 ### 7.6.3 Translation table walks

1568 It is the AArch64.TranslateAddress function which begins the process that performs the actual translation table
1569 walk, converting the input virtual address to the physical one. The full ASL code is too much to contain in a single
1570 figure, and so it can be found in §7.8 at the end of this chapter. This section will reference the relevant lines from
1571 the translation table walk ASL.

1572 Figure 7.9 is an example trace of the execution of the STR Xt,[Xn] instruction, as it would happen if we were to
1573 execute it from EL1 in the EL1&0 two-stage regime. Each node represents an event in the trace (a memory or
1574 register access), and the arrows between them represent control flow. **TODO: Generate from an actual isla**
1575 **trace rather than by hand? at least to be proper... TODO: Give labels to each event?**

1576 As described before, the instruction starts by reading the Xt and Xn registers, before beginning the call to
1577 AArch64.TranslateAddress.

1578 The events drawn inside the dotted box come from accesses during the call to the translation table walk functions.
1579 It first calls FullTranslate (in AArch64.TranslateAddress, page 58, line 2), which calls S1Translate (in
1580 AArch64.FullTranslate, page 59, line 12), which calls S1Walk (in AArch64.S1Translate, page 60, line 29) to
1581 do the actual first-stage translation table walk. It begins by reading the relevant TTBR register to get the root
1582 table address (in AArch64.S1Walk, page 63, line 9). This is stored in a walkstate struct, which the ASL code
1583 uses to keep track of the state that changes as the walk progresses, notably, the next-level table address and
1584 any accumulated permissions. It then begins the loop to do the walk, starting from the table address read from
1585 the TTBR. On each iteration of the loop, the intermediate-physical address of the entry to be read is computed
1586 (in AArch64.S1Walk, page 63, line 38), and passed through a second stage of translation (in AArch64.S1Walk,
1587 page 63, line 47).

1588 This second stage translation calls S2Walk, which behaves similarly to the S1Walk function, taking the following
1589 steps: it reads the VTTBR (in AArch64.S2Walk, page 67, line 11); computes the (now) physical address of the entry
1590 to read (in AArch64.S2Walk, page 67, line 41); and reads it (in AArch64.S2Walk, page 67, line 44), eventually
1591 calling PhysMemRead (in AArch64.FetchDescriptor, page 69, line 23), which appears as the first R S2 L0 node
1592 in Figure 7.9.

1593 S2Walk continues to loop, each time updating the running walkstate with the next-level table address from
1594 the decoded descriptor (in AArch64.S2Walk, page 67, line 53), until a leaf entry is found. It is either invalid (in
1595 AArch64.S2Walk, page 68, line 65), or, a valid page or block entry (in AArch64.S2Walk, page 68, line 70). These
1596 correspond to the next three R S2 Ln events in the figure.

1597 Assuming the walk did not fail with a fault, the S2Translate function returns with the physical address of
1598 the stage 1 level 0 table. S1Walk can continue, performing a read of the physical memory in the table (in
1599 AArch64.S1Walk, page 63, line 52). From there, S1Walk continues in much the same way as the stage 2 walk did:
1600 computing the current table intermediate-physical address, translating it to get the physical address, performing
1601 the read of memory to get the descriptor, until a leaf entry is found.

1602 This process generates all the events up to, and including, the final stage 1 entry read (the R S1 L3 event),
1603 returning the intermediate-physical address that S1Walk computed.

1604 Finally, FullTranslate calls S2Translate one last time (in AArch64.FullTranslate, page 59, line 22) on the
1605 intermediate-physical address, generating the last Rreg(VTTBR) and R S2 Ln events, and producing the final PA
1606 of the translation.

1607 This output PA is what is passed to the PhysMemWrite of the MemSingle[] call we saw earlier, generating the
1608 final W [pa]=data event in the trace.

1609 ## 7.7 Caching in TLBs

1610 Hardware does not simply perform the (up to) 24 additional memory accesses for every instruction-fetch, read,
1611 or write. This would have an unacceptable performance penalty. Instead, the results of previous translations
1612 of the same address are cached, in specialised structures called *Translation Lookaside Buffers*, or simply TLBs.
1613 These TLBs can store whole translation results, or the separate virtual and intermediate-physcical mappings, or
1614 individual translation table entries, or a mix of the above, which we will explore more in the next chapter.

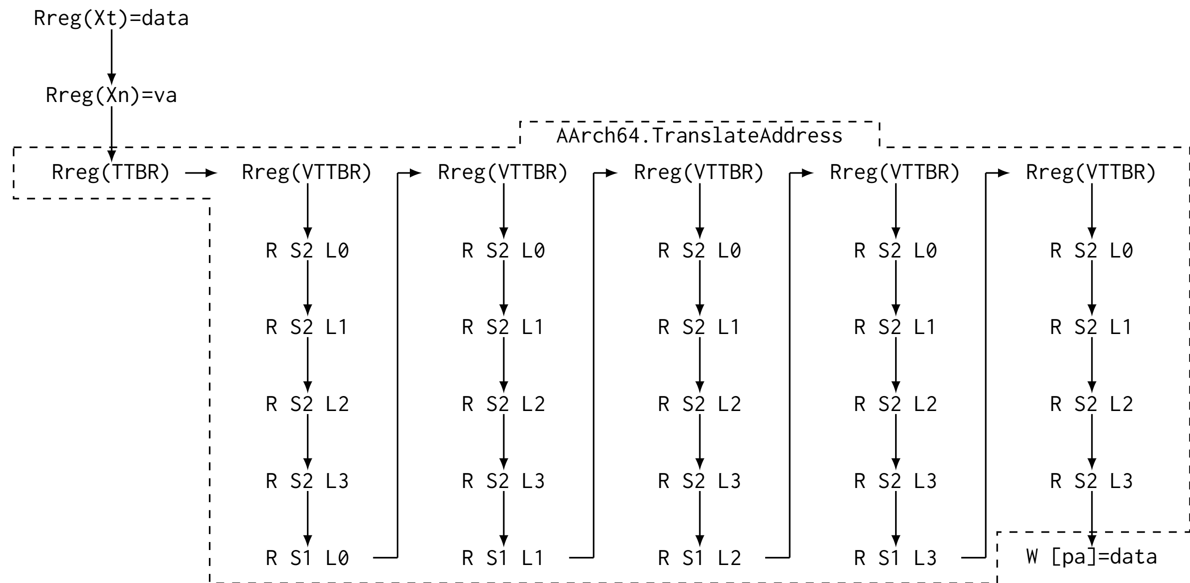


Figure 7.9: Memory and register accesses during a 'STR Xt,[Xn]' instruction.

When the processor translates a virtual address, it first looks for it in the TLB. If there is no entry, then this is called a *TLB miss* and a translation table walk must be performed. The results of this walk are typically then cached in the TLB, so future translations of the same address can directly grab the physical address, memory attributes, and permissions, without needing to do another translation table walk. This process and the various microarchitectural structures are explored more in §8.3.1.

If there is an entry, this is referred to as a *TLB hit*. In this case, the result can be taken directly from the TLB.

Under normal circumstances, the TLB is invisible to userspace programs. However, systems code is expected to manage the TLBs explicitly, using a set of instructions which Arm provide specifically for this purpose: the family of `TLBI` TLB-maintenance instructions. When context switching, the systems software must manually manage the TLB, invalidating stale entries for old mappings out of the cache. The behaviours that arise from reading from potentially stale TLB entries are explored in detail in §8.5.

Address space identifiers TLB misses and TLB maintenance are both expensive operations, and so to reduce the burden, Arm provide a mechanism to permit multiple processes' address spaces to be loaded into the TLB at the same time, by allowing the software to mark each address space with a numeric label. Arm call these *address space identifiers* (or ASIDs).

Entries in the TLB are tagged with the current ASID, and so only that process will see entries in the TLB with that ASID.

The current ASID is encoded in the high order bits of the current `TTBR`. During a context switch, the system software needs only switch to the new translation tables for the new address space of the other process, without doing TLB maintenance, so long as it ensures the ASIDs are distinct.

There are only finitely many ASIDs available (typically it is an 8-bit field), and so eventually TLB maintenance is required to re-use a previously allocated ASID for a new address space. But this happens far less frequently than the context switches themselves. The provided TLB maintenance instructions can target specific ASIDs, avoiding the need to over-invalidate other cached address space translations, preventing a cascade of TLB misses in other processes, further improving the runtime performance for a small amount of additional effort on the software side.

VMIDs Address space identifiers are used only for stage 1 translations. Stage 2 has virtual machine identifiers (VMIDs).

As before, the current VMID is encoded in the `VTTBR_EL2` register, and the TLB entries are additionally tagged with the current VMID (as well as the ASID), and a translation will only use TLB entries that match the current ASID and VMID.

TLB maintenace instructions Arm define a whole family of instructions under the `TLBI` mnemonic. The format for a `TLBI` instruction is a product of fields:

```
TLBI <type><level><broadcast>{,<reg>}

<type> =
  ALL | VMALL | ASID | VA{A|L} | IPAS2
<level> =
  E1 | E2
<broadcast> =
  {IS}
<reg> =
  X0 | X1 | ... | X30
```

Again, see the full description in the Arm manual for a more complete description [1, D5-4915].

The most common, and the ones that will be discussed in the following chapters, are as follows:

- `TLBI VAE1,Xn`: Invalidate this CPU's cached copies of entries used to translate the virtual address in register `Xn`, for the EL1&0 regime, for the current ASID and VMID.
- `TLBI VALE1,Xn`: Invalidate this CPU's cached copies of any last-level entries used to translate the virtual address in register `Xn`, for the EL1&0 regime, for the current ASID and VMID.
- `TLBI VAAE1,Xn`: Invalidate this CPU's cached copies of any last-level entries used to translate the virtual address in register `Xn`, for the EL1&0 regime, for the current VMID, for any ASID.
- `TLBI VAE1IS,Xn`: Invalidate all CPU's cached copies of entries used to translate the virtual address in register `Xn`, for the EL1&0 regime, for the current ASID and VMID.
  (...and equivalent `TLBI VAE2`, `TLBI VALE2`, `TLBI VAE2IS` instructions for virtual addresses in the EL2 regime)
- `TLBI IPAS2E1,Xn`: Invalidate this CPU's cached copies of entries used to translate the intermediate physical address in register `Xn`, for the EL1&0 regime, for the current VMID.
- `TLBI IPAS2LE1,Xn`: Invalidate this CPU's cached copies of any last-level entries used to translate the intermediate physical address in register `Xn`, for the EL1&0 regime, for the current VMID.
- `TLBI IPAS2E1IS,Xn`: Invalidate all CPU's cached copies of entries used to translate the intermediate physical address in register `Xn`, for the EL1&0 regime, for the current VMID.
- `TLBI VMALLE1`: Invalidate this CPU's cached copies of entries for the EL1&0 regime, for the current VMID.
- `TLBI VMALLE1IS`: Invalidate all CPU's cached copies of entries for the EL1&0 regime, for the current VMID.
- `TLBI ALLE1`: Invalidate this CPU's cached copies of entries for the EL1&0 regime, for any ASID or VMID.
- `TLBI ALLE1IS`: Invalidate all CPU's cached copies of entries for the EL1&0 regime, for any ASID or VMID.
  (...and equivalent `TLBI ALLE2`, and `TLBI ALLE2IS` instructions for the EL2 regime)
- `TLBI ASIDE1,Xn`: Invalidate this CPU's cached copies of entries for the EL1&0 regime, for the ASID specified in register `Xn`.
- `TLBI ASIDE1IS,Xn`: Invalidate this CPU's cached copies of entries for the EL1&0 regime, for the ASID specified in register `Xn`.
  (Note that the EL2 regime does not have ASIDs)

1685 ## 7.8 Arm ASL Reference

1686 Here I include the actual Arm ASL for the various parts of the translation machinery. This listing contains a
1687 verbatim subset of the ASL pseudocode for the translation table walk.

1688 The sources have per-function line numbers and are annotated to direct the reader to those parts highlighted in
1689 §7.6.3. Lines which handle out-of-scope features (access flags, dirty bits, shareability domains, debugging, realms,
1690 secure states, atomics) are greyed out. Key lines have coloured annotations.

1691 The ASL code listed here (minus the annotations) is copyright Arm, publicly available on Arm's webpage [55],
1692 and is restricted to only those parts of the translation table walk we are discussing. They are included here under
1693 *fair dealings*, for the purpose of critism and review [56, s. 30].

7.8.1 AArch64.TranslateAddress

```
1  AddressDescriptor AArch64.TranslateAddress(bits(64) va, AccType acctype, boolean
       iswrite, boolean aligned, integer size)
2    result = AArch64.FullTranslate(va, acctype, iswrite, aligned);<— Do the translation
3
4    if !IsFault(result) && acctype != AccType_IFETCH then
5      result.fault = AArch64.CheckDebug(va, acctype, iswrite, size);
6
7    if HaveRME() && !IsFault(result) && (acctype != AccType_DC ||
8      boolean IMPLEMENTATION_DEFINED "GPC Fault on DC operations") then
9      accdesc = CreateAccessDescriptor(acctype);
10     result.fault.gpcf = GranuleProtectionCheck(result, accdesc);
11
12     if result.fault.gpcf.gpf != GPCF_None then
13       result.fault.statuscode = Fault_GPCFOnOutput;
14       result.fault.paddress   = result.paddress;
15       result.fault.acctype    = acctype;
16       result.fault.write      = iswrite;
17
18   if !IsFault(result) && acctype == AccType_IFETCH then
19     result.fault = AArch64.CheckDebug(va, acctype, iswrite, size);
20
21   // Update virtual address for abort functions
22   result.vaddress = ZeroExtend(va);
23
24   return result;
```

7.8.2 AArch64.FullTranslate

```
AddressDescriptor AArch64.FullTranslate(bits(64) va, AccType acctype, boolean
    iswrite, boolean aligned)

  fault = NoFault();
  fault.acctype = acctype;
  fault.write   = iswrite;

  ispriv = PSTATE.EL != EL0 && !(acctype IN {AccType_UNPRIV, AccType_UNPRIVSTREAM});
  regime = TranslationRegime(PSTATE.EL, acctype);
  ss = SecurityStateAtEL(PSTATE.EL);

  AddressDescriptor ipa;
  (fault, ipa) = AArch64.S1Translate(fault, regime, ss, va, acctype, aligned,
    iswrite, ispriv);

  if fault.statuscode != Fault_None then
    return CreateFaultyAddressDescriptor(va, fault);

  assert (ss == SS_Realm) IMPLIES EL2Enabled();
  if regime == Regime_EL10 && EL2Enabled() then
    s1aarch64 = TRUE;
    s2fs1walk = FALSE;
    AddressDescriptor pa;
    (fault, pa) = AArch64.S2Translate(fault, ipa, s1aarch64, ss, s2fs1walk, acctype,
     aligned, iswrite, ispriv);

    if fault.statuscode != Fault_None then
      return CreateFaultyAddressDescriptor(va, fault);
    else
      return pa;
  else
    return ipa;
```

Do the first stage of translation

Check for stage 1 translation fault

Do the second stage of translation

Check for stage 2 translation fault

7.8.3 AArch64.S1Translate

```
1  (FaultRecord, AddressDescriptor) AArch64.S1Translate(FaultRecord fault_in, Regime
     regime, SecurityState ss, bits(64) va, AccType acctype, boolean aligned_in,
     boolean iswrite_in, boolean ispriv)
2    FaultRecord fault = fault_in;
3    boolean aligned = aligned_in;
4    boolean iswrite = iswrite_in;
5    // Prepare fault fields in case a fault is detected
6    fault.secondstage = FALSE;
7    fault.s2fs1walk   = FALSE;
8
9    if !AArch64.S1Enabled(regime) then
10     return AArch64.S1DisabledOutput(fault, regime, ss, va, acctype, aligned);
11
12   walkparams = AArch64.GetS1TTWParams(regime, va);
13
14   if (AArch64.S1InvalidTxSZ(walkparams) ||
15       (!ispriv && walkparams.e0pd == '1') ||
16       (!ispriv && walkparams.nfd == '1' && IsDataAccess(acctype) && TSTATE.depth >
     0) ||
17       (!ispriv && walkparams.nfd == '1' && acctype == AccType_NONFAULT) ||
18       )AArch64.VAIsOutOfRange(va, acctype, regime, walkparams)) then
19     fault.statuscode = Fault_Translation;
20     fault.level     = 0;
21     return (fault, AddressDescriptor UNKNOWN);
22
23   AddressDescriptor descaddress;
24   TTWState walkstate;
25   bits(64) descriptor;
26   bits(64) new_desc;
27   bits(64) mem_desc;
28   repeat
29     (fault, descaddress, walkstate, descriptor) = AArch64.S1Walk(fault, walkparams,
     va, regime, ss, acctype, iswrite, ispriv);
30
31     if fault.statuscode != Fault_None then
32       return (fault, AddressDescriptor UNKNOWN);
33
34     if acctype == AccType_IFETCH then
35       // Flag the fetched instruction is from a guarded page
36       SetInGuardedPage(walkstate.guardedpage == '1');
37
38     if AArch64.S1HasAlignmentFault(acctype, aligned, walkparams.ntlsmd, walkstate.
     memattrs) then
39       fault.statuscode = Fault_Alignment;
40     elsif IsAtomicRW(acctype) then
41       if AArch64.S1HasPermissionsFault(regime, ss, walkstate, walkparams, ispriv,
     acctype, FALSE) then
42         // The Permission fault was not caused by lack of write permissions
43         fault.statuscode = Fault_Permission;
44         fault.write     = FALSE;
45       elsif AArch64.S1HasPermissionsFault(regime, ss, walkstate, walkparams, ispriv,
       acctype, TRUE) then
46         // The Permission fault was caused by lack of write permissions
47         fault.statuscode = Fault_Permission;
48         fault.write     = TRUE;
49     elsif AArch64.S1HasPermissionsFault(regime, ss, walkstate, walkparams, ispriv,
     acctype, iswrite) then
50       fault.statuscode = Fault_Permission;
51
52     new_desc = descriptor;
53     if walkparams.ha == '1' && AArch64.FaultAllowsSetAccessFlag(fault) then
54       // Set descriptor AF bit
55       new_desc<10> = '1';
56
```

Check VA is valid

Do the translation table walk

Check for S1 translation fault

Check for permission fault

```
57      // If HW update of dirty bit is enabled, the walk state permissions
58      // will already reflect a configuration permitting writes.
59      // The update of the descriptor occurs only if the descriptor bits in
60      // memory do not reflect that and the access instigates a write.
61      if (fault.statuscode == Fault_None &&
62          walkparams.ha  == '1' &&
63          walkparams.hd  == '1' &&
64          descriptor<51> == '1' && // Descriptor DBM bit
65          (IsAtomicRW(acctype) || iswrite) &&
66          !(acctype IN {AccType_AT, AccType_ATPAN, AccType_IC, AccType_DC})) then
67        // Clear descriptor AP[2] bit permitting stage 1 writes
68        new_desc<7> = '0';
69
70      AddressDescriptor descupdateaddress;
71      FaultRecord s2fault;
72      // Either the access flag was clear or AP<2> is set
73      if new_desc != descriptor then
74        if regime == Regime_EL10 && EL2Enabled() then
75          s1aarch64 = TRUE;
76          s2fs1walk = TRUE;
77          aligned   = TRUE;
78          iswrite   = TRUE;
79          (s2fault, descupdateaddress) = AArch64.S2Translate(fault, descaddress,
        s1aarch64, ss, s2fs1walk, AccType_ATOMICRW, aligned, iswrite, ispriv);
80
81          if s2fault.statuscode != Fault_None then
82            return (s2fault, AddressDescriptor UNKNOWN);
83        else
84          descupdateaddress = descaddress;
85
86        (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
        walkparams.ee, descupdateaddress);
87
88    until new_desc == descriptor || mem_desc == new_desc;
89
90    if fault.statuscode != Fault_None then
91      return (fault, AddressDescriptor UNKNOWN);
92
93    // Output Address
94    oa = StageOA(va, walkparams.tgx, walkstate);<-- Compute IPA
95    MemoryAttributes memattrs;
96    if (acctype == AccType_IFETCH &&
97      (walkstate.memattrs.memtype == MemType_Device || !AArch64.S1ICacheEnabled(regime
        ))) then
98      // Treat memory attributes as Normal Non-Cacheable
99      memattrs = NormalNCMemAttr();
100     memattrs.xs = walkstate.memattrs.xs;
101   elsif (acctype != AccType_IFETCH && !AArch64.S1DCacheEnabled(regime) &&
102       walkstate.memattrs.memtype == MemType_Normal) then
103     // Treat memory attributes as Normal Non-Cacheable
104     memattrs = NormalNCMemAttr();
105     memattrs.xs = walkstate.memattrs.xs;
106
107     // The effect of SCTLR_ELx.C when '0' is Constrained UNPREDICTABLE
108     // on the Tagged attribute
109     if HaveMTE2Ext() && walkstate.memattrs.tagged then
110       memattrs.tagged = ConstrainUnpredictableBool(Unpredictable_S1CTAGGED);
111   else
112     memattrs = walkstate.memattrs;
113
114   // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION
        DEFINED
115   // to be either effective value or descriptor value
116   if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
117       !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1"))
        then
```

```
118       memattrs.shareability = walkstate.memattrs.shareability;
119     else
120       memattrs.shareability = EffectiveShareability(memattrs);
121
122     if acctype == AccType_ATOMICLS64 && memattrs.memtype == MemType_Normal then
123       if memattrs.inner.attrs != MemAttr_NC || memattrs.outer.attrs != MemAttr_NC then
124         fault.statuscode = Fault_Exclusive;
125         return (fault, AddressDescriptor UNKNOWN);
126
127     ipa = CreateAddressDescriptor(va, oa, memattrs);
128     return (fault, ipa);
```

Return IPA and Memory Attributes

7.8.4 AArch64.S1Walk

```
1  (FaultRecord, AddressDescriptor, TTWState, bits(64)) AArch64.S1Walk(FaultRecord
     fault_in, S1TTWParams walkparams, bits(64) va, Regime regime, SecurityState ss,
     AccType acctype, boolean iswrite_in, boolean ispriv)
2    FaultRecord fault = fault_in;
3    boolean iswrite = iswrite_in;
4    if HasUnprivileged(regime) && AArch64.S1EPD(regime, va) == '1' then
5      fault.statuscode = Fault_Translation;
6      fault.level     = 0;
7      return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);
8
9    walkstate = AArch64.S1InitialTTWState(walkparams, va, regime, ss);<------ read TTBR
10
11   // Detect Address Size Fault by TTB
12   if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
13     fault.statuscode = Fault_AddressSize;
14     fault.level     = 0;
15     return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);
16
17   bits(64) descriptor;
18   AddressDescriptor walkaddress;
19
20   walkaddress.vaddress = va;
21   if !AArch64.S1DCacheEnabled(regime) then
22     walkaddress.memattrs = NormalNCMemAttr();
23     walkaddress.memattrs.xs = walkstate.memattrs.xs;
24   else
25     walkaddress.memattrs = walkstate.memattrs;
26
27   // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION
       DEFINED
28   // to be either effective value or descriptor value
29   if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
30       !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1"))
       then
31     walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
32   else
33     walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);
34
35   DescriptorType desctype;
36   repeat<------------ For each level in {0,1,2,3}
37     fault.level = walkstate.level;
38     FullAddress descaddress = AArch64.TTEntryAddress(walkstate.level, walkparams.tgx
       , walkparams.txsz, va, walkstate.baseaddress);
39                                                      Get IPA of entry to read
40     walkaddress.paddress = descaddress;
41
42     if regime == Regime_EL10 && EL2Enabled() then
43       s1aarch64 = TRUE;
44       s2fs1walk = TRUE;                               Do S2 translation to get the
45       aligned   = TRUE;                               PA of the entry
46       iswrite   = FALSE;
47       (s2fault, s2walkaddress) = AArch64.S2Translate(fault, walkaddress, s1aarch64,
       ss, s2fs1walk, AccType_TTW, aligned, iswrite, ispriv);
48
49       if s2fault.statuscode != Fault_None then<----- Check for S2 fault
50         return (s2fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64)
       UNKNOWN);
51
52       (fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, fault);
53     else                                  Read memory to get descriptor
54       (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault);
55
56     if fault.statuscode != Fault_None then<----- Check for external abort
57       return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);
58
```

```
59      desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.ds, walkparams.
        tgx, walkstate.level);
60
61      case desctype of
62        when DescriptorType_Table
63          walkstate = AArch64.S1NextWalkStateTable(walkstate, regime, walkparams,
        descriptor);
64
65          // Detect Address Size Fault by table descriptor
66          if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
67            fault.statuscode = Fault_AddressSize;
68            return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64)
        UNKNOWN);
69
70        when DescriptorType_Page, DescriptorType_Block
71          walkstate = AArch64.S1NextWalkStateLast(walkstate, regime, ss, walkparams,
        descriptor);
72
73        when DescriptorType_Invalid
74          fault.statuscode = Fault_Translation;
75          return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN
        );
76
77        otherwise
78          Unreachable();
79
80    until desctype IN {DescriptorType_Page, DescriptorType_Block};
81
82    if (walkstate.contiguous == '1' &&
83        AArch64.ContiguousBitFaults(walkparams.txsz, walkparams.tgx, walkstate.level))
         then
84      fault.statuscode = Fault_Translation;
85    elsif desctype == DescriptorType_Block && AArch64.BlocknTFaults(descriptor) then
86      fault.statuscode = Fault_Translation;
87    // Detect Address Size Fault by final output
88    elsif AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
89      fault.statuscode = Fault_AddressSize;
90    // Check descriptor AF bit
91    elsif (descriptor<10> == '0' && walkparams.ha == '0' &&
92        !(acctype IN {AccType_DC, AccType_IC} &&
93        !boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC
        operations")) then
94      fault.statuscode = Fault_AccessFlag;
95
96    return (fault, walkaddress, walkstate, descriptor);
```

Extract next level table address

Extract page start address

Return fault if invalid

7.8.5 AArch64.S2Translate

```
1  (FaultRecord, AddressDescriptor) AArch64.S2Translate(FaultRecord fault_in,
     AddressDescriptor ipa, boolean s1aarch64, SecurityState ss, boolean s2fs1walk,
     AccType acctype, boolean aligned, boolean iswrite, boolean ispriv)
2    walkparams = AArch64.GetS2TTWParams(ss, ipa.paddress.paspace, s1aarch64);
3    FaultRecord fault = fault_in;
4
5    // Prepare fault fields in case a fault is detected
6    fault.statuscode  = Fault_None; // Ignore any faults from stage 1
7    fault.secondstage = TRUE;
8    fault.s2fs1walk   = s2fs1walk;
9    fault.ipaddress   = ipa.paddress;
10
11   if walkparams.vm != '1' then   <-- Check if in a two-stage regime
12     // Stage 2 translation is disabled
13     return (fault, ipa);
14
15   if (AArch64.S2InvalidTxSZ(walkparams, s1aarch64) ||
16       AArch64.S2InvalidSL(walkparams) ||
17       AArch64.S2InconsistentSL(walkparams) ||
18       AArch64.IPAIsOutOfRange(ipa.paddress.address, walkparams)) then
19     fault.statuscode = Fault_Translation;
20     fault.level    = 0;
21     return (fault, AddressDescriptor UNKNOWN);
22
23   AddressDescriptor descaddress;
24   TTWState walkstate;
25   bits(64) descriptor;
26   bits(64) new_desc;
27   bits(64) mem_desc;
28   repeat
29     (fault, descaddress, walkstate, descriptor) = AArch64.S2Walk(fault, ipa,
     walkparams, ss, acctype, iswrite, s1aarch64);
                                                      ^-- Do translation table walk
30
31     if fault.statuscode != Fault_None then   <-- Check for stage 2 translation fault
32       return (fault, AddressDescriptor UNKNOWN);
33
34     if AArch64.S2HasAlignmentFault(acctype, aligned, walkstate.memattrs) then
35       fault.statuscode = Fault_Alignment;
36     elsif IsAtomicRW(acctype) then
37       if AArch64.S2HasPermissionsFault(s2fs1walk, walkstate, ss, walkparams, ispriv,
      acctype, FALSE) then
38         // The Permission fault was not caused by lack of write permissions
39         fault.statuscode = Fault_Permission;
40         fault.write    = FALSE;
41       elsif AArch64.S2HasPermissionsFault(s2fs1walk, walkstate, ss, walkparams,
     ispriv, acctype, TRUE) then
42         // The Permission fault was caused by lack of write permissions.
43         // However, HW updates, which are atomic writes for stage 1
44         // descriptors, permissions fault reflect the original access.
45         fault.statuscode = Fault_Permission;
46         if !fault.s2fs1walk then
47           fault.write  = TRUE;
48     elsif AArch64.S2HasPermissionsFault(s2fs1walk, walkstate, ss, walkparams, ispriv
     , acctype, iswrite) then   <-- Check for stage 2 permission fault
49       fault.statuscode = Fault_Permission;
50
51     new_desc = descriptor;
52     if walkparams.ha == '1' && AArch64.FaultAllowsSetAccessFlag(fault) then
53       // Set descriptor AF bit
54       new_desc<10> = '1';
55
56     // If HW update of dirty bit is enabled, the walk state permissions
57     // will already reflect a configuration permitting writes.
58     // The update of the descriptor occurs only if the descriptor bits in
```

```
59      // memory do not reflect that and the access instigates a write.
60      if (fault.statuscode == Fault_None &&
61          walkparams.ha  == '1' &&
62          walkparams.hd  == '1' &&
63          descriptor<51> == '1' && // Descriptor DBM bit
64          (IsAtomicRW(acctype) || iswrite) &&
65          !(acctype IN {AccType_AT, AccType_ATPAN, AccType_IC, AccType_DC})) then
66        // Set descriptor S2AP[1] bit permitting stage 2 writes
67        new_desc<7> = '1';
68
69      // Either the access flag was clear or S2AP<1> is clear
70      if new_desc != descriptor then
71        (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
      walkparams.ee, descaddress);
72
73    until new_desc == descriptor || mem_desc == new_desc;
74
75    if fault.statuscode != Fault_None then
76      return (fault, AddressDescriptor UNKNOWN);
77
78    ipa_64 = ZeroExtend(ipa.paddress.address, 64);
79    // Output Address
80    oa = StageOA(ipa_64, walkparams.tgx, walkstate);<-- Compute final PA
81    MemoryAttributes s2_memattrs;
82    if ((s2fs1walk &&
83        walkstate.memattrs.memtype == MemType_Device && walkparams.ptw == '0') ||
84      (acctype == AccType_IFETCH &&
85        (walkstate.memattrs.memtype == MemType_Device || HCR_EL2.ID == '1')) ||
86      (acctype != AccType_IFETCH &&
87         walkstate.memattrs.memtype == MemType_Normal && HCR_EL2.CD == '1')) then
88      // Treat memory attributes as Normal Non-Cacheable
89      s2_memattrs = NormalNCMemAttr();
90      s2_memattrs.xs = walkstate.memattrs.xs;
91    else
92      s2_memattrs = walkstate.memattrs;
93
94    if !s2fs1walk && acctype == AccType_ATOMICLS64 && s2_memattrs.memtype ==
      MemType_Normal then
95      if s2_memattrs.inner.attrs != MemAttr_NC || s2_memattrs.outer.attrs !=
      MemAttr_NC then
96        fault.statuscode = Fault_Exclusive;
97        return (fault, AddressDescriptor UNKNOWN);
98
99    MemoryAttributes memattrs;
100   if walkparams.fwb == '0' then
101     memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs);<-- Merge memory attributes
102   else
103     memattrs = s2_memattrs;
104
105   pa = CreateAddressDescriptor(ipa.vaddress, oa, memattrs);
106   return (fault, pa);<-- Return PA and Memory Attributes
```

7.8.6 AArch64.S2Walk

```
1  (FaultRecord, AddressDescriptor, TTWState, bits(64)) AArch64.S2Walk(
2      FaultRecord fault_in, AddressDescriptor ipa, S2TTWParams walkparams,
       SecurityState ss, AccType acctype, boolean iswrite, boolean s1aarch64)
3
4    FaultRecord fault = fault_in;
5    ipa_64 = ZeroExtend(ipa.paddress.address, 64);
6
7    TTWState walkstate;
8    if ss == SS_Secure then
9      walkstate = AArch64.SS2InitialTTWState(walkparams, ipa.paddress.paspace);
10   else
11     walkstate = AArch64.S2InitialTTWState(ss, walkparams);<-- read VTTBR
12
13   // Detect Address Size Fault by TTB
14   if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64) then
15     fault.statuscode = Fault_AddressSize;
16     fault.level     = 0;
17     return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);
18
19   bits(64) descriptor;
20   AddressDescriptor walkaddress;
21
22   walkaddress.vaddress = ipa.vaddress;
23   if HCR_EL2.CD == '1' then
24     walkaddress.memattrs = NormalNCMemAttr();
25     walkaddress.memattrs.xs = walkstate.memattrs.xs;
26   else
27     walkaddress.memattrs = walkstate.memattrs;
28
29   walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);
30
31   DescriptorType desctype;
32   repeat<-- For each level in {0,1,2,3}
33     fault.level = walkstate.level;
34
35     FullAddress descaddress;
36     if walkstate.level == AArch64.S2StartLevel(walkparams) then
37       // Initial lookup might index into concatenated tables
38       descaddress = AArch64.S2SLTTEntryAddress(walkparams, ipa.paddress.address,
       walkstate.baseaddress);
39     else
40       ipa_64 = ZeroExtend(ipa.paddress.address, 64);
41       descaddress = AArch64.TTEntryAddress(walkstate.level, walkparams.tgx,
       walkparams.txsz, ipa_64, walkstate.baseaddress);<-- Get PA of entry to read
42
43     walkaddress.paddress = descaddress;
44     (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault);
45                                           ^-- Read descriptor from memory
46     if fault.statuscode != Fault_None then<-- Check for external abort
47       return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);
48
49     desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.ds, walkparams.
       tgx, walkstate.level);
50
51     case desctype of
52       when DescriptorType_Table
53         walkstate = AArch64.S2NextWalkStateTable(walkstate, walkparams, descriptor);
54                                                 ^-- Extract next level table address
55         // Detect Address Size Fault by table descriptor
56         if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64)
       then
57           fault.statuscode = Fault_AddressSize;
58           return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64)
       UNKNOWN);
59
60       when DescriptorType_Page, DescriptorType_Block
```

```
61          walkstate = AArch64.S2NextWalkStateLast(walkstate, ss, walkparams, ipa,
        descriptor);                                    Extract page start address
62
63        when DescriptorType_Invalid
64          fault.statuscode = Fault_Translation;
65          return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN
        );               Return fault if invalid
66
67        otherwise
68          Unreachable();
69
70    until desctype IN {DescriptorType_Page, DescriptorType_Block};
71
72    if (walkstate.contiguous == '1' &&
73          AArch64.ContiguousBitFaults(walkparams.txsz, walkparams.tgx, walkstate.level))
        then
74      fault.statuscode = Fault_Translation;
75    elsif desctype == DescriptorType_Block && AArch64.BlocknTFaults(descriptor) then
76      fault.statuscode = Fault_Translation;
77    // Detect Address Size Fault by final output
78    elsif AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64) then
79      fault.statuscode = Fault_AddressSize;   Check output address is within bounds
80    // Check descriptor AF bit
81    elsif (descriptor<10> == '0' && walkparams.ha == '0' &&
82          !(acctype IN {AccType_DC, AccType_IC} &&
83          !boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC
        operations")) then
84      fault.statuscode = Fault_AccessFlag;
85
86    return (fault, walkaddress, walkstate, descriptor);
```

7.8.7 AArch64.FetchDescriptor

```
1  (FaultRecord, bits(N)) FetchDescriptor(bit ee, AddressDescriptor walkaddress,
     FaultRecord fault_in)
2    // 32-bit descriptors for AArch32 Short-descriptor format
3    // 64-bit descriptors for AArch64
4    //or AArch32 Long-descriptor format
5    assert N == 32 || N == 64;
6    bits(N) descriptor;
7    FaultRecord fault = fault_in;
8    AccessDescriptor walkacc;
9
10   walkacc.acctype = AccType_TTW;
11   // MPAM PARTID for translation table walk is determined by the access invoking the
       translation
12   walkacc.mpam  = GenMPAMcurEL(fault.acctype);
13
14   if HaveRME() then
15     fault.gpcf = GranuleProtectionCheck(walkaddress, walkacc);
16     if fault.gpcf.gpf != GPCF_None then
17       fault.statuscode = Fault_GPCFOnWalk;
18       fault.paddress   = walkaddress.paddress;
19       fault.gpcfs2walk = fault.secondstage;
20       return (fault, bits(N) UNKNOWN);
21
22   PhysMemRetStatus memstatus;
23   (memstatus, descriptor) = PhysMemRead(walkaddress, N DIV 8, walkacc);
24   if IsFault(memstatus) then
25     fault = HandleExternalTTWAbort(memstatus, fault.write, walkaddress, walkacc, N
      DIV 8, fault);
26     if IsFault(fault.statuscode) then
27       return (fault, bits(N) UNKNOWN);
28
29   if ee == '1' then
30     descriptor = BigEndianReverse(descriptor);
31
32   return (fault, descriptor);
```

1694

Chapter 8

1695

Relaxed virtual memory

1696 *This chapter is based, in part, on: Relaxed virtual memory in Armv8-A [54] by Ben Simner, Alasdair Armstrong, Jean*
1697 *Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. Published in the proceedings of the 31st*
1698 *European Symposium on Programming (ESOP, 2022).*

1699 Now we will introduce the main concurrency architecture design questions that arise for virtual memory in
1700 Arm. As usual, the architecture defines the *envelope* of behaviours which hardware must guarantee and on which
1701 software may rely. This envelope must be tight enough to give the guarantees software needs to function, but still
1702 loose enough to admit the range of existing and conceivable microarchitectures whose optimization techniques
1703 are necessary for performance.

1704 This chapter therefore will discuss both the relevant microarchitecture as we understand it, and also the behaviours
1705 which it is believed software relies upon. The discussion will touch on points of several kinds: some which are
1706 clear in the current Arm prose documentation; some where Arm are in the process of architecting a change; some
1707 that are not documented but where the semantics is (perhaps, after discussion with Arm) clear or constrained by
1708 current hardware or software practice; and, some where their modelling raised questions for which the architecture
1709 is not yet well-defined, and Arm must make an architectural decision.

1710 Ideally, we would be able to specify which points belong to which kind. It is, however, not so easy. There is
1711 no clean separation between aspects there are clearly defined in the architecture reference, and those that are
1712 not; instead, the manual has a shallow covering of many of the behaviours described here. In other places, the
1713 reference may have been updated or changed over the course of the work, clarifying parts of the architecture, and
1714 while this may have happened concurrently with discussing those and other points with Arm, the reference text
1715 itself is solely the responsibility of Arm. In §8.8 we will return to this question, and more directly address the
1716 kinds of each point discussed.

1717 **Chapter overview** The body of this chapter will explore a sequence of key behaviours, some of which the
1718 architecture guarantee and some that it does not. Each contains a description of the behaviour, including whether
1719 software relies on it or known hardware guarantees it; a short discussion of the architectural intent as we
1720 understand it; and any associated litmus tests.

1721 This chapter will discuss a variety of interesting behaviours. In an attempt to make this chapter more approachable,
1722 it is broken down into a logical progression: slowly building up from the most simple and fundamental parts of
1723 the architecture, to increasingly more complex cases.

1724 We will first discuss (in §8.2) how translation affects the prior 'data memory' **TODO: PS: user-mode?** tests
1725 covered in previous work. Then, we shall see how the caching of translation entries is limited (§??) and the
1726 fundamental behaviours of the translation table walk (§8.4). Building upon that, we will see that these translation
1727 table walks may be cached and re-used in later translations, which is explored in detail in §8.5. Then (in §8.6),
1728 we will explore how the various kinds of TLB maintenance interact with those cached translations, and other
1729 translation table walks. Finally, we touch on how all of the above fit together with system registers and other
1730 context changing and synchronising operations in §8.7.

1731 *Chapter Contents*

1732 §8.2 **Aliased data memory** . . . 74
1733 §8.2.1 **Virtual coherence** . . . 74
1734 §8.2.2 **Aliasing different locations** . . . 78
1735 §8.2.3 **Might be same (physical) address** . . . 79
1736 §8.3 **What can be cached in TLBs** . . . 79
1737 §8.4 **Reads not from TLB** . . . 82
1738 §8.4.1 **Out-of-order execution** . . . 82
1739 §8.4.2 **Enforcing thread-local ordering** . . . 84
1740 §8.4.3 **Enhanced Translation Synchronization** . . . 90
1741 §8.4.4 **Forwarding to the translation table walker** . . . 91
1742 §8.4.5 **Speculative execution** . . . 93
1743 §8.4.6 **Single-copy atomicity** . . . 94
1744 §8.4.7 **Multi-copy atomicity** . . . 94
1745 §8.4.8 **Translation-table-walk intra-walk ordering** . . . 96
1746 §8.4.9 **Multiple translations within a single instruction** . . . 96
1747 §8.5 **Caching of translations in TLBs** . . . 97
1748 §8.5.1 **Cached translations** . . . 97
1749 §8.5.2 **TLB fills** . . . 98
1750 §8.5.3 **μTLBs** . . . 98
1751 §8.5.4 **Partial caching of walks** . . . 100
1752 §8.5.5 **Reachability** . . . 101
1753 §8.6 **TLB maintenance** . . . 101
1754 §8.6.1 **Recovering coherence** . . . 101
1755 §8.6.2 **Thread-local ordering and `TLBI`** . . . 105
1756 §8.6.3 **Broadcast** . . . 106
1757 §8.6.4 **Virtualization** . . . 108
1758 §8.6.5 **Break-before-make** . . . 111
1759 §8.6.6 **ASIDs and VMIDs** . . . 111
1760 §8.6.7 **Access permissions** . . . 113
1761 §8.7 **Context synchronisation** . . . 117
1762 §8.7.1 **Relaxed system registers** . . . 117
1763 §8.8 **Contributions** . . . 118

8.1 Virtual memory litmus tests

As previously discussed, one fundamental idea to come out of the field of relaxed memory is the concept of litmus tests. Virtual memory is no different, and exploring the architectural intent is best done through the creation, discussion and evaluation of small programs which are representative examples of common patterns.

However, as we explore more of the system semantics more and more of the system state plays an integral role in the behaviours we see. For this reason we need a new language for describing the state of the system, with features not supported by the language supported by the previous litmus, rmem, herd, and diy tools [?, 28, ?, ?], in particular, the translation table state.

The litmus tests here are given in the `isla-axiomatic` test format. I describe the isla tool itself, and the extended test format syntax, in more detail in **TODO: ?REF?**.

A virtual memory litmus test To illustrate this isla test format, Figure 8.1 contains the test listing for a non-trivial virtual memory litmus test called **CoW** (or “Copy-on-Write”).

```
AArch64
                                        Initial Stat
0:R0=0x2
0:R1=x
0:R3=y
0:R4=z
0:R5=z
0:R6=0b0
0:R7=pte3(x)
0:R8=page(x)
0:R9=mkdesc3(oa=pa2)
0:R10=pte3(x)
0:R20=0b0
0:VBAR_EL1=0x1000
0:PSTATE.EL=0b00
virtual x y z;
physical pa1 pa2;
x ↦ pa1 with [AP = 0b11] and default;
x ↦ invalid;
x ↦ pa2 with [AP = 0b01] and default;
y ↦ pa1;
z ↦ pa2;
identity 0x1000 with code;
*pa1 = 1;
*pa2 = 0;
                                        Thread 0
01. STR X0,[X1]
                                        Thread 0 EL1 H
01. 0x1400:
02.     CBNZ X20,exit
03.     LDR X2,[X3]
04.     STR X2,[X4]
05.     DC CIVAC,X5
06.     DSB SY
07.     STR X6,[X7]
08.     DSB SY
09.     TLBI VALE1IS,X8
10.     DSB SY
11.     STR X9,[X10]
12.     MOV X20,#1
13.     ERET
14. exit:
15.     MRS X21,ELR_EL1
16.     ADD X21,X21,#4
17.     MSR ELR_EL1,X21
18.     ERET
                                        Final Stat
pa1=1 & pa2=2
```

Figure 8.1: Test CoW: code listing

This test is derived from sequence of operations the Linux kernel takes when performing copy-on-write. Thread 0 tries to write to a location (call it x) that is currently read-only (line 1 in the thread code), then when the fault is taken the Linux exception handler begins executing (line 1 in the handler), Linux performs some checks that it's okay to copy and that it hasn't already done so (not part of the test), and then copies the physical page (lines 3 and 4 in the handler, although the test here only copies one value as demonstration), before flushing the data caches (line 5) so that later reads will be guaranteed to see the copied values. Then Linux needs to swap over the pagetable entry for x from a read-only view on the original page to a writeable mapping on the freshly copied page. It does this by first 'breaking' the entry, making it invalid (line 7), then performing the necessary TLB maintenance (line 9), before writing a new mapping to the new page (line 11). Now, Linux can return from the handler (line 13) and re-try the store instruction, hopefully this time successfully writing to the new page.

The test format is split into 4 main parts:

- ▷ The initial state, comprised of:
  - the per-thread register state.
  - the global memory and pagetable state.
- ▷ The thread code and any exception-handler code.
- ▷ The interesting final state, as a predicate over the final register and memory state.
- ▷ And, optionally, whether the outcome is allowed or forbidden by the model.

Initial state The initial state has three virtual addresses (`x`, `y` and `z`), and two physical addresses (`pa1` and `pa2`). Initial register values are written like `0:R4=z`, meaning register `R4` on Thread `0` initially contains the value `z` (in this case, a virtual address). Helper functions like `pte3`, `page` and `mkdesc3` are used to get the address of the leaf entry, the page offset and to create a new valid descriptor with the given OA, a more detailed description of the functions are given later.

Behind the scenes, isla creates a full instantiation of the Arm translation tables, but with some holes for symbolic values where the test may modify the tables. There is a default translation table, where the code and the tables themselves are mapped by default and everything else is invalid.

The pagetable setup is then defined in a small DSL which defines a delta to that default table, specifying that certain pages should be mapped or unmapped initially, as well as being able to specify the set of locations and

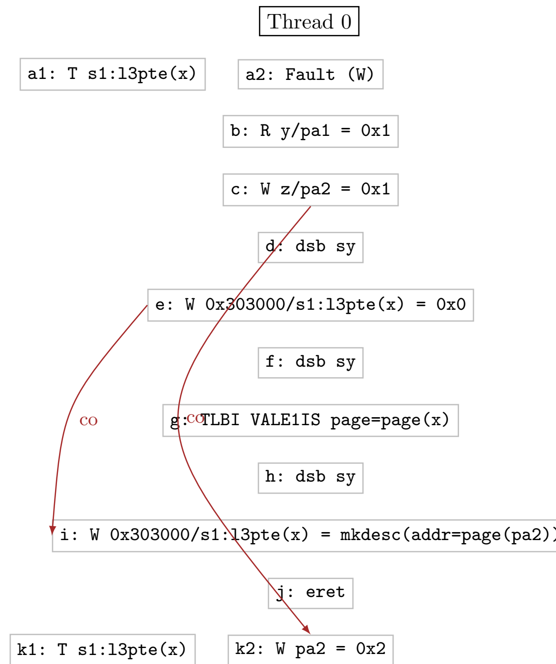


Figure 8.2: Test CoW: execution diagram

their initial memory values the test will need.

Fundamentally we categorise those locations as either virtual, intermediate, or physical. The line `virtual x y z` in the CoW test allocates 3 virtual contiguous pages, and labels their page-aligned addresses as `x`, `y`, and `z`. It then allocates two physical pages with addresses `pa1` and `pa2`. Next, the setup defines the initial value of the translation tables, as well as specifying the set of potential translation tables that may be in use by the test (for isla to create symbolic 'holes' for those). Namely, the initial state starts with `x` mapped to `pa1` with the access permissions bits set to `0b11` (read-only). The next two lines tell isla that during the test `x` may become unmapped (the descriptor may be invalid), or mapped to `pa2` with `AP=0b01`. The test also defines two other variables, `y` and `z` as aliases to the two physical pages, to help with copying the data between them, just as Linux would. Since there is an exception handler in this test, we need to ensure that the code page of the handler is mapped executable at EL1, which is what the `identity 0x1000 with code` line does (note that the handler section starts within the `0x1000` page). Finally, we say that the initial values of `pa1` and `pa2` are `1` and `0` respectively.

Register translation helpers The initial register state can reference parts of the initial state related to pagetables through the use of helper functions. Here are the helpers used by CoW, and most of the tests in this section. The full description of this format is given in **TODO: ?REF?**if more information is needed.

- `pte<N>(va)`: The (intermediate) physical address of the level N entry in the default translation tables that maps `va`.
- `desc<N>(va)`: The 64-bit descriptor from the initial state of the level N entry that maps `va` (the value of `pte<N>(va)` in the initial state).
- `page(va)`: The page number that `va` is in (equivalently: `va` $\gg$ `12`).
- `mkdesc<N>(oa=pa)`: A fresh 64-bit descriptor for a valid leaf entry at level N where the output address is given by the `oa` parameter.
- `mkdesc<N>(table=pa)`: A fresh 64-bit descriptor for a valid table entry at level N where the next-level-table address is given by the `table` parameter.

Entries listed as `f<N>` mean a family of functions `f1`, `f2`, `f3` and so on.

Execution diagrams Figure 8.2 is the isla-generated execution diagram for the CoW test. It illustrates a candidate execution which isla found (with any symbolic holes filled with concrete values) which matched the final state of the execution, and was consistent with the axioms of the model (given in Chapter 9).

The execution is rendered as a diagram, with separate traces for each thread, with multiple columns per thread, for translations and explicit events. In the diagram, there is one thread (Thread 0), and all events belong to

its trace. There are two columns; the right-hand side are the explicit events rendered in program-order, and the left-hand side contains translation events alongside any explicit events from the same instruction. Not all events from the trace are displayed in the execution diagram; many uninteresting events, of register reads and writes, and translation reads of unchanged entries, are suppressed. The execution displayed here is one where the initial store's translation table walk (event a1) reads an valid entry from the initial state but which did not have permissions to do a write, and so generates a Fault event (a2). The execution continues, copying the memory over to a new page (events b-c), before updating the translation tables to point to the new page (d-h, see §8.6.5), before returning from the exception handler (j) and re-trying the store which succeeds in writing to the new page (k2), giving a final state consistent with the expected final state from the test listing in Figure 8.1.

In general, while there could be multiple executions that correspond to the final execution, the tests are usually written in a way to ensure that there is only one consistent candidate execution which corresponds to the final state. In cases where the test is forbidden by the model, we still have isla induce a concrete candidate, and render a diagram of the interesting forbidden execution.

8.2 Aliased data memory

Much of the previous work on relaxed memory has been concerned with what we shall call 'data memory': the weak behaviour of concurrent loads and stores to memory. For Arm, we shall see that these previous models were implicitly assuming that all locations in the test were virtual addresses, with well-formed, constant, and injective, address translation mappings, which mapped all locations as readable, writable, and executable, normal cacheable memory.

Consider a non-injective mapping. Such mappings give rise to *aliasing*: the situation where two distinct virtual addresses in the same address space map to the same output physical address. This section will explore how the behaviours of those data memory tests change in the presence of aliasing.

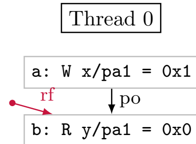
8.2.1 Virtual coherence

For data memory accesses, one of the most fundamental guarantee that architectures provide is *coherence*: in any execution, for each memory location, there is a total order of the accesses to that location, consistent with the program order of each thread, with reads reading from the most recent write in that order. Hardware implementations provide this, despite their elaborate cache hierarchies and out-of-order pipelines, by a combination of coherent cache protocols and pipeline hazard checking, identifying and restarting instructions when possible coherence violations are detected.

For Arm, coherence is with respect to physical addresses [1, B2.3.1 (p157)] [1, D5.11.1 (p4931)] . This means that if two virtual addresses alias to the same physical address, then:

- ▷ a load from one virtual address cannot ignore a program-order previous store to the other, as seen in the following CoWR.alias test [Figure 8.3]:

AArch64	CoWR.alias
Initial State	
0:R0=0x1 0:R1=x 0:R3=y	
physical pa1; x |-> pa1; y |-> pa1; *pa1 = 0;	
Thread 0	
STR X0,[X1] LDR X2,[X3]	
Final State	
0:X2=0	
Forbid	



This test is a variation on the standard CoWR test, where the VA is replaced with two distinct VAs, which both alias to the same PA.
The initial state is a configuration with two virtual addresses, `x` and `y`, which are both mapped to the physical address `pa1`, whose initial value is `0`. The thread then stores `1` to `x`, then loads `y`. It is then forbidden for this load to read `0`.
While the Armv8-A architecture reference manual describes data caches as being physically-indexed [1, D5.11.1 (p4931)] and so accesses via the same PA are 'fully coherent', further discussions with Arm clarify that this implies not just this coherence test, but that all prior data memory behaviours previously examined still apply when subjected to aliasing.

Figure 8.3: CoWR.alias test

- ▷ a load from one virtual address cannot ignore the write that a program-order previous load of the other address saw (CoRR0.alias+po [Figure 8.4], CoRR2.alias+po [Figure 8.5]).
- ▷ a load from one virtual address can have its value forwarded from a store to the other, and similarly on a speculative branch (MP.alias3+rfi-data+dmb [Figure 8.6], PPOCA.alias [Figure 8.6]).

AArch64	CoRR0.alias+po
Initial State	
0:R0=0b1 0:R1=x	1:R1=x 1:R3=y 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00
physical pa1; x |-> pa1; y |-> pa1; *pa1 = 0;	
Thread 0	Thread 1
STR X0,[X1]	**LDR** X0,[X1] **LDR** X2,[X3]
Final State	
1:X0=1 & 1:X2=0	
Forbid	

This test is a variation of the data memory CoRR0 test, where one of the loads has been replaced with a load of a distinct virtual address which aliases to the same underlying physical address.
Note that, like the original test, it is forbidden to read from the initial state in the later load, as this would violate coherence: exactly what the earlier text from the manual explicitly forbade.

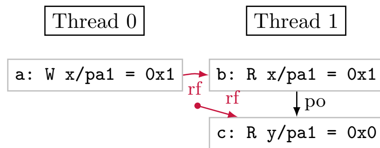
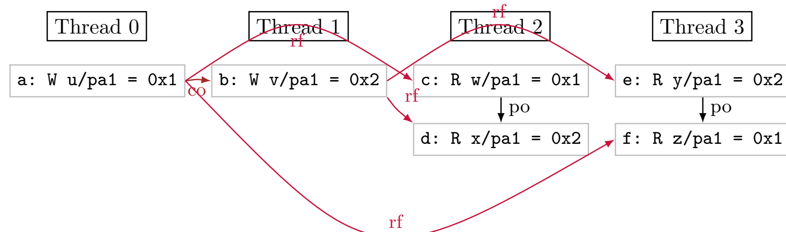


Figure 8.4: CoRR0.alias+po test

AArch64			CoRR2.alias+po
Initial State			
0:R0=0b01 0:R1=u	1:R0=0b10 1:R1=v	2:R1=w 2:R3=x 2:PSTATE.SP=0b0 2:PSTATE.EL=0b00	3:R1=y 3:R3=z 3:PSTATE.SP=0b0 3:PSTATE.EL=0b00
physical pa1; u |-> pa1; v |-> pa1; w |-> pa1; x |-> pa1; y |-> pa1; z |-> pa1; *pa1 = 0;			
Thread 0	Thread 1	Thread 2	Thread 3
STR X0,[X1]	**STR** X0,[X1]	**LDR** X0,[X1] **LDR** X2,[X3]	**LDR** X0,[X1] **LDR** X2,[X3]
Final State			
2:X0=1 & 2:X2=2 & 3:X0=2 & 3:X2=1			
Forbid			



This test is a variation of the data memory CoRR2 test. Here there are many options for adding aliasing, so we choose the maximally aliased version where each individual store and load uses a distinct virtual address but where all those virtual addresses alias to the same physical one.
This gives us a classic coherence shape, where it is forbidden for different threads to observe writes to the same physical location in different orders.

Figure 8.5: CoRR2.alias+po test

AArch64	MP.alias3+rfi-data+dmb
Initial State	
0:R0=0x1 0:R1=x 0:R3=z 0:R5=y	1:R1=y 1:R3=x

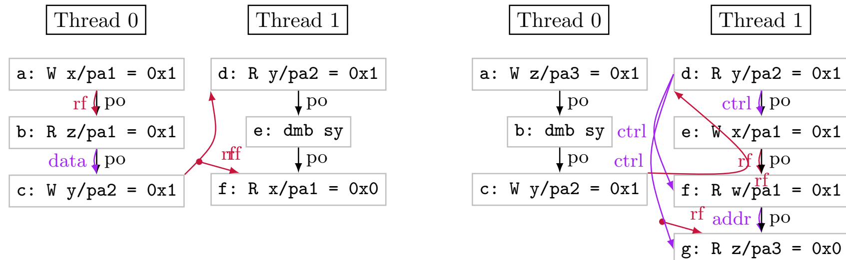
physical pa1 pa2;
x |-> pa1;
y |-> pa2;
z |-> pa1;
*pa1 = 0;
*pa2 = 0;	

Thread 0	Thread 1
STR X0,[X1] **LDR** X2,[X3] **STR** X2,[X5]	**LDR** X0,[X1] **DMB** SY **LDR** X2,[X3]
Final State	
1:X0=1 & 1:X2=0	
Allow	

AArch64	PPOCA.alias
Initial State	
0:R0=0x1 0:R1=z 0:R2=0x1 0:R3=y	1:R1=y 1:R2=0x1 1:R3=x 1:R5=w 1:R7=z

physical pa1 pa2 pa3;
w |-> pa1;
x |-> pa1;
y |-> pa2;
z |-> pa3;
*pa1 = 0;
*pa2 = 0;
*pa3 = 0;	

Thread 0	Thread 1
STR X0,[X1] **DMB** SY **STR** X2,[X3]	**LDR** X0,[X1] **CBNZ** X0,L0 L0: **STR** X2,[X3] **LDR** X4,[X5] **EOR** X8,X4,X4 **LDR** X6,[X7,X8]
Final State	
1:X0=1 & 1:X4=1 & 1:X6=0	
Allow	



These tests are variations of the standard PPOCA and MP+rfi-data+dmb tests, but with some aliasing. Both are examples of *forwarding*: a thread-local read of a write before that write has been propagated to memory. These two tests, determined to be allowed architecturally from our discussions with Arm, show that the processor can forward from a write even if the read was for a different virtual address so long as the physical addresses match, even down a speculative path.

Figure 8.6: PPOCA.alias and MP.alias3+rfi-data+dmb tests.

8.2.2 Aliasing different locations

In the previous section, we explored taking tests over a single location, and rewriting the test to use many locations, which all alias to the same address. One can also take a test that has multiple locations and make some of them alias to the same address.

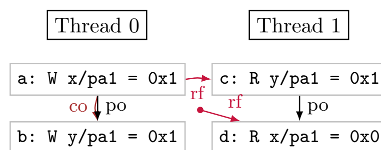
Multi-location data memory tests, which are architecturally allowed, may become forbidden in the presence of aliasing. For example, taking the traditional MP+pos test, when the two locations are aliased to the same physical address then we get the forbidden MP.alias+pos test [Figure 8.7]. This new test is, essentially, equivalent to the old CoRR0 test: coherence with two writes and two reads to the same location; just using different aliases.

AArch64	MP.alias+pos
Initial State	

`0:R0=0x1`
`0:R1=x`
`0:R2=0x1`
`0:R3=y`
	`1:R1=y`
	`1:R3=x`

`physical pa1;`
`x |-> pa1;`
`y |-> pa1;`
`*pa1 = 0;`	

Thread 0	Thread 1
`STR X0,[X1]` `STR X2,[X3]`	`LDR X0,[X1]` `LDR X2,[X3]`
Final State	
`1:X0=1 & 1:X2=0`	
Forbid	



Because x and y alias to the same phsical address pa1, the two loads (c and d) read the same location, and so cannot read different writes out-of-order.

Figure 8.7: Test MP.alias+pos

8.2.3 Might be same (physical) address

There is a corner case that we now should consider. For load and store instructions, when the last register used in the calculation of the address is read, the address becomes known. This allows, in the flat model, for program-order later instructions to begin execution (or at least, know they will not be restarted) at that point.

With the introduction of address translation, however, this point happens much later, after the whole translation table walk is performed. Between the read of the register and the completion of the translation table walk, other instructions may perform some part of their functionality. This may include reading from a different virtual address, before the physical address of a program-order previous instruction is known, but after the virtual address is known.

One might expect that, when deciding whether to propagate a store, if the page offset of the virtual address is different to that of the in-flight program-order earlier instructions, then the write could go ahead early, knowing that the access could not be to the same physical address as any of those instructions. However, this is not the case. Although the accesses definitely will not access the same physical address, the program-order earlier access may still fault, meaning the write will not be reached. This means that writes must wait for program-order earlier translations to finish (or at least, be known to not fault) before they can be propagated to other threads.

8.3 What can be cached in TLBs

As was described in §7.7, Arm hardware can have TLBs, caching previously seen translations. But, there are some restrictions to this; both in what information a TLB must cache when it does so, but also in what kind of information it is not permitted to cache at all.

8.3.1 Microarchitectural TLBs

Here we must make a clear distinction between the actual *microarchitectural* translation caching one may encounter inspecting hardware, and the architectural model being discussed here.

While there are possibly many different ways to describe the same architectural intent, here we carefully choose one which will make building tooling, extending the model, discussions with architects, and explaining individual tests easier. We will first look at a specific example to pin down terminology and gain some intuition for hardware, before giving a model MMU and TLB that abstracts away from the details.

Microarchitectural MMU – A53 Let us explore more closely how the actual hardware fill and walk works on a modern microprocessor. The Arm Cortex A53 is an Arm-designed application class processor. Previous relaxed memory work included exercising this core design extensively during litmus testing validation of the models, finding it to be relaxed, exhibiting many relaxed behaviours, but not aggressively so. This makes the A53 a good candidate as a demonstrator of an average relaxed processor design. While other processors by Arm are more aggressive in their optimisations, the MMU and TLB layout of the A53 seems typical: other cores, such as the A57 **TODO: ?CITE?**, A72 **TODO: ?CITE?**, A76 **TODO: ?CITE?**, A78 **TODO: ?CITE?** and A715 **TODO: ?CITE?** all have comparable, or simpler, TLB configurations.

The Arm A53 Technical Reference Manual (TRM) describes, in detail, the structure of the Memory Management Unit [57, 5-2] of the A53, and its constituent parts. Figure 8.8 shows a hand-written block diagram representing the key information from the TRM.

We see that each core has its own MMU, and that each MMU contains a unit that will perform the translation table walk, in addition to a selection of translation caching structures:

- ▷ one instruction micro-TLB;
- ▷ one data micro-TLB;
- ▷ one unified TLB;
- ▷ one walk cache; and,
- ▷ one IPA cache.

The microarchitectural TLBs store whole translations: virtual to physical mappings, plus permissions and so-on, tagged with their context. The TLBs are arranged hierarchically. With small, 10-entry, ‘micro’ TLBs for instruction and data streams separately, and one large 512-entry unified TLB.

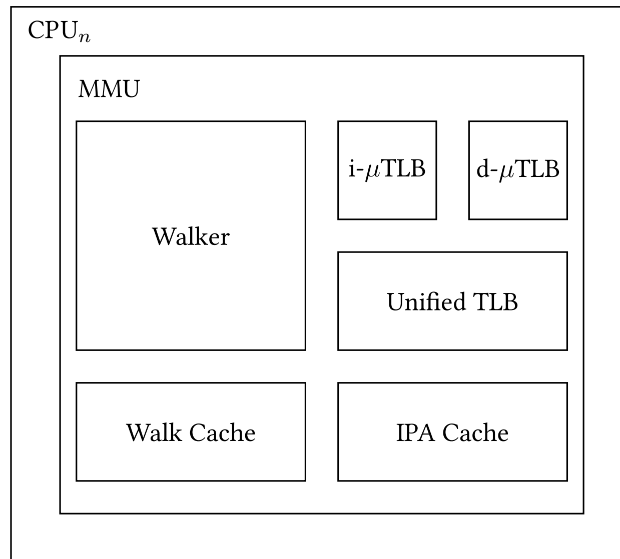


Figure 8.8: A53 Memory Management Block Diagram.

1937 On a TLB miss, the MMU performs a translation table walk using the walker, computing the Arm translation
1938 table walk ASL code which we previously explored in §7.6.

1939 When it begins this walk, the MMU first checks the walk cache for a matching entry. Walk cache entires are
1940 mappings from virtual address to the physical address of the last level translation table. If an entry is present the
1941 MMU can skip most of the walk entirely, performing just the very last read to read the leaf entry.

1942 If a second stage of translation is required during the walk, the IPA cache is used (and may be, or not, used many
1943 times during the same walk). The IPA cache stores mappings from intermediate physical to physical memory —
1944 with no associated virtual address — which can be used during both the final stage 2 walk and any intermediate
1945 stage 2 walks during a stage 1 walk.

1946 **TODO: PS: walk cache s1 only? BS: that is one of thibaut's questions to RG**

1947 The MMU is free to save the result of any translation table walk into these structures, including for walks due to
1948 speculation, prefetching, or architectural execution. This, essentially, allows the MMU to perform a walk for any
1949 arbitrary VA or IPA, at any point in time.

1950 8.3.2 Model MMU

1951 To abstract away from any specific microarchitecture, we will model the MMU as if it were a separate asynchronous
1952 unit, one for each thread, each with an overapproximate 'TLB'.

1953 Later, we will see tests that justify and ground this particular choice of abstraction, and we will explore this model
1954 and the mathematics which corresponds to it in more rigorous detail. But for now, we can imagine this model
1955 MMU as a set of (concurrently) executing translation table walks and a 'model TLB' cache of translation table
1956 entries.

1957 **Model TLB entries** In general, the architecture permits hardware to cache whatever information from the
1958 translation process the hardware sees fit, this may include the output of whole translation table walks (complete
1959 virtual to physical mappings) or individual translation table entries, or even the result of partial walks (the address
1960 of the last-level table, for example).

1961 It would not be feasible to even attempt to enumerate all the possible shapes of TLBs and the kinds of information
1962 they can cache. Instead, we will define a *model* TLB. This model will treat the TLB as a cache of writes of translation
1963 table entries, each tagged with some context. This allows the model to cache any combination of entries read from
1964 a translation table walk, making it weak enough to allow all known TLB implementations, but strong enough to
1965 not break any of the guarantees Arm require of those TLB implementations. These guarantees are explored, in
1966 detail, in §8.4 and §8.5.

```
TranslationTableEntry ≡ u64
Context ≡ ArchContext × Stage × option VA × option IPA × PA × Level
ArchContext ≡ VMID × ASID × Regime
CachedTranslationTableEntry ≡ PA × TranslationTableEntry × Context
TLB ≡ set CachedTranslationTableEntry
```

Figure 8.9: Model TLB type definitions.

1967 Each entry in the model TLB contains the information about the write itself: the physical address of the entry,
1968 and the cached 64-bit entry. But it must also be tagged with some contextual information, some used during TLB
1969 lookup and some used to identify cached entries during TLB invalidation. Figure 8.9 gives a consise summary of
1970 the model TLB definition in some psuedo-type-definitions.

1971 This contextual information includes:

1972 ▷ the architectural context information of the translation: the VMID, ASID (or a "global indicator"), and the
1973 translation regime;
1974 ▷ some *extended context* information, required for implementing TLB maintenance:
1975 – the virtual address, intermediate physical address, and/or physical address of the translation;
1976 – the translation stage and level at which the write was used;
1977 – the system register values used in the translation (those which can be cached); and,
1978 – for an entry used for a Stage 1 translation, whether it has been invalidated at both stages.

1979 The model MMU then performs all translations by doing a full translation table walk, but being able to optionally
1980 satisfy any read during that walk from a matching entry in the model TLB which matches the architectural context
1981 and input address.

1982 We imagine that any behaviour exhibited by a specific micro-architectural MMU and TLB configuration would
1983 also be explainable in this model.

1984 **TLB fills** Hardware has a variety of mechanisms which may lead to a translation table walk: direct architectural
1985 execution of instructions, pre-fetching of data or instructions, and speculation down branches. These translation
1986 table walks may result in TLB misses, and those misses then result in reads from memory and the MMU 'filling'
1987 the TLB with a copy of the information it can use in future.

1988 Arm do not wish to enumerate all the possible speculation machinery or prefetchers so instead opt for a model
1989 that is weaker: at any point in time, any thread's MMU can spontaneously perform a translation table walk for any
1990 virtual or intermediate-physical address for the current architectural context (VMID, ASID, etc, as in §8.3.2), and
1991 any reads that the translation table walk performs can either read from other TLB entries, or perform a non-TLB
1992 read of memory and then potentially cache a copy of the write it reads from in the TLB tagged with the extended
1993 context information from the walk. The behaviour of those non-TLB reads are explored more in §8.4.

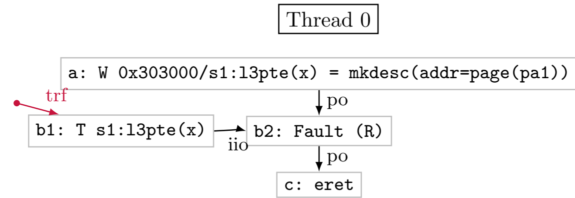
1994 ### 8.3.3 Invalid entries

1995 It is architecturally forbidden to cache information from attempted translations which result in translation faults,
1996 access flag faults, or address size faults (Note that a translation table walk may give rise to other faults as well,
1997 as discussed in §7.3.2, such as permission faults and alignment faults, which do not impose restrictions on TLB
1998 caching). More specifically, a TLB entry cannot be a write of a translation table entry which is the *direct* cause of
1999 such a fault. In particular, the TLB cannot cache translation table entries whose valid bit is not set.

2000 This is important, as it gives software a mechanism in which it can safely update a mapping without potentially
2001 having multiple entries in the TLB for the same virtual address. These problems are described in more detail
2002 during the exploration of break-before-make in §8.6.5.

2003 **TODO: PS: no forward refs to tests?**

```
AArch64                                CoWTf.inv+po
                 Initial State
0:R0=desc3(y)
0:R1=pte3(x)
0:R3=x
0:VBAR_EL1=0x1000
0:PSTATE.SP=0b0
physical pa1;
x |-> invalid;
x ↦ pa1;
y |-> pa1;
*pa1 = 1;
identity 0x1000 with code;
                   Thread 0
STR X0,[X1]
LDR X2,[X3]
              Thread 0 EL1 Handler
0x1400:
MOV X2,#0
MRS X20,ELR_EL1
ADD X20,X20,#4
MSR ELR_EL1,X20
ERET
                  Final State
0:X2=0
                     Allow
```



Thread local re-ordering lets the translation (b1) of the load instruction happen earlier than the write to the translation table (a). This allows the load to trigger a data abort (a translation fault, b2).

Figure 8.10: Test CoWTf.inv+po

8.4 Reads not from TLB

The requirement that invalid entries are not cached in the TLB gives us a way to directly observe non-TLB reads: translation table reads which result in a translation fault *must* have come from a non-TLB read.

We will see that these reads have some important properties that software can rely on, but that some of those properties will depend on certain architecture features being enabled (namely FEAT_ETS).

In this section will we explore the properties these reads have, and the guarantees software can rely on. We shall see that these reads are affected by thread-local re-ordering, even to a greater extent than data memory reads, and the synchronization that recovers the sequential semantics. We will see how these reads from the translation table walk relate to data memory reads, with respect to coherence, multi-copy atomicity, write forwarding and so on. Finally, we will see how the FEAT_ETS architectural feature can change the required synchronization software needs to perform.

8.4.1 Out-of-order execution

First, let us consider whether reads that do not come from the TLB preserve the original program order.

po-previous writes One of the simplest questions one might ask is whether a translation-table-walk non-TLB read can ignore a program-order previous store.

This scenario is captured by the CoWTf.inv+po test [Figure 8.10]. Starting with a VA x initially invalid at level 3, and so cannot have its level 3 entry cached in any TLB (directly or indirectly), the test then overwrites the invalid entry with a new valid entry pointing to the physical address pa1. Program-order later, the thread then attempts to read x.

We see that the thread can take a translation fault. This fault is caused by reading an invalid entry, which was read from a stale entry in memory, ignoring the program-order previous store to the translation table entry's location.

One explanation that suffices to allow this outcome is that the instructions can be locally re-ordered; the translation table walk of the later load instruction can happen much earlier than the program-order previous store, and satisfy its read from memory first.

po-previous reads Similarly, the reads of a translation table walk can be locally re-ordered with respect to program-order earlier loads of the translation table entry, as demonstrated in the CoRpteTf.inv+po test [Figure 8.11].

AArch64	CoRpteTf.inv+po
Initial State	
0:R0=**desc3**(y) 0:R1=**pte3**(x)	1:R1=**pte3**(x) 1:R3=x 1:VBAR_EL1=0x1000 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00
option default_tables = true; **physical** pa1; **intermediate** ipa1; x |-> invalid; x ↦ pa1; y |-> pa1; **identity** 0x1000 **with code**; *pa1 = 1;	
Thread 0	Thread 1
STR X0,[X1]	**LDR** X0,[X1] **LDR** X2,[X3]
	Thread 1 EL1 Handler
	0x1400: **MOV** X2,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=**desc3**(y) & 1:X2=0	
Allow	

The translation read (event c1) can be re-ordered with respect to the program-order previous load of `l3pte(x)` (b), even though the load read the new translation table entry, for the same location the translation reads from.

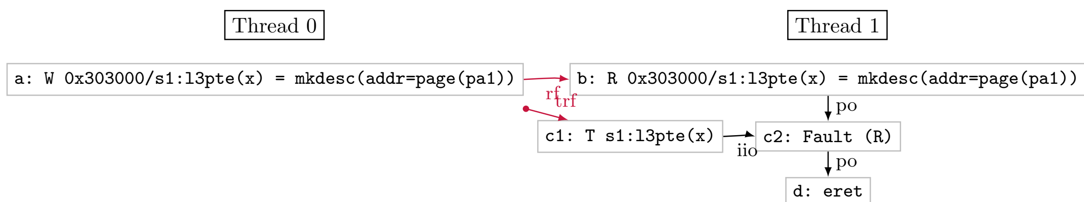


Figure 8.11: Test CoRpteTf.inv+po

AArch64	LB.TT.inv+pos
Initial State	
`0:R1=x`	`1:R1=y`
`0:R2=mkdesc3(oa=pa1)`	`1:R2=mkdesc3(oa=pa1)`
`0:R3=pte3(y)`	`1:R3=pte3(x)`
`0:VBAR_EL1=0x1000`	`1:VBAR_EL1=0x2000`
`0:PSTATE.SP=0b0`	`1:PSTATE.SP=0b0`
`0:PSTATE.EL=0b00`	`1:PSTATE.EL=0b00`

```
physical pa1;
x |-> invalid;
y |-> invalid;
x ↦ pa1;
y ↦ pa1;
*pa1 = 1;
identity 0x1000 with code;
identity 0x2000 with code;
```

Thread 0	Thread 1
`MOV X0,#0`	`MOV X0,#0`
`LDR X0,[X1]`	`LDR X0,[X1]`
`STR X2,[X3]`	`STR X2,[X3]`
Thread 0 EL1 Handler	**Thread 1 EL1 Handler**
`0x1400:`	`0x2400:`
`MRS X13,ELR_EL1`	`MRS X13,ELR_EL1`
`ADD X13,X13,#4`	`ADD X13,X13,#4`
`MSR ELR_EL1,X13`	`MSR ELR_EL1,X13`
`ERET`	`ERET`

Final State
`0:X0=1 & 1:X0=1`
Forbid

The writes to the translation tables (b and d) are forbidden from propagating to other threads before the program-order earlier translations (a1 and c1) are satisfied, forbidding them from reading from each other's writes.

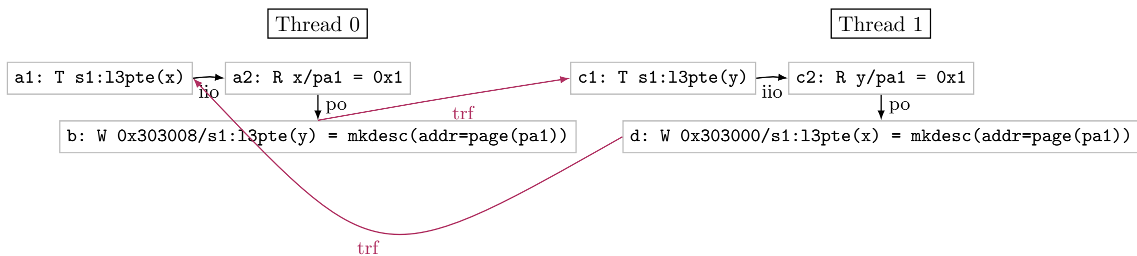


Figure 8.12: Test LB.TT.inv+pos

po-future writes A translation table walk read may not, in general, be re-ordered with program-order later stores.

This is consistent with the description in §8.2.3, as the program-order later store might not architecturally happen if the translation table walk read were to fault. So, the later writes are speculative until the translation has finished, preventing the write from propagating until then.

This forbids both the general re-ordering of the propagation of the write to other threads (LB.TT.inv+pos [Figure 8.12]) with program-order earlier translation table walks, and, translations reading from program-order later writes (CoTW1.inv [Figure 8.13]).

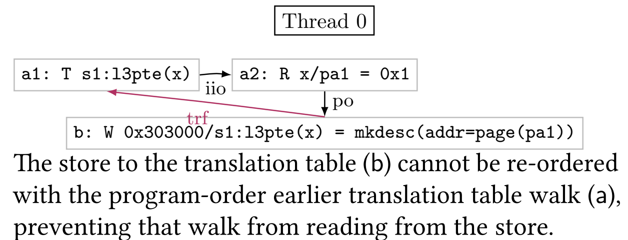
8.4.2 Enforcing thread-local ordering

Since non-TLB reads do not necessarily preserve the program order, it appears that there are no coherence guarantees one can make about them. However, by introducing some thread-local ordering constructs, we can recover some of the strong guarantees we are used to.

To force a non-TLB read to happen after some program-order earlier event we can insert the two-instruction sequence `DSB SY ; ISB` between them. The `DSB` ("Data Synchronization Barrier") waits for all loads to satisfy and for all stores to have finished and be visible to translation table walkers, before the `ISB` ("Instruction Synchronization Barrier") flushes the pipeline and restarts any program-order later instructions, including any translation table walks they perform.

Locally-ordered-previous writes If we introduce this sequence into the previous CoWTf.inv+po test we obtain the CoWTf.inv+dsb-isb test [Figure 8.14], which is forbidden by Arm. This is because the non-TLB reads, in the

```
AArch64                                  CoTW1.inv
                  Initial State
0:R1=x
0:R2=desc3(y)
0:R3=pte3(x)
0:VBAR_EL1=0x1000
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
physical pa1;
x |-> invalid;
x ↦ pa1;
y |-> pa1;
*pa1 = 1;
identity 0x1000 with code;
                    Thread 0
LDR X0,[X1]
STR X2,[X3]
               Thread 0 EL1 Handler
0x1400:
MOV X0,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
                   Final State
0:X0=1
                     Forbid
```



The store to the translation table (b) cannot be re-ordered with the program-order earlier translation table walk (a), preventing that walk from reading from the store.

Figure 8.13: Test CoTW1.inv

absence of non-coherent TLB caching structures (discussed more in §8.6.1), will read from the coherent storage subsystem, and so will be required to see the new write, or something coherence after it.

Locally-ordered-previous reads If a program-order previous load has already seen some other-thread write, either through a translation (CoTTf.inv+dsb-isb [Figure 8.15]), or through a normal data load of the translation table (CoRpteTf.inv+dsb-isb [Figure 8.16]), then translation table non-TLB reads which are ordered after that read must also see that write, or a write coherence after it. These tests use the DSB; ISB sequence previously described, but any ordering to the translation table walk (described in §8.4.3) will suffice.

Microarchitecturally this is because translation table walkers are 'separate observers'. The idea is that the MMU performs reads of memory the same way any of the other observers (threads) do, meaning that those reads behave almost exactly like normal data memory reads.

This 'separate observers' principle is a reasonable model, however, we will see later on in §8.4.4 where it begins to break down.

Instruction synchronization barrier and control dependencies The ISB instruction naturally orders all translation table walks of program-order later instructions with the ISB itself. This is because the ISB effectively restarts all program-order later instructions, including any translations they do.

However, an ISB is not naturally ordered with respect to program-order *earlier* instructions. That is why in the previous tests we introduced a DSB. But a control-dependency would also work (CoTTf.inv+ctrl-isb [Figure 8.17]).

Address dependencies In previous work, address dependencies were assumed fundamental, but now we can define what an address dependency is: a register dataflow dependency into the translation table walk reads.

Address dependencies remain a strong way to order events. Arm, here and in general, avoid speculation of the values and addresses of the explicit reads and writes to memory. This means that a translation table walk will not start until after its address dataflow dependent registers are fully determined. Note, that this does not mean that pre-fetching and caching of the walk cannot happen, it's just that the architectural translation table walk must retrieve any cached values after it is known what the address will be, see §**TODO: ?REF?**.

For non-TLB translation reads this means that a non-TLB read is locally ordered after any read whose value flows into the non-TLB read, as in CoRpteTf.inv+addr [Figure 8.18].

Memory barriers Much of the earlier work in relaxed-memory concurrency was dedicated to the behaviour of *barriers*. The Arm data memory barrier (DMB) creates ordering between memory events program-order earlier than the barrier, with memory events program-order after the barrier.

AArch64	CoWTf.inv+dsb-isb
Initial State	

`0:R0=desc3(y)`
`0:R1=pte3(x)`
`0:R3=x`
`0:VBAR_EL1=0x1000`
`0:PSTATE.SP=0b0`	

`physical pa1;`
`x |-> invalid;`
`x ↦ pa1;`
`y |-> pa1;`
`*pa1 = 1;`
`identity 0x1000 with code;`	

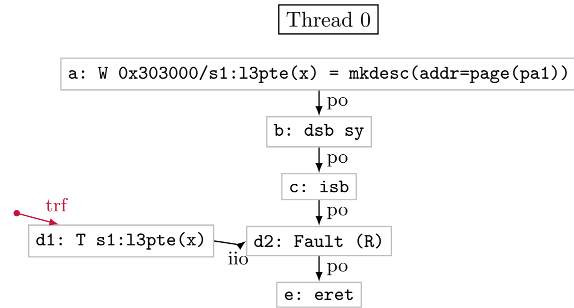
Thread 0	

`STR X0,[X1]`
`DSB SY`
`ISB`
`LDR X2,[X3]`	

Thread 0 EL1 Handler	

`0x1400:`
`MOV X2,#0`
`MRS X20,ELR_EL1`
`ADD X20,X20,#4`
`MSR ELR_EL1,X20`
`ERET`	

Final State	
`0:X2=0`	
Forbid	



The write to the translation table (a) is ordered before the non-TLB read of the entry (d1) because of the intervening DSB;ISB sequence, creating local order. This ordering ensures that the non-TLB read respects the coherence order up to the point of the write a, preventing the non-TLB read from reading from a write coherence-before a.

Figure 8.14: Test CoWTf.inv+dsb-isb

AArch64	CoTTf.inv+dsb-isb
Initial State	

`0:R0=desc3(y)`
`0:R1=pte3(x)`
	`1:R1=x`
	`1:R3=x`
	`1:VBAR_EL1=0x1000`
	`1:PSTATE.SP=0b0`
	`1:PSTATE.EL=0b00`

`physical pa1;`
`x |-> invalid;`
`x ↦ pa1;`
`y |-> pa1;`
`*pa1 = 1;`
`identity 0x1000 with code;`	

Thread 0	Thread 1

`STR X0,[X1]`	`LDR X2,[X1]`
	`MOV X0,X2`
	`DSB SY`
	`ISB`
	`LDR X2,[X3]`

	Thread 1 EL1 Handler

	`0x1400:`
	`MOV X2,#0`
	`MRS X13,ELR_EL1`
	`ADD X13,X13,#4`
	`MSR ELR_EL1,X13`
	`ERET`

Final State	
`1:X0=1 & 1:X2=0`	
Forbid	

The second translation-table non-TLB read of x (e1) is locally ordered after the first translation table walk (b1) because of the intervening dsb; isb sequence, and so cannot see a write coherence-before the write the earlier (b1) translation-read read from.

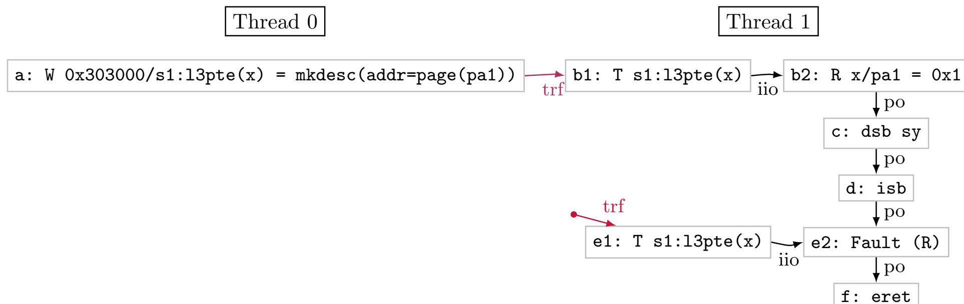


Figure 8.15: Test CoTTf.inv+dsb-isb

AArch64	CoRpteTf.inv+dsb-isb
Initial State	

`0:R0=desc3(y)`
`0:R1=pte3(x)`
	`1:R1=pte3(x)`
	`1:R3=x`
	`1:VBAR_EL1=0x1000`
	`1:PSTATE.SP=0b0`
	`1:PSTATE.EL=0b00`

`option default_tables = true;`
`physical pa1;`
`intermediate ipa1;`
`x |-> invalid;`
`x ↦ pa1;`
`y |-> pa1;`
`identity 0x1000 with code;`
`*pa1 = 1;`	

Thread 0	Thread 1

`STR X0,[X1]`	`LDR X0,[X1]`
	`DSB SY`
	`ISB`
	`LDR X2,[X3]`

	Thread 1 EL1 Handler

	`0x1400:`
	`MOV X2,#0`
	`MRS X13,ELR_EL1`
	`ADD X13,X13,#4`
	`MSR ELR_EL1,X13`
	`ERET`

Final State	
`1:X0=desc3(y) & 1:X2=0`	
Forbid	

The final translation table walk of `x` (e1) cannot be re-ordered with the program-order previous load of `pte3(x)` (b), because of the intervening `DSB;ISB` sequence. The non-TLB translation read of `pte3(x)` (e1) therefore must read from the same write as the earlier load, or something coherence-after it.

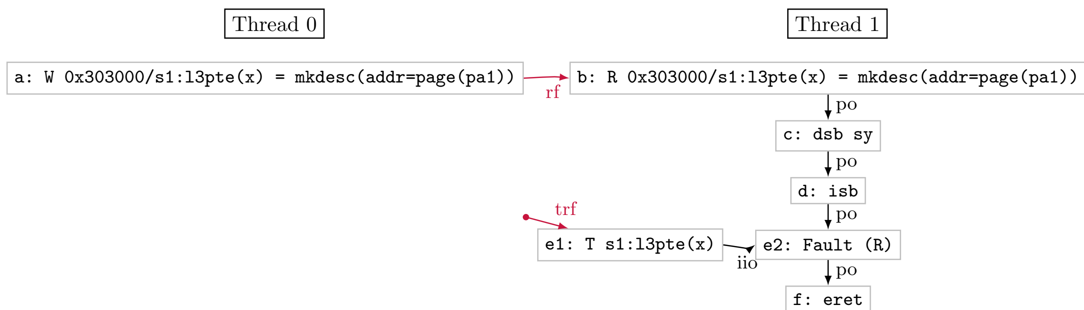


Figure 8.16: Test CoRpteTf.inv+dsb-isb

AArch64	CoTTf.inv+ctrl-isb
Initial State	

`0:R0=desc3(y)`
`0:R1=pte3(x)`
	`1:R1=x`
	`1:R3=x`
	`1:VBAR_EL1=0x1000`
	`1:PSTATE.SP=0b0`
	`1:PSTATE.EL=0b00`

`physical pa1;`
`x |-> invalid;`
`x ↦ pa1;`
`y |-> pa1;`
`*pa1 = 1;`
`identity 0x1000 with code;`	

Thread 0	Thread 1

`STR X0,[X1]`	`MOV X0,#0`
	`LDR X0,[X1]`
	`EOR X4,X0,X0`
	`CBNZ X4,LC00`
	`LC00:`
	`ISB`
	`MOV X2,#0`
	`LDR X2,[X3]`

	Thread 1 EL1 Handler

	`0x1400:`
	`MRS X13,ELR_EL1`
	`ADD X13,X13,#4`
	`MSR ELR_EL1,X13`
	`ERET`

Final State	
`1:X0=1 & 1:X2=0`	
Forbid	

Control-ISB locally-orders the later translation table walk (d1) after the resolution of the control flow, which happens only after the satisfaction of the read b2.

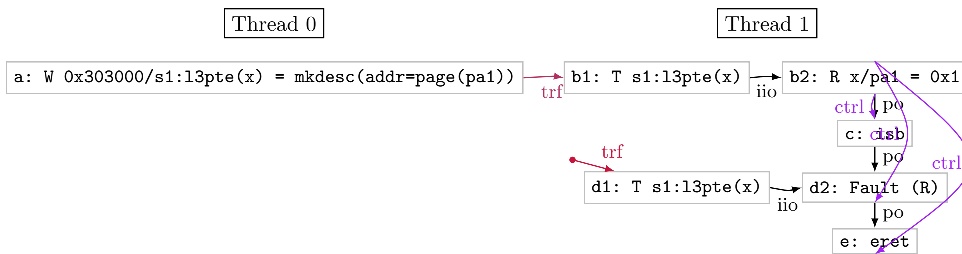


Figure 8.17: Test CoTTf.inv+ctrl-isb

AArch64	CoRpteTf.inv+addr
Initial State	
0:R0=**desc3**(y) 0:R1=**pte3**(x)	1:R1=**pte3**(x) 1:R3=x 1:VBAR_EL1=0x1000 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00
option default_tables = true; **physical** pa1; **intermediate** ipa1; x |-> invalid; x ↦ pa1; y |-> pa1; **identity** 0x1000 **with code**; *pa1 = 1;	
Thread 0	Thread 1
STR X0,[X1]	**LDR** X0,[X1] **EOR** X4,X0,X0 **LDR** X2,[X3,X4]
	Thread 1 EL1 Handler
	0x1400: **MOV** X2,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=**desc3**(y) & 1:X2=0	
Forbid	

The address dependency from the load b to the second load, orders the reads due to the translation table walk of that load (c1) after b. Since c1 is a non-TLB read, it cannot read from a write coherence-before the write b read from.

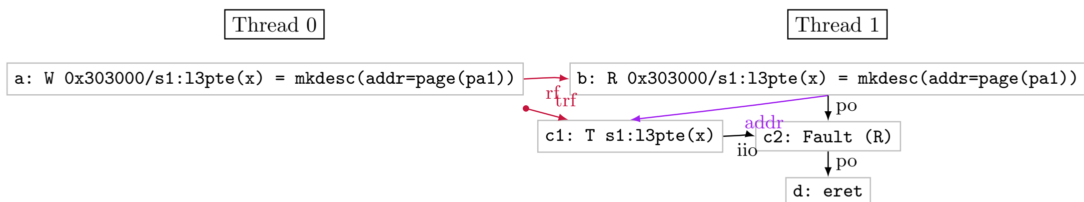
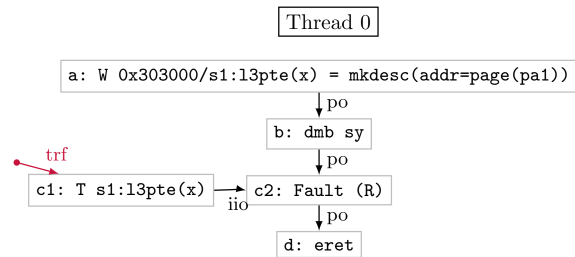


Figure 8.18: Test CoRpteTf.inv+addr

AArch64	CoWTf.inv+dmb
Initial State	
0:R0=**desc3**(y) 0:R1=**pte3**(x) 0:R3=x 0:VBAR_EL1=0x1000 0:PSTATE.SP=0b0	
physical pa1; x |-> invalid; x ↦ pa1; y |-> pa1; *pa1 = 1; **identity** 0x1000 **with code**;	
Thread 0	
STR X0,[X1] **DMB** SY **LDR** X2,[X3]	
Thread 0 EL1 Handler	
0x1400: **MOV** X2,#0 **MRS** X20,ELR_EL1 **ADD** X20,X20,#4 **MSR** ELR_EL1,X20 **ERET**	
Final State	
0:X2=0	
Allow (if not ETS)	



The non-TLB read c1 is not locally ordered after the write a, despite the intervening dmb sy barrier (b).

Figure 8.19: Test CoWTf.inv+dmb

We will see that this applies to *explicit* memory events only: the principle reads and writes that load and store instructions perform, not the implicit reads and writes they do during translations (or instruction fetching, **TODO: ref: ifetch chapter**).

Ordering of the explicit memory events does not, automatically, induce ordering between those explicit events and any reads due to translation table walks performed by those instructions. In the next subsection, we will see how FEAT_ETS (§8.4.3) extends the architecture to include more orderings between translations and other memory events in the same thread.

Figure 8.19 shows a simple coherence test, with a data memory barrier between a store to the translation tables and a load whose translation table walk might read from that. We can see that the barrier does not enforce that the translation table walk sees the update to the translation tables. From the previous tests, we know this means that the translation table walk happened (microarchitecturally) before the store was propagated to memory.

The arm DMB vs DSB instructions **TODO: PS: discuss DMB v DSB**

The architectural intent for DMB's ordering with respect to translation table walkers in the absence of FEAT_ETS is still tentative, so we shall focus on the fragment with FEAT_ETS**TODO: ... and continue.**

8.4.3 Enhanced Translation Synchronization

TODO: PS: litmus tests?

Recent versions of the Arm architecture require support for FEAT_ETS: Enhanced Translation Synchronization. This feature does not change the ISA, but instead, requires implementations to enforce extra ordering.

The Arm Architecture Reference Manual says the following [1, D5.2.5 (p4802)] :

> If FEAT_ETS is implemented, and a memory access RW1 is Ordered-before a second memory access RW2, then RW1 is also Ordered-before any translation table walk generated by RW2 that generates any of the following:
> - ▷ A Translation fault.
> - ▷ An Address size fault.
> - ▷ An Access flag fault.

This prose description is a little ambiguous, and we feel, needs some clarification: The scenario being described here is a case with two instructions, $\mathtt{I}_1$ and $\mathtt{I}_2$, each either a load or store. Imagine $\mathtt{I}_1$ and $\mathtt{I}_2$ both executing to completion, without generating any translation, address size, or access flag faults. Then, each instruction would

have generated one or more explicit memory events. For example, a store might generate up to 8 separate write events (one for each byte). Call those events $\mathtt{E}_{ij}$ for the jth explicit event of instruction $\mathtt{I}_i$.

Each explicit event $\mathtt{E}_{ij}$ would have required a translation table walk, generating translation read events which we can call $\mathtt{T}_{ijk}$ for the kth translation-table-walk read for the jth explicit memory event for instruction $\mathtt{I}_i$.

Then, if $\mathtt{I}_2$ generates a translation, address size, or access flag fault, and $\mathtt{E}_{1n}$ would have been locally-ordered-before $\mathtt{E}_{2m}$ in the imagined execution without the fault, and `FEAT_ETS` is enabled, then, $\mathtt{E}_{1n}$ is locally ordered before any translation table read $\mathtt{T}_{2m_}$ in the execution with the fault.

The intuition here is that, microarchitecturally, on implementations that support `FEAT_ETS`, when an instruction takes an exception, the access that caused it is re-tried once the prefix of instructions is non-restartable. This reduces *spurious aborts*: faults that come from an out-of-order read of a (what is now) stale value from memory.

Other effects For ETS to have the desired effect — of forbidding spurious aborts with standard local orderings such as barriers — then ETS must implicitly enforce more than just the aforementioned ordering constraints.

Specifically, `TLBI` instructions must have stronger thread-local orderings to translation-table walks (described in more detail later); translation table walks must be (other) multi-copy atomic; and, translation table walk reads must be coherent and single-copy atomic.

non-ETS fragment There is a question here as to whether we should consider the non-ETS behaviours of the architecture. On the one hand, hardware in use today is from a pre-ETS version of the architecture and so we cannot assume that the behaviours of those devices are consistent with ETS. On the other hand, ETS is a feature that is widely assumed by software, even if not present on hardware.

Linux, for example, assumes implementations are ETS compatible even when they are not. Building models that capture the full extent of the non-ETS fragment would have questionable benefits as one would have to assume an ETS model when verifying software. Additionally, as ETS is becoming a mandatory feature, the concerns over non-ETS hardware will diminish over time, perhaps even by the publication of this thesis, they will be questions of the past. Finally, the semantics of this non-ETS fragment is still unclear; there are numerous questions, especially around forwarding and multi-copy atomicity generally, which are grey areas in the non-ETS fragment which Arm have yet to explicitly decide one way or another.

For these reasons we will assume `FEAT_ETS` is present and enabled, unless explicitly stated otherwise.

Ordering to the translation table walk We can now define which constructs give rise to local ordering into a translation table walk. Address dependencies, and locally-ordered context-synchronisation (in particular, the `DSB; ISB` sequence) always give rise to ordering to the translation table walks. Control dependencies, on their own, never give rise to such ordering. If using `FEAT_ETS`, then a plain `DSB` orders translation table walks of program-order later instructions after it. **TODO: BS: even if there's no fault?** Other barriers may give ordering to the translation table walker, if using `FEAT_ETS` and the translation results in a translation fault, and those barriers would have ordered the event that would have happened otherwise.

8.4.4 Forwarding to the translation table walker

Writes take time to propagate out to memory to other cores. One common performance optimization is *gathering*: collecting multiple writes together in a store buffer and propagating them all out together.

To maintain uniprocessor semantics, the core can read from its own store buffer, in effect, allowing it to read from writes before they've been propagated out to other cores. This behaviour is referred to as *write forwarding*.

Although the translation table walker is described as a 'separate' observer, it is also part of the core that hosts it, and is allowed to read from that core's store buffer, effectively allowing writes to be 'forwarded' to the walker, as shown in the R.TR.inv+dmb+trfi test [Figure 8.20].

The simplest model here is one where non-TLB translation reads behave as a normal data memory read, reading either from forwarding from the store buffer, or from the coherence-latest write in the storage subsystem.

AArch64		R.TR.inv+dmb+trfi
Initial State		
0:R0=0x2 0:R1=x 0:R2=0x2 0:R3=**pte3**(w)	1:R0=**mkdesc3**(oa=pa1) 1:R1=**pte3**(w) 1:R3=w 1:VBAR_EL1=0x1000 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00	2:R1=**pte3**(w)
physical pa1; w |-> invalid; w ↦ pa1; w ↦ raw(2); x |-> pa1; *pa1 = 0; **identity** 0x1000 **with code**;		
Thread 0	Thread 1	Thread 2
STR X0,[X1] **DMB** SY **STR** X2,[X3]	**STR** X0,[X1] **MOV** X2,#1 **LDR** X2,[X3]	**LDR** X0,[X1] **LDR** X2,[X1]
	Thread 1 EL1 Handler	
	0x1400: **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**	
Final State		
1:X2=0 & 2:X0=2 & 2:X2=**mkdesc3**(oa=pa1)		
Allow		

The write of the new valid entry (d) can be forwarded locally to the translation of `w` (e1) allowing the read of `w` (e2) to satisfy early. **TODO: PS: Thread2 needs explaining**

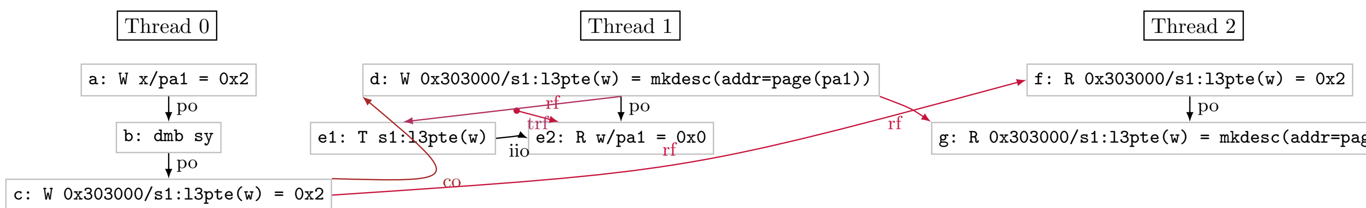


Figure 8.20: Test R.TR.inv+dmb+trfi

AArch64	MP.RTf.inv+dmb+ctrl
Initial State	
0:R0=**desc3**(z) 0:R1=**pte3**(x) 0:R2=0b1 0:R3=y	1:R1=y 1:R3=x 1:VBAR_EL1=0x1000 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00
physical pa1 pa2; x |-> invalid; x ↦ pa1; z |-> pa1; *pa1 = 1; y |-> pa2; **identity** 0x1000 **with code**;	
Thread 0	Thread 1
STR X0,[X1] **DMB** SY **STR** X2,[X3]	**LDR** X0,[X1] **CBNZ** X0,L0 L0: **LDR** X2,[X3]
	Thread 1 EL1 Handler
	0x1400: **MOV** X2,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=1 & 1:X2=0	
Allow	

The non-TLB read in Thread 1 (e1) is not locally ordered after the earlier load (d), despite the control dependency. This is because the processor can speculatively perform the translation table walk, before the earlier read is satisfied.

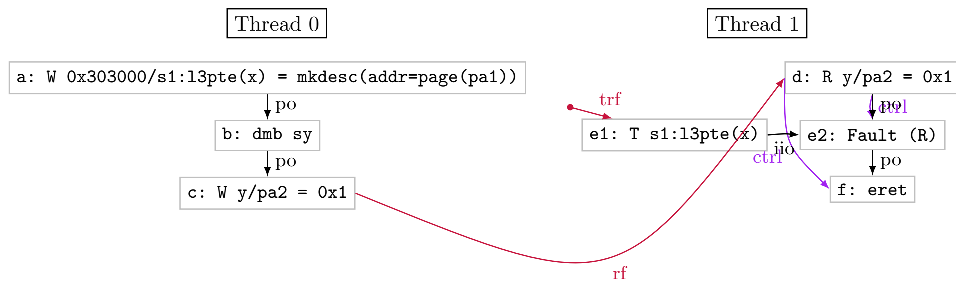


Figure 8.21: Test MP.RTf.inv+dmb+ctrl

8.4.5 Speculative execution

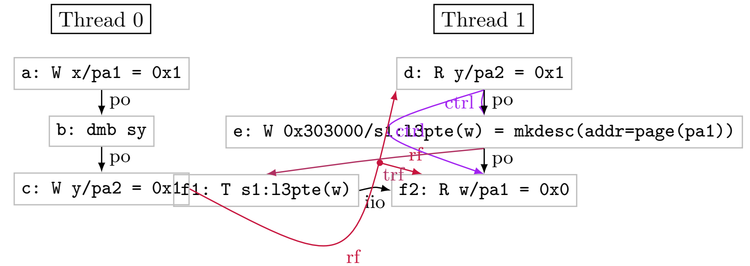
To facilitate fast out-of-order pipelines the machine has to begin fetching and executing the next instruction before the earlier instructions are finished. But, those instructions might control the flow of execution through the program. Executing later instructions before they are finished means that those later instructions are being executed *speculatively*: they may, if the predicted flow turns out to be incorrect, need to be discarded, **TODO: PS: what about restarting on coherence violations?** to avoid the need for rollback across threads.

When executing down a speculative path like this, there are additional restrictions that the core must adhere to. For example, stores should not be propagated out to memory, although they can still be read from by program-order later reads in the same thread.

Since we know reads and writes can be performed speculatively, their associated translations must also be allowed to have been performed speculatively. This is what allows the MP.RTf.inv+dmb+ctrl test [Figure 8.21] to see an old value for the translation table entry, as the translation can be performed speculatively. **TODO: PS: If this were a "user" test, I'd say that e1 was satisfied out-of-order w.r.t. d, not that e1 was "performed speculatively". Or I'd expect to see a test with control-flow speculation, or argument that the second instruction is speculative until the first is known not to fault. Are you not distinguishing between out-of-order and speculative execution any more? TODO: BS: but speculation implies OoO?**

However, forwarding from a speculative write to the translation table walker is disallowed. Since reads to read-sensitive locations (such as devices) can have side-effects, software can protect those locations by marking them as device memory in the translation tables, or leaving them unmapped altogether. A speculative write could update the translation tables arbitrarily, including allowing reads to read-sensitive locations, so it must be forbidden for a

AArch64	MP.RT.inv+dmb+ctrl-trfi
Initial State	
0:R0=0b1 0:R1=x 0:R2=0b1 0:R3=y	1:R1=y 1:R2=**mkdesc3**(oa=pa1) 1:R3=**pte3**(w) 1:R5=w 1:VBAR_EL1=0x1000 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00
physical pa1 pa2; w |-> invalid; w ↦ pa1; x |-> pa1; *pa1 = 0; y |-> pa2; **identity** 0x1000 **with code**;	
Thread 0	Thread 1
STR X0,[X1] **DMB** SY **STR** X2,[X3]	**LDR** X0,[X1] **CBZ** X0,LC00 LC00: **STR** X2,[X3] **LDR** X4,[X5]
	Thread 1 EL1 Handler
	0x1400: **MOV** X4,#2 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=1 & 1:X4=0	
Forbid	



The non-TLB read of the translation table entry (f1) cannot read from a forwarded thread-local write (event e) when on a speculative path, requiring that f1 be ordered after d. **TODO: PS: manual layout this**

Figure 8.22: Test MP.RT.inv+dmb+ctrl-trfi

translation read to read from a still speculative write. The MP.RT.inv+dmb+ctrl-trfi test [Figure 8.22] demonstrates this, requiring that the translation table walk on the speculative path cannot read from the still-speculative store to the translation tables.

Instruction restarts A related, but separate, concept, is that of instruction restarts. In the **TODO: PS: user-mode?** base memory model a read might be satisfied early, out-of-order with respect to program-order previous instructions, even before those instructions' accesses addresses are known. If such an earlier access turned out to be to the same address, and the later access is not a read of the same write, then the later access must be restarted to avoid coherence violations.

Translation table walk reads, while they are reads, do not do this hazard checking, and so are not required to be restarted to recover coherence. See §8.2 for more discussion on this. **TODO: PS: 8.2 has a lot of stuff, point to specifics?**

8.4.6 Single-copy atomicity

In the base memory model, there are two key guarantees on the *atomicity* of reads and writes: single-copy and multi-copy atomicity.

Recall that, single-copy atomic reads always read the maximum it can from another single-copy atomic write; in particular a 64-bit atomic never partially reads from another 64-bit atomic write.

Translation table walk reads are 64-bit single-copy-atomic reads of memory. This means that each of the reads generated by a translation table walk will read the entire descriptor in one shot. This causes the CoWroW.inv+dsb-isb test [Figure 8.23] to be forbidden, disallowing reading the output address obtained from one write, and access permissions from another.

8.4.7 Multi-copy atomicity

Multi-copy atomicity is a guarantee that requires any update to memory to propagate to all other threads simultaneously. This is one of the core guarantees Armv8 and RISC-V give, but earlier versions of Arm and IBM's

AArch64	CoWroW.inv+dsb-isb
Initial State	

```
0:R0=mkdesc3(oa=pa1, AP=0b11)
0:R1=pte3(x)
0:R2=0x1
0:R3=x
0:VBAR_EL1=0x1000
0:PSTATE.SP=0b0

physical pa1;
x |-> invalid;
x ↦ pa1 with [AP = 0b11] and default;
*pa1 = 0;
identity 0x1000 with code;
```

Thread 0

```
STR X0,[X1]
DSB SY
ISB
STR X2,[X3]
```

Thread 0 EL1 Handler

```
0x1400:
MRS X20,ELR_EL1
ADD X20,X20,#4
MSR ELR_EL1,X20
ERET
```

Final State

```
pa1=1
```

Forbid

The translation table walk of the second store must read from the entire write from the earlier store, or not at all, forbidding its translation walk from reading a mix of both the initial state and the earlier write. This means there should be no way the final store can happen, as it must either be invalid or read-only.
Note that, isla does not generate candidates with non-atomic reads which are supposed to be single-copy atomic, and so the diagram is hand-drawn **TODO: Draw it**.

Figure 8.23: Test CoWroW.inv+dsb-isb

TODO: file not found ./islatests/gen/-TODO: ./islatests/gen/diagrams/pdfs/[/[.pdf tex/[.tex]

Figure 8.24: Test [

current Power architectures do not. This has a caveat for Armv8, which is described as *other*-multi-copy atomic: threads can observe their own writes early (through write forwarding).

Microarchitecturally, a thread can only read another thread's write by reading from a global coherent storage subsystem. This ensures that after the thread reads from that write, any other thread must also see that write, or something coherence after it. While this is a property that the base model seems to have, whether it is true for accesses during translation table walks is a separate question.

The non-TLB reads during a translation table walk, in fact, do seem to respect this property: if one other thread has observed a write through a translation table walk then future translation table walk non-TLB reads by other threads will also observe that write (or something newer). Axiomatically, if one thread translation-reads-from a write, then all translation-table-walk reads locally-ordered after another memory event, which is itself ordered after the other thread's translation-table-walk read, will be ordered after that translation-table-walk read.

There are three combinations of multi-thread reads of interest, where a weaker architecture (with separate pagetable and data memory storage) might have mixed non-multi-copy atomic behaviours. The first of these is the most basic; translation-read to translation-read, that is, the pagetable accesses are multi-copy atomic, and this is what forbids reading the old translation table value in Thread 2 in the WRC.TRTf.inv+po+dsb-isb test [Figure ??]. The other two are combinations of read-to-translation-read and translation-read-to-read, these show us that the translation accesses and explicit data accesses are architecturally unified: information about the memory state learned through one kind of access apply to accesses of the other. This is what forbids the following WRC.RRTf.inv+dmb+dsb-isb [Figure ??] and WRC.TRR.inv+po+dsb [Figure ??] tests, from reading the old value from memory at the end of Thread 2.

TODO: PS: these all need text captions WRC.TRTf.inv+po+dsb-isb WRC.RRTf.inv+dmb+dsb-isb **TODO: PS: why DSB not just any R/R ordering.** WRC.TRR.inv+po+dsb **TODO: PS: why DSB not just any R/R ordering.**

TODO: file not found ./islatests/gen/-TODO: ./islatests/gen/diagrams/pdfs/[/[.pdf tex/[.tex]

Figure 8.25: Test [

TODO: file not found ./islatests/gen/-tex/[.tex **TODO: ./islatests/gen/diagrams/pdfs/[/[.pdf]**

Figure 8.26: Test [

TODO: file not found ./islatests/gen/-tex/[.tex **TODO: ./islatests/gen/diagrams/pdfs/[/[.pdf]**

Figure 8.27: Test [

2210 ### 8.4.8 Translation-table-walk intra-walk ordering

2211 All the tests so far have been concerned with changes to at most one of the translation table entries during a
2212 single walk, however, as we saw in §7 a translation table walk may perform many reads for a single translation.

2213 The ASL for the translation table walker performs each translation, in order, starting with the root, and ending
2214 with the leaf entry.

2215 While reads in a thread can be re-ordered, translation-reads within a translation table walk cannot, as this would
2216 require the hardware to do value speculation on the next-level table address, and as discussed in §8.4.5 reading
2217 from speculative values in a translation table walk is generally forbidden.

2218 Requiring the translation reads from a translation table walk to be satisfied in translation walk order has an
2219 observable effect, for example in the following ROT.inv+dsb test [Figure **??**] the translation table walk of the read
2220 in Thread 1 must see the writes to the translation table done by Thread 0 in the order they were propagated out to
2221 memory, and so reading from the old level 3 entry is forbidden.

2222 ROT.inv+dsb The translation-table walk from the read of `x` in Thread 1 must perform its translation non-TLB
2223 reads in the order they appear in the walk, forbidding reading from the new level 2 table entry in `d1`, but then
2224 reading the stale initial value for that entry from memory.

2225 The test listing contains some concrete values to make it executable in isla, namely fixing the location of the new
2226 table at `0x280000` so it's not symbolic, and the exact location of the level 3 entry within the new table will be at
2227 `0x283000` (known from the fixed isla configuration). Whether the exception comes from the level 2 or the level 3
2228 entry can be determined by reading the `ISS` field of the `ESR_EL1` register, which the exception handler does.

2229 ### 8.4.9 Multiple translations within a single instruction

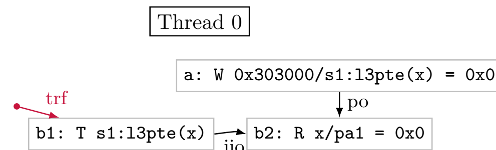
2230 Some instructions generate multiple explicit memory events, such as for the load pair and store pair instructions,
2231 or misaligned accesses, or potentially some read-modify-writes. When there are multiple explicit memory events,
2232 there will be a dedicated translation for each of them, with its own translation table walk.

2233 Here the architecture as it is written today is overly sequentialised: the ASL for these cases performs each
2234 translation (and the respective access) in some order, but the architectural intent is that the separate translations
2235 should be unordered with respect to each other.

2236 Misaligned accesses, and the load and store pair instructions, should generate explicit memory events and
2237 associated translations which are unordered with respect to each other.

2238 **TODO: PS: litmus test with misalgined?**

```
AArch64                          CoWinvT+po
Initial State
0:R0=0b0
0:R1=pte3(x)
0:R3=x
0:VBAR_EL1=0x1000
0:PSTATE.SP=0b0
physical pa1 pa2;
x |-> pa1;
x ↦ invalid;
identity 0x1000 with code;
Thread 0
STR X0,[X1]
LDR X2,[X3]
Thread 0 EL1 Handler
0x1400:
MOV X2,#1
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
Final State
0:X2=0
Allow
```



The translation read (b1) of the last-level entry for `x` can be re-ordered with respect to the program-order earlier store (a) to `pte3(x)`.

Figure 8.28: Test CoWinvT+po

8.5 Caching of translations in TLBs

We have seen in §8.4 that, while non-TLB reads do not necessarily preserve the program-order without additional synchronisation due to the out-of-order execution of instructions, those translation table reads get satisfied from the coherent storage subsystem or from forwarding from earlier stores, much like the normal explicit data reads do. This section will explore what happens when translation table walk reads may instead be satisfied from the TLB.

Unfortunately for the programmer, the TLB need not be coherent with memory: it can have stale values. This section explores the behaviours that arise from this caching of stale values.

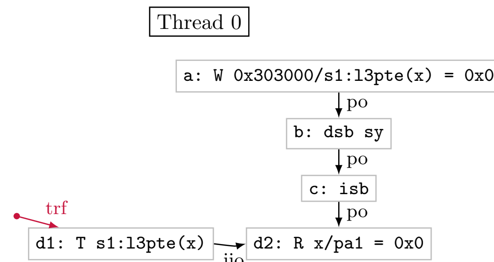
8.5.1 Cached translations

In the previous section we carefully constructed tests which began with an initially invalid translation, to avoid TLB caching issues. Here, we will generally start with entries that are valid, and so might be present in the TLB.

The following CoWinvT+po test [Figure 8.28] begins with an *initially valid* (and therefore potentially initially *cached in the TLB*) translation for the virtual address `x`. It then updates the last-level translation table entry for `x`, setting it to `0`, making it invalid (and thus unmapping `x`). Then, program order later, the same thread tries to read `x`.

The read can succeed, as its translation can read from the old value from memory. We saw earlier that translation table walks can be re-ordered with respect to program order, but even inserting thread-local ordering to the translation, such as in test CoWinvT+dsb-isb [Figure 8.29], does not forbid it.

```
AArch64                    CoWinvT+dsb-isb
Initial State
0:R0=0b0
0:R1=pte3(x)
0:R3=x
0:VBAR_EL1=0x1000
0:PSTATE.SP=0b0
physical pa1 pa2;
x |-> pa1;
x ↦ invalid;
identity 0x1000 with code;
Thread 0
STR X0,[X1]
DSB SY
ISB
LDR X2,[X3]
Thread 0 EL1 Handler
0x1400:
MOV X2,#1
MRS X20,ELR_EL1
ADD X20,X20,#4
MSR ELR_EL1,X20
ERET
Final State
0:X2=0
Allow
```



The translation read (d1) of the last-level entry for x is required to be satisfied after the earlier store (a) to the entry's location because of the intervening `dsb sy; isb` sequence, but can be satisfied from a cached value in the TLB, allowing d1 to read from a stale value.

Figure 8.29: Test CoWinvT+dsb-isb

8.5.2 TLB fills

Translation table walks can be requested by the core in two different ways: (1) through the architectural execution of an instruction; or, (2) from a spontaneous translation table walk (for example, due to speculation and prefetching of data or instructions). In either case, the result of that walk can be cached in the TLB and recalled for other translation table walks.

Architecturally a TLB fill is no different to a normal translation table walk; each fill originates from a non-TLB read, with all the behaviours described in the previous sections. Later translation table walks are allowed, however, to recall an earlier value and then reuse that rather than doing a fresh read.

Spontaneous walks The hardware may, at any time, try to prefetch or speculatively read some address. Architecturally these appear as spontaneous translation table walks. Those spontaneous walks may be cached. We can see this occuring in the following MP.RT.inv+poloc-dmb+ctrl-isb test [Figure 8.30], where a spontaneous translation and the resulting TLB fill allows a future translation table walk to see a stale value.

Speculative paths Since translation table walks, and therefore TLB fills from the result of those walks, can happen at any point, there is no need to consider TLB fills of architectural translation table walks down speculative paths as any such behaviour is subsumed by a spontaneous fill.

However, as described earlier, we saw that writes cannot be forwarded to translation table walks when down speculative paths (§8.4.5), as this would lead to security violations. This naturally excludes TLB fills of still speculative writes; since a speculative write cannot be used in the result of a translation table walk, it cannot end up cached in a TLB.

8.5.3 μTLBs

So far we have covered the idea of something either being in the TLB, or not. But hardware may have multiple micro-TLBs, each with their own potential cached value.

In effect, these micro-TLBs together behave like a larger non-deterministic TLB with potentially many values. The presence of these smaller caching structures in a superscalar machine means that different instructions may be accessing different TLBs at the same time. This allows later instructions to 'skip' over a previously seen cached entry, and then see it again later.

This is most obvious in the CoTfT+dsb-isb test [Figure 8.31], where the presence of these micro-TLBs (or other distributed caching structures) allow later events (even locally-ordered later) to see old cached entries after earlier events witnessed a TLB miss.

AArch64	MP.RT.inv+poloc-dmb+ctrl-isb
Initial State	
0:R0=**mkdesc3**(oa=pa1) 0:R1=**pte3**(x) 0:R2=0b0 0:R3=**pte3**(x) 0:R4=0b1 0:R5=y	1:R1=y 1:R3=x 1:VBAR_EL1=0x1000 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00

physical pa1 pa2;
x |-> invalid;
x ↦ pa1;
y |-> pa2;
*pa1 = 0;
*pa2 = 0;
identity 0x1000 **with code**;	

Thread 0	Thread 1
 STR X0,[X1] **STR** X2,[X3] **DMB** SY **STR** X4,[X5]	**LDR** X0,[X1] **CBNZ** X0,L0 L0: **ISB** **MOV** X2,#1 **LDR** X2,[X3]
	Thread 1 EL1 Handler
	0x1400: **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=1 & 1:X2=0	
Allow	

A spontaneous walk and fill can happen on Thread 1 after the write of the valid entry to `pte3(x)` (a), but before the immediate re-invalidation of that entry (b), allowing the later translation table walk to see the old cached entry (g1), even though the architectural translation table walk could not have happened while the valid entry was visible.

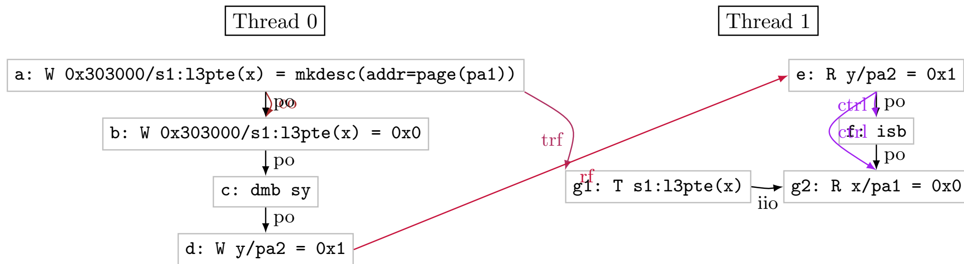
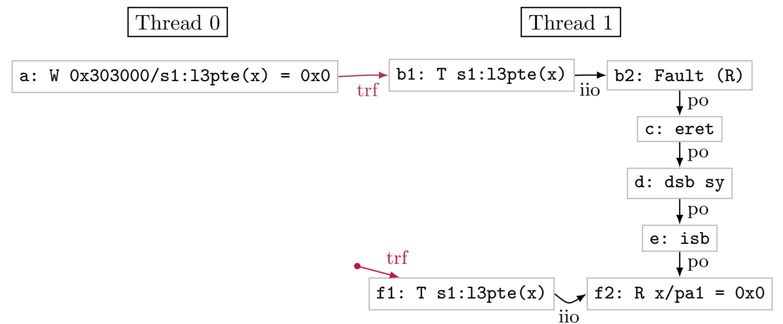


Figure 8.30: Test MP.RT.inv+poloc-dmb+ctrl-isb

AArch64	CoTfT+dsb-isb
Initial State	
0:R0=0b0 0:R1=**pte3**(x)	1:R1=x 1:R3=x 1:VBAR_EL1=0x1000 1:PSTATE.SP=0b0 1:PSTATE.EL=0b00
physical pa1; x |-> pa1; x ↦ invalid; y |-> pa1; *pa1 = 0; **identity** 0x1000 **with code**;	
Thread 0	Thread 1
  STR X0,[X1]	**LDR** X2,[X1] **MOV** X0,X2 **DSB** SY **ISB** **LDR** X2,[X3]
	Thread 1 EL1 Handler
	0x1400: **MOV** X2,#1 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=1 & 1:X2=0	
Allow	



The earlier translation read (b1) reads from the new invalid entry, reading from memory (as it cannot have been in the TLB), but a later translation read (f1) of the same location can still potentially see a stale cached entry.

Figure 8.31: Test CoTfT+dsb-isb

AArch64	MP.RTT.inv3+dmb-dmb+dsb-isb
Initial State	
`0:R0=0b0`	`1:R1=y`
`0:R1=pte2(x)`	`1:R3=x`
`0:R2=mkdesc3(oa=pa1)`	`1:VBAR_EL1=0x1000`
`0:R3=pte3(x)`	`1:PSTATE.SP=0b0`
`0:R4=0b1`	`1:PSTATE.EL=0b00`
`0:R5=y`	

`virtual x y;`
`physical pa1 pa2;`
`assert x[48..21]` = y[48..21];!
`x |-> invalid;`
`x ↦ pa1;`
`x ↦ invalid at level 2;`
`y |-> pa2;`
`*pa1 = 0;`
`*pa2 = 0;`
`identity 0x1000 with code;`	

Thread 0	Thread 1

`STR X0,[X1]`
`DMB SY`
`STR X2,[X3]`
`DMB SY`
`STR X4,[X5]`
	`LDR X0,[X1]`
	`DSB SY`
	`ISB`
	`MOV X2,#1`
	`LDR X2,[X3]`

	Thread 1 EL1 Handler

	`0x1400:`
	`MRS X13,ELR_EL1`
	`ADD X13,X13,#4`
	`MSR ELR_EL1,X13`
	`ERET`

Final State	
`1:X0=1 & 1:X2=0`	
Allow	

The translation-read of the level 2 entry for x (i1) can read from stale writes from a translation that the subsequent level 3 translation-read (i2) does not read from, as the level 2 entry could have been cached in the 'TLB' (in this case, a co-located 'walk cache' structure), while the level 3 entry gets read from memory.

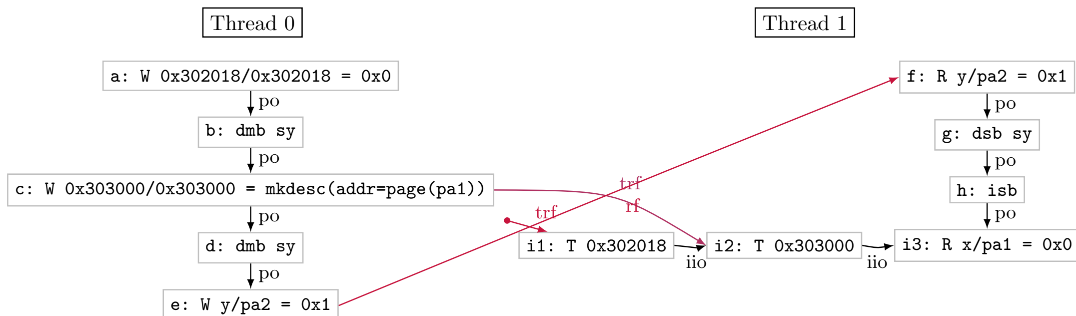


Figure 8.32: Test MP.RTT.inv3+dmb-dmb+dsb-isb

2286 ## 8.5.4 Partial caching of walks

2287 TLBs need not cache entire virtual to physical translations. Instead, they are free to cache any subset of the reads
2288 from the walk separately.

2289 **Caching of last-level table** The most common kind of caching structure we see in microarchitecture is the
2290 walk cache (see §8.3.1). These structures allow a translation table walk to read a stale value for all the translation
2291 reads up to the last level, but then do a separate access for that read. This can be see in the MP.RTT.inv3+dmb-
2292 dmb+dsb-isb test [Figure 8.32], where a walk cache could allow the table entry to be cached separately from the
2293 last-level entry, allowing the last translation read to read from a much newer value.

2294 **Caching of whole translation** A common configuration for the TLB is to cache whole translation walks, from
2295 virtual to physical. This kind of caching has an important caveat: there is no requirement for the TLB to remember
2296 the intermediate physical address of any stage 2 translations that were done during the walk. This includes the
2297 final stage 2 walk of the access address itself.

2298 **Independent caching of IPAs** In a two-stage regime, the virtual addresses are first translated into intermediate
2299 physical address. The secondary translations based on the intermediate physical addresses, either of the final

output address or of any of the intermediate table addresses, may be cached in the TLB without remembering the originating virtual address.

This means these cached translations may be recalled for translations of different virtual addresses.

Pre-fetching may perform translations of arbitrary IPAs, and so these cached translations might not correspond to any valid whole translation table walk, but may still be used during such walks.

This is most clear in ROT.invs1+dmb2 [Figure 8.33], where, although the IPA was never reachable from the stage 1 translations, the old IPA to PA mapping was cached and used later.

Caching of individual entries Architecturally, Arm wish to allow many more implementations of TLBs and translation caching structures than currently known hardware contains.

The weakest variation on this is allowing each individual translation table entry to be cached separately and independently.

One could construct litmus tests for each of the possible combination of translation table entries, or even a 'most relaxed' version where every translation table entry comes from different previous translations. But these tests would be overwhelmingly large, so for simplicity I give just one of them, ROT.inv2+dmb [Figure 8.34]; where the last-level entry came from a newer value than the previous levels.

8.5.5 Reachability

One key property that the TLB must have is that it can only cache translation table entries which are *reachable*. That is, it can only cache an entry which is the result of a valid translation table walk, either using values from memory or other valid translation table entries from the TLB.

In effect, the TLB can not synthesize translation table walks that are not valid.

This means that writes coherence-before the most recent write at the time a translation table entry location becomes reachable are not visible to the walker, and cannot have been cached in any TLB.

Importantly, it is not allowed for the TLB to contain entries for a translation table entry's location from a time when that location was not a valid translation table entry location.

Notably, writes from memory before that memory was reachable by a translation table walk of any VA should not be visible once the location becomes reachable. This is captured in the RUE+isb [Figure 8.35] ("Read-unreachable-entry") test, which is forbidden as the write to the translation table from before the time the location becomes reachable by translation table walkers cannot have been cached in any TLBs, or read from by any spontaneous walks.

This area is currently under discussion with Arm.

8.6 TLB maintenance

Recovering coherence for translation reads in the presence of TLB caching can be achieved through the use of the TLB *maintenance* instruction: `TLBI`.

TLB maintenance generally causes two microarchitectural effects: to erase stale entries from the TLB, ensuring future TLB fills (for example, due to a translation read) will see the coherent value from memory; and, to discard any partially executed instructions, on other cores, which had already begun execution using a stale entry but had not yet finished executing. We will explore both of these effects and the subtle interaction with other parts of the virtual memory systems architecture in more detail throughout this section.

8.6.1 Recovering coherence

By inserting the correct `TLBI` in the previous CoWinvT+dsb-isb test [Figure 8.29], we can produce a new test, CoWinvT.EL1+dsb-tlbi-dsb-isb [Figure 8.36], which is forbidden.

There are many flavours of `TLBI` that could have been inserted into this test, the one in the figure is `TLBI VAE1`, or, TLB invalidation by virtual address, for the EL1&0 translation regime. Using a `TLBI`-by-VA means the programmer

AArch64	ROT.invs1+dmb2
Initial State	
0:R0=**mkdesc3**(oa=pa1) 0:R1=**pte3**(x, s2_table) 0:R2=0b0 0:R3=**pte3**(x, s2_table) 0:R4=**mkdesc3**(oa=ipa1) 0:R5=**pte3**(x) 0:PSTATE.EL=0b01	1:R1=x 1:VBAR_EL1=0x1000 1:VBAR_EL2=0x2000
physical pa1; **intermediate** ipa1; x |-> invalid at level 2; x ↦ ipa1; ipa1 |-> pa1; ipa1 ↦ invalid; *pa1 = 1; **identity** 0x1000 **with code**; **identity** 0x2000 **with code**;	
Thread 0	Thread 1
STR X0,[X1] **DMB** SY **STR** X2,[X3] **DMB** SY **STR** X4,[X5]	**MOV** X0,#0 **LDR** X0,[X1]
	Thread 1 EL1 Handler
	0x1400: **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
	Thread 1 EL2 Handler
	0x2400: **MRS** X13,ELR_EL2 **ADD** X13,X13,#4 **MSR** ELR_EL2,X13 **ERET**
Final State	
1:X0=1	
Allow	

The translation read of the stage 2 leaf entry for x (f2) can read from an old cached version, from a write (a) that was not reachable by any translation table walk for any VA, but only from an orphan IPA.

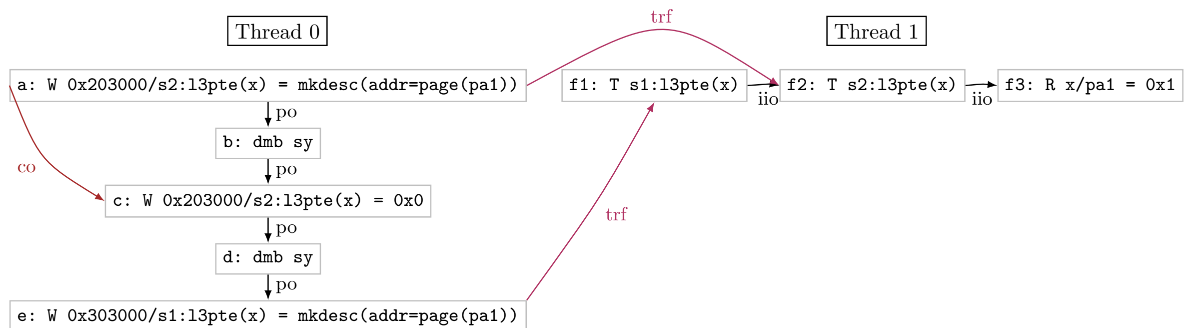


Figure 8.33: Test ROT.invs1+dmb2

AArch64	ROT.inv2+dmb
Initial State	

`0:R0=0b0`
`0:R1=`**`pte3`**`(x, new_table)`
`0:R2=`**`mkdesc2`**`(table=0x283000)`
`0:R3=`**`pte2`**`(x)`
`0:PSTATE.EL=0b01`
	`1:R1=x`
	`1:VBAR_EL1=0x1000`

```
physical pa1;
intermediate ipa1;
assert pa1 == ipa1;
ipa1 |-> pa1;
x |-> invalid at level 2;
x ↦ table(0x283000) at level 2;
s1table new_table 0x280000 {
  x |-> ipa1;
  x ↦ invalid;
};
identity 0x1000 with code;
```

Thread 0	Thread 1

`STR X0,[X1]`
`DMB SY`
`STR X2,[X3]`
	`MOV X0,#1`
	`LDR X0,[X1]`

	Thread 1 EL1 Handler

	`0x1400:`
	`MRS X13,ELR_EL1`
	`ADD X13,X13,#4`
	`MSR ELR_EL1,X13`
	`ERET`

Final State
`1:X0=0`
Allow

The translation-read of the level 3 entry (d2) can read from a stale cached translation, which was cached before the write to the level 2 entry (c). Note that this test assumes that the original `new_table` was reachable (and therefore could be cached) before the write `c`. See §8.5.5 for a discussion on this.

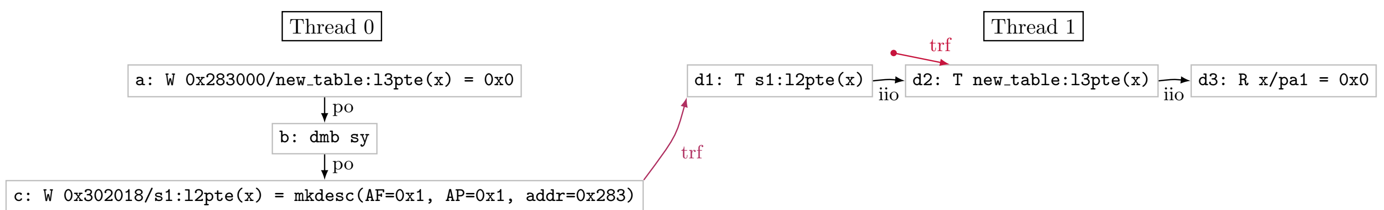


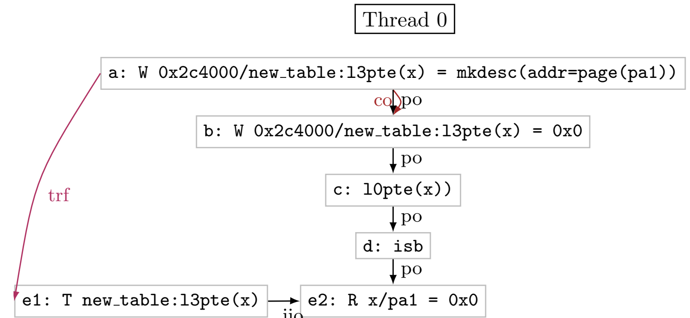
Figure 8.34: Test ROT.inv2+dmb

```
AArch64                                      RUE+isb
                    Initial State
0:R0=mkdesc3(oa=pa1)
0:R1=0x0
0:R2=pte3(x, new_table)
0:R3=ttbr(asid=0x01, base=new_table)
0:R4=x
0:VBAR_EL1=0x1000
0:PSTATE.EL=0b01
0:PSTATE.SP=0b1
intermediate ipa1;
physical pa1;

*pa1 = 0;

s1table new_table 0x2C0000 {
    identity 0x1000 with code;
    x |-> invalid;
    x ↦ pa1;
};

identity 0x1000 with code;
                    Thread 0
01. STR X0,[X2]
02. STR X1,[X2]
03. MSR TTBR0_EL1,X3
04. ISB
05. MOV X1,#1
06. LDR X1,[X4]
               Thread 0 EL1 Handler
01. 0x1200:
02. MRS X20,ELR_EL1
03. ADD X20,X20,#4
04. MSR ELR_EL1,X20
05. ERET
                    Final State
0:X1=0
```

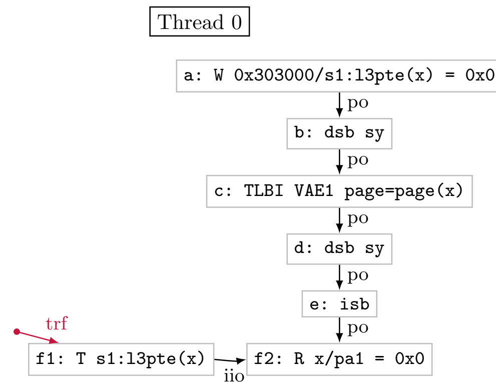


The write to the `new_table` translation table entry for `x` (a) is not visible at the point of the change of `TTBR` (c), and so the later translation table walk (e1) cannot read from it.

Note that isla currently does not do any kind of reachability analysis, and so does not forbid this test.

Figure 8.35: Test RUE+isb

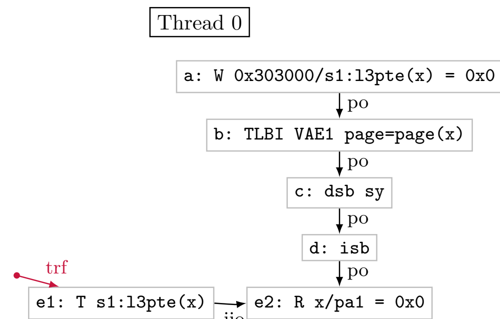
```
AArch64                       CoWinvT.EL1+dsb-
                                     tlbi-dsb-isb
                    Initial State
0:R0=0b0
0:R1=pte3(x)
0:R3=x
0:R5=page(x)
0:VBAR_EL1=0x1000
0:PSTATE.EL=0b01
physical pa1 pa2;
x |-> pa1;
x ↦ invalid;
identity 0x1000 with code;
                    Thread 0
STR X0,[X1]
DSB SY
TLBI VAE1,X5
DSB SY
ISB
LDR X2,[X3]
               Thread 0 EL1 Handler
0x1000:
MOV X2,#1
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
                    Final State
0:X2=0
                      Forbid
```



The read of the translation table entry for `x` (f1) is required to happen after the earlier store (a), because of the intervening `dsb sy; isb` sequence (d and e), and cannot be satisfied from the TLB, because of the `TLBI` (c), forbidding it from still seeing a stale value. Note that `TLBI` instructions can only be executed from EL1, so this test starts execution at EL1 rather than the usual default of EL0.

Figure 8.36: Test CoWinvT.EL1+dsb-tlbi-dsb-isb

```
AArch64                CoWinvT.EL1+tlbi-dsb-isb
Initial State
0:R0=0b0
0:R1=pte3(x)
0:R3=x
0:R5=page(x)
0:VBAR_EL1=0x1000
0:PSTATE.EL=0b01
physical pa1 pa2;
x |-> pa1;
x ↦ invalid;
identity 0x1000 with code;
Thread 0
STR X0,[X1]
TLBI VAE1,X5
DSB SY
ISB
LDR X2,[X3]
Thread 0 EL1 Handler
0x1000:
MOV X2,#1
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
Final State
0:X2=0
```



The TLBI (b) can be re-ordered with program-order earlier events, due to the lack of DSBs ordering it after them, allowing the store (a) to happen later, letting the final translation read (e1) still see the old stale translation.

Figure 8.37: Test CoWinvT.EL1+tlbi-dsb-isb

has to provide the virtual page to invalidate, and the TLBI only affects addresses for that specific invalidated entry, not all of them.

Using the incorrect TLBI leads to insufficient invalidation occuring. For example, if in the aforementioned CoWinvT.EL1+dsb-tlbi-dsb-isb the TLBI had the wrong page, then it would have no effect and the test would remain allowed.

FEAT_nTLBPA

Armv8.4-A introduced a new optional Arm feature, FEAT_nTLBPA [1, A2.2.1 (p79)] .

This feature adds a field to the memory model feature register (AA64MMFR1_EL1) which can identify whether the current processor's TLB (and related microarchitectural caching structures) may contain non-coherent copies of stage 1 entries indexed by those entries intermediate physical address. Microarchitecturally, this corresponds to there being non-coherent caches associated with the TLB, which must be flushed on a TLBI.

These caches would allow TLB misses to read from a non-coherent cache, thus not seeing the most up-to-date value from the coherent storage subsystem like described in §8.4.

Note that the text in the reference manual is a little ambiguous, the entry in A2.2.1 describes it as a "mechanism to identify if [TLB caching] does not include non-coherent caches [of old translation entries] since the last completed TLBI". This change adds a field to the register, whose reserved value in Armv8.0 corresponds to the non-coherent caches existing. This implies that implementation of the feature is not only the existence of the runtime identification register's field, but additionally that its value is 0b0001 (that is, that non-coherent caches do not exist). This further implies that in processors without FEAT_nTLBPA one should assume that TLBs may contain non-coherent caching structures.

8.6.2 Thread-local ordering and TLBI

TLB maintenance instructions are not naturally locally ordered with respect to other instructions in the instruction stream, this means that they can be re-ordered with other instructions. To ensure they are synchronized with other instructions, the programmer can use the DSB barrier instruction to order instructions before and after it.

Leaving out one, or both, of the DSBs around the TLBI leads to insufficient ordering around the TLBI and allows the invalidation to occur at the wrong time. For example, the CoWinvT.EL1+tlbi-dsb-isb test [Figure 8.37] is allowed as the initial write and TLBI may be re-ordered, negating the architectural effect of the TLBI.

TODO: talk about FEAT_ETS

AArch64	MP.RT.EL1+dsb-tlbiis-dsb+dsb-isb
Initial State	
`0:R0=0b0`	`1:R1=y`
`0:R1=pte3(x)`	`1:R3=x`
`0:R2=0b1`	`1:VBAR_EL1=0x1000`
`0:R3=y`	`1:PSTATE.SP=0b0`
`0:R4=page(x)`	`1:PSTATE.EL=0b00`
`0:PSTATE.EL=0b01`	

`physical pa1 pa2;`
`x |-> pa1;`
`x ↦ invalid;`
`y |-> pa2;`
`identity 0x1000 with code;`	

Thread 0	Thread 1

`STR X0,[X1]`
`DSB SY`
`TLBI VAE1IS,X4`
`DSB SY`
`STR X2,[X3]`
	`LDR X0,[X1]`
	`DSB SY`
	`ISB`
	`LDR X2,[X3]`

	Thread 1 EL1 Handler

	`0x1400:`
	`MOV X2,#1`
	`MRS X13,ELR_EL1`
	`ADD X13,X13,#4`
	`MSR ELR_EL1,X13`
	`ERET`

Final State	
`1:X0=1 & 1:X2=0`	
Forbid	

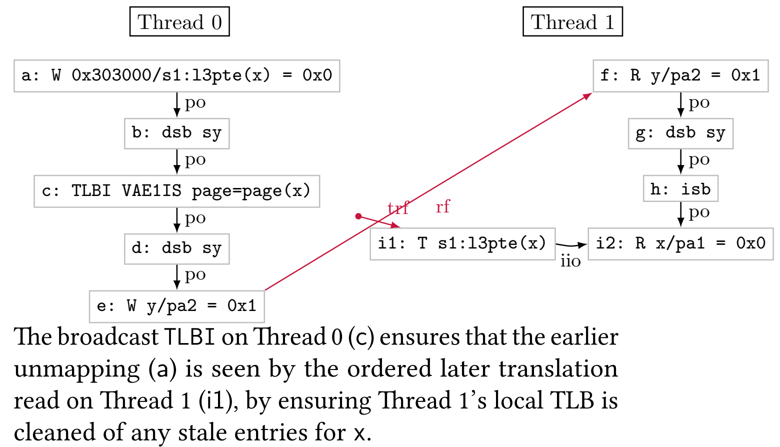


Figure 8.38: Test MP.RT.EL1+dsb-tlbiis-dsb+dsb-isb

8.6.3 Broadcast

Arm provide broadcast variants of the TLBI instructions. These are generally suffixed with the letters IS (for "Inner-shareable").

Broadcast TLBIs, sometimes referred to as TLB *shootdowns*, allow one processor to perform maintenance on another core's TLB.

This is in contrast to other systems, such as for IBM's Power architecture, where maintenance of other cores must be achieved in software through the use of only thread-local invalidation instructions.

TLB invalidation on another core One of the simplest examples is a message passing invalidation pattern, where the old entry is removed and a message is sent to another core. This can be seen in the MP.RT.EL1+dsb-tlbiis-dsb+dsb-isb test [Figure 8.38].

Instruction restarts Broadcast TLBIs must do more than touch the other thread's TLB. If the other processor had already performed translation, using the old stale value, but has not yet finished execution, then that instruction must be restarted.

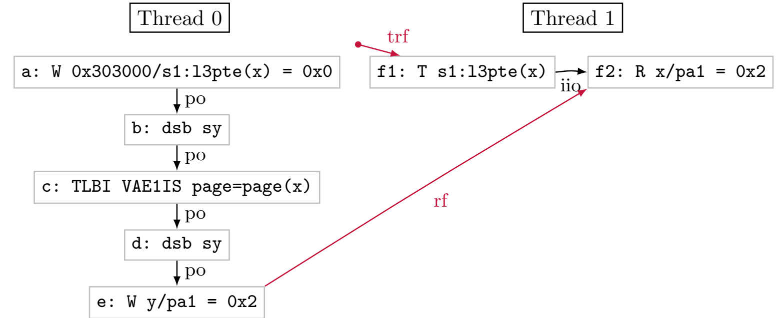
This ensures that Arm broadcast TLBIs have the same behaviour as the traditional software IPI-based shootdown (With context synchronization); but also provides a needed security guarantee.

If a mapping is taken away from a process, then future writes to the physical location it used to map to, should not be visible to that process anymore.

This guarantee is captured in the RBS+dsb-tlbiis-dsb [Figure 8.39] (**Read-broken-secret**) test. Once a mapping has been *broken*, and sufficient TLB maintenance performed, any future reads or writes to the original physical location will not be visible through that mapping anymore. Note, however, that this does not mean that instructions which have already completed their execution will be restarted, even if they occur after an earlier restarted instruction. This can be seen in the RBS+dsb-tlbiis-dsb+poloc test [Figure 8.40], where the program-order later load can see the old value, even after the first faults.

While here I describe things in terms of instruction restarting, these behaviours can be (and presumably are) implemented in terms of waiting: instead of the TLBI forcibly restarting instructions that already started but haven't finished, the TLBI can simply wait for them to complete. This phrasing of waiting for completion is how this process is described in the Arm ARM [1, D5.10.2 (p4928)] .

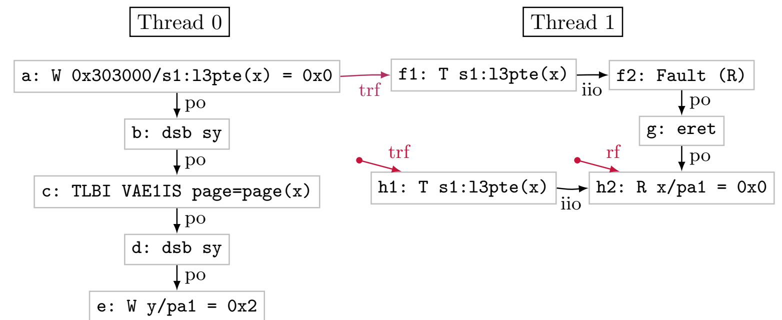
AArch64	RBS+dsb-tlbiis-dsb
Initial State	
0:R0=0b0 0:R1=**pte3**(x) 0:R5=**page**(x) 0:R2=0x2 0:R3=y 0:PSTATE.EL=0b01	1:R1=x 1:VBAR_EL1=0x1000
physical pa1; x |-> pa1; x ↦ invalid; y |-> pa1; *pa1 = 0; **identity** 0x1000 **with code**;	
Thread 0	Thread 1
STR X0,[X1] **DSB** SY **TLBI** VAE1IS,X5 **DSB** SY **STR** X2,[X3]	**LDR** X0,[X1]
	Thread 1 EL1 Handler
	0x1400: **MOV** X0,#1 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=2	
Forbid	



The broadcast TLBI of x (c) ensures that the execution of the load of x in Thread 1 either entirely executes using the old translation and finishes before the TLBI does, or begins execution after the TLBI finishes.

Figure 8.39: Test RBS+dsb-tlbiis-dsb

AArch64	RBS+dsb-tlbiis-dsb+poloc
Initial State	
0:R0=0b0 0:R1=**pte3**(x) 0:R5=**page**(x) 0:R2=0x2 0:R3=y 0:PSTATE.EL=0b01	1:R1=x 1:R3=x 1:VBAR_EL1=0x1000
physical pa1; x |-> pa1; x ↦ invalid; y |-> pa1; *pa1 = 0; **identity** 0x1000 **with code**;	
Thread 0	Thread 1
STR X0,[X1] **DSB** SY **TLBI** VAE1IS,X5 **DSB** SY **STR** X2,[X3]	**MOV** X0,#1 **LDR** X0,[X1] **MOV** X2,#1 **LDR** X2,[X3]
	Thread 1 EL1 Handler
	0x1400: **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
Final State	
1:X0=1 & 1:X2=0	



Even though the broadcast TLBI on Thread 0 (c) ensures that not-yet-completed instructions using the old mapping are restarted, it does not require that the second load of x in Thread 1 (h) be restarted if it has already satisfied its value, as that value must have come from a write before the TLBI.

Figure 8.40: Test RBS+dsb-tlbiis-dsb+poloc

2398 **Atomic `TLBI`s** In the previous RBS-shaped tests, I describe the behaviour in terms of writes that occur 'before'
2399 the `TLBI`.

2400 Microarchitecturally a `TLBI` instruction is very non-atomic: it sends messages to all other cores, performs some
2401 action, and sends messages back to the originating core. The program-order earlier `DSB` ensures that program-order
2402 earlier instructions are complete before sending the messages. The program-order later `DSB` ensures that all
2403 program-order later instructions wait for those messages to return.

2404 The presence of these `DSB`s ensure that the `TLBI`'s effect happens entirely at that point in the instruction stream,
2405 and cannot be broken up and re-ordered amongst the other instructions in the stream. This, coupled with the fact
2406 that these messages *strengthen* and never weaken the behaviour of other cores, means that you cannot observe a
2407 partial `TLBI` effect. So long as the programmer takes care to maintain the required thread-local ordering.

2408 Because of this, we can think of the `TLBI` as executing either before an instruction or after an instruction, but
2409 do not need to consider a `TLBI` executing in the middle of another instruction. This allows us to simplify things,
2410 fitting `TLBI`s into a (generalised) coherence order, with other writes occurring either before or after.

2411 ### 8.6.4 Virtualization

2412 Throughout this section we have considered tests for stage 1 translation with virtual mappings. But many of these
2413 questions and behaviours also apply to the stage 2 intermediate physical mappings, with some key differences.

2414 **Virtual to physical and IPA caches** The existence of TLBs that cache virtual to physical mappings (§8.5.4)
2415 complicates the TLB maintenance sequence required for changes to the intermediate physical mappings.

2416 When invalidating stale second stage entries from the TLB, it is required for the programmer to do *two* sets of
2417 invalidations: first one TLB invalidation to remove any of the old entries for the old IPA to PA, then, perhaps
2418 surprisingly, a second TLB invalidation is needed to remove any stale whole translation, VA to PA mappings or
2419 any combination thereof, as these could have indirectly cached the result of a second stage translation without
2420 remembering the IPA.

2421 This can be seen in MP.RT.EL2+dsb-tlbiipais-dsb+dsb-isb [Figure 8.41], where invalidation of *just* the IPA is not
2422 enough. Adding an invalidation of the VA (or all VAs), like in MP.RT.EL2+dsb-tlbiipais-dsb-tlbiis-dsb+dsb-isb
2423 [Figure 8.42], ensures that later translations cannot see the stale value anymore.

AArch64	MP.RT.EL2+dsb-tlbiipais-dsb+dsb-isb
Initial State	
0:R0=0b0 0:R1=**pte3**(ipa1, s2_table) 0:R2=0b1 0:R3=z 0:R4=**page**(ipa1) 0:PSTATE.EL=0b10	1:R1=y 1:R3=x 1:VBAR_EL2=0x2000 1:PSTATE.EL=0b00

physical pa1 pa2;
intermediate ipa1 ipa2;
x |-> ipa1;
ipa1 |-> pa1;
ipa1 ↦ invalid;
y |-> ipa2;
ipa2 |-> pa2;
z |-> pa2;
identity 0x2000 **with code**;
*pa1 = 0;
*pa2 = 0;	

Thread 0	Thread 1
STR X0,[X1] **DSB** SY **TLBI** IPAS2E1IS,X4 **DSB** SY **STR** X2,[X3]	**LDR** X0,[X1] **DSB** SY **ISB** **MOV** X2,#1 **LDR** X2,[X3]
	Thread 1 EL2 Handler
	0x2400: **MRS** X13,ELR_EL2 **ADD** X13,X13,#4 **MSR** ELR_EL2,X13 **ERET**
Final State	
1:X0=1 & 1:X2=0	
Allow (if not ETS)	

Despite the TLB invalidation of the stale IPA (c), a later stage 2 translation-read of that IPA (i1) can still see the old stale value.

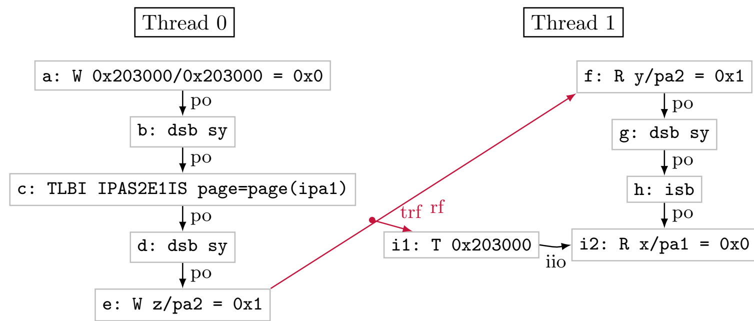


Figure 8.41: Test MP.RT.EL2+dsb-tlbiipais-dsb+dsb-isb

AArch64	MP.RT.EL2+dsb-tlbiipais-dsb-tlbiis-dsb+dsb-isb
Initial State	

`0:R0=0b0`
`0:R1=pte3(ipa1, s2_table)`
`0:R2=0b1`
`0:R3=z`
`0:R4=page(ipa1)`
`0:PSTATE.EL=0b10`
	`1:R1=y`
	`1:R3=x`
	`1:PSTATE.EL=0b00`
	`1:PSTATE.SP=0b0`
	`1:VBAR_EL2=0x2000`

```
physical pa1 pa2;
intermediate ipa1 ipa2;
x |-> ipa1;
ipa1 |-> pa1;
ipa1 ↦ invalid;
y |-> ipa2;
ipa2 |-> pa2;
z |-> pa2;
identity 0x2000 with code;
*pa1 = 0;
*pa2 = 0;
```

Thread 0	Thread 1

`STR X0,[X1]`
`DSB SY`
`TLBI IPAS2E1IS,X4`
`DSB SY`
`TLBI VMALLE1IS`
`DSB SY`
`STR X2,[X3]`
	`LDR X0,[X1]`
	`DSB SY`
	`isb`
	`LDR X2,[X3]`

	Thread 1 EL2 Handler

	`0x2400:`
	`MOV X2,#1`
	`MRS X13,ELR_EL2`
	`ADD X13,X13,#4`
	`MSR ELR_EL2,X13`
	`ERET`

Final State	
`1:X0=1 & 1:X2=0`	
Forbid	

By performing TLB invalidation of the stage 1 entries (e) after invalidating the stage 2 ones (c1), it is guaranteed that the later translation-read (k1) cannot see the old stale value anymore.

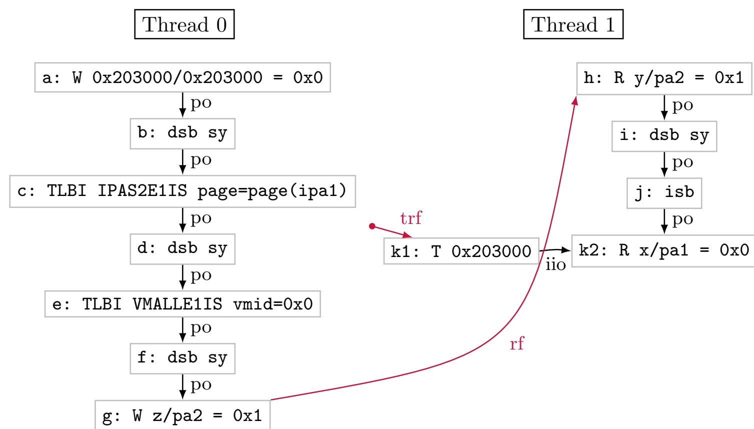


Figure 8.42: Test MP.RT.EL2+dsb-tlbiipais-dsb-tlbiis-dsb+dsb-isb

8.6.5 Break-before-make

TLBs are not required to store only a single cached translation for a given address. There may, in general, be multiple valid translations cached in the TLB.

To avoid this possibility, the architecture provides a *break-before-make* sequence, which will ensure that there cannot be two cached translations existing in the TLB at the same time.

The architecture requires break-before-make when writing to the translation tables to update an already valid entry with a new valid entry, and the change involves any of the following[1]:

- A change in output address, if the new or old entry is writeable.
- A change in output address, if the new and old locations have different contents.
- A change in memory type.
- A change in block size (e.g. replacing a page of 4KiB leaf with a 2MiB block mapping).

For those cases where break-before-make is required, the programmer must:

(1) write an invalid entry to overwrite the currently valid translation table entry in memory;
(2) perform a `dsb sy` (or equivalent);
(3) perform any TLB maintenance required to sufficiently invalidate the old entry from any TLB(s) required;
(4) perform a `dsb sy` (or equivalent);
(5) write the new valid translation table entry, overwriting the old invalid entry.

Litmus test For completeness, the BBM+dsb-tlbiis-dsb [Figure 8.43] gives possibly the simplest valid to valid concurrent update test,

Break-before-make violations

Architecturally, there is no hard requirement to perform break-before-make. Failure to do so simply leads to a degraded state, defined by ConstrainedUnpredictable behaviour.

The Arm reference manual does make it clear that failure to perform break-before-make when required can lead to failure of single-copy atomicity, coherence or even the full breakdown of uniprocessor semantics. While the reference manual does not give motivation for this, we can speculate that this is to allow hardware to perform multiple translations during execution of the instruction, for example, during hazard checking. As such, we do not try to give a full picture of ConstrainedUnpredictable behaviour arising from break-before-make not being followed.

Understanding ConstrainedUnpredictable in full is future work, but a quick summary might be 'any behaviour that this program could have performed if it wanted to'. That is, an instantenous change in the state to a random new state that would have been reachable by executing arbitrary code at that same exception level, security state and translation regime.

8.6.6 ASIDs and VMIDs

In an effort to reduce the expense of TLB maintenance the architecture provides a mechanism to separate out the address spaces by tagging translations with *address space identifiers* (or ASIDs). These ASIDs allow TLB entries to be tagged with only the address space they are used with, and allow TLB maintenance operations to selectively target only the address space being updated.

Crucially, this allows software to switch between address spaces without having to invalidate the TLB.

This idea is extended not just to address spaces at EL1 (used primary for the operating system and its processes), but to EL2 with *virtual machine identifiers* (or VMIDs). These VMIDs serve the same function as ASIDs, giving IDs to address spaces, except in this case IDs to second-stage IPA to PA address spaces.

[1] See the Arm ARM "TLB maintenance requirements and the TLB maintenance instructions" [1, D5.10.1 (p4913)] . for the full list of conditions.

AArch64	BBM+dsb-tlbiis-dsb
Initial State	
`0:R0=0b0`	`1:R1=x`
`0:R1=pte3(x)`	`1:VBAR_EL1=0x1000`
`0:R2=mkdesc3(oa=pa2)`	`1:PSTATE.SP=0b0`
`0:R4=0b1`	`1:PSTATE.EL=0b00`
`0:R6=page(x)`	
`0:PSTATE.EL=0b01`	
`physical pa1 pa2;` `x |-> pa1;` `x ↦ invalid;` `x ↦ pa2;` `identity 0x1000 with code;` `*pa2 = 2;`	
Thread 0	Thread 1
`STR X0,[X1]` `DSB SY` `TLBI VAE1IS,X6` `DSB SY` `STR X2,[X1]`	`LDR X0,[X1]`
	Thread 1 EL1 Handler
	`0x1400:` `MOV X0,#1` `MRS X13,ELR_EL1` `ADD X13,X13,#4` `MSR ELR_EL1,X13` `ERET`
Final State	
`1:X0=0`	
Allow	

The update of the translation table entry for x in Thread 0 follows the break-before-make sequence, first *breaking* x (a), then performing the necessary TLBI sequence (b-c-d), before *making* x be the new mapping (e). This ensures the concurrent access in Thread 1 is guaranteed to see either the old value, the intermediate broken page (and so a page fault), or the new value. This test is the variant whose final state asserts that the old value was read.

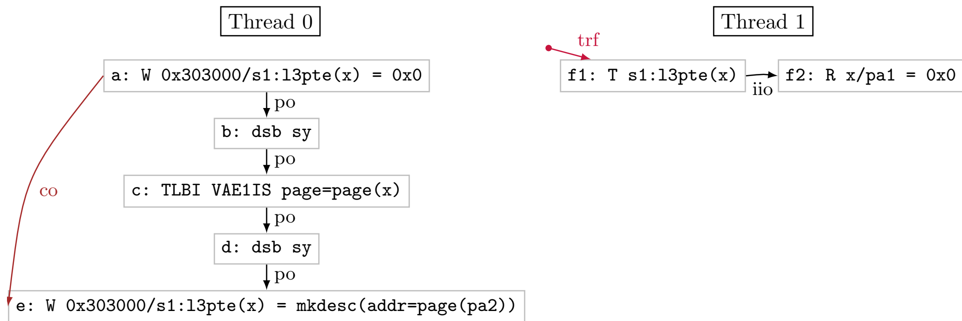


Figure 8.43: Test BBM+dsb-tlbiis-dsb

2465 ### 8.6.7 Access permissions

2466 Accesses which result in permission faults can have been satisfied from the TLB, and writes which update
2467 translation table entries AP field can be cached in the TLB.

2468 Translations can give rise to permission faults. These are unlike translation faults, in that, they are based not just
2469 upon the descriptor read, but also on the *kind* of access requested: whether a read, or a write.

2470 Accesses which result in permission faults result in exceptions, much like translation faults do, but may have been
2471 read from the TLB. This can clearly be seen in the CoWinvTp.ro+dsb-isb test [Figure 8.44], where ordered after a
2472 write to the translation tables a permission failure is experienced, whose descriptor must have come from the TLB.

2473 **Multiple cached entries** The changing of access permissions not necessarily being break-before-make vi-
2474 olations allows us to observe multiple cached entries within the TLB. It is permitted for these entries to exist
2475 simultaneously.

2476 When reading from the TLB, and there existing multiple entries for the same input address, it is allowed for the
2477 hardware to generate a *TLB conflict abort*. These aborts are reported as data aborts.

2478 If the hardware does not generate a conflict abort, then translation reads of that address are CONSTRAINEDUN-
2479 PREDICTABLE, nondeterministically able to read one or the other or an "amalgamation" of the values [1, K1.2.3
2480 (p11243)] .

2481 Here there seems a contradiction:it is not required to perform break-before-make, but there is no requirement that
2482 only one entry be cached in the TLB. We can side step this issue by constructing a test that only changes a single
2483 bit of the descriptor, in a way that is not a break-before-make violation, and therefore avoiding any questions
2484 about how 'amalgamation' of entries happens. This can be seen with the MP.RTpT.ro+dmb-dmb+dsb-isb-dsb-isb
2485 test [Figure 8.45], where the existence of multiple cached entries in the TLB allows multiple translation-reads to
2486 read from different stale writes.

2487 **Atomic TLB reads** Existence of multiple cached translation table entries in the TLB, without break-before-
2488 make violations, introduces the question of whether those TLB fills and subsequence TLB reads must read from
2489 entire single-copy atomic writes of the original translation table entries (much like a read of memory would)
2490 or whether a translation read can read from a mix of different writes. RMD+dmb [Figure 8.46] ("Read-mixed-
2491 descriptor") shows that translation reads cannot read partially read from a write, it must read from the entire
2492 write or none of it.

AArch64	CoWinvTp.ro+dsb-isb

Initial State

```
0:R0=0x0
0:R1=pte3(x)
0:R2=0x1
0:R3=x
0:VBAR_EL1=0x1000
0:PSTATE.SP=0b0
```

```
physical pa1;
x |-> pa1 with [AP = 0b11] and default;
x ↦ invalid;
*pa1 = 0;
identity 0x1000 with code;
```

Thread 0

```
STR X0,[X1]
DSB SY
ISB
MOV X13,#0
STR X2,[X3]
```

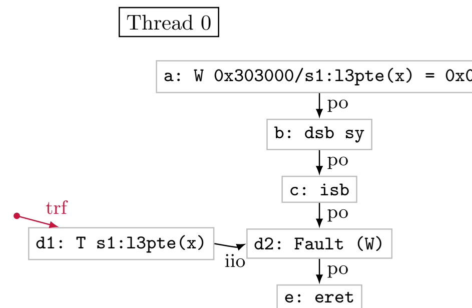
Thread 0 EL1 Handler

```
0x1400:
// read ESR_EL1.ISS to see if Permission or Tra
    MRS X14,ESR_EL1
    AND X14,X14,#0b1111
    CMP X14,#0b1111
    MOV X15,#1 // Permission
    MOV X16,#2 // Translation
    CSEL X13,X15,X16,eq
    MRS X20,ELR_EL1
    ADD X20,X20,#4
    MSR ELR_EL1,X20
    ERET
```

Final State

```
0:X13=1
```

Allow



The translation-read (d1) of x, which happens after the program-order earlier store to the translation tables (a) because of the intervening dsb; isb sequence (b-c), can read from a stale value and result in a permission fault, as the read-only entry from the initial state may be cached in the TLB.

Figure 8.44: Test CoWinvTp.ro+dsb-isb

AArch64	MP.RTpT.ro+dmb-dmb+dsb-isb-dsb-isb
Initial State	

```
0:R0=mkdesc3(oa=pa1, AP=0b10)   1:R1=y
0:R1=pte3(x)                    1:R4=x
0:R2=0b0                        1:VBAR_EL1=0x1000
0:R3=pte3(x)                    1:PSTATE.SP=0b0
0:R4=0b1
0:R5=y
```

```
physical pa1 pa2;
x |-> pa1 with [AP = 0b11] and default;
x ↦ pa1 with [AP = 0b10] and default;
x ↦ invalid;
y |-> pa2;
*pa1 = 0;
identity 0x1000 with code;
```

Thread 0	Thread 1
	`LDR X0,[X1]`
	`DSB SY`
`STR X0,[X1]`	`ISB`
`DMB SY`	`LDR X13,[X4]`
`STR X2,[X3]`	`MOV X2,X13`
`DMB SY`	`DSB SY`
`STR X4,[X5]`	`ISB`
	`LDR X13,[X4]`
	`MOV X3,X13`

	Thread 1 EL1 Handler
	`0x1400:`
	`// read ESR_EL1.ISS to see`
	`MRS X14,ESR_EL1`
	`AND X14,X14,#0b1`
	`CMP X14,#0b1111`
	`MOV X15,#1 // Pern`
	`MOV X16,#2 // Tran`
	`CSEL X13,X15,X16`
	`MRS X20,ELR_EL1`
	`ADD X20,X20,#4`
	`MSR ELR_EL1,X20`
	`ERET`

Final State

`1:X0=1 & 1:X2=1 & 1:X3=0`

Allow

The first translation-read of x (i1) reads from the write that removes read permissions (a) and this write must have come from the TLB because of the intervening invalidation (c), message pass (e-f), and dsb; isb sequence (g-h). The later translation-read of x (m1) can still see an even older value with read permissions, from the initial state, as it may *also* have been cached in the TLB.

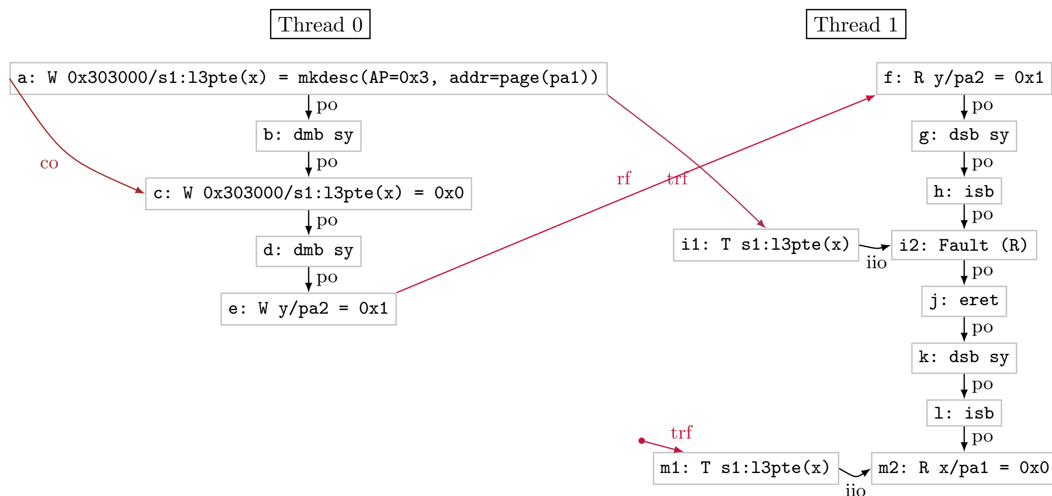


Figure 8.45: Test MP.RTpT.ro+dmb-dmb+dsb-isb-dsb-isb

```
AArch64                                        RMD+dmb
                     Initial State
0:R0=mkdesc3(oa=pa2, AP=0b10)  1:R1=x
0:R1=pte3(x)                   1:VBAR_EL1=0x1000
0:R2=0x1                       1:PSTATE.SP=0b0
0:R3=y
physical pa1 pa2;
x |-> pa1 with [AP = 0b11] and default;
x ↦ pa2 with [AP = 0b10] and default;
y |-> pa2;
*pa1 = 0;
*pa2 = 1;
identity 0x1000 with code;
         Thread 0          |        Thread 1
STR X0,[X1]                | MOV X0,#0
DMB SY                     | LDR X0,[X1]
STR X2,[X3]                |
                           |  Thread 1 EL1 Handler
                           | 0x1400:
                           | MRS X20,ELR_EL1
                           | ADD X20,X20,#4
                           | MSR ELR_EL1,X20
                           | ERET
                      Final State
1:X0=1
                        Forbid
```

The translation-read of x (d1) cannot read from both the 64-bit single-copy atomic write a as well as from the initial state. Note that this test does not, as far as we can see, violate the break-before-make requirements, as currently prescribed by the Arm manual, as the contents in memory of both locations pa1 and pa2 are the same at the time of the write to the translation tables.
This diagram was generated by hand, as isla does not generate a candidate execution of this shape.

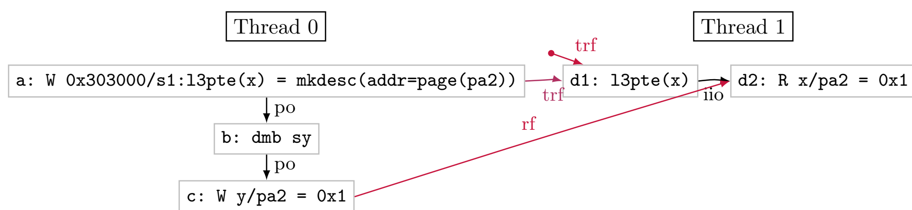


Figure 8.46: Test RMD+dmb

8.7 Context synchronisation

There are many operations which change the current context the system is in. We will focus in on two of these: taking and returning from exceptions, and writing to system registers.

These actions can change the context that the system is executing in: the current exception level, the translation regime, the translation table base, the ASID or VMID, and a variety of other system configuration state.

8.7.1 Relaxed system registers

So far, in this and previous work, register reads and writes have been completely coherent: instructions program-order after a write to a register will always read from that write (or an intervening write) when it reads that register. System registers break this guarantee.

Arm System registers may require the programmer to insert explicit synchronization, as stated clearly in the Arm reference manual [1, D13.1.2 (p5235)] :

> Reads of the System registers can occur out of order with respect to earlier instructions executed on the same PE, provided that both:
> - Any data dependencies between the instructions, including read-after-read dependencies, are respected.
> - The reads to the register do not occur earlier than the most recent Context synchronization event to its architectural position in the instruction stream.

This means a read of a system register might not read from the most recent write to that system register.

To ensure that writes to system registers are seen by program-order later reads, the programmer can ensure that a *Context synchronization* event occurs. These are typically things which flush the pipeline causing future instructions to restart: The `ISB` instruction and taking and returning from exceptions.

There are two important caveats: (1) this does not apply to non-System registers, such as special-purpose or general-purpose registers, and they never require synchronization; and (2), the synchronization required for System registers depends on the *kind* of accesses.

There are typically two kinds of accesses to System registers: direct, and indirect. Direct accesses are the way we think of registers: instructions which specifically read or write to those registers. Indirect accesses happen when an instruction which does not explicitly mention the register by name performs an access, a read or a write, to that register, during the execution of its behaviour.

Because of the out-of-order nature of the pipeline, these indirect register reads and writes may occur out-of-order with respect to any program-order earlier direct reads or writes of that register. This means that before any direct read, and after any direct write, the programmer must perform a context-synchronizing event to ensure that these direct accesses occur in-order with respect to other indirect accesses. The programmer does not have to insert context-synchronization *after* any direct read, as it is guaranteed that register reads or writes cannot be affected by program-order later accesses.

System register ASL In the previous chapter we explored the Arm ASL code for the translation table walk and for one of the store instructions. We saw that this ASL code reads from system registers (as indirect reads).

A naive attempt at a first interpretation of the relaxed semantics is to allow these reads to read-from the most recent indirect write and any program-order later direct writes since the last context synchronization event. However, this would not give the correct behaviour.

The Arm ASL is not written to accommodate relaxed system register behaviours. It leaves questions open about whether these registers can be redundant re-read during execution, whether the instruction reads the entire register at once or piecemeal over the course of execution, and whether repeated accesses to the same register within an instruction are able to read-from different writes. These questions, and others, are still under discussion with Arm.

We will see later in §**TODO: ?REF?** that we give a simple incomplete (and possibly unsound) interpretation in our model in the *pointed set* semantics of system registers, which allows the model to observe some of the known behaviours in this area, without yet fully exploring the architecture.

Caching of system registers in TLBs In addition to being out-of-order due to pipeline effects, some system registers may be indirectly cached within the TLB.

We have already seen one of these: the `MAIR` register. Direct writes to the `MAIR` may not be seen by program-order later translations, even after context-synchronization, as the translations may get their value from the TLB and the TLB may have stored a result which depended on the previous value of the `MAIR`, effectively causing a stale read of it at that point in the instruction stream.

To ensure that an update to the MAIR is observed by later translations therefore requires both TLB maintenance and context synchronization, in that order.

The registers which can be cached in this way, and the behaviours that arise from this caching, are still under current investigation with Arm.

8.8 Contributions

We have now covered all the relaxed memory behaviours, and will, in the next chapter, move on to discuss the extant models created to capture those behaviours. But before that, it may at this point be unclear what the *contribution* of this chapter is. They come in three forms: (1) the attempt at some systematic coverage of the kinds of behaviours which systems software must account for; (2) the precise, formal description (in prose, and as litmus tests) of those behaviours; and, (3) the clarification of the architecture where such behaviours were otherwise unclear.

Coverage of behaviours While this chapter attempts to systematically cover the behaviours we imagine software may try to rely on, starting from the basics of translation table walks and exploring the effects of out-of-order pipelines, caching, and barriers, we cannot claim it is *exhaustive*. As this is a manually compiled and curated list of behaviours, from reading the text and talking with architects, there are surely corner cases missed and software patterns overlooked. However, we believe we have covered those patterns known and required for the features we cover enough for software verification efforts of microkernels and hypervisors.

Clarification of architecture Attempts to clarify the architecture come primarily from the confidential discussions with architects. The behaviours discussed usually fell into one of three categories, whether they were clear already, needed further exploration or are, still, under invesitgation by Arm.

The first major category are those behaviours which were already clear and potentially covered in the architecture text. As alluded to right at the start of this chapter, these are not whole sections or sub-sections or even necessarily whole tests. The most obvious cases are §8.3.3 ('Invalid entries'), §8.2.1 ('Virtual coherence'), and §8.6.5 ('Break-before-make'). These are fundamental behaviours to the correctness of all modern systems software, and for which the architecture reference manual has clear words (at least, enough to cover the basic sequences software rely upon).

Most of the subsections fall into a more general category, of things that either had some associated reference materials, or was otherwise clear from discussion with architects, but for which further invesitgation was needed. This includes: forwarding (§8.4.4) and speculation (§8.4.5) for translation table walks; multi-copy atomic translation table walks (§8.4.7); intra-instruction ordering (§8.4.8,§8.4.9); micro-TLBs (§8.5.3) and partial walk caching (§8.5.4); a variety of `TLBI` questions (§8.6); and, system register accesses (§8.7.1).

Despite the work conducted here, from reading the architecture reference text, discussions with architects, and the testing of existing hardware, there are still many questions which are under current invesitgation with Arm. These include further questions about the scope of `TLBI`s, interaction with exceptions and interrupts, changes in cacheability, translations for instruction fetching, and relaxed system register accesses. Those areas will require more work before giving a concrete semantics.

Chapter 9

An axiomatic VMSA model

This chapter is based, in part, on: Relaxed virtual memory in Armv8-A [54] by Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. Published in the proceedings of the 31st European Symposium on Programming (ESOP, 2022).

We now define a semantic model for Armv8-A relaxed virtual memory that, to the best of our knowledge, captures the Arm architectural intent for the questions discussed in §**??**, including Stage 1 and Stage 2 translation-table walks and the required TLB maintenance.

In §8 we described the design issues in microarchitectural terms, discussing the behaviour of translation table walks and TLB caching, along with the needs of system software. We now abstract from microarchitecture: constructing a model based on ordering between translation-read events and others, avoiding modelling TLBs and out-of-order pipelines directly.

This chapter will present this model, as an extention to the 'user-mode' Armv8-A axiomatic model presented in §**TODO: ?REF?**.

9.1 Extended candidate executions

The base Armv8-A axiomatic model is defined as a predicate over *candidate executions*, each of which is a graph with various events (reads, writes, barriers) and relations over them, notably the per-thread program order `po`, the per-location coherence order `co`, the reads-from relation `rf` from writes to reads, the `addr`, `data`, and `ctrl`-dependency subsets of `po`, and others.

We extend these candidates with both new events and new relations over those events, as well as modifying some of the original ones.

9.1.1 Candidate events

In addition to the events of the original model, we add the following events to the candidates:

- `T` for reads originating from architected translation-table walks.
  These roughly correspond to the actual satisfaction from memory which with TLBs may happen very early.
- `TLBI` events for each `TLBI` instruction, with a single such event per `TLBI` instruction, corresponding to the `TLBI` being completed on all relevant cores.
- `TE` and `ERET` events for taking and returning from an exception (these might not correspond to changes in exception level).
- `MSR` events for writes to relevant system registers, such as the `TTBR`.
- `DSB` events for `DSB` instructions.
- `Fault` events for translation and permission faults.

Translation-reads During execution of the ASL `TranslateAddress` function (§7.6) there will be many reads, which would usually generate `R` events. When those reads happen during the `TranslateAddress` call, they instead generate `T` events. This means that each translation table walk may generate up to 24 `T` events, before the instruction generates the (explicit) `R|W` event.

Alternative representations were explored, including leaving them as `R` events or collecting all reads into a single large translation event. But these options did not give the clarity and fine granularity we desired in the model, and would require more relations and axioms than presented here.

We also choose not to include TLB hits and misses in the model directly, but instead model the TLB as a relaxation of the values the walk can read from, much like normal `R` data memory read events and modelling load buffering, write gathering and caches.

We add a helper set, `T_f`, for translation reads which read-from a write whose value is even. That is, an entry whose invalid bit is `0`. If a translation read results in a fault, either because it was an invalid entry and we get a translation fault, or because the access permissions of the resulting translation do not permit the kind of requested access and so result in a permission fault, the candidate will contain a `Fault` event (partitioned into `Fault_t` and `Fault_p` for translation and permission faults) in `po` order where the explicit memory event would have been. See text on `obETS` for more discussion of these 'ghost' fault events.

We partition the `T` set into two subsets: `Stage1` and `Stage2` for translation read events from a stage 1 or stage 2 walk respectively (stage 2 reads during a stage 1 walk are marked as stage 2, not stage 1).

Finally, we leave the `M` set unchanged, which contains only the explicit reads and writes performed by instructions.

TLBIs As described in §7.7 Arm have a variety of `TLBI` instructions, with varying arguments. All of these `TLBI`s generate a single `TLBI` event.

To aide in modelling, there are a set of subsets of `TLBI` for various kinds of `TLBI`:

- ▷ `TLBI-S1` for invalidations of Stage 1 entries.
- ▷ `TLBI-S2` for invalidations of Stage 2 entries.
- ▷ `TLBI-IPA` for invalidations by intermediate physical address.
- ▷ `TLBI-VA` for invalidations by virtual address.
- ▷ `TLBI-ASID` for invalidations by ASID.
- ▷ `TLBI-VMID` for invalidations by VMID.
- ▷ `TLBI-ALL` for the `TLBI ALL` instructions.
- ▷ `TLBI-IS` for broadcast `TLBI`s.
- ▷ `TLBI-EL1` for invalidations of the EL1&0 regime.
- ▷ `TLBI-EL2` for invalidations of the EL2 regime.

These events do not *cut* the `TLBI` set into partitions, but rather any `TLBI` event may belong to multiple. For example, a `TLBI VAE1IS` event would belong to `TLBI-VA`, `TLBI-VMID`, `TLBI-EL1`, and `TLBI-IS`.

We also include all `TLBI`s in a general `C` ("Cache maintenance") set.

Exceptions Despite not modelling exceptions in general in this work, we do need to include some exception machinery in the model to capture the minimal ordering requirements arising from both their context synchronisation effects and also behaviours from crossing exception level boundaries.

To support this we add two new events to capture taking and returning from exceptions: `TE` ("Take-exception") and `ERET`.

Barriers The Arm `DSB` ("Data synchronization barrier") instruction is required for TLB maintenace, as was seen in the previous chapter. We include `DSB` events, one for each kind of `DSB` instruction:

- ▷ `DSB.SY` and `DSB.ISH` (here, equivalent as we do not model shareability domains)
- ▷ `DSB.NSH`
- ▷ `DSB.ST`
- ▷ `DSB.LD`

Arm define a hierarchy of barriers where, for example: `DMB.LD` < `DMB.SY` < `DSB.SY` That is, any ordering imposed by a `DMB.LD` is also imposed by a `DMB.SY`, and therefore also a `DSB.SY`.

To help capture this, and reduce the explosion in the number of relations in the model later on, we simplify and update the barrier story in the Arm model and include the helper sets given in Figure 9.1.

```
let dsbsy = DSB.ISH | DSB.SY | DSB.NSH
let dsbst = dsbsy | DSB.ST | DSB.ISHST | DSB.NSHST
let dsbld = dsbsy | DSB.LD | DSB.ISHLD | DSB.NSHLD
let dsbnsh = DSB.NSH
let dmbsy = dsbsy | DMB.SY
let dmbst = dmbsy | dsbst | DMB.ST | DSB.ST | DSB.ISHST | DSB.NSHST
let dmbld = dmbsy | dsbld | DMB.LD | DSB.ISHLD | DSB.NSHLD
let dmb = dmbsy | dmbst | dmbld
let dsb = dsbsy | dsbst | dsbld
```

Figure 9.1: Barrier helper sets.

Context changing and synchronisation Finally, we add events for context-changing and context-synchronising operations. Context changes involve updates to system registers which change the current translation regime, which generate MSR events. We add a general context-synchronisation event set CSE which includes ISB, TE and ERET.

Changes to system registers may have relaxed behaviours, as described in §8.7.1, but full relaxation of the system register reads done by the Arm psueocode is unlikely to be valid, consistent or meaningful. Instead, we introduce a *pointed-set semantics*: when generating a candidate, we keep a per-system-register set of writes to that register, remembering which one is the most recent. On a write to that system register, we add it to the set. On a read of that system register, we generate one candidate for each value in the set, and then 'lock' the remainder of the execution of that instruction to that value so repeated reads will see the new value. When a context-synchronization event is generated (that is, an event that will be in the CSE set) all the sets are reduced to singleton sets containing only the most recent write.

This gives us some relaxed behaviours, enough to see relaxed behaviours around changes to the TTBR, but we note that this is unlikely to be the full story for relaxed system registers.

9.1.2 Candidate relations

In addition to those new events, we introduce new relations over those (and other) events:

- trf and tfr as analogues to rf and fr but for translation-read (T) events.
- iio ("intra-instruction order") which relates events of the same instruction in the order they occur during execution of that instruction's intra-instruction semantics as defined by the Arm ASL.
- same-va, same-ipa, same-pa relations which relate events whose virtual, intermediate physical or physical address of the associated explicit memory access are the same.
- same-va-page, same-ipa-page, same-pa-page which relate events whose associated explicit memory events are in the same page (e.g. 4KiB chunk) of the virtual, intermediate physical or physical address space.
- same-asid, same-vmid relates events for which translations for the associated memory event are using the same ASID or VMID.
- wco, a generalised coherence order which includes TLBIs.

Addresses, ASIDs and VMIDs Each translation table walk will read from registers and system registers and get a value for the (input) address, the current ASID and current VMID. We then relate each T with any other T where the translation associated with it is for the same virutal address (with same-va), the same intermediate-physical address (with same-ipa), or the same resulting physical address (same-pa). This means that all T events within a translation have the same same-* relations. We also include relations which match translation's virtual, intermediate physical and physical addresses if they are in the same page rather than exactly, with the same rules, but as a same-*-page relation.

If two translations are for the same ASID, their translation reads are related by same-asid. If two translations are for the same VMID, their translation reads are related by same-vmid.

To use these relations we also include TLBI events. A TLBI-X is related to T by same-X if the parameter to the TLBI instruction (the page, vmid, or asid) either passed by register, immediate or through the current context, if the T event's associated translation matches X. For example, a TLBI-IPA event would be same-ipa-page related to a T whose translation was for an intermediate physical address in the page provided as the parameter to the TLBI IPA instruction.

Generalised coherence order We create an extended coherence order `wco`, which is the usual `co` (a per-location total order of writes to that location) as well as their relative ordering to all `TLBI` events.

One might be concerned at the validity of doing this, on two fronts. Generally, extending coherence to a total order over all locations is sound [9, §10.5 p174], and so there is no issue in doing this. Secondly, for broadcast `TLBI`s, microarchitecture will implement these with message passing to and from each core separately, and so there is no single moment the `TLBI` 'happens'. However, as described in §8.6.7 we seem to be able to consider `TLBI` instructions as executing 'atomically' so long as there are no break-before-make violations. This is a similar justification as to including `DC` and `IC` events in a similar generalised coherence order for instruction fetching [58, §5 p29].

Dependencies A candidate execution consists not only of events, and reads-from relations but also a set of dependencies: `addr`, `data`, `ctrl`, `po` and `loc`. We add `iio` and `tdata` to these.

The intra-instruction ordering `iio` relation relates two events in the same instruction in the order the Arm pseudocode generated the events. This relation therefore captures a total order over all events within an instruction, regardless of the intra-instruction dependencies (control, data) or unordered accesses (for example, for misaligned accesses). We are currently invesitagting a relaxation of this ordering, and associated changes in the underlying Arm pseudocode definitions, to enable a more relaxed definition of the ordering within an instruction to handle these cases.

We make `loc` relate events with the same physical address (for `T` events, this is the physical location of the translation table entry).

Program order (`po`) is restricted to explicit events: `R`, `W`, `F`, `C`, `CSE` and `MSR`. Implicit translation reads (`T`) and any indirect reads or writes of registers are not included in `po`.

Address dependencies were once fundamental, but now we can define address dependencies in the presence of address translation as dependencies into the translation table walk. To do this, we include a new relation `tdata` that relates reads with the translation read events of a translation which reads from the register written by that read to compute the address. The traditional `addr` can be recovered as `tdata ; iio* ; [M]`.

9.2 Cat model

The base Arm axiomatic model had three axioms: `internal`, `external`, and `atomic`. These were acyclicity and emptyness checks of unions of set of relations: `obs, dob, aob` and `bob`. We will slightly modify three of these relations `obs`, `bob` and `dob`, and add 5 new ones (`tob`, `obtlbi`, `ctxob`, `obfault`, `obETS`) to handle the ordering between translations and `TLBI`s, and include them in the `external` acyclicity check. Then we will introduce one final new axiom `translation-internal`.

Figure 9.2 contains the axioms and relations for the updated Armv8-A relaxed virtual memory axiomatic model (`RVM`). Unchanged parts from the original are greyed out. Note that some helper relations are elided here, and will be described in more detail later.

9.3 Axioms

The RVM model axioms are, mostly, a syntactic extension to the original Armv8-A axiomatic model presented in §**TODO: ref intro**. This is deliberate. Although there may be other, perhaps even nicer or more succinct, ways of phrasing the given model, the variation presented here is designed to be syntactically as close as possible to the original. This helps with readability for those familiar with the original; it allows us to present the differences to the original in an easier form; it makes recovery of the original model easier; and, it makes it easier to prove equivalance of the axiomatic models in the presence of constant address translation, increasing the confidence we have in the model.

The model has 3 kinds of axioms: internal ones for per-location guarantees, an external axiom for the global happens-before ordering, and the atomic axiom for RMWs (untouched in this work).

Internal axioms The new model has two per-location axioms: `internal` and `translation-internal`.

```
let speculative =
    ctrl
  | addr; po
  | [T]; instruction-order

(* translation-ordered-before *)
let tob =
    [T_f]; tfre
  | [T]; iio; [R|W]; po; [W]
  | speculative; trfi

(* observed by *)
let obs = rfe | fr | wco
  | trfe

(* ordered-before TLBI and translate *)
let obtlbi_translate =
    [T & Stage1] ; tlb_barriered ; [TLBI-
   S1]
  | ([T & Stage2] ; tlb_barriered ; [TLBI
   -S2])
    & (same-translation ; [T & Stage1] ;
    trf^-1 ; wco^-1)
  | (([T & Stage2] ; tlb_barriered ; [
   TLBI-S2]) ; wco? ; [TLBI-S1])
   & (same-translation ; [T & Stage1] ;
   maybe_TLB_cached)

(* ordered-before TLBI *)
let obtlbi =
    obtlbi_translate
  | [R|W|Fault_T]; iio^-1; (
   obtlbi_translate & ext); [TLBI]

(* context-change ordered-before *)
let ctxob =
    speculative; [MSR]
  | [CSE]; instruction-order
  | [ContextChange]; po; [CSE]
  | speculative; [CSE]
  | po; [ERET]; instruction-order; [T]

(* ordered-before a translation fault *)
let obfault =
    data; [Fault_T & IsFromW]
  | speculative; [Fault_T & IsFromW]
  | [dmbst]; po; [Fault_T & IsFromW]
  | [dmbld]; po; [Fault_T & (IsFromW|
   IsFromR)]
  | [A|Q]; po; [Fault_T & (IsFromW |
    IsFromR)]
  | [R|W]; po; [Fault_T & IsFromW &
    IsReleaseW]

(* ETS-ordered-before *)
let obETS =
    (obfault; [Fault_T]); iio^-1; [T_f]
  | ([TLBI]; po; [dsb]; instruction-order
    ; [T]) & tlb-affects

(* dependency-ordered-before *)
let dob =
    addr | data
  | speculative; [W]
  | addr; po; [W]
  | (addr | data); rfi
  | (addr | data); trfi

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

(* barrier-ordered-before *)
let bob = [R]; po; [dmbld]
  | [W]; po; [dmbst]
  | [dmbst]; po; [W]
  | [dmbld]; po; [R|W]
  | [L]; po; [A]
  | [A | Q]; po; [R | W]
  | [R | W]; po; [L]
  | [F | C]; po; [dsbsy]
  | [dsb]; po

(* Ordered-before *)
let ob = (obs | dob | aob | bob
  | iio | tob | obtlbi | ctxob | obfault
    | obETS)^+

(* Internal visibility requirement *)
acyclic po-loc | fr | co | rf as internal
(* External visibility requirement *)
irreflexive ob as external
(* Atomic requirement *)
empty rmw & (fre; coe) as atomic
(* Writes cannot forward to po-future
   translates *)
acyclic (po-pa | trfi) as translation-
   internal
```

Figure 9.2: RVM axioms and relations

```
(* Internal visibility requirement *)
acyclic po-loc | fr | co | rf as internal
(* Writes cannot forward to po-future translates *)
acyclic (po-pa | trfi) as translation-internal
```

2744

2745 Unchanged from the original, the `internal` axiom captures the SC-per-location guarantee briefly discussed in
2746 §TODO: ?REF?. Translations, however, do not have the same per-location guarantees. To account for this,
2747 we introduce a second axiom, `translation-internal`, which captures the weaker per-location guarantee for
2748 translation table walks. Since translation reads, in the presence of TLB caching and out-of-order pipelines, do not
2749 guarantee even coherence, the only behaviour this axiom ends up preventing is translation reads reading from
2750 program-order later stores.

2751 **External axiom** The `external` axiom asserts acyclicity of the global happens-before ordering for Arm. The
2752 happens-before (called ob, 'ordered-before', in Arm) relation is the union of all the ordering relations, given in
2753 §9.4.

```
(* Ordered-before *)
let ob = (obs | dob | aob | bob | iio | tob | obtlbi | ctxob | obfault |
    obETS)^+
(* External visibility requirement *)
irreflexive ob as external
```

2754

2755 We choose to include all the pipeline and TLB effects as ordering requirements, rather than introducing new
2756 ordering axioms just for translation and TLB invalidation. This produces a model that is more consistent with
2757 the previous Arm memory models, and ensures ordering information gained through observing translation table
2758 walks are respected by non-translation-table accesses.

2759 **Atomic axiom** The `atomic` axiom remains unchanged. In this work we do not consider the interaction of
2760 translation with atomic accesses.

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

2761

2762 9.4 Relations

2763 The RVM model modifies some of the original, and introduces some new, ordering relations. This section goes
2764 through each in detail, describing the mechanism and justifying the existence or non-existence of particular
2765 clauses.

2766 9.4.1 obs

```
(* observed by *)
let obs = rfe | fr | wco | trfe
```

2767

2768 The 'observed-by' relation. It includes the original `rf` and `fr` (over physical locations), the *generalised coherence*
2769 *order* `wco` (§9.1.2), and the `trfe` (translation-reads-from-external) relation.

2770 **Generalised coherence** Including `wco`, which is existentially quantified over the candidates, fixes some global
2771 order the writes and `TLBI`s happen in. Consider, informally, some microarchitectural execution. It would propagate
2772 writes to the coherent storage subsystem, and would complete `TLBI` instructions, and these events would be
2773 interleaved in some whole-machine trace. The generalised `wco` relation captures the relative ordering of these
2774 events in the axiomatic model, as they would have happened in the traces of machine executions. The model is
2775 then quantified over all such orderings, accounting for any interleaving of these events.

External translation reads Inclusion of `trfe` enforces that translation-table-walk translation reads, which could not come from forwarding, must have originally come from the coherent storage subsystem and so the write must have been globally propagated before the translation read happened (§8.4.2, §8.4.7).

However, the translation read might have happened much later, either due to extreme out-of-order (§8.4.1) or TLB caching (§8.5.1), and so we do not include `tfre` (translation-from-reads-external) in `ob`.

Additionally, writes may be propagated to that thread's translation table walker before they are propagated to the coherent storage subsystem (§8.4.4), in other words, they can be forwarded. Therefore we do not include `trfi` (translation-reads-from-internal) in `obs`.

9.4.2 dob

```
let dob =
    addr | data
  | speculative; [W]
  | addr; po; [W]
  | (addr | data); rfi
  | (addr | data); trfi
```

The dependency-ordered-before relation is mostly unchanged, we add a single `(addr | data); trfi` clause to the end to forbid thin-air creation of values (§8.4.1, §8.4.2, **TODO: need dedicated thin air paragraph/test in prev chapter**) similarly to the original model for data memory reads.

9.4.3 bob

```
let bob =
    [R]; po; [dmbld]
  | [W]; po; [dmbst]
  | [dmbst]; po; [W]
  | [dmbld]; po; [R|W]
  | [L]; po; [A]
  | [A | Q]; po; [R | W]
  | [R | W]; po; [L]
  | [F | C]; po; [dsbsy]
  | [dsb]; po
```

We rewrite the original barrier-ordered-before relation to use the barrier helpers defined in Figure 9.1. This does not change the underlying model for `DMB` instructions, but allows those same clauses to capture the barrier hierarchy imposing the same ordering when using stronger barriers (namely, `DSB`s).

The Arm `DSB` instruction has some extra ordering however. Firstly that a `DSB SY` orders `TLBI` instructions (§8.6.2) and so we include `[F|C];po;[dsbsy]`. Secondly, all program-order later events must wait for an earlier `DSB` to finish before performing its explicit memory events, so we also include `[dsb];po` in `ob`.

9.4.4 tob

```
let tob =
    [T_f]; tfre
  | [T]; iio; [R|W]; po; [W]
  | speculative; trfi
```

Translation table walks themselves impose ordering on the surrounding events.

Invalid writes The first of these is one of the key behaviours described in §8.3.3, that reads of invalid entries must not have come from the TLB. So we add the `[T_f];tfre` edge to capture this, that any translation-reads which read an invalid entry must happen before any writes coherence after the one it read from.

There is a major caveat here: write forwarding to the translation table walker. We cannot simply have `[T_f];tfr` as a thread-local write may be forwarded to the translation table walker before it's propagated to memory (§8.4.4).

2805 However, it should not be the case that the write is forwarded from a write that is too old or behind a DSB if
2806 FEAT_ETS, except it may be the case that there might be other intervening writes in between. For now, we are
2807 unable to give a precise bound on the ordering for thread-local [T_f];tfr, and this area is still currently under
2808 investigation with Arm.

2809 **Speculation** As we saw earlier, speculation interacts with translation in two ways: first, it is forbidden to
2810 read-from a still speculative write (§8.4.5), and, secondly, events program-order after an instruction which does a
2811 translation table walk are speculative until the translation table walk completes (§8.4.1).

2812 To capture these we first define when one event is considered speculative until another event happens, with a
2813 new speculative relation, defined as following:

```
let speculative = ctrl | addr; po | [T]; instruction-order
```

2814

2815 This captures all the control-flow dependencies that we model here, the classic ctrl and addr; po, as well as a
2816 new general [T]; instruction-order which says that all events ordered (iio|po)+ after a translation read are
2817 speculative until the translation read satisfies. We can then include speculative ; trfi to succinctly forbid
2818 any forwarding of still-speculative writes to translation table walks.

2819 Finally, we include [T]; iio; [M]; po; [W] which captures that writes cannot propagate until program-order
2820 earlier instructions have their physical address (so, do not fault). Although, this edge is subsumed by the
2821 speculative; [W] edge in dob, it is kept here for clarity.

2822

9.4.5 ctxob

```
let ctxob =
    speculative; [MSR]
  | [CSE]; instruction-order
  | [ContextChange]; po; [CSE]
  | speculative; [CSE]
  | po; [ERET]; instruction-order; [T]
```

2823

2824 The ctxob relation captures the orderings required from context changing and synchronising operations, without
2825 trying to capture the full extent of the relaxed behaviours. As such, these orderings are likely to be incomparable
2826 to the real semantics.

2827 **Speculation** The first guarantee we see is that context changes and synchronisation should not happen specu-
2828 latively. Speculative context changes may end up creating translation table roots and therefore translation table
2829 walks using unreachable writes (§8.5.5). To prevent this we ensure that context changing operations only happen
2830 once they are non-speculative, by enforcing speculative; [MSR] in ob. Forbidding speculative execution of
2831 context synchronisation is done through the inclusion of speculative; [CSE] in ob.

2832 **Context synchronising** A context synchronisation event (such as an ISB or ERET instruction) should ensure that
2833 program-order earlier context-changing events are seen by program-order later instructions. Microarchitecturally
2834 this is achieved by having context-synchronisation events flushing the pipeline, restarting all program-order later
2835 instructions. For now this effect seems fixed in the architecture (§**TODO: the CSE section needs expanding**
2836 **in prev chapter**), and so we get [CSE]; instruction-order in ob subsuming the earlier ISB orderings.

2837 To ensure that context changes are seen after the synchronisation we include [ContextChange]; po; [CSE],
2838 and the union of these two relations ensures the context change is ordered before any program-order later events.

2839 **Exceptions** Taking and returning from exceptions are context synchronising (§**TODO: the CSE section needs**
2840 **expanding in prev chapter**), and so those are captured by the previous clauses. However, translation reads of a
2841 lower exception level should not satisfy during execution at a higher exception level. We over approximate this
2842 with po; [ERET]; instruction-order; [T] ensuring all translation reads after an ERET wait.

2843 ### 9.4.6 obfault and obETS

```
(* ordered-before a translation fault *)
let obfault =
    data; [Fault_T & IsFromW]
  | speculative; [Fault_T & IsFromW]
  | [dmbst]; po; [Fault_T & IsFromW]
  | [dmbld]; po; [Fault_T & (IsFromW|IsFromR)]
  | [A|Q]; po; [Fault_T & (IsFromW | IsFromR)]
  | [R|W]; po; [Fault_T & IsFromW & IsReleaseW]

(* ETS-ordered-before *)
let obETS =
    (obfault; [Fault_T]); iio^-1; [T_f]
  | ([TLBI]; po; [dsb]; instruction-order; [T]) & tlb-affects
```

2844

2845 To capture the specific guarantees described by FEAT_ETS (§8.4.3, §8.6.2), we include 'ghost' Fault events in the
2846 candidate executions. These events sit in the execution (in po order) where the explicit memory event would have
2847 been if there was no fault, and tags the fault with the kind of fault it was (translation or permission).

2848 **Ordering to a fault** To fully capture the strength of FEAT_ETS we keep track of syntactic dependencies *into*
2849 the instruction which faulted, and apply those dependencies to the Fault event itself. obfault then the syntactic
2850 subset of bob and dob where the right-hand side of each clause is substituted with a Fault_T (a translation fault).

2851 Using obfault we can then keep track of the (syntactic) subset of ob that would have ordered the explicit event
2852 after, and associate those relations with the Fault_T event instead. obETS's first clause then adds to ob this
2853 ordering, but attached to the translation read of the invalid entry itself, as architected by FEAT_ETS.

2854 Note that dependencies and orderings *from* a faulting instruction seem not respected, and so we do not induce
2855 orderings out of a Fault_T.

2856 **FEAT_ETS and TLBI** The second clause of obETS captures the second architected behaviour of FEAT_ETS
2857 (§**TODO: TLBI ordering needs ETS explained**), that faults after a thread-local TLBIs do not need context
2858 synchronisation to be ordered after the TLBI. Note that one still needs a DSB to complete the TLBI in that case.

2859 ### 9.4.7 obtlbi

```
(* ordered-before TLBI *)
let obtlbi =
    obtlbi_translate
  | [R|W|Fault_T]; iio^-1; (obtlbi_translate & ext); [TLBI]

(* translate ordered-before TLBI *)
let obtlbi_translate =
  [T & Stage1] ; tlb_barriered ; [TLBI-S1]
| (([T & Stage2] ; tlb_barriered ; [TLBI-S2]) ; wco? ; [TLBI-S1])
  & (same-translation ; [T & Stage1] ; maybe_TLB_cached)
| ([T & Stage2] ; tlb_barriered ; [TLBI-S2])
    & (same-translation ; [T & Stage1] ; trf^-1 ; wco^-1)
```

2860

2861 Finally, there is the obtlbi relation which captures the ordering from translations (and their explicit memory
2862 events) and the TLB invalidations which affect them. The relation is split in two: the obtlbi_translate clause
2863 enforces order between stale translations and the TLBIs they are invalidated by, the second clause covers broadcast
2864 TLBIs.

2865 **Capturing stale TLB entries** When a translation read happens, it is allowed for it to read from a stale write
2866 (§8.5.1). That is, the translation need not be ordered before writes which come after the write it actually reads
2867 from. Consequently the tfre relation is not included in ob.

2868 We strengthen this, by including some edges from translations to TLBIs, when there is an interposing newer write.
2869 The general shape of this ordering is illustrated in Figure 9.3.

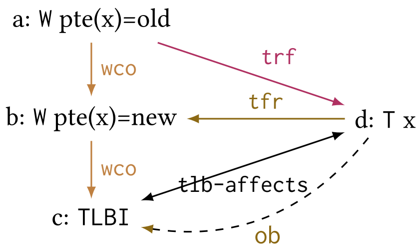


Figure 9.3: General `obtlbi_translate` shape.

2870 This shape is succinctly captured by the `tlb_barriered` auxiliary relation, which relates any translate-read that
2871 reads from a write which is `wco` before another write which is `wco` before a `TLBI` which targets the address, ASID
2872 or VMID of the translation:

```
let tlb_barriered =
  ([T] ; tfr ; wco ; [TLBI]) & tlb-affects^-1
```

2873

2874 We cannot simply include `tlb_barriered` in `ob`, however. Instead, we must consider the orderings for stage 1
2875 and stage 2 translation reads separately.

2876 **Stale stage 1 reads** For stage 1 translation reads, either in single-stage regimes or as part of a two-stage
2877 regime, we can include a variant of `tlb_barried` specialised to stage 1 translation-reads and `TLBI`s which affect
2878 stage 1 entries.

2879 **Stale stage 2 reads** Stage 2 walks are more subtle. The requirement to perform stage 1 invalidation (§8.6.4)
2880 means that, in those instances, we do not get `tlb_barriered` directly.

2881 Instead, we have to case split on the execution: either, (1) the translation table walk does a stage 1 translation
2882 read which reads-from an older write, in which case there may have been a whole cached translation that must
2883 be invalidated; or, (2) one of the stage 1 translation reads of the translation table walk reads from a write that is
2884 newer than the stage 2 `TLBI` and so there cannot have been any cached whole translation entries in the TLB and
2885 so, logically, we only need the stage 2 invalidation. These cases are illustrated in Figure 9.4, and correspond to the
2886 two clauses of `obtlbi_translate` which match on stage 2 translation reads.

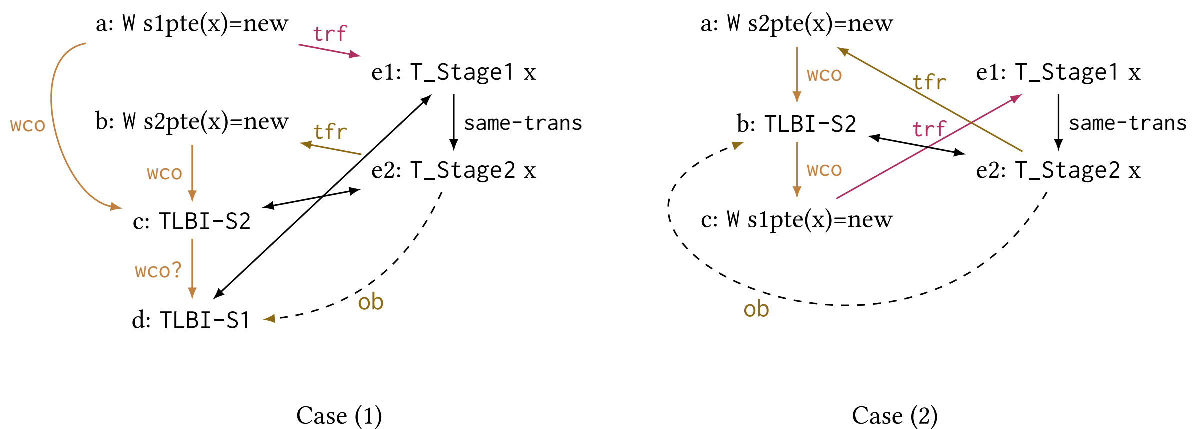


Figure 9.4: `obtlbi` stage 2 scenarios.

2887 We capture the general shape of (1), where a translation-read may have been cached in the TLB, with the following
2888 `maybe_TLB_cached` relation:

```
let maybe_TLB_cached = ([T]; trf^-1; wco; [TLBI]) & tlb-affects^-1
```

2889

2890 We then use this relation to add ordering from a stage 2 translation-read to the stage 1 `TLBI`, wco-after a stage 2
2891 `TLBI` that removed any stale IPA mappings, which would remove any cached whole-translation any stage 1
2892 translation-read might have read from, and after which any fresh translation table walk would be required to not
2893 see the stale stage 2 entry the translation-read read from.

2894 **Broadcast `TLBI`s** Recall that broadcast `TLBI`s impose restrictions on other threads (§8.6.3). When a broadcast
2895 `TLBI`'s invalidation affects a translation on another core, then it must also affect the explicit memory effect
2896 associated with it. This shape is illustrated in Figure 9.5, and corresponds to the final clause of `obtlbi`.

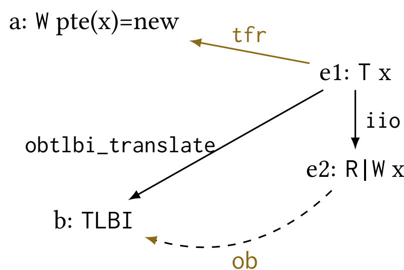


Figure 9.5: `obtlbi` broadcast `TLBI` shape.

2897 **Connecting TLB invalidations to translation reads** The final part of the puzzle is how to relate `TLBI` events
2898 with translations which may be affected by the invalidation. Recall that the `TLBI`s are grouped into subsets of
2899 `TLBI-S1`, `TLBI-VA`, and so on. We define a `tlb_might_affect` that is the cross-product of these with the `same-*`
2900 relations:

```
let tlb_might_affect =
  [ TLBI-S1 & ~TLBI-S2 &  TLBI-VA  &  TLBI-ASID & TLBI-VMID] ; (same-va-
    page & same-asid & same-vmid) ; [T & Stage1]
| [ TLBI-S1 & ~TLBI-S2 & ~TLBI-VA  &  TLBI-ASID & TLBI-VMID] ; (same-
    asid & same-vmid) ; [T & Stage1]
| [ TLBI-S1 & ~TLBI-S2 & ~TLBI-VA  & ~TLBI-ASID & TLBI-VMID] ; same-vmid
     ; [T & Stage1]
| [~TLBI-S1 &  TLBI-S2 &  TLBI-IPA & ~TLBI-ASID & TLBI-VMID] ; (same-ipa
    -page & same-vmid) ; [T & Stage2]
| [~TLBI-S1 &  TLBI-S2 & ~TLBI-IPA & ~TLBI-ASID & TLBI-VMID] ; same-vmid
     ; [T & Stage2]
| [ TLBI-S1 &  TLBI-S2 & ~TLBI-IPA & ~TLBI-ASID & TLBI-VMID] ; same-vmid
     ; [T]
| ( TLBI-S1 &            ~TLBI-IPA & ~TLBI-ASID & ~TLBI-VMID) * (T &
    Stage1)
| (            TLBI-S2 & ~TLBI-IPA & ~TLBI-ASID & ~TLBI-VMID) * (T &
    Stage2)
```

2901

2902 Finally, we get `tlb-affects` by attaching `tlb_might_affect` to events in the same thread, and if a `TLBI-IS`, to
2903 ones in other threads too:

```
let tlb-affects =
  [TLBI-IS]; tlb_might_affect
| ([~TLBI-IS]; tlb_might_affect) & int
```

2904

2905 Chapter 10

2906
Validating the VMSA model

2907
10.1 Extending isla-axiomatic

2908
10.2 Running on hardware: system-litmus-harness

2909
10.2.1 Harness overview

2910
10.2.2 Results from hardware

Chapter 11

An operational VMSA model

11.1 Introduction

In the previous chapters I give a formalisation of the Arm virtual memory systems architecture (VMSA) as an extension to Arm's own official memory consistency model.

Here I describe an informal sketch of a microarchitectural-style structural operational semantics, as an extension to Flat; let us call this extension PFlat (for Flat over *Physical* memory).

11.2 Structure of the state

PFlat's state is much like the state of Flat, except with two additions:

- ▷ A per-thread MMU.
- ▷ A per-MMU TLB.

11.2.1 The MMU

In flat each thread executes the intra-instruction semantics as defined by each instruction's sail code sequentially.

We alter this slightly to remove the sequential execution of the address translation function, and instead add a dedicated MMU.

When the thread wishes to perform a translation of some given virtual address, it sends a *translation request* to the MMU which may return a *translation result* (or simply a *translation*) in response.

The MMU is a set of in-progress calls to the translation function plus a TLB.

At any point in time, the MMU can spontaneously begin

11.2.2 Its TLB

11.3 Virtual memory axioms

Describe each new ‘ob‘ edge in detail, and the new axioms.

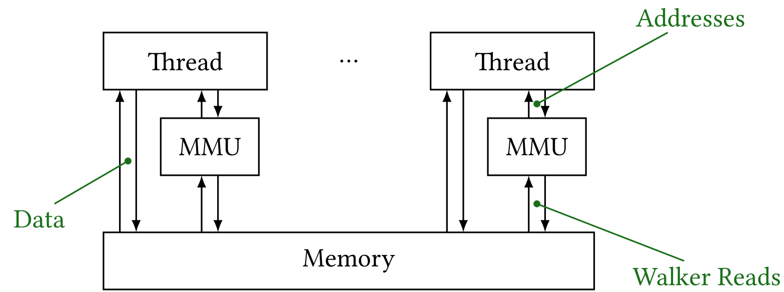


Figure 11.1: Operational VM machine.

2933
11.4 Break-before-make violation detection

2934
11.5 A weaker VMSA model

2935
11.6 Executing the models with Isla

2936 Just briefly describe (post-cav) extensions to isla and optimisations for virtual memory.

Chapter 12

Limitations

12.1 Exceptions and Interrupts

We care about this part of the architecture, we give an under/overapproximate model earlier but it's not correct. Describe Ohad work and a brief overview of the issue and some tests perhaps.

12.2 Cross-interaction of instruction fetch and virtual memory

Ifetches need translation, why can't we just copy/paste our vmsa model for ifetch talk about non-pipt icaches and aliasing, talk about "po".

12.3 Other architectures

Here we discussed just Armv8, but we could do the same for x86, Power or RISC-V.

Describe possible extensions and differences.

2948 Chapter 13

Conclusion

2950 Recap of contributions, limitations and remaining open questions.

2951 # Glossary

Bibliography

[1] Arm Limited. Arm architecture reference manual. Armv8, for Armv8-A architecture profile. https://developer.arm.com/documentation/ddi0487/ha/?lang=en, February 2022. H.a Armv9 EAC. ARM DDI 0487H.a (ID020222). 11530pp.

[2] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual combined volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d and 4. https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4, accessed 2019-06-30, May 2019. 325462-070US.

[3] ARM Limited. ARM architecture reference manual. ARMv8, for ARMv8-A architecture profile, October 2018. v8.4. ARM DDI 0487D.a (ID103018).

[4] Alastair Reid. Trustworthy specifications of ARM v8-A and v8-M system level architecture. In *FMCAD 2016*, pages 161–168, October 2016.

[5] Alastair Reid. ARM releases machine readable architecture specification. https://alastairreid.github.io/ARM-v8a-xml-release/, April 2017.

[6] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of processors with ISA-Formal. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 42–58. Springer, 2016.

[7] Alasdair Armstrong, 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, and Peter Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2019. Proc. ACM Program. Lang. 3, POPL, Article 71.

[8] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Shaked Flur Jon French Kathryn E. Gray, Gabriel Kerneis, Neel Krishnaswami, Prashanth Mundkur, Robert Norton-Wright, Christopher Pulte, Alastair Reid, Peter Sewell, Ian Stark, and Mark Wassell. Sail. https://www.cl.cam.ac.uk/~pes20/sail/, 2019.

[9] Christopher Pulte. *The Semantics of Multicopy Atomic ARMv8 and RISC-V*. PhD thesis, University of Cambridge, 2019. https://doi.org/10.17863/CAM.39379.

[10] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2018.

[11] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. Mixed-size concurrency: ARM, POWER, C/C++11, and SC. In *The 44st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France*, pages 429–442, January 2017.

[12] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *Proceedings of POPL: the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[13] Kathryn E. Gray, Gabriel Kerneis, Dominic Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER

2993 multiprocessors. In *Proc. MICRO-48, the 48th Annual IEEE/ACM International Symposium on Microarchitecture*,
2994 December 2015.

2995 [14] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding Cats: Modelling, Simulation, Testing, and
2996 Data Mining for Weak Memory. *ACM TOPLAS*, 36(2):7:1–7:74, July 2014.

2997 [15] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the ARM and POWER relaxed
2998 memory models. Draft available from http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.
2999 pdf, 2012.

3000 [16] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and
3001 Derek Williams. Synchronising C/C++ and POWER. In *Proceedings of PLDI 2012, the 33rd ACM SIGPLAN*
3002 *conference on Programming Language Design and Implementation (Beijing)*, pages 311–322, 2012.

3003 [17] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: running tests against hardware. In
3004 *Proceedings of TACAS 2011: the 17th international conference on Tools and Algorithms for the Construction and*
3005 *Analysis of Systems*, pages 41–44, Berlin, Heidelberg, 2011. Springer-Verlag.

3006 [18] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER
3007 multiprocessors. In *Proceedings of PLDI 2011: the 32nd ACM SIGPLAN conference on Programming Language*
3008 *Design and Implementation*, pages 175–186, 2011.

3009 [19] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A
3010 rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97,
3011 July 2010. (Research Highlights).

3012 [20] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *Proceedings of*
3013 *TPHOLs 2009: Theorem Proving in Higher Order Logics, LNCS 5674*, pages 391–407, 2009.

3014 [21] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In *Proc. CAV*,
3015 2010.

3016 [22] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco
3017 Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In *Proc. DAMP 2009*, January
3018 2009.

3019 [23] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus
3020 Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *Proceedings of POPL*
3021 *2009: the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages
3022 379–391, January 2009.

3023 [24] Nathan Chong and Samin Ishtiaq. Reasoning about the ARM weakly consistent memory model. In *MSPC*,
3024 2008.

3025 [25] Allon Adir, Hagit Attiya, and Gil Shurek. Information-flow models for shared memory with an application
3026 to the PowerPC architecture. *IEEE Trans. Parallel Distrib. Syst.*, 14(5):502–515, 2003.

3027 [26] Will Deacon. The ARMv8 application level memory model. https://github.com/herd/herdtools7/
3028 blob/master/herd/libdir/aarch64.cat (accessed 2019-07-01), 2016.

3029 [27] Andrew Waterman and Krste Asanović, editors. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*.
3030 December 2018. Document Version 20181221-Public-Review-draft. Contributors: Arvind, Krste Asanović,
3031 Rimas Avižienis, Jacob Bachmeyer, Christopher F. Batten, Allen J. Baum, Alex Bradbury, Scott Beamer,
3032 Preston Briggs, Christopher Celio, Chuanhua Chang, David Chisnall, Paul Clayton, Palmer Dabbelt, Roger
3033 Espasa, Shaked Flur, Stefan Freudenberger, Jan Gray, Michael Hamburg, John Hauser, David Horner, Bruce
3034 Hoult, Alexandre Joannou, Olof Johansson, Ben Keller, Yunsup Lee, Paul Loewenstein, Daniel Lustig, Yatin
3035 Manerkar, Luc Maranget, Margaret Martonosi, Joseph Myers, Vijayanand Nagarajan, Rishiyur Nikhil, Jonas
3036 Oberhauser, Stefan O'Rear, Albert Ou, John Ousterhout, David Patterson, Christopher Pulte, Jose Renau,
3037 Colin Schmidt, Peter Sewell, Susmit Sarkar, Michael Taylor, Wesley Terpstra, Matt Thomas, Tommy Thorn,
3038 Caroline Trippel, Ray VanDeWalker, Muralidaran Vijayaraghavan, Megan Wachs, Andrew Waterman, Robert
3039 Watson, Derek Williams, Andrew Wright, Reinoud Zandijk, and Sizhuo Zhang.

3040 [28] Shaked Flur, Jon French, Kathryn Gray, Christopher Pulte, Susmit Sarkar, and Peter Sewell. rmem. www.cl.
3041 cam.ac.uk/~pes20/rmem/, 2017.

3042 [29] Jade Alglave and Luc Maranget. The herd7 tool. http://diy.inria.fr/doc/herd.html/, 2019. Accessed
3043 2019-07-08.

3044 [30] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot
3045 Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*,
3046 32(1):2:1–2:70, February 2014.

3047 [31] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao
3048 Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. In *Proceedings*
3049 *of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018,*
3050 *Philadelphia, PA, USA, June 18-22, 2018*, pages 646–661, 2018.

3051 [32] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David
3052 Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX*
3053 *Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4,*
3054 *2016.*, pages 653–669, 2016.

3055 [33] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification
3056 to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating*
3057 *Systems Principles, Shanghai, China, October 28-31, 2017*, pages 287–305, 2017.

3058 [34] Roberto Guanciale, Hamed Nemati, Mads Dam, and Christoph Baumann. Provably secure memory isolation
3059 for linux on ARM. *J. Comput. Secur.*, 24(6):793–837, 2016.

3060 [35] Christoph Baumann, Oliver Schwarz, and Mads Dam. Compositional verification of security properties for
3061 embedded execution platforms. In *PROOFS@CHES 2017, 6th International Workshop on Security Proofs for*
3062 *Embedded Systems, Taipei, Taiwan, Friday September 29th, 2017*, pages 1–16, 2017.

3063 [36] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish.
3064 The verified CakeML compiler backend. *J. Funct. Program.*, 29:e2, 2019.

3065 [37] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation
3066 of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*
3067 *'14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192, 2014.

3068 [38] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.

3069 [39] Jade Alglave, Luc Maranget, Kate Deplaix, Keryan Didier, and Susmit Sarkar. The litmus7 tool. http:
3070 //diy.inria.fr/doc/litmus.html/, 2019. Accessed 2019-07-08.

3071 [40] Jade Alglave and Luc Maranget. The diy7 tool. http://diy.inria.fr/, 2019. Accessed 2021-07-01.

3072 [41] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering
3073 of real-world semantics. In *Proceedings of ICFP 2014: the 19th ACM SIGPLAN International Conference on*
3074 *Functional Programming*, pages 175–188, 2014.

3075 [42] Azalea Raad, John Wickerson, and Viktor Vafeiadis. Weak persistency semantics from the ground up:
3076 Formalising the persistency semantics of ARMv8 and transactional models. *Proc. ACM Program. Lang.*,
3077 3(OOPSLA):135:1–135:27, October 2019.

3078 [43] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the Intel-x86
3079 architecture. *PACMPL*, 4(POPL):11:1–11:31, 2020.

3080 [44] Magnus O. Myreen. Verified just-in-time compiler on x86. In *Proceedings of the 37th Annual ACM SIGPLAN-*
3081 *SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 107–118, New York, NY, USA,
3082 2010. ACM.

3083 [45] Shilpi Goel, Warren A. Hunt, Matt Kaufmann, and Soumava Ghosh. Simulation and formal verification of
3084 x86 machine-code programs that make system calls. In *Proceedings of the 14th Conference on Formal Methods*
3085 *in Computer-Aided Design*, FMCAD '14, pages 18:91–18:98, Austin, TX, 2014. FMCAD Inc.

3086 [46] Shilpi Goel. The x86isa books: Features, usage, and future plans. In *Proceedings 14th International Workshop*
3087 *on the ACL2 Theorem Prover and its Applications, Austin, Texas, USA, May 22-23, 2017.*, pages 1–17, 2017. arXiv
3088 version: https://arxiv.org/abs/1705.01225.

3089 [47] Rishiyur S. Nikhil and Niraj Nayan Sharma. Forvis: A formal RISC-V ISA specification. https://github.
3090 com/rsnikhil/Forvis_RISCV-ISA-Spec, 2019. Accessed 2019-07-01.

3091 [48] Ian J Clester, Thomas Bourgeat, Andy Wright, Samuel Gruetter, and Adam Chlipala. riscv-plv risc-v isa
3092 formal specification. https://github.com/mit-plv/riscv-semantics, 2019. Accessed 2019-07-01.

3093 [49] Hira Syeda and Gerwin Klein. Reasoning about translation lookaside buffers. In *LPAR-21, 21st International*
3094 *Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*,
3095 pages 490–508, 2017.

3096 [50] Hira Taqdees Syeda and Gerwin Klein. Program verification in the presence of cached address translation.
3097 In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic*
3098 *Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, pages 542–559, 2018.

3099 [51] Bogdan F. Romanescu, Alvin R. Lebeck, and Daniel J. Sorin. Specifying and dynamically verifying address
3100 translation-aware memory consistency. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural*
3101 *Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 323–334, New York, NY, USA,
3102 2010. ACM.

3103 [52] Bogdan Romanescu, Alvin Lebeck, and Daniel J. Sorin. Address translation aware memory consistency. *IEEE*
3104 *Micro*, 31(1):109–118, January 2011.

3105 [53] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. COATCheck: Verifying memory
3106 ordering at the hardware-OS interface. *SIGOPS Oper. Syst. Rev.*, 50(2):233–247, March 2016.

3107 [54] Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter
3108 Sewell. Relaxed virtual memory in Armv8-A. In *Proceedings of the 31st European Symposium on Programming,*
3109 *ESOP 2022*, April 2022.

3110 [55] Arm Limited. Exploration Tools – Arm Developer. https://developer.arm.com/downloads/-/
3111 exploration-tools, 2022. Accessed May 2022.

3112 [56] UK. Copyright, designs and patents act. c. 48, 1988.

3113 [57] Arm Limited. Arm Cortex-A53 MPCore Processor Technical Reference Manual, 2022. ARM DDI 0500J.

3114 [58] Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and
3115 Peter Sewell. ARMv8-A system semantics: instruction fetch in relaxed architectures. In *Proceedings of the*
3116 *29th European Symposium on Programming, ESOP 2020*, 2020.