Arm systems semantics

Ben Simner

University of Cambridge

February 1, 2024

Abstract

Computing relies on architecture specifications to decouple hardware and software development. Historically these have been prose documents, with all the problems that entails, but research over the last ten years has developed rigorous and executable-as-test-oracle specifications of mainstream architecture instruction sets and "user-mode" concurrency, clarifying architectures and bringing them into the scope of programming-language semantics and verification.

However, the *system semantics*, of address translation and TLB maintenance, instruction-fetch and its required cache maintenance, remains mostly obscure, leaving us without a solid foundation for verification of security-critical systems software.

We produce precise mathematical models, for those aspects of the Arm A-class architecture. We implement these models as executable models, in both microarchiectural-flavoured operational and declarative axiomatic style formats. We validate these models, against currently available hardware through the production and evaluation of hardware test harnesses and test suites, and against the architectural intent through discussions with Arm architects. We produce a variety of hand-written and machine-generated litmus tests, exercising parts of the architecture previously unexplored.

We discuss the nature of producing such models, the challenges that writing specifications of existing systems entails, and briefly touch upon how these models have evolved over time, and how we imagine they will evolve in the future as the remaining questions are resolved.

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 66501 total words as counted by `detex | wc -w`.

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

Contents

CONTENTS

Introduction

The computers we use every day are complex machines, made of many components, all working together to execute the software we run on them. These machines act as interpreters for a custom binary programming language, with commands made up of the instructions of the underlying *architecture*. These architectures can be thought of as abstractions of the underlying hardware, as programming languages whose syntax is defined by the binary encoding of the instructions from the ISA (*Instruction Set Architecture*), and whose semantics is the composition of the sequential behaviours of the individual instructions and registers from the ISA, with the whole machine execution model.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.

Over the years much work has gone into defining, mathematically and precisely, the architectures that the processors we use every day implement. This previous work covers Intel/AMD's x86 [1, 2, 3, 4], Arm's ARMv7-A [5] and Armv8-A [6, 7] architectures, IBM's Power [8], RISC-V [9], and others. In theory, this interface is straightforward to define. One can give precise formal semantics to the individual instructions, as Arm does with its *Architecture Specification Language* (or ASL for short) [10, 11], and then tie instructions together in 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 [12], and 4922 [1] pages respectively, covering everything from the individual instructions to the interactions between those instructions and the way they interact with memory.

The complexity of these interfaces becomes most apparent with the interaction with *multiprocessor* systems [13]. 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*, affecting the semantics of the machine code, that is the values capable of being read or written to registers or memory by those processors. 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 precise mathematical semantics to the concurrent behaviours of 'userland' machine code programs [14, 15, 3, 4, 16, 7, 17]. Analogously for high-level languages, there is similar work in understanding their relaxed memory behaviours which arise from both their compilation to such low-level machine programs, and also from the compiler's optimisations [18, 19, 20, 16].

We now seek to expand this work on relaxed memory for the Arm architecture, 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 power a large proportion of modern mobile devices. There are a few reasons to focus on Arm: (1) they are ubiquitous and millions (perhaps even billions, with over a trillion devices running Arm hardware today) of people rely on software running on Arm hardware 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 in their ASL language, enabling us to give a faithful specification to the architectural envelope.

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

1.1 Arm-A architecture overview

Arm's A-class architecture is general-purpose, intended for mobile devices, tablets, laptops, and even servers. Arm has three A-class architectures which can currently be found in modern hardware: ARMv7-A, Armv8-A, and Armv9-A. ARMv7-A is 32-bit only. Armv8-A and Armv9-A have 32-bit and 64-bit execution modes. Armv8-A and Armv9-A's 64-bit modes use the same base ISA and execution modes, except where Armv9 has some additional features, or required extensions, or bugfixes. We will focus here on the 64-bit architecture found in Armv8-A and Armv9-A, and will use the term *Arm-A* to refer to both Armv8-A and Armv9-A interchangeably.

Execution of an Arm-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 use either the T32 or A32 instruction sets. This is illustrated in Figure 1.1.
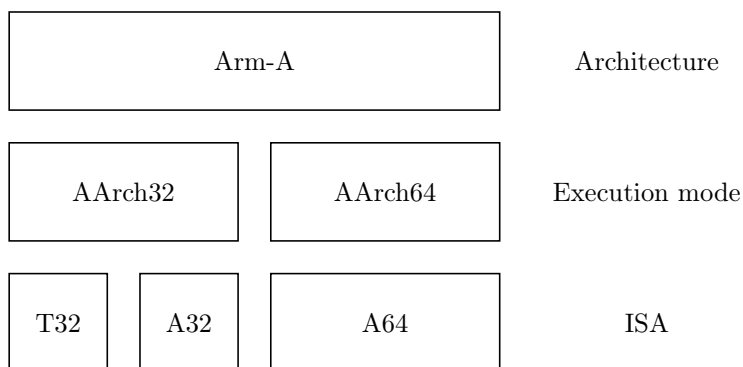
Figure 1.1: Arm-A structure.

A64, currently, has 402 'base' instructions and another 1,205 vector, matrix and floating-point instructions. It has 31 general-purpose registers, accessible through either 32-bit views as w0-w30, or as 64-bit views as x0-x30, as shown in Figure 1.2. It has a dedicated zero register (wzr/xzr), and stack pointer register (sp). Instructions are fixed-width, with 32-bit opcodes, and in the typical RISC style: with most instructions reading operands from registers, and writing results back to registers, with only limited support for immediate values. Execution in AArch64 is split into 4 'exception levels', these demark the levels of privilege that a process may have, ranging from EL0 (least privileged) to EL3 (most privileged). Typically userland 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. There are also secure modes, which we do not consider here.

Each CPU, called PEs (processing elements) in Arm nomenclature, has: its own bank of registers; is executing in either AArch64 or AArch32 execution mode; is fetching, decoding and executing instructions from either the A64, A32 or T32 ISAs; is executing at at one of EL0, EL1, EL2 or EL3.
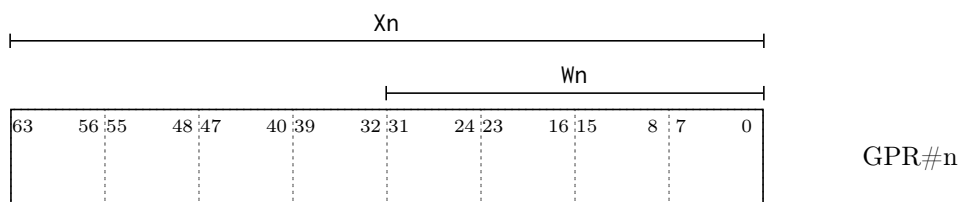
Figure 1.2: Arm-A W and X register views for a general-purpose register.
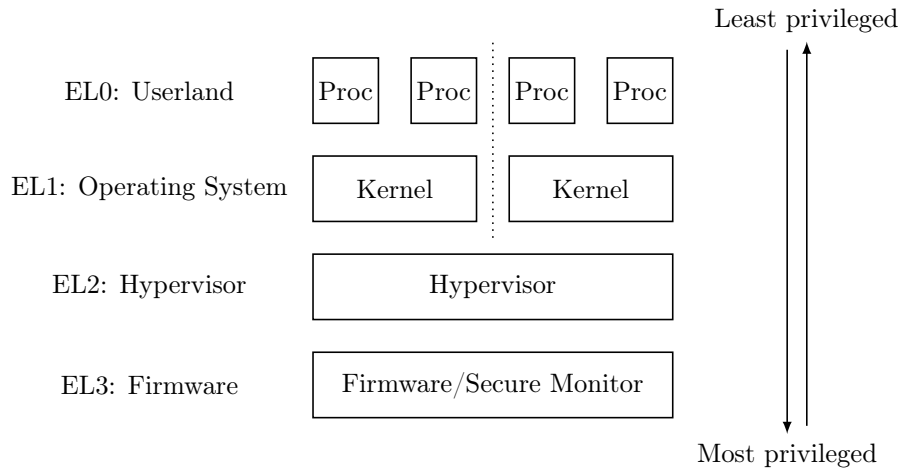
Figure 1.3: Arm-A exception levels.

1.2 Systems software

The programs we interact with on a day-to-day basis on our computers, our word processors and internet browsers, are typically unprivileged programs, with restricted access to hardware. Such programs are often referred to as executing in *userland*. 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 (in Arm, this means at EL0, as in Figure 1.3), and with the operating systems and hypervisors below them restricting the access to memory they have through the use of *virtual memory* (see Chapter 7).

Operating systems typically split userland execution into *processes*: discrete instances of programs, each with some associated dedicated (virtual) memory [21, p. 85]. It is then the operating system, executing with more privilege (at EL1), that configures and schedules these processes.

Modern operating systems seek to enforce isolation between these processes primarily through the application of a virtual memory abstraction [21, pp. 185,194,604][22, p 227], with each process behaving as if it has direct access to memory, when in fact the operating system (and the hardware supporting it) are redirecting the accesses at runtime.

This virtual memory abstraction can be layered, with an extra level of abstraction below the operating systems controlled by a *hypervisor*. Hypervisors behave similarly, but instead of controlling many processes at EL0 they instead can control multiple operating systems at EL1.

Finally, at EL3 executes any firmware or secure monitor. Generally, the firmware performs hardware-specific actions, especially during boot (reading and writing implementation-defined configuration registers and performing any functionality required by the System-on-Chip). The Secure Monitor is a part of the Arm architecture's TrustZone security extensions, and we will not discuss these features here.

Figure 1.3 demonstrates a typical setup, with firmware running at EL3, a hypervisor at EL2, which is controlling a couple of operating systems, each of which has multiple processes under its control.

1.3 Relaxed memory

The implementations of many programming languages, the compilers and interpreters, either in software and hardware, are not just direct implementations of the simple in-order sequential semantics one might expect. Instead, as time progressed these implementations have acquired multiple layers of abstraction, made with increasing complexity. Compilers and hardware re-write programs to be faster, use less space, and be more compact. They propagate and duplicate reads, subsume or outright eliminate writes, reorder operations in the program, replace one computation with another, or even just remove entire sections of the program entirely.

These optimisations may be *semantics preserving* with respect to the simple sequential semantics: that they, aside from the timing effects they are designed to cause, are invisible to the programmer. This is, however, not true in all cases, with many highly desirable optimisations not preserving the source program's semantics [23].

It is multithreaded programs, and multicore processors, which often breaks the assumptions made by these optimisations. As an example, take Intel's x86 microprocessor architecture. It allows its implementations to perform an innocuous-sounding optimisation: to buffer writes together locally. This *store buffering* optimisation is ubiquitous in the hardware world, but, it permits multiple cores to have mutually inconsistent views of memory [23, 3, 4]; where, at the same point in time, different cores see different values for the same memory address. If the programmer was unaware of these behaviours and the required mitigation in software, then this could break key invariants of software, leading to critical bugs in synchronisation primitives [23], data structures, or software more generally [24].

Intel, and their x86 architecture, is not the only example of hardware architectures performing such optimisations, and store buffering is not the only behaviour hardware exhibits. Arm [12], RISC-V [25], and IBM's Power [26] architectures all exhibit their own behaviours, with consequential requirements on software. Each of these microprocessor architectures comes with its own reference manual, comprised of thousands, or tens of thousands, of pages with a mix of prose and pseudocode, attempting to describe these behaviours. These architectures are incomparable, the behaviours they allow are not subsets of one another. Instead, there are several optimisations that some architectures allow as observable behaviour, where others do not. These include things such as reordering of instructions, prefetching and caching of data and instructions, buffering of loads and stores, hierarchical cache layouts, and branch prediction with speculation down those branches. It is not that some implementations perform these optimisations while others do not, but that those architectures which allow such behaviours to be observed do not require that the hardware include relevant hazard checking or invalidations which would recover from 'bad' states.

It is not just hardware that has these concerns. A variety of software languages, including C and C++ [27, 28], Java [29, §17.4], Rust [30], and Haskell [31], 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.

Over the decades, the community has spent a large amount of effort in understanding the behaviours that the hardware use every day actually exhibit, in talking with architects and hardware designers about what they imagine hardware could do, now or in the future, and building precise mathematical models which capture the architectural 'envelope' of allowable behaviours. These models come in many flavours, and in Chapter 2 we will explore two such models for Arm, and the set of behaviours they are intended to capture.

1.4 Contributions

In this thesis, we extend the previous relaxed memory work on Arm into the realm of systems software: instruction fetch and cache maintenance, pagetables and TLB maintenance, and a start on exception handling. We will produce both axiomatic-style declarative semantics and microarchitectural-style operational semantics to cover a variety of those parts of the architecture.

1.4.1 Artefacts

This work will present:

▷ A set of litmus tests for instruction fetching and cache maintenance (Ch. 3), covering many areas and features and clarifying the architectural intent in those areas.

▷ A microarchitectural-style structural-operational-semantics for Arm-A (Ch. 4), covering ifetch and cache maintenance, as an extension to the existing Flat model.

▷ An equivalent formulation as an axiomatic-style declarative semantics (Ch. 5), as an extension to the herd-style Armv8 axiomatic model.

▷ An extension of the litmus7 tool, and a set of results from testing against a range of hardware (Ch. 6).

▷ A set of litmus tests for virtual memory and TLB maintenance, using the whole Arm translation table walk with both stages (Ch. 8).

▷ An axiomatic-style declarative semantics (Ch. 9) as an extension to the original Armv8 model.

▷ A new hardware testing harness, and validation of the models by experimentation against hardware, and through abstraction proofs (Ch. 10).

1.4.2 Publications and collaborations

The work presented in Chapters 3 to 10 were done in collaboration with a variety of other people on different aspects, and resulted in the production of the following publications:

▷ "**ARMv8-A system semantics: instruction fetch in relaxed architectures**", in the Proceedings of the 29th European Symposium on Programming (ESOP 2020), by **Ben Simner**, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell [32].

▷ "**Isla: Integrating full-scale ISA semantics, axiomatic concurrency models**", in the Proceedings of the 33rd International Conference on Computer Aided Verification (CAV 2021), by Alasdair Armstrong, Brian Campbell, **Ben Simner**, Christopher Pulte, and Peter Sewell [33].

▷ "**Relaxed virtual memory in Armv8-A**", in the Proceedings of the 31st European Symposium on Programming (ESOP 2022), by **Ben Simner**, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell [34].

▷ "**Relaxed exceptions in Arm-A (pre-publication)**", in the unpublished work, by **Ben Simner**, Ohad Kammar, Jean Pichon-Pharabod, and Peter Sewell [35].

▷ "**Isla: Integrating full-scale ISA semantics, axiomatic concurrency models (extended version)**", in the Formal Methods in System Design (May, 2023), by Alasdair Armstrong, Brian Campbell, **Ben Simner**, Christopher Pulte, and Peter Sewell [36].

Many of the aspects of the work presented in this thesis were done jointly with many of the people listed above. In particular: the Isla tooling was primarily written by Alasdair Armstrong; the work on the litmus and diy tools by Luc Maranget; production of litmus tests and discussions with architects and microarchitects was done jointly with Shaked Flur, Christopher Pulte, Ohad Kammar, Thibaut Pérami, Jean-Pichon Pharabod, and Peter Sewell; the writing of models was done with Christopher Pulte and Shaked Flur (for ifetch), Christopher Pulte and Thibaut Pérami (for VMSA), and Jean Pichon-Pharabod and Ohad Kammar (for exceptions); and, the validation of the models, through proof and hardware testing, was done jointly with Jean Pichon-Pharabod (on the VMSA abstraction proofs) and Luc Maranget (test generation and hardware testing for ifetch).

Much of the above work was done in collaboration with Arm and their staff, in particular their chief architect, Richard Grisenthwaite. He is our primary contact within Arm, and we have a close collaboration with him characterised by discussions on Arm hardware, the requirements of the software that runs on them, the consequences of the models we propose, and, where relevant, the history of the architecture. In cases where we present some behaviour and declare that it is 'allowed by Arm', it usually means we have confirmation from the chief architect directly. However, it is not just the chief architect we collaborate with, but many members of Arm's staff: Will Deacon, and later Jade Alglave, as the primary maintainer of the Arm memory models; and Ian Caulfield, Nikos Nikoleris, Gustavo Petri, Anthony Fox, and others, who discussed Arm modelling efforts, Arm hardware implementations, and provided feedback individually on many of the aforementioned publications.

1.5 Thesis overview

This document is split into four main parts:

- ▷ Introduction and background (Chapters 1 and 2)
- ▷ Instruction fetch (Chapters 3-6)
- ▷ Virtual memory (Chapters 7-10)
- ▷ Limitations and Conclusion (Chapter 11)

Background   Chapter 2 covers the fundamental concepts behind relaxed memory: the idea of litmus testing as a means to clarify and understand architecture, including a selection of important and useful litmus tests from the literature; how Arm defines their intra-instruction semantics and how such semantics compose with a concurrency model; the two kinds of concurrency models we will explore in this thesis, microarchitectural-style operational semantics and axiomatic-style declarative semantics; and describe instantiations of these for Arm-A.

Instruction fetching   We start with a brief overview of the existing prose text for instruction fetch (*ifetch*) and the related instruction (and data) cache maintenance operations. Focusing primarily on self-modifying (and concurrent-modification) of code, such as what is required for JITs, dynamic loaders, and operating systems schedulers, we produce a set of litmus tests (Ch. 3) to capture the key relaxed behaviours that arise from the optimisations found in modern microprocessors, and clarify where such behaviours were unclear. We produce a microarchitectural-style operational semantics (Ch. 4) based on our discussions with architects and microarchitects. We then produce an axiomatic model (Ch. 5) intended equivalent to the operational model. We then validate that these models (Ch. 6), confirming they coincide for the litmus tests given in the chapter. We automatically generate a large test suite of novel tests and check the two models do not diverge on these tests. We additionally check that they do not forbid behaviours exhibited on hardware by running the test suite on a selection of modern Arm processors.

Virtual memory   Structured similarly to the instruction-fetching chapters, but independent of the work presented in it, we begin with an overview of the Arm *Virtual Memory Systems Architecture* or *VMSA* (Ch. 7), which describes the structure and behaviour of the Arm address translation and memory management architecture without considering concurrency or caching effects, then we produce a set of litmus tests (Ch. 8) which clarifies the architectural intent. We produce an axiomatic-style model for relaxed virtual memory (Ch. 9), as an extension to the original (user mode) model, using the whole Arm translation table walk, including multiple stages, and TLB maintenance. Finally, there is a discussion on the validation of this model (Ch. 10) achieved by discussion with the Arm chief architect, along with some limited testing of current Arm hardware, and some proofs over the axiomatic model for some expected key abstraction results.

Conclusion   Finally, Chapter 11 presents a short recap of the presented work, its limitations, and relation to other work in the area. Additionally, there is a discussion on what we learned, both consequently of the resulting models, but also through the process itself, before finally touching on what remains as potential future work.

Modelling Arm: background

Now we turn our attention to the current well-established methods of precisely and formally modelling relaxed memory behaviours, specifically for Arm-A. In this chapter, we will cover two methods: microarchitectural-style operational semantics, which mimic the mechanisms seen on hardware; and, axiomatic-style declarative models which filter out whole-program execution graphs based on some predicate.

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 of the underlying architecture that the models should allow or forbid. 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 Relaxed behaviours and litmus testing

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 [23]. The code listing for the AArch64 (Arm-A) variant can be found in Figure 2.1. The 'MP' portion of the name captures the *shape*: the code pattern, or sequence of events, that acts as the skeleton for a family of related tests. 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, in this case, that both threads have accesses just program-order after each other with no other barriers or dependencies. Typically these variations are defined as the sequence of orderings between events (separated by – in the name) for each thread (separated by +). Thus, we get a whole family of litmus tests based on the basic MP shape: MP+pos (the one shown here), MP+dmbs (with an Arm dmb memory barrier on each thread), MP+dmb.st+addr (with an Arm dmb.st memory barrier on the writer thread and an address dependency on the reader thread), and so on.

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	**LDR** X0,[X1]
STR X0,[X1]	**LDR** X2,[X3]
MOV X2,#1	
STR X2,[X3]	
Allowed: 1:X0=1, 1:X2=0	

Figure 2.1: MP test code listing.

The code listing given is totally standard [37]: the top line contains the name of the litmus test (MP+pos), and the architecture that this variant is for (AArch64); the second section contains the initial register and memory state;

the next section contains the assembly code listing for each thread; and finally at the bottom is the interesting outcome we wish to explore, given as a constraint on the final register and memory state.

On Arm, the given outcome is allowed. On a sequentially-consistent (SC) machine, whose executions are an interleaving of the instructions of both threads [38], there are many such executions of the listed code, each giving rise to a (potentially distinct) final state. To see the highlighted outcome, where Thread 1 reads 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 combination is not consistent with any of the simple interleavings of the instructions a sequentially consistent machine would perform. We represent these executions not as an interleaving of the instructions, but as a graph of the events of those instructions (the reads and writes they perform) connected by their implicit orderings. There may be, and in this case, are, multiple different operational traces that lead to the same execution witness, which we shall explore later. The execution graph that corresponds with the allowed outcome can be found in Figure 2.2.


```
 ┌──────────┐        ┌──────────┐
 │ Thread 0 │        │ Thread 1 │
 └──────────┘        └──────────┘
   a:W x=1             c:R y=1
        │    rf      
fr      │po  ───────►      │po
        ▼                  ▼
   b:W y=1             d:R x=0
```
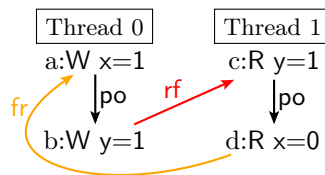
Figure 2.2: MP test execution diagram.

The nodes on the left, under the Thread 0 label, correspond to events from executing Thread 0 of the program, where the event labelled a:W x=1 corresponds to the propagation of the first store in Thread 0 (the write of 1 to x) to memory, and event b corresponds to the second store being propagated. They are related by program-order (po) which says that instruction the event a comes from is earlier than that of b's in the instruction stream of the processor; that is, a's instruction was earlier in the fetch-decode-execute cycle of the processor than b's was. Similarly, under Thread 1 we see the event labelled c:R y=1 for the first load reading from y, and seeing 1. This value was read from the write event b, therefore b is related to c (the read of y) by the reads-from (rf) relation. 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 d of x read a value from a write to x from before the event a happened, since it read an older value than it, in this case that is the initial memory from the 'Initial state' of the test, and so d is related to a by the from-reads (fr) relation.

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 either the reads or writes. The execution is therefore allowed, and we readily observe this outcome on hardware. In some cases, the execution may be *architecturally allowed*, that is, the final state constraint is permitted to occur in practice, but has not been experimentally observed on any hardware so far. In other cases, there may be no architecturally allowed execution that permits a particular outcome, but it is still observed on hardware: these are (or at least imply there exists) *hardware errata*, more commonly referred to as 'bugs'.

We use litmus tests to explore *behaviours*: particular patterns in code, or specific hardware mechanisms that are responsible for allowing or forbidding the test. Many litmus tests exercise many microarchitectural mechanisms whose composition or confluence leads to the final result, or where there may be multiple different mechanisms or choices that could each independently lead to the same result. For example, in the MP+pos test we just saw, there are three well-understood microarchitectural explanations: that the stores are committed out-of-order (re-ordered within the pipeline, store queue, or other thread-local storage), that the stores propagate out-of-order (are pushed out-of-order into the shared memory), or that the loads satisfy out-of-order (either requested out-of-order in the pipeline, or requests returned out-of-order from the memory subsystem). Any of the above explanations are alone sufficient to allow the relaxed outcome highlighted by the test. One needs to prevent out-of-order execution on both sides of the test (through the use of memory barriers, for example) to forbid that relaxed outcome.

Previous work has systematically 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 exhaustive review of all the behaviours that are allowed and forbidden in Arm, instead referring the reader to the existing literature [14, 37, 39, 16, 7, 6, 40]. However, we will briefly look at some of the behaviours that the reader should be familiar with in order to understand future chapters, namely coherence, barriers and dependencies, and multi-copy atomicity.

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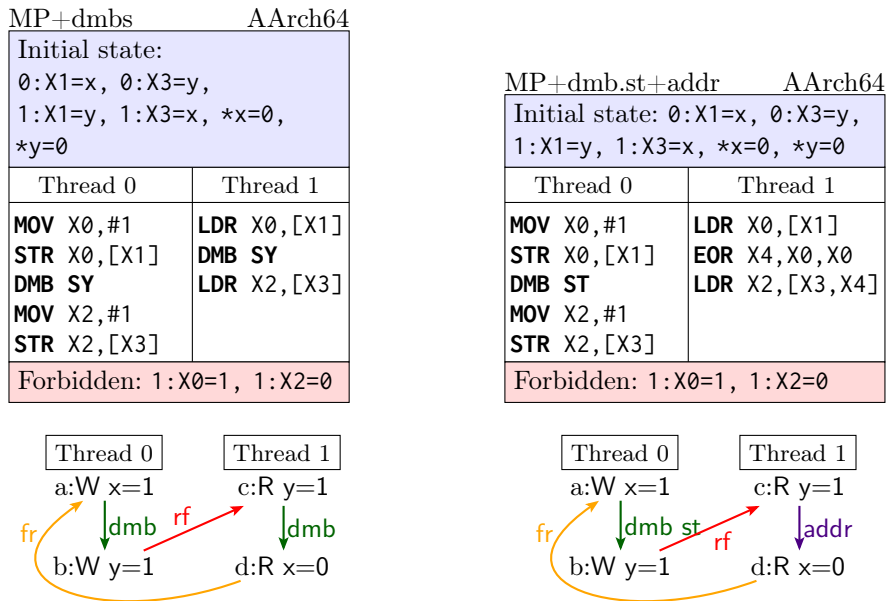
Figure 2.3: Two variants of MP with thread-local ordering.
On the left: MP+dmbs with Arm `DMB` barrier between instructions.
On the right: MP+dmb.st+addr with an address dependency between the reads.

2.1.1 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/AMD's x86, where only writes can be re-ordered with respect to program-order later reads (through store buffering) [1, 23, 3]. Note that here I do not mean that the *hardware* is not allowed to re-order the instructions, but that if it does so, it must preserve the illusion of in-order execution to the programmer.

Not all re-orderings are permissible; Arm requires that single-threaded programs should behave as if executed sequentially, at least for loads and stores. 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 must 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.

Arm have *syntactic dependencies*. These are the intrinsic relations which arise from the dataflow during execution of the program. Usually, they are categorised into three kinds: address dependencies (`addr`), from reads to memory events that use that read in the computation of the address the memory event accesses; data dependencies (`data`), from reads to writes, where the value read is used in the computation of the value written; and control dependencies (`ctrl`), from reads to events of instructions program-order after a (conditional) branch in the program where the value of the read was used in the computation of the value used in the condition. Note that these are not 'static' properties of the source program, instead, the 'syntactic' part reflects that Arm have no *fake* dependencies – where the dependent value does not actually depend on the value of the dependency – such as conditional jumps where both branches go the same location, or `XOR`ing values with themselves. These fake dependencies are just as real as any other on Arm, and examples of them can be seen in Figures 2.3 and 2.4.

Not all dependencies are equal. On Arm, address and data dependencies enforce both read-to-read and read-to-write ordering, control dependencies enforce read-to-write but not read-to-read ordering. Speculation allows reads to happen 'early', but not writes; this gives an asymmetry where control dependencies provide strength to a write but not a read. This can be seen in the two tests in Figure 2.4.

2.1.2 Coherence

A guarantee provided by most modern microprocessor architectures is *coherence*: that there is for each location, a total order that writes to that location happens in, that all threads agree on [8].

This property is one that sets processor consistency models apart from those you would find in databases and

MP+dmb.st+ctrl	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	**LDR** X0,[X1]
STR X0,[X1]	**CBNZ** X0,LC00
DMB ST	LC00:
MOV X2,#1	**LDR** X2,[X3]
STR X2,[X3]	
Allowed: 1:X0=1, 1:X2=0	

LB+ctrls	AArch64
Initial state: 0:X1=x, 0:X3=y,	
1:X1=y, 1:X3=x, *x=0, *y=0	

Thread 0	Thread 1
LDR X0,[X1]	|LDR X0,[X1]|
CBNZ X0,LC00	**CBNZ** X0,LC01
LC00:	LC01:
MOV X2,#1	**MOV** X2,#1
STR X2,[X3]	**STR** X2,[X3]
Forbidden: 0:X0=1, 1:X0=1	
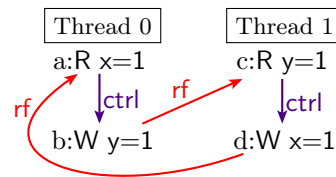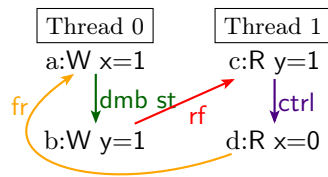
Figure 2.4: Two litmus tests with speculation.
On the left: MP+dmb.st+ctrl with Arm `DMB` barrier between the writes, but a control dependency between the reads.
On the right: LB+ctrls, a variant of the classic 'load buffering' litmus test, with control dependencies to both writes.

other distributed systems, which generally do not require it, such as the classic *causal consistency* model for distributed systems [41].

Two of the key litmus tests for coherence can be found in Figure 2.5.

2.1.3 Multi-copy atomicity

Coherence alone does not guarantee that all threads agree on what the most recent write is at the same point in time. Eventually, they will all have seen the same writes to the same location in the same order, but at any particular moment, some threads may not have caught up to the latest write yet. Architectures that have this property are *non-multi-copy* atomic [42].

Arm has a kind of partial multi-copy atomicity, which they call *other*-multi-copy atomicity in their manual. This other-multi-copy atomicity gives guarantees similar to normal multi-copy-atomic architectures, but allows writes to be read by the writing thread itself earlier than they can be seen by other threads, however, once a write has propagated to another thread then all threads must see that write as normal [7]. This happens by *write forwarding*; the processor can satisfy a read from a same-thread same-location program-order-earlier write, if that write has committed, even before the write has propagated out to memory. Figure 2.6 contains the classic PPOCA (preserved-program-order–control–address) litmus test, which shows that writes can be observed locally before being propagated to other threads, even down speculative branches. Figure 2.7 shows the classic IRIW (independent-reads independent-writes) litmus test, which demonstrates the latter point, that writes propagate to all threads simultaneously.

CoRR1   AArch64

Initial state:
0:X1=x, 1:X1=x, 1:X3=x,
*x=0,

Thread 0	Thread 1
MOV X0,#1	**LDR** X0,[X1]
STR X0,[X1]	**LDR** X2,[X3]

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

CoWR   AArch64

Initial state:
0:X1=x, 1:X1=x,
0:X3=x, *x=0

Thread 0
MOV X0,#1
STR X0,[X1]
LDR X2,[X3]

Forbidden: 0:X2=0
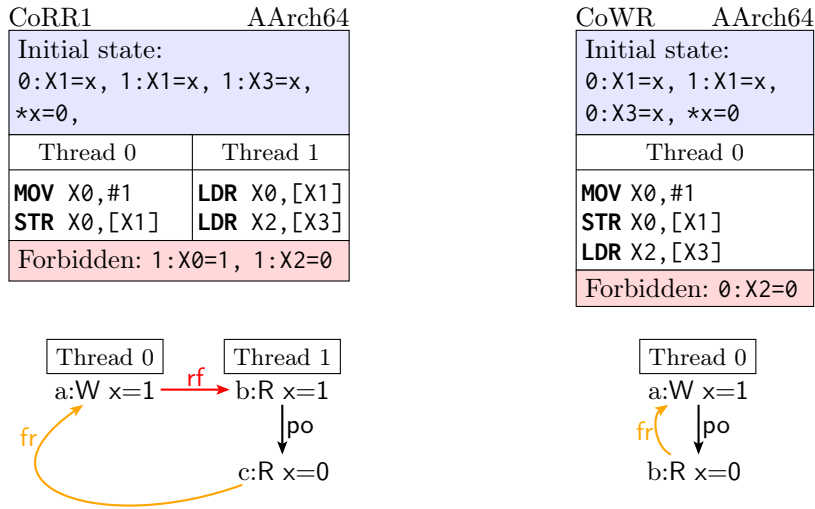
Figure 2.5: Two coherence litmus tests.
On the left: CoRR1, that two subsequent reads of the same location in the same thread should be consistent with the coherence order. On the right: CoWR, that a read of a location cannot skip over a newer program-order earlier write from the same thread.

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	**LDR** X0,[X1]
STR X0,[X1]	**EOR** X8,X0,X0
DMB ST	**MOV** X2,#1
MOV X2,#1	**STR** X2,[X3,X8]
STR X2,[X3]	**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	**LDR** X0,[X1]
STR X0,[X1]	**CBNZ** X0,LC00
MOV X2,#1	LC00:
STR X2,[X3]	**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			
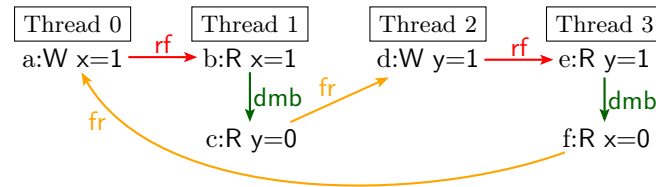
Figure 2.7: IRIW+dmbs: a classic multi-copy atomicity litmus test.

2.2 Intra-instruction semantics

Much of the work presented here will be dedicated to understanding the *inter*-instruction and concurrency aspects of the semantics. Previous work has, at least for Arm and RISC-V, established high-fidelity models for the semantics of individual instructions, that is, the sequential behaviour of a single instruction: the register reads and writes they perform, along with any memory effects they do.

Arm produces such models as part of their architecture specifications, in their custom ASL (*architecture specification language*) programming language [10], which can be found in the manual [12] or otherwise acquired from Arm [43].

The ASL and Sail specification languages Although the work here is focused on Arm-A, and Arm use their ASL language, the tools we build upon are generally architecture agnostic and use the Sail specification language for instruction semantics [44]. For compatibility with those tools we use the `asl_to_sail` generated translations [44, 45] throughout the work presented here. Sometimes the listings given will be extracted from the Arm documentation (and therefore will be in ASL) or from the tooling (and so be in Sail); the captions of any figures or listings should make it clear which language the presented code is in. Sail and ASL are very similar languages, and are used for broadly the same purposes, with similar syntax and semantics; we will not go into depth here into the history or minutiae of them, instead, we will look at just one aspect of Sail (its effect system) as it is important to the function of the tools we will use later on.

Outcomes Sail programs are *effectful*, they have effects such as *read register*, *write register*, *read memory*, and so on.

These effects make Sail programs monadic computations over the sail effect datatype (called `outcome`). Figure 2.8 lists the outcomes defined by the Sail effect system [15], it contains one pure value (DONE), and the other values are each a paused computation containing a continuation.

Read_mem(`read_kind, address, size, read_continuation`)	Read request
Write_ea(`write_kind, address, size, next_state`)	Write effective address
Write_memv(`memory_value, write_continuation`)	Write value
Barrier(`barrier_kind, next_state`)	Barrier
Read_reg(`reg_name, read_continuation`)	Register read request
Write_reg(`reg_name, register_value, next_state`)	Write register
Internal(`next_state`)	Pseudocode internal step
Done	End of pseudocode

Figure 2.8: Outcomes (the Sail effect datatype).

```
1   function execute_aarch64_instrs_integer_arithmetic_add_sub_shiftedreg (d,
        datasize, m, n, setflags, shift_amount, shift_type, sub_op) = {
2       result : bits('datasize) = undefined;
3       let operand1 : bits('datasize) = X_read(datasize, n);
4       operand2 : bits('datasize) = ShiftReg(datasize, m, shift_type, shift_amount)
        ;
5       nzcv : bits(4) = undefined;
6       carry_in : bits(1) = undefined;
7       if sub_op then {
8           operand2 = not_vec(operand2);
9           carry_in = 0b1
10      } else {
11          carry_in = 0b0
12      };
13      (result, nzcv) = AddWithCarry(operand1, operand2, carry_in);
14      if setflags then {
15          (PSTATE.N @ PSTATE.Z @ PSTATE.C @ PSTATE.V) = nzcv
16      };
17      X_set(datasize, d) = result
18  }
```

Figure 2.9: Sail pseuodcode for the `ADD Xd,Xn,Xm` instruction.

An example instruction   As an example, take the Arm `ADD Xd,Xn,Xm` instruction, whose Sail code can be found in Figure 2.9, as extracted from the original source ASL code in the Arm manual. It takes two input registers (Xn,Xm), adds the values stored in them together, and stores the result in the output register (Xd), updating any flags as it does so.

The calls to `X_read` and `X_set`, and (not shown) `EndOfInstruction`, each have an effect, and emit an outcome in the trace. Omitting the outcomes for the flag registers, and the exact arithmetic calculation, then this code results in the following trace of outcomes:

```
Read_reg(n, fun v1 ->
    Read_reg(m, fun v2 ->
        Write_reg(d, (v1 + v2), Done)
    )
)
```

The set of traces for an instruction define the semantics of that instruction, and the concurrency models described later in this chapter are parameterised over such traces.

2.3 Arm-A operational model

The canonical multi-copy atomic operational semantics for Arm is the *Flat* model [7].

Flat is a small-step operational semantics, with transitions designed to (abstractly) match the kinds of actions we see in hardware.

Flat is implemented, as an executable-as-a-test-oracle model, in the RMEM tool [46]. RMEM is written in a combination of OCaml and the Lem [47, 48] language for operational semantics. It can either be run through a command-line interface for example, to run batches of tests, or can be used interactively, including through a version compiled to JavaScript which can be run in a web browser [49].

Flat has an explicit flat memory (from which it derives its name), which stores the most recent write that propagated to memory for each location, and a set of hardware threads, with each thread containing a tree of concurrently executing instructions (abstractly modelling modern microprocessor pipelines) with explicit out-of-order execution.

Figure 2.10 demonstrates a snapshot of an example instruction tree from a thread executing 10 instructions. Some instructions (i_2, in grey) have finished executing, some (i_3, i_6, i_7, i_9, blank/white) have not begun executing, and some (i_0, i_1, i_4, i_8, i_5, in pink) are currently in-progress. Flat has explicit speculation down branches, and

re-ordering of instructions. This can be seen in the diagram: there is a fork in the tree at i_3 (a branch in the program) which has not yet been executed while some earlier instructions (i_0, i_1) have not finished (and so it is not yet known whether the program will execute down branch i_4 or i_8), but later instructions down both branches have already begun executing.
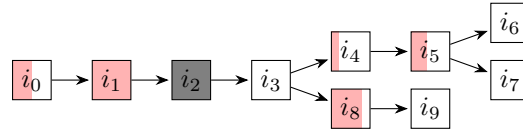
Figure 2.10: A tree of 10 concurrently executing instructions.

Flat is composed of two subsystems, a storage subsystem which contains a flat array for memory, and the thread subsystem which contains a set of threads which can only communicate with the flat memory and not between each other, as sketched in Figure 2.11.
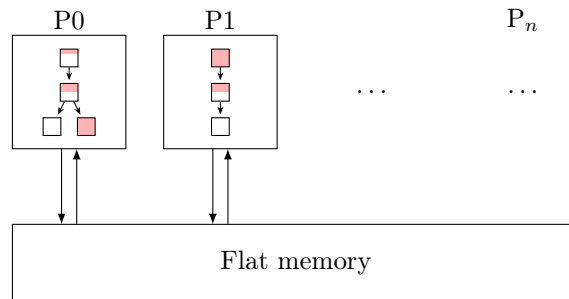
Figure 2.11: Flat state (diagram).

Thread subsystem   The thread subsystem has, for each thread, a tree of instructions. Each node in the tree is an instruction *instance*, a piece of state representing a single instruction in the process of being fetched, decoded and executed; its state includes the current pseudocode state (such states are listed in Figure 2.12), as well as any other ancillary data required by the operational model (pending addresses and values and so on).

The thread system then has a set of guarded transitions, split into two groups: the local transitions, each of which calls the continuation contained within the outcome of an instance and updates the instruction instance state with the new outcome; and, the synchronised transitions which can also update the storage subsystem state, these generally update the `micro_op_state` (containing the current pseudocode state) without calling the `outcome` continuation. Figure 2.13 contains a fragment of the Lem code from RMEM which defines the thread subsystem state and the relevant transitions (but not their guards).

Plain(`next_state`)	Ready to make a pseudocode step
Pending_mem_reads(`read_cont`)	Performing the read(s) from memory of a load
Pending_mem_writes(`write_cont`)	Performing the write(s) to memory of a store

Figure 2.12: Operational pseudocode states.

```
 1  type threadSubsystem =          22         of reg_name * value
 2    nat → instruction_tree;        23    | T_satisfy_read
 3  type instruction_tree =          24         of value
 4    list (instruction_instance *   25    | T_mem_write_footprint
         instruction_tree);          26         of list write
 5  type instruction_instance =      27    | T_mem_potential_write
 6    <| id: nat;                    28         of list write
 7       program_loc: address;       29    | T_commit_store
 8       micro_op_state: micro_op_state;  30    | T_complete_store
 9       mem_reads: set address;     31    | T_commit_barrier
10       ... |>                      32         of barrier_kind
11  type micro_op_state =            33    | ...
12    | MOS_plain                    34  type sync_trans =
13         of outcome                35    | T_propagate_write
14    | MOS_pending_mem_read         36         of write
15         of (value → outcome)      37    | T_satisfy_read
16    | MOS_potential_mem_write      38         of read_request * value
17         of outcome                39    | T_propagate_barrier
18  type thread_trans =              40         of barrier_kind
19    | T_register_read              41    | ...
20         of reg_name * value
21    | T_register_write
```

Figure 2.13: Lem fragment of thread subsystem state.

Storage subsystem The Flat storage subsystem is comparatively straightforward, a finite map from location to the most-recently propagated write to that location. Figure 2.14 contains a fragment of the Lem sources from RMEM for the (non-mixed-size) Flat storage subsystem.

```
type flat_storage_subsystem_state = <| memory: nat → write; ... |>
```

Figure 2.14: Simplified Lem listing of the Flat storage subsystem state from RMEM.

Transitions Flat defines a set of common transitions for all instructions, as well as a set of key transitions specific to stores, loads and barriers. Below is a complete list of the local and synchronised transitions.

Common transitions:
  ▷ Fetch instruction.
  ▷ Pseudocode internal step.
  ▷ Register read.
  ▷ Register write.
  ▷ Finish instruction.

Transitions on a Load instruction:
  ▷ Initiate read.
  ▷ Satisfy read from forwarding.
  ▷ Satisfy read from flat memory.
  ▷ Complete load.

Transitions on a Barrier instruction:
  ▷ Commit barrier.

Transitions on a Store instruction:
  ▷ Initiate write address.
  ▷ Initiate write data.
  ▷ Commit write.
  ▷ Propagate write to memory.
  ▷ Complete store.

Each transition has a guard (a predicate over the state that must be true in order for the transition to be valid) and an action (a function that updates the whole system state from one configuration to another). Figure 2.15 gives the informal description of one transition, the 'Initiate read' on a load, including its guard and action. I do not describe the entire Flat model here in detail.

Transition: Initiate memory reads for instruction i.
Guard: Instruction i is in state MOS_plain(O_read_mem(addr,cont)).
Action:
  ▷ Add addr to i.mem_reads
  ▷ Update the state of i to MOS_pending_mem_read(fn v → cont v)

Figure 2.15: Example Flat transition in full.

2.4 Arm-A axiomatic model

In contrast to the operational model presented in the previous section, a model with equivalent behaviour can be given declaratively, as an *axiomatic*-style model. These axiomatic models describe the allowed behaviour of programs by a set of *axioms* constraining the event graphs of the *candidate* executions of that program.

In an axiomatic model, the executions are the graphs of events of a single run of the program, with the events related by a set of intrinsic relations capturing the order of events and their dependencies.

The model first considers an overapproximate set of *candidate* executions: executions consistent with the intra-instruction semantics, but where the values used in the program are unconstrained. The model then has axioms, generally acyclicity of some relation over the events of the execution, which reject some of these executions as *inconsistent*.

The resulting model can then be used to check whether some given program can reach a final state satisfying a given constraint. If there is a candidate executions of the program, which is consistent with the axioms of the model, then the model is said to *allow* that execution, and if the final state satisfies the given constraint, that outcome is permitted by the model.

Succinctly, an axiomatic model winnows down a large set of graphs of potential whole-program executions to a small set of allowed executions by checking that the events of those executions do not violate any of the axioms of the model.

2.4.1 Arm-A candidate executions

Arm-A candidate executions are composed of two parts. First, there is the set of events of the program, for Arm these are the memory access and barrier events, labelled with their access type (read or write, or barrier kind). In addition, there are the fundamental candidate relations over those events, derived from the intrinsic dependencies in the program; some of which we have already seen: program order, and address/data/control dependencies.

It is often useful to split the candidate execution definition into two steps: first, to define the *pre-execution* which contains all the events, and the relations which are intrinsic to the program; then to complete these into a *candidate* execution with existentially-quantified relations (coherence-order and reads-from) which witness a particular choice of runtime execution order.

More formally, we can define an Arm-A candidate execution as: a set of event IDs (here just assuming IDs are the natural numbers); a labelling function (from $\mathbb{N}$ to Label); a collection of the candidate relations ($\mathcal{C}_R$) satisfying some constraints (described in more detail later on), and a candidate witness ($\mathcal{C}_W$) describing the existentially quantified coherence-order and reads-from relations.

$$\text{Candidate Pre-Execution} \equiv \mathcal{P}(\mathbb{N}) \times (\mathbb{N} \to \text{Label}) \times \mathcal{C}_R$$
$$\text{Candidate Execution} \equiv \text{Pre-Execution} \times \mathcal{C}_W$$

The candidate relations, and the candidate witness, are sets of named relations over the events of the pre-execution, subject to some well-formedness constraints (discussed later):

$$\xrightarrow{\mathsf{L}} \subseteq \mathbb{N} \times \mathbb{N}$$
$$\mathcal{C}_R \equiv \langle \xrightarrow{\mathsf{po}}, \xrightarrow{\mathsf{loc}}, \xrightarrow{\mathsf{addr}}, \xrightarrow{\mathsf{ctrl}}, \xrightarrow{\mathsf{data}}, \xrightarrow{\mathsf{rmw}}, \xrightarrow{\mathsf{ext}} \rangle$$
$$\mathcal{C}_W \equiv \langle \xrightarrow{\mathsf{co}}, \xrightarrow{\mathsf{rf}} \rangle$$

Events The labelling function maps each event ID to an event label, describing the kind of access and, if applicable, what data or address it operates over.

A simplified version of the labels, sufficient for the model described here, contains memory events with location and values, namely reads (R) including acquire reads (A) and weak-acquire reads (Q), writes (W) including release writes (L); and a set of Arm barriers (DMB, ISB) and their variants. More precisely, these labels can be described as

follows:

$$\text{Label} \equiv \text{Reads} \cup \text{Writes} \cup \text{Barriers}$$
$$\text{Reads} \equiv \{\mathbf{R}, \mathbf{A}, \mathbf{Q}\} \times \text{Loc} \times \text{Val}$$
$$\text{Writes} \equiv \{\mathbf{W}, \mathbf{L}\} \times \text{Loc} \times \text{Val}$$
$$\text{Barriers} \equiv \{\mathbf{DMB.LD}, \mathbf{DMB.ST}, \mathbf{DMB.SY}, \mathbf{ISB}\}$$
$$\text{Loc} \equiv \text{Bitvec}_{48}$$
$$\text{Val} \equiv \text{Bitvec}_{64}$$

In §2.5.1 we will see a more realistic definition of the event types for a production architecture (Armv9-A), and their correspondence to the underlying effects of the Sail definition, as used by the `isla-axiomatic` tool.

Candidate relations   The candidate relations capture the relationships and orderings between the events of the execution. These are often separated into two kinds: the *pre-execution* relations (which are intrinsic to the program), and the existentially-quantified coherence-order and reads-from relations of the witness, combined these two sets make up the relations of the candidate execution. For Arm, the relations in a pre-execution are, with their intended meaning:

- ▷ program order: E_1 po E_2 iff the instruction generating E_1 occurs before the instruction generating E_2 in the instruction stream.
- ▷ same-location: M_1 loc M_2 iff the address of M_1 is the same location as used by M_2.
- ▷ address dependent: R_1 addr M_2 iff the value read by R_1 is used in the calculation of the address M_2.
- ▷ data dependent: R_1 data W_2 iff the value read by R_1 is used in the calculation of the value written by W_2.
- ▷ control dependent: R_1 ctrl E_2 iff the value read by R_1 is used to determine whether or not the instruction E_2 originates from would have executed at all.
- ▷ read-modify-write: R_1 rmw W_2 for the separate read and write events of an atomic update.
- ▷ external: E_1 ext E_2 iff the instructions which generated events E_1 and E_2 originated from different hardware threads.

Plus the existentially quantified witness:

- ▷ reads-from (rf), from W_1 to R_2 when R_2 reads the value that W_1 wrote.
- ▷ coherence-order (co), from W_1 to W_2 where W_1 appears before W_2 in the coherence order of that location, (informally, that W_1 propagated to memory before W_2).

(E_n represents events of any kind, M_n is a memory effect event, R_n is a read event, and W_n is a write event)

Well-formedness   Each of the relations of the candidate relations and witness are subject to some well-formedness constraints.

Note that a well-formed execution does not necessarily correspond to a consistent execution of the underlying ISA (see 'Fundamental candidates and ISA-Consistency').

Well-formedness requires that the candidate relations are all properly constructed: they have the right type, and satisfy some basic relational properties (symmetry, reflexivity, transitivity and so on) depending on the relation. Figure 2.16 contains the types and some basic well-formedness properties of the pre-execution relations.

Relation	Type	Properties
po	E × E	transitive, asymmetric, irreflexive
loc	M × M	transitive, symmetric, reflexive
ext	E × E	transitive, symmetric, irreflexive
addr,ctrl	R × M	asymmetric, irreflexive
data	R × W	asymmetric, irreflexive
rmw	R × W	asymmetric, irreflexive

Figure 2.16: Non-ISA-dependent well-formedness properties of pre-execution relations.

For the existentially-quantified coherence-order and reads-from relations, they are arbitrary, but subject to the constraints given in Figure 2.17.

$$\forall W_1, R_2.\ \mathrm{rf}(W_1, R_2) \implies \mathrm{loc}(W_1, R_2) \qquad \text{read and write must be same location}$$
$$\forall W_1, R_2.\ \mathrm{rf}(W_1, R_2) \implies \text{r-value}(R_2) = \text{w-value}(W_1) \qquad \text{value read matches value written}$$
$$\forall W_1, W_2, R_3.\ \mathrm{rf}(W_1, R_3) \wedge \mathrm{rf}(W_2, R_3) \implies W_1 = W_2 \qquad \text{each read reads-from at most one write}$$
$$\forall R_2.\ \exists W_1.\ \mathrm{rf}(W_1, R_2) \qquad \text{every read reads from somewhere}$$

$$\forall W_1, W_2.\ W_1 \neq W_2 \wedge \mathrm{loc}(W_1, W_2)$$
$$\implies \mathrm{co}(W_1, W_2) \vee \mathrm{co}(W_2, W_1) \qquad \text{co is per-location total}$$
$$\forall W_1, W_2, W_3.\ \mathrm{co}(W_1, W_2) \wedge \mathrm{co}(W_2, W_3) \implies \mathrm{co}(W_1, W_3) \qquad \text{co is transitive}$$
$$\forall W_1, W_2.\ \mathrm{co}(W_1, W_2) \implies \neg\mathrm{co}(W_2, W_1) \qquad \text{co is antisymmetric}$$
$$\nexists W_1.\ \mathrm{co}(W_1, W_1) \qquad \text{co is irreflexive}$$

Figure 2.17: Well-formedness conditions of co and rf.
r-value and w-value extract the Val from a read or write respectively.
(Hand transcribed from the versions used in isla-axiomatic, see §2.5)

675 We say a candidate execution is *well-formed* if all the constraints of all the relations are satisfied:

$$\textsc{Well-Formed}(E : \text{Execution}) = \text{see Figures 2.16 and 2.17}$$

676 **Fundamental candidates and ISA-Consistency** Candidate executions are constructed from a limited set
677 of events: reads, writes, and barriers. Eventually, we will see that later models extend this set, both with
678 more instructions and further architectural features, but also with an expanded set of intrinsic events from the
679 intra-instruction semantics.

For a candidate execution to be consistent with a given architecture's intra-instruction semantics, as defined by
its ISA, then there must be a corresponding execution in a model whose events have been expanded to include *all*
the events of the underlying ISA. We can imagine taking the candidate execution and 'completing' the events to
include all the relevant register reads and writes, and instruction fetches, and other intrinsic events the ISA *would
have produced*, and we get a 'fundamental' candidate execution.

$$\text{Fundamental Execution} \equiv \mathcal{P}(\mathbb{N}) \times (\mathbb{N} \to \text{Label}_F) \times \mathcal{C}_{\text{Rf}} \times \mathcal{C}_{\text{W}}$$
$$\text{Complete}(E : \text{Execution}) : \text{Fundamental Execution}$$

Fundamental executions are much like their candidate counterparts, except that the labels are simply the set
of possible outcomes as defined by the ISA, with continuations replaced by their arguments; and the various
candidate relations are replaced by intra-instruction causality orders.

$$\text{Label}_F \equiv \text{Outcome (see Figure 2.8)}$$
$$\mathcal{C}_{\text{Rf}} \equiv \langle \xrightarrow{\text{po}}, \xrightarrow{\text{iico-addr}}, \xrightarrow{\text{iico-ctrl}}, \xrightarrow{\text{iico-data}} \rangle$$

680 As an example, take the reader thread of an MP-shaped test, with a barrier between the loads. Figure 2.18 shows a
681 sketch for a completion of that reader thread to a fundamental execution in Arm, with introduced events in blue
682 (assuming translation disabled, and eliding voluminous ISA intricacy).

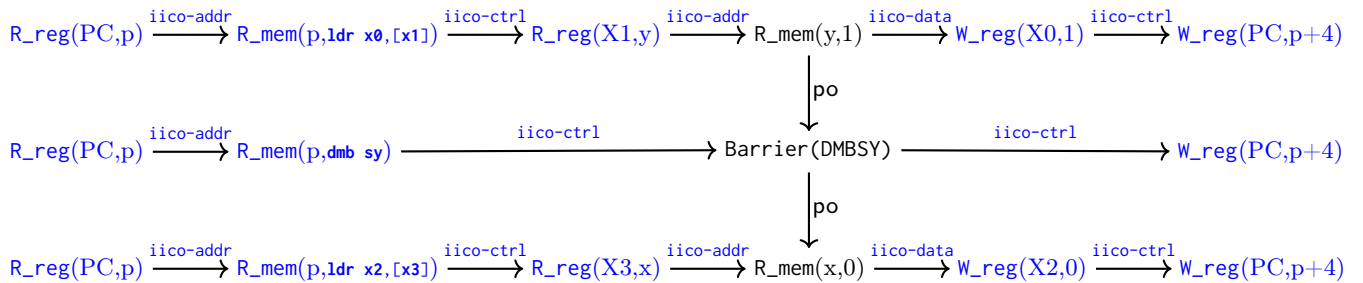**Figure 2.18:** Completion of reader thread of MP+dmb.sys into a fundamental candidate.
Nodes and edges in black are original, the ones in blue complete the execution.

683 The previously primitive inter-instruction dependencies (addr, ctrl, data) become derived relations, and part of the
684 ISA-consistency check requires that the candidate dependencies matches the derived ones in the fundamental
685 execution.

Given a fundamental candidate we can partition it into each thread (by grouping by `int`) and then into instructions (by grouping by `iico`). For each instruction we can extract a trace of events, by following `iico`. Recall that the intra-instruction semantics defines a set of traces, so we can ask whether the extracted trace from the graph corresponds to one of these traces defined by the intra-instruction semantics, which is precisely asking whether the extracted trace simulates the ISA:

$$\text{Instr}(I : \mathcal{P}(\mathbb{N}), F : \text{Execution}) : I \subseteq E.iico^+[I]$$
$$\text{SimulatesISA}(F : \text{Fundamental Execution}) : \forall I. \text{Instr}(I, F) \Rightarrow I \in \text{ISA}$$

Where r^+ is the symmetric closure of r.

We can now define what it means for an execution to be consistent with the ISA (with respect to some given intra-instruction semantics). If there exists a completed fundamental candidate, such that, for each instruction, the sequence of events in `iico` order is an execution of the intra-instruction semantics, then we can say the original execution is *ISA-Consistent*:

$$\text{ISA-Consistent}(E) = \exists F. F = \text{Complete}(E) \land \text{SimulatesISA}(F)$$

In practice, tools generally to go the other way: producing complete traces from the intra-instruction semantics defined by the ISA, and discarding or hiding events down to a smaller set — thereby producing ISA-Consistent executions by construction. However, it is still useful to think in terms of *completing* the executions up to a larger fundamental candidate, as not all models explicitly appeal to the intra-instruction semantics in their definitions, especially historically.

Consistency   Given an arbitrary pre-execution, that is, a graph with any choice of events and relations, one can define whether or not such a graph corresponds to a valid execution. This can be done by checking that: there exists some witness (`co` and `rf`) such that that candidate is well-formed; that the candidate is consistent with the ISA; and, that does not violate any of the axioms of the model.

$$\text{Axiom-Consistent}(E : \text{Execution}) = \text{see §2.4.2}$$
$$\text{Consistent}(E : \text{Execution}) = \text{Well-Formed}(E)$$
$$\land \ \text{ISA-Consistent}(E)$$
$$\land \ \text{Axiom-Consistent}(E)$$
$$\text{Consistent}(E : \text{Pre-Execution}) = \exists \text{co, rf.} \ \text{Consistent}((E, \langle \text{co, rf} \rangle))$$

Program semantics   Consistent executions correspond to *some* execution of the underlying architecture, but not necessarily to one that follows from a given initial state.

We do not need consistency checks to be aware of the initial state, instead, we can generate the set of executions with some 'initial write' events corresponding to the initial state prefixed onto the execution (coherence-before any other writes) and do the usual consistency check.

Each execution then has a 'final' state, which is the concrete register values for each thread at the end of execution, and the coherence-final write for each location.

We can then define whether a particular outcome is permitted by the model, by checking whether there exists any consistent execution, prefixed with the initial writes from the program, whose final state matches the desired outcome:

$$\text{State} \equiv \text{Memory} \times (\text{ThreadId} \rightarrow \text{Registers})$$
$$\text{Program} \equiv \text{State}$$
$$\text{Final}(E : \text{Execution}) = \text{`Final register and memory state of } E\text{'}$$
$$\text{Prefixed}(P : \text{Program}, E : \text{Execution}) = \text{`E has co-initial writes corresponding to P's memory'}$$
$$\text{Permitted}(P : \text{Program}, S : \text{State}) = \exists E : \text{Pre-Execution, co, rf.}$$
$$\text{let } C = (E, \langle \text{co, rf} \rangle) \text{ in}$$
$$\text{Prefixed}(P, C)$$
$$\land \ \text{Consistent}(C)$$
$$\land \ S = \text{Final}(C)$$

Giving semantics to an Arm-A program can be done by collecting the set of consistent executions:

$$\llbracket P : \text{Program} \rrbracket = \{S : \text{State} \mid \textsc{Permitted}(P, S)\}$$

(Note that this means $\llbracket \_ \rrbracket$ is not defined compositionally as a traditional denotational semantics would be, instead, here we have a whole-program consistency check)

An example Consider the classic MP+dmb.sy+addr litmus test, whose code listing is contained in Figure 2.19. The test has two threads, the first with two store instructions and a barrier, the second with two loads with a syntactic address dependency between them, forming the classic message-passing shape seen earlier. Figure 2.20 contains six potential candidate executions for this test:

  ▷ Candidate 1 is not consistent with the intra-instruction semantics: it has read events in Thread 0, but the intra-instruction semantics requires store instructions generate write events not read events.

  ▷ Candidate 2 has events consistent with the intra-instruction semantics, but the relations are not consistent with the well-formedness conditions (specifically, rf does not satisfy the 'read and write must be same location' constraint), and so this candidate is not well-formed.

  ▷ Candidates 3, 4 and 5, are well-formed, and consistent with the ISA, and consistent with the axioms of the model (given in §2.4.2).

  ▷ Candidate 6 is well-formed, and consistent with the ISA, but not consistent with the axioms.

The four well-formed candidate executions in Figure 2.20 are the *only* well-formed and ISA-Consistent candidates for this test. Executions with other events would not be ISA-Consistent; or with co and rf other than those shown would not be well-formed; or, with read or write values, other than those shown, would also not be ISA-Consistent, as those values must have arisen from an execution of the intra-instruction semantics. Only Candidate 6 has a final state which satisfies the 1:X0=1,1:X2=0 constraint of the test. Since no candidate satisfying the final state constraint is consistent with the axioms, the test is *forbidden*.

MP+dmb.sy+addr	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	**LDR** X0,[X1]
STR X0,[X1]	**EOR** X4,X0,X0
DMB SY	**LDR** X2,[X3,X4]
MOV X2,#1	
STR X2,[X3]	

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

Figure 2.19: MP+dmb.sy+addr test code listing.
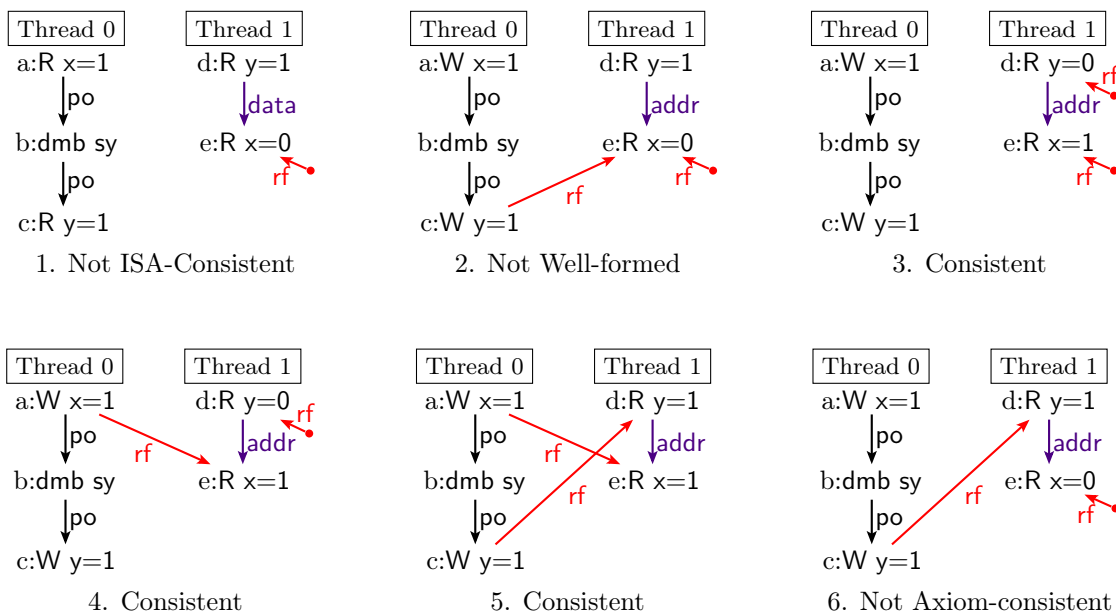

Figure 2.20: Six potential candidate executions for MP+dmb.sy+addr.

2.4.2 Arm-A axioms

Axiomatic models define *axioms* over sets of events, primarily as acyclicity requirements over a given set of relations over those events. The axioms of the model define which executions are *Axiom-consistent*. Final states from consistent executions are those states that are permitted by the model to be observed on hardware.

Formally one often defines an axiomatic model as a statement in a relation[1] algebra over events, as done by Alglave et al [39]. These relations are constructed composing the candidate relations $\mathcal{C}_R$, and the restricted identity relation ($\mathbf{id}_E$, for identity over events with label E), with some standard relation operators: union (|), intersection (&), relation composition (usually written with the reversed ; operator), transitive closure (*), and relation inverse ($^{-1}$). The model is then a set of relations defined in this algebra, describing the set of preserved orderings, with axioms requiring some of them to be acyclic.

We write these models in the herd model definition language (often commonly referred to as simply *Cat*), introduced by Alglave et al [39]. Cat is a general language that allows one to express first-order quantifier-free relations, in a relatively concise syntax, using a set of built-in relations and relational operators. Values in Cat are either sets of events, or relations (sets of pairs of events). Cat extends the usual set of relational operators with some custom syntax, reproduced here for quick reference: R+ is equivalent to the Cat expression (R; R*); [E] corresponds to the mathematical relation $\mathbf{id}_E$, [E1 | E2] corresponds to the relation $\mathbf{id}_{E1}$ | $\mathbf{id}_{E2}$, and so on for any number of unions; domain(R) and range(R) give the sets of events that are the domain and codomain of a relation; (E1 * E2) is the cartesian product of the sets of events with labels E1 and E2, that is, the mathematical relation range($\mathbf{id}_{E1}$)$\times$range($\mathbf{id}_{E2}$); (E1 * _) and (_ * E2) are also the relations formed by the cartesian product of sets of events, but where _ is a wildcard that matches events with any label; id for the generalised identity relation over events, which corresponds to $\mathbf{id}_{\_}$; and, R? for relation option, equivalent to R | id. The original herdtools Cat language and the isla-axiomatic Cat-like model language have diverged somewhat over time, but the features described in this section remains common to both.

An Arm-A Cat model A modified version of the original non-mixed-size multi-copy-atomic Armv8-A model [7, 50], can be found in Figure 2.21. The other models presented in this thesis will be an extension to the one presented here. Note that this particular presentation of the model is slightly different from the original, with the transitive relations over barriers split into multiple edges explicitly relating events to barriers, and lifting coi and fri into obs. Although equivalent to the original, this presentation will be easier to extend, the reason for which will become apparent later on. Additionally, the current official Arm models have diverged from the original model this one is based on, either through the addition of new features (mixed-size, memory tagging extensions, and so on), or through iterative refactors of the model over time. An isla-axiomatic-executable version of the model can be found at https://github.com/rems-project/system-semantics-arm-axiomatic-models/blob/main/models/aarch64_interface.cat.

```
1   (* observed by *)
2   let obs = rfe | fr | co
3
4   (* dependency-ordered-before *)
5   let dob =
6       addr | data
7       | ctrl; [W]
8       | addr; po; [W]
9       | (ctrl | (addr; po)); [ISB]
10      | (addr | data); rfi
11
12  (* atomic-ordered-before *)
13  let aob = rmw
14      | [range(rmw)]; rfi; [A | Q]
15
16  (* barrier-ordered-before *)
17  let bob = [R] ; po ; [dmbld]
18      | [W] ; po ; [dmbst]
19      | [dmbst]; po; [W]
20      | [dmbld]; po; [R|W]
21      | [ISB]; po; [R]
22      | [L]; po; [A]
23      | [A | Q]; po; [R | W]
24      | [R | W]; po; [L]
25
26  (* Ordered-before *)
27  let ob1 = obs | dob | aob | bob
28  let ob = ob1+
29
30  (* Internal visibility
       requirement *)
31  acyclic po-loc | fr | co | rf as
       internal
32
33  (* External visibility
       requirement *)
34  irreflexive ob as external
35
36  (* Atomic: Basic LDXR/STXR
       constraint to forbid
       intervening writes. *)
37  empty rmw & (fre; coe) as atomic
38
```

Figure 2.21: Armv8-A multi-copy atomic 'user' axiomatic model.

---

[1]To distinguish from relational algebra, as one would find in discussions of relational databases.

The Cat model relies on a set of built-in event sets and relations, these are:

	Events			Relations
R	Reads		`po,rmw`	program-order and read-modify-write
W	Writes		`po-loc`	program-order same-location (po & loc)
M	All explicit memory events (R	W)	`addr,ctrl,data`	dependencies
A	Read-acquire		`co,rf`	existentially-quantified (candidate) relations
L	Write-release		`rfe,rfi`	rf-external (rf & ext), rf-internal (rf & ~ext)
Q	Weak read-acquire		`coe,coi`	co-external, co-internal
F	Fences (barriers)		`id`	identity
ISB	Instruction sychronization barrier			
dmbXY	Barrier, with type XY={st,ld,sy}			

The axioms The Arm-A model is made up of three axioms: `external` (line 34), which asserts acyclicity of the primary ordered-before relation, capturing most of the ordering constraints of the Arm memory model; the `internal` axiom (line 31), sometimes called 'SC-per-location', which ensures that when restricted to a single location the accesses are consistent with an SC semantics; and, the `atomic` axiom (line 37)which asserts that there are no same-location writes interposing between events of what is supposed to be an atomic action.

Ordered-before The main ordered-before relation, defined on line 28, is called ob in the Arm Cat model, and is defined as the transitive closure of the union of a set of auxiliary ordering relations. These auxiliary relations are: observed-by (obs, line 2) which orders events after the events they observe the effect of, namely, writes must happen before other-thread reads which read from them; dependency-ordered-before (dob, line 5), which orders events which must not be re-ordered due to syntactic dependencies in the original source program; atomic-ordered-before (aob, line 13) which asserts that the read of an atomic read-modify-write happens before the write, and that acquire reads of an atomic write are ordered; and, barrier-ordered-before (bob, line 17) between events where there is an intervening barrier instruction ordering them. A candidate execution with a cycle in ordered-before is forbidden. For example, in the following MP+dmb.st+addr test, whose code listing and event diagram for the forbidden execution can be found in Figure 2.22.
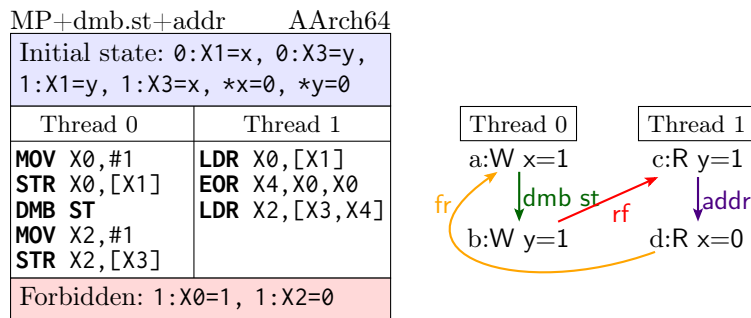
Figure 2.22: MP+dmb.st+addr test code listing and execution diagram.

The interesting candidate execution of MP+dmb.st+addr shown in Figure 2.22, results in the final state `1:X0=1 ∧ 1:X2=0`, and contains the following cycle:

> ▷ a `dmb st` b
> ▷ b `rfe` c
> ▷ c `addr` d
> ▷ d `fr` a

This cycle is forbidden in the Arm model, as each of the relations are contained in ob, and a cycle in ob is forbidden by the `external` axiom:

> ▷ (`[W]; dmb st; [W]`) is in bob, which is in ob.
> ▷ `rfe` is in obs, which is in ob.
> ▷ `addr` is in dob, which is in ob.
> ▷ `fr` is in obs, which is in ob.

Internal and atomic   The other axioms of the model forbid behaviours that the ordered-before acyclicity check does not recognise, such as non-SC behaviours for single locations or supposedly atomic actions (such as exclusives or read-modify-writes) which were interrupted by an intervening write. Figure 2.23 contains two example tests, a coherence test forbidden by the `internal` axiom and an LB-shaped atomic increment failure forbidden by the `atomic` axiom.

CoRR0	AArch64
Initial state:	
0:X1=x, 1:X1=x, *x=0,	

Thread 0	Thread 1
MOV X0,#1	**LDR** X0,[X1]
STR X0,[X1]	**LDR** X2,[X1]
MOV X2,#2	
STR X2,[X1]	

Forbidden: 1:X0=2, 1:X2=1

LB+po-locxxs	AArch64
Initial state: 0:X1=x 1:X1=x *x=0	

Thread 0	Thread 1
LDXR X0,[X1]	**LDXR** X0,[X1]
ADD X0,X0,#1	**ADD** X0,X0,#1
STXR W3,X0,[X3]	**STXR** W3,X0,[X3]

Forbidden: 0:X3=0, 1:X3=0, *x=1
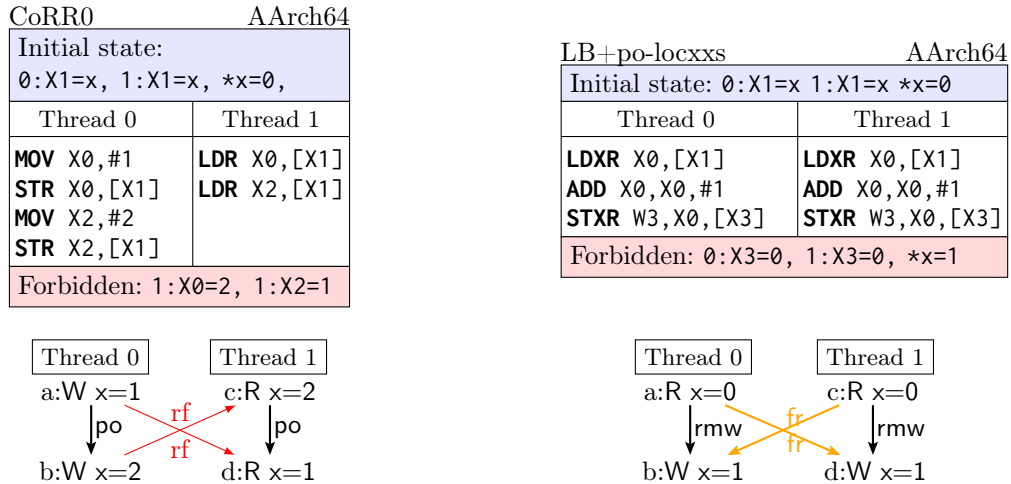
Figure 2.23: Two tests forbidden by the other axioms.
On the left, a variation on coherence which relies on `po-loc` and so is forbidden by the `internal` axiom.
On the right, an atomic increment that failed to atomically update the location, forbidden by the `atomic` axiom.

Note that this is not the only possible presentation of the model. A separate internal/SC-per-location axiom is classic, but the current official herdtools version of the Arm model has separate axioms for each of the forbidden coherence shapes [51]. The external axiom usually considers a partially-ordered happens-before (ordered-before) relation, which is built from smaller primitive relations, as was presented here, but other formulations sometimes pick some linearisation of some total order, equivalent to but more operational in presentation than the one presented here. Finally, there may be an equivalent formulation of the Arm model with only a single axiom, but no known formulation exists.

2.5   The isla-axiomatic tool

Throughout this work we will use the `isla-axiomatic` [33] tool to implement executable versions of our axiomatic models.

The `isla-axiomatic` tool uses the full Arm ISA's ASL specification, converted to Sail. The generation of candidates then uses whole machine states, including all instruction fetch and translation table walks as real memory accesses; unlike in herd where the instruction semantics are ad-hoc.

Using `isla-axiomatic` allows us to use the Arm ASL definitions which already exist (for instruction fetching, decoding, and translation table walks in particular), giving us those fundamental executions 'for free' for those features, and enabling us to focus on modelling the concurrent aspects of them.

isla-axiomatic candidates   Underpinning the `isla-axiomatic` tool is `isla`, a generic symbolic evaluator for Sail programs [33].

`isla-axiomatic` uses `isla` to generate candidate executions, by producing traces of Sail outcomes for each thread, with concrete control flow but potentially symbolic values for reads and writes. `isla-axiomatic` then produces the relevant dependency relations (which it does in an ad-hoc way), then applies a restriction to the events of the traces (discarding all events except reads, writes and barriers for the base model), and takes the cartesian product of these restricted traces of events for each thread; the result is precisely the set of well-formed pre-executions, in a compact representation (by being symbolic).

Internally, `isla-axiomatic` uses a non-architected fetch-decode-execute loop for each thread, which sequentially fetches the next instruction and runs the Sail (converted from ASL) decode and execute functions, until a pre-determined point is reached (usually a particular 'end-of-test' opcode) which signifies the end of that trace. The top-level fetch-decode-execute function we use roughly matches the following pseuocode definition, but is implemented for real as a hand-written part of our Arm Sail model[1]:

```
1   function Step() {                  12   // magic opcode not part of ISA
2     if pending interrupts then {     13   if opcode == 0xfee1dead {
3       TakePendingInterrupt();        14     EndOfTrace();
4     };                               15   };
5                                      16
6     let pc = Read_reg(PC);           17   let instr = ArmASL_Decode(opcode);
7                                      18
8     let opcode = \                   19   ArmASL_Execute(instr);
9       Read_mem(                      20
10        ReadKind_IFETCH,             21   Write_reg(PC, pc+4)
11        pc, 4);                      22 }
```

At each branch in the Sail code, the symbolic execution forks. This gives a set of traces of outcomes for each thread, with concrete opcodes and register names, but with constrained symbolic values.

We can then use this as an executable oracle for litmus tests. By taking the well-formed pre-executions generated from those symbolic traces, `isla-axiomatic` can produce a single SMT problem for each candidate whose satisfiability encodes whether the candidate is consistent. It does this by creating SMT definitions of: the events from the pre-execution with constraints on symbolic values; the candidate relations (in particular, coherence-order and reads-from); the axioms of the model and any auxiliary relations from the Cat model; with the final assertion from the litmus test. Giving this SMT problem to an off-the-shelf SMT solver (such as Z3) allows automatic consistency checking: if the SMT solver can find a satisfying assignment of the symbolic values, then the execution is allowed; if the SMT solver says it is unsatisfiable then the execution is either forbidden by the axioms, or does not satisfy the constraint on the final state. If no SMT-compiled execution is found satisfiable by the SMT solver, `isla-axiomatic` reports the test as forbidden.

2.5.1 ISA/concurrency interface

This section is based on in-progress work with Thibaut Pérami, Alasdair Armstrong, Thomas Bauereiss, and Peter Sewell.

As `isla-axiomatic` uses the full ISA outcomes, the model should be able to utilise any information exposed in the Sail `outcome` type. To achieve this the `isla-cat` language is extended with the structs and enums from the Sail definition, and an *accessor* construct allowing the model writer to define event sets predicated on the values of fields of the underlying Sail structs.

As previously mentioned, each event in an `isla-axiomatic` candidate execution corresponds to an `outcome` in the trace of the intra-instruction semantics. The outcomes then form the interface between the sequential ISA semantics and the concurrency model. The current Sail ISA/concurrency interface is defined in https://github.com/rems-project/sail/tree/sail2/lib/concurrency_interface.

For example, the Arm Sail model contains the `sail_barrier` outcome[2] :

```
outcome sail_barrier : 'barrier -> unit
```

---

[1]https://github.com/rems-project/sail-arm/blob/master/arm-v9.3-a/src/step.sail#L217
[2]https://github.com/rems-project/sail/blob/sail2/lib/concurrency_interface/barrier.sail#L75

Each architecture's Sail specification can then instantiate the `'barrier` type variable with architecture-specific data. For instance, in Armv9-A the `'barrier` kind is instantiated with a custom `Barrier` type[1], derived from the Arm barrier kind in the official ASL specification:

```
1  enum MBReqDomain = {
2      MBReqDomain_Nonshareable,
3      MBReqDomain_InnerShareable,
4      MBReqDomain_OuterShareable,
5      MBReqDomain_FullSystem
6  }
7
8  enum MBReqTypes = {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All}
9
10 struct DxB = {
11     domain : MBReqDomain,
12     types : MBReqTypes,
13     nXS : bool
14 }
15
16 union Barrier = {
17     Barrier_DSB : DxB,
18     Barrier_DMB : DxB, // The nXS field is ignored from DMBs
19     Barrier_ISB : unit,
20     Barrier_SSBB : unit,
21     Barrier_PSSBB : unit,
22     Barrier_SB : unit,
23 }
24
25 instantiation sail_barrier with
26     'barrier = Barrier
```

Then the Sail Arm specification can use the `sail_barrier` outcome to generate events in the trace, for example in the `DataSynchronizationBarrier` function call which the ASL left uninterpreted is implemented in the Sail model by a `sail_barrier` effect[2] which generates a barrier event in the trace when executed:

```
1  function DataSynchronizationBarrier (domain, types, nXS) = {
2      sail_barrier(Barrier_DSB(struct { domain = domain, types = types, nXS = nXS
       }))
3  }
```

2.5.2 Extended Cat with Sail interface

The extended `isla-cat` language is very similar to the original Cat language but with some differences. Since `isla-axiomatic` does not support mutually recursive bindings, procedures, or inline function definitions, we will not use them in our models.

Unlike Cat, `isla-axiomatic` does not define a large set of built-in relations and sets. Instead, it adds *accessors*: point-free functions over events which can access the fields of the underlying Sail structures to allow the model author to define their own relations and sets based on the underlying ISA definitions.

For example, the Armv9-A accessor for barrier access types matches on the `Barrier` union we saw earlier, and if it is one of `Barrier_DMB` or `Barrier_DSB` it extracts the `.types` field from its `DxB` struct, and otherwise returns the default value for that type. The `isla-cat` definition of such an accessor is given below:

```
1  accessor barrier_types: MBReqTypes = .match {
2      Barrier_DMB => .types,
3      Barrier_DSB => .types,
4      _ => default
5  }
```

These accessors can be used in simple function declarations, using the `isla-cat` *define* command. For example, the Armv9-A model defines the F (*fence*) event type and the various Arm barrier event kinds (`dmbld`,`dmbsy`,...)

---

[1]https://github.com/rems-project/sail-arm/blob/interface-v9/arm-v9.3-a/src/interface.sail#L286
[2]https://github.com/rems-project/sail-arm/blob/interface-v9/arm-v9.3-a/src/stubs.sail#L105

with accessors. An extract of the `isla-cat` definition for Armv9-A[1], for the parts defining the `dmbld` event (which is the event set that includes all barrier events that are at least as strong as a `DMB.LD` instruction), is given below:

```
1   accessor F: bool = is sail_barrier
2
3   define has_barrier_type(ev: Event, t: MBReqTypes): bool =
4       (barrier_types(ev) == t)
5
6   accessor is_DxB: bool =
7       .match {
8           Barrier_DMB => true,
9           Barrier_DSB => true,
10          _ => false
11      }
12
13  accessor is_DMB: bool =
14      .match {
15          Barrier_DMB => true,
16          _ => false
17      }
18
19  define ArmBarrierRM(ev: Event): bool =
20      is_DxB(ev) & has_barrier_type(ev, MBReqTypes_Reads)
21
22  define DMB(ev: Event): bool =
23      F(ev) & is_DMB(ev)
24
25  define DMBLD(ev: Event): bool = DMB(ev) & ArmBarrierRM(ev)
26
27  define dmbld(ev: Event): bool =
28      (* see full code for definitions of dmbsy and dsbld *)
29      DMBLD(ev) | dmbsy(ev) | dsbld(ev)
```

---

[1]Full definition can be found at https://github.com/rems-project/system-semantics-arm-axiomatic-models/blob/main/models/armv9-interface/barriers.cat

Relaxed instruction fetching

These chapters are based, in part, on: ARMv8-A system semantics: instruction fetch in relaxed architectures [32] by Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell. Published in the proceedings of the 29th European Symposium on Programming (ESOP, 2020).

We now describe the main instruction fetch phenomena and architecture design questions for Arm-A. As usual, this will be done through the creation of handwritten litmus tests, which we will use to guide model design later on.

Chapter contents

3.1 Introduction

Self-modifying code is a software pattern relied on by nearly all software, but only explicitly managed by few: some are systems software, such as dynamic loaders, operating system kernels, and hypervisors; and some usermode, like just-in-time (JIT) compilers. This software is exactly the security-critical computing base, currently trusted but not trustworthy, that is especially in need of verification, and which will require a precise and well-validated definition of the architectural abstraction. While more esoteric software may make use of such patterns, potentially in ways otherwise envisaged by the architects, we will focus on the primary patterns used by those aforementioned kinds of software.

The semantics required for self-modifying code, of instruction fetch and cache maintenance, are areas where microarchitectural optimisations can have surprising programmer-visible effects, especially in the concurrent context. Previous work has scarcely touched on this: none of seL4 [52], CertiKOS [53, 54], Komodo [55], or the works of Guanciale et al. [56], or Baudmann et al. [57], address realistic architecture concurrency, and they use (at best) idealised models of the sequential systems architecture. The CakeML [58, 59] and CompCert [60] verified compilers target only sequential user-mode ISA fragments, without self-modifying code. Previous attempts at verification of self-modifying code have typically focused on MIPS or x86, such as in the works of Cai et al. and Myreen [61, 62]. However, those architectures have a very different programmer model than Arm presents, not requiring explicit instruction cache maintenance.

In the following four chapters we focus on instruction fetch and its required cache maintenance, for Arm-A. The ability to execute code that has previously been written to data memory is fundamental to computing: fine-grained self-modifying code is now rare, and (rightly) deprecated, but program loading, dynamic linking, JIT compilation, debugging, and OS configuration, all rely on executing code from data writes. However, because these are relatively infrequent operations, hardware designers have been able to optimise by partially separating the instruction and data paths, with distinct instruction caching, which by default may not be coherent with data accesses. This can introduce programmer-visible behaviour analogous to that of user-mode relaxed-memory concurrency, and require specific additional synchronisation to correctly pick up code modifications. Exactly what these are is not entirely clear in the current Arm-A architecture text.

We will clarify this situation, developing precise abstractions that bring the instruction-fetch part of Arm-A system behaviour into the domain of rigorous semantics. Arm have stated [private communication] that they intend to officially incorporate a version of this into their architecture.

We aim thereby to enable future work on system software verification using the techniques of programming languages research: program analysis, model-checking, program logics, and so on.

We begin (§3.2) by recalling the informal architectural guarantees that Arm provide, and the ways in which real-world software systems such as Linux, JavaScript, and WebAssembly change instruction memory. We then survey the fundamental phenomena and architecture design questions with a series of examples, and explore the interactions between instruction fetching, cache maintenance and the 'usual' relaxed memory stores and loads, showing that instruction fetches are more relaxed, and how even fundamental coherence guarantees for data memory do not apply to instruction fetches.

We give an operational semantics for Arm instruction fetch and icache maintenance (Ch. 4) in an abstract-microarchitectural style (following §2.3) capturing the architectural intent.

We give a more concise presentation of the model in an axiomatic style (Ch. 5), extending the "user-mode" axiomatic model from previous work (§2.4), and intended to be functionally equivalent to the presented operational semantics.

We validate all this (Ch. 6), in two ways: by the extensive discussion with Arm staff mentioned above, and by experimental testing of hardware behaviour, on a selection of Armv8-A cores designed by multiple vendors. We run tests on hardware with a mild extension of the Litmus tool [63, 64]. We make the operational model executable as a test oracle by integrating it into the RMEM tool and its web interface [49], introducing optimisations that make it possible to exhaustively execute the examples. We make the axiomatic model executable as a test oracle by extending our `isla-axiomatic` tool. We then compare hardware and the two models for the handwritten tests (modulo two tests not supported by the axiomatic checker), compare hardware and the operational model on a suite of 1456 tests, automatically generated with an extension of the diy tool [65], and check the operational and 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.

We focus on 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.

Caveats and Limitations Our operational semantics 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 [44, 10, 11, 66]; this is a matter of additional engineering and is future work. We only handle the 64-bit AArch64 part of Arm-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, while the equivalence between our operational and axiomatic models is validated experimentally, we do not have a formal proof of equivalence. A proof of this equivalence will be essential in the long term, but represents a major step and substantial work 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 [6], and we expect the addition of instruction fetch to require global changes to the argument.

3.2 Industry practice and the existing Arm 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).

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 typical 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, loop buffers, and caching of decoded micro-ops. 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'* [67, B2.4.4 (B2-114)]; instead, programmers must use explicit cache maintenance instructions[1] . 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.'*

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

At first sight, this may be entirely mysterious. This and the following chapters establish precise semantics for each of the above instructions, explaining why each is required. However, for now, a rough intuition for each is:

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 work, so this is equivalent to a DSB SY.

---

[1]Version J.a of the Arm architecture reference manual has a minor typographical error here.

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.

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` fields, described more in §3.14.

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.'* [68, p. 201]

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 [67, 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'*.

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 `dlopen`); 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.

3.3   Modifiable instructions

As was mentioned in §3.2, concurrent modification and execution is only permitted if the original and modified instructions are in a particular set: various branches, supervisor/hypervisor/secure-monitor calls, the `ISB` (instruction synchronisation) barrier, and NOP. Otherwise, the architecture permits *constrained unpredictable* behaviour, meaning that the resulting machine state could be anything that would be reachable by arbitrary instructions at the same exception level. The following W+F test (Figure 3.1) illustrates this.

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	

Figure 3.1: Code listing for test W+F.

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

Conditional branches   In version D.a (and earlier) of the Arm architecture reference manual, it made clear that, for branches with conditions (`B.cond`), the Arm architecture provided a specific non-single-copy-atomic fetch guarantee: that the execution will be consistent with either the old or new target, with either the old or new condition [69, B2-94]. In version E.a, this condition was removed entirely, meaning `B.cond` instructions were not permitted to be concurrently updated at all [70, B2-112]. In version G.b, `B.cond` was added to the list of concurrently-modifiable instructions, once more permitting replacement of (and with) a `B.cond` instruction [71, B2-130], with the stronger semantics that you will see either the old instruction or the new instruction entirely.

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
Final state: execute "B.NE g"	

Figure 3.2: Code listing for test W+F+branches.

For example, the W+F+branches test (Figure 3.2) overwrites a `B.EQ g` with a `B.NE h`. Under the D.a and earlier text, the result could be consistent with executing `B.NE g` or `B.EQ h` instead, and thus the test is allowed; under the E.a-G.a text, the test has CONSTRAINEDUNPREDICTABLE behaviour; under the G.b and later text, the test has well-defined behaviour but is forbidden.

To avoid this unfortunate confusion, and any possible constrained unpredictable behaviours due to it, our examples will be restricted to modifying only `NOP`s and *unconditional* branches.

Synchronising branches   Arm does not include any instruction synchronisation effects on the branch instruction, instead, the architecture relies on explicit synchronisation instructions (see §3.6). This is in contrast to other architectures such as x86 which does not require any explicit cache maintenance or pipeline flushing when jumping to newly-modified code.

3.4 Coherence

Data writes and reads are coherent, in Arm and in other major architectures: in any execution, for each address, the reads of each hardware thread must see a subsequence of the total *coherence order* of all writes to that address (see §2.1.2). The plain-data CoRR1 test (Figure 2.5, p18) illustrates one case of this: it is forbidden for a thread to read a new write of x and then the initial state for x. However, instruction fetches are not necessarily coherent: one instruction fetch may be inconsistent with a program-order-previous fetch, and the data and instruction streams can become out of sync with each other. We explore three new kinds of coherence:

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

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

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

These new kinds of coherence describe the relationship between the instruction 'stream' with the instruction and data caches.

3.4.1 Instruction-to-Instruction coherence

Arm explicitly do not guarantee any consistency between fetches of the same location: fetching an instruction does not mean that a later fetch of that same location will not see an older instruction [67, B2.4.4]. This is illustrated by the CoFF test (Figure 3.3), which is a variant of CoRR1 test (Figure 2.5, p18), but where the reads in the relaxed cycle of events are implicit reads caused by an instruction fetch.
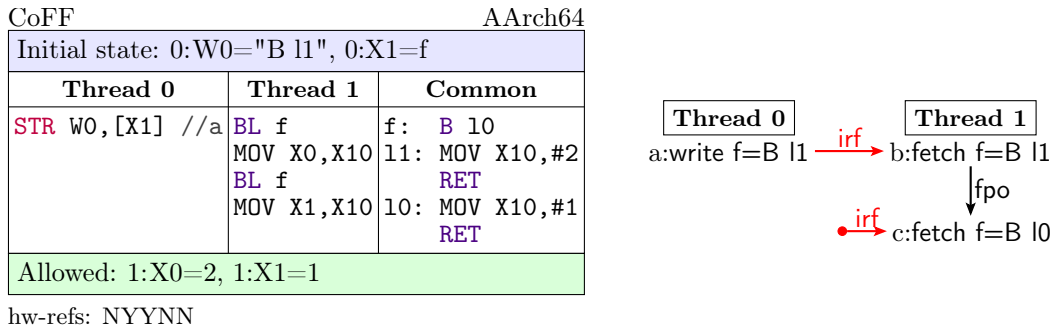
Figure 3.3: Code listing and execution diagram for CoFF.

Here, Thread 1 makes two calls to address f (recall BL is the branch-and-link 'call' instruction), while Thread 0 overwrites the instruction at that address with the opcode for the instruction B l1 (a branch to the location labelled l1). The interesting potential execution is that in which the first call to f fetches and executes the newly-written B l1, but the second call fetches and executes the original B l0. The execution shown in Figure 3.3 is the well-formed candidate execution consistent with the final state of the test. Candidate executions for self-modifying tests are similar to those of previous axiomatic models, but augmented with new fetch events, one per instruction, and new edges relating those events. We will discuss these new candidates in more detail in Ch5.

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, and/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).

As usual, edges from the initial state are shown as originating from a small circle.

Since we do not modify the code of most locations, or perform any cache maintenance operations over those locations, we usually omit the fetch events for the instructions at those locations; instead, showing only a subgraph of the interesting events, as in the CoFF execution diagram in Figure 3.3.

Here, and in future tests, we assume some common library 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.

For Arm, this execution is both architecturally allowed, and experimentally observed. This is shown in the test listing in Figure 3.3 in the line underneath the final state beginning with hw-refs. This line is a minature table, where each column represents one hardware device, and the value whether it was observed on that device (Y), not observed on that device (N), or whether there are no results for that device (indicated by -). Here is that final hw-refs line from CoFF (Figure 3.3), annotated with the names of the devices (see §6.2.3 for a more detailed discussion of the hardware testing):

nexus9	s905	h955-a53	h955-a57	openq820
N	Y	Y	N	N

3.4.2 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 (Figure 3.4, p40). Here, if Thread 1 happens to fetch the newly-written B l1 at f (in the 'Common' function code), then a data read of f cannot see 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"			

hw-refs: NNNNN
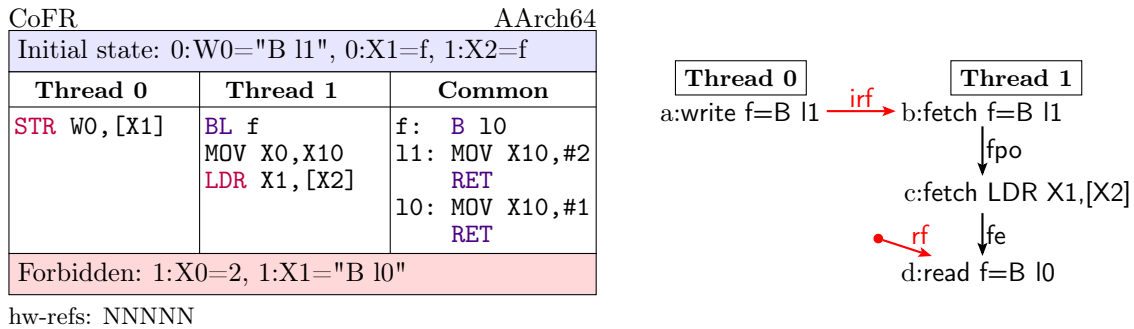
Figure 3.4: Code listing and execution diagram for CoFR.

This was not clear in the Arm prose specification at the time of writing [69, 72, 71, 68], but the architectural intent that emerged during discussion with Arm is that the given execution should be forbidden. This architectural decision was motivated by microarchitectural design: (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 the coherent data storage system, so the instruction cache cannot be ahead of the available data. This ensures that observing a write with an instruction fetch implies that all threads are now guaranteed to read from that write (or another coherence-after it).

3.4.3 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 (Figure 3.5).

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			

hw-refs: NYYYY
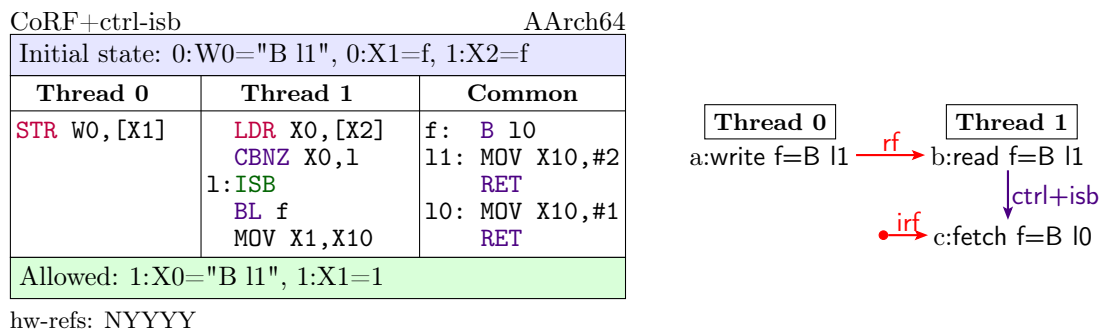
Figure 3.5: Code listing and execution diagram for CoRF+ctrl-isb.

Here Thread 1 has a control dependency (the CBNZ conditional branch, dependent on the value read by its load) and an instruction synchronisation barrier (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 prose at the time of writing [69, 72, 71, 68], 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.

3.5 Cross-thread synchronisation

We now consider modifying code that can be fetched by other threads, by considering variants of the standard message-passing shape (MP+pos (Figure 2.1, p14) and friends). 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, with some thread-local ordering on each thread ordering the writes on the left-hand thread, and ordering the read to the fetch on the right-hand side. We call this test MP.RF+dmb+ctrl-isb (Figure 3.6).

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	

hw-refs: -----
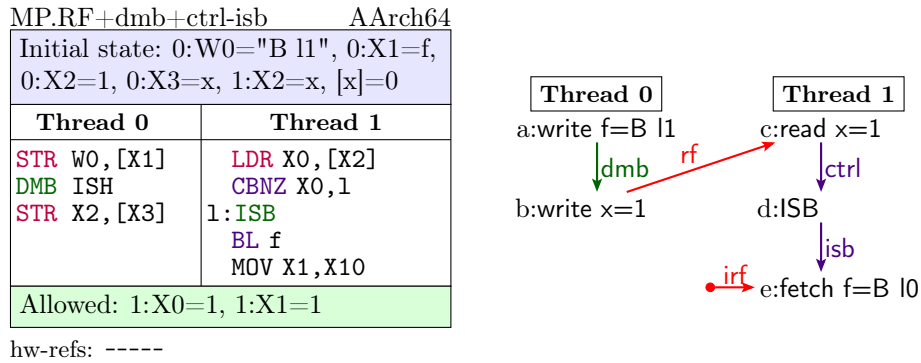
Figure 3.6: Code listing and execution diagram for MP.RF+dmb+ctrl-isb.

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 Arm-A. Thread 0 needs the entire cache synchronization sequence (described in §3.2) not just a DMB, to forbid this outcome. Adding that full cache synchronisation sequence gives test MP.RF+cachesync+ctrl-isb (Figure 3.10, p43), described in more detail in §3.6.2.

Synchronisation with memory by fetching   Another variant of this MP-shape test, where the message passing itself is done using modification of code, gives a much stronger guarantee. This can be seen in MP.FR+dmb+fpo-fe (Figure 3.7), in this test Thread 0 writes a message (to x) and then writes to the code concurrently being executed by Thread 1. If Thread 1 fetches the new instructions written by Thread 0, then Thread 1 must also see the new value of x.

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	

hw-refs: N-NNN
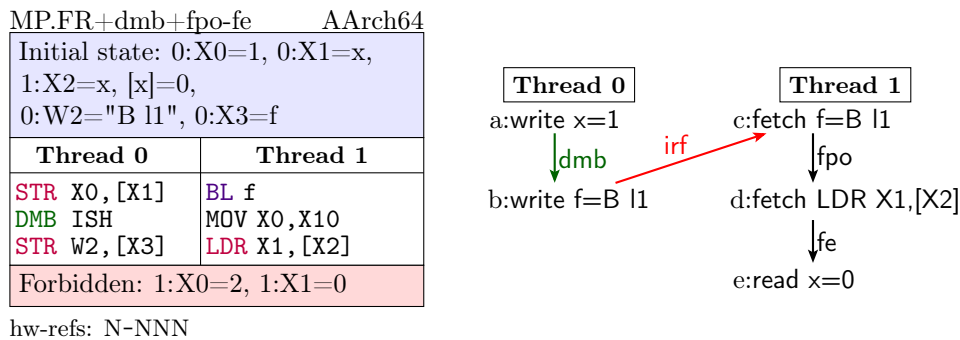
Figure 3.7: Code listing and execution diagram for MP.FR+dmb+fpo-fe.

This was not clear from the architectural prose at the time of writing, but this outcome is forbidden. This is for similar reasons as the previous CoFR test (Figure 3.4, p40): since Thread 1 fetched the updated value for f, the 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.

3.6 Cache maintenance

As we have seen, instruction fetches satisfy few guarantees, so explicit synchronisation must be performed when modifying the instruction stream.

Test SM (Figure 3.8) 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		

hw-refs: YYYYY
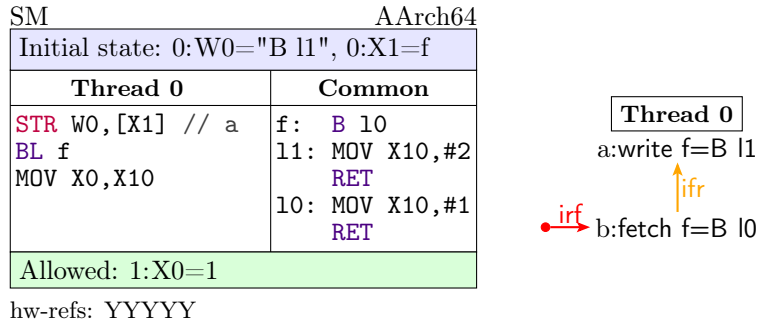
Figure 3.8: Code listing and execution diagram for SM.

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 $ifr = irf^{-1};co$. 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 [67, B2.4.4], and it is experimentally observed on all devices we have tested. Microarchitecturally, this is simply due to fetches from old instruction cache entries.

3.6.1 Synchronisation on a single thread

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	

hw-refs: NNNNN
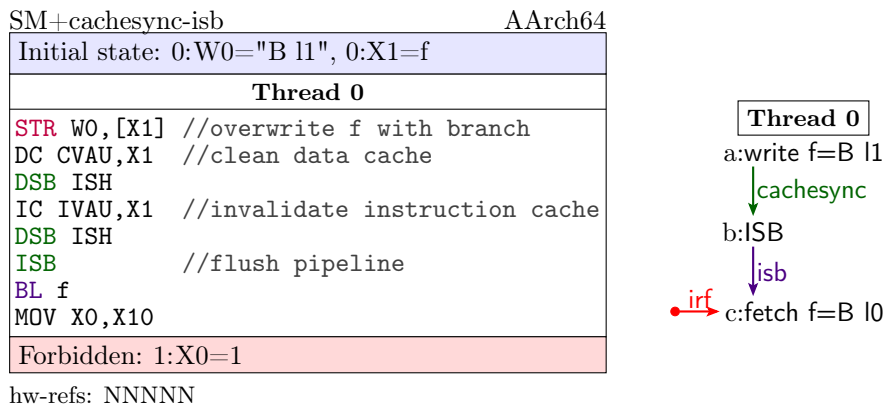
Figure 3.9: Code listing and execution diagram for SM+cachesync-isb.

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 sequence was in §3.2. Our §4.1 microarchitecturally-flavoured operational model will describe the semantics of this sequence using that microarchitectural intuition in

a way that gives precise and well-defined semantics to each instruction individually, such that their composition results in the correct system-wide synchronisation. This will be discussed in much more detail in Ch4.

3.6.2 Broadcast cache maintenance

The hardware thread writing new instructions and performing the necessary cache maintenance, need not be the same hardware thread as the one that will try fetch those instructions. 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.

The simplest example of this is in MP.RF+cachesync+ctrl-isb (Figure 3.10), where the 'producer' thread (Thread 0) writes the new instructions, and performs all the cache maintenance, before writing a flag informing the 'consumer' thread (Thread 1) that the instructions are ready to be fetched. Although the cache maintenance happened on a different thread to the one that will try fetch the new instructions, their effect is enforced system wide; the consumer needs only to flush the pipeline (with an ISB) to be guaranteed to see the new instructions.

MP.RF+cachesync+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]	LDR X0,[X2]
DC CVAU,X1	CBNZ X0,l
DSB ISH	l:ISB
IC IVAU,X1	BL f
DSB ISH	MOV X1,X10
STR X2,[X3]	
Forbidden: 1:X0=1, 1:X1=1	

hw-refs: NNNNY

Figure 3.10: Code listing and execution diagram for MP.RF+cachesync+ctrl-isb.

In-order fetches Test MP.FF+dmb+fpo (Figure 3.11) has both of Thread 0's writes be of new instructions. This idiom is quite common in practice; this was how Chrome's WebAssembly JIT synchronised its updates to modified code, up until the code was redesigned to use Arm's FEAT_BTI (branch-target-identification) feature [73, 74].

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]	BL f2
DMB ISH	MOV X0,X10
STR W2,[X3]	BL f1
	MOV X1,X10
Allowed: 1:X0=2, 1:X1=1	

hw-refs: NYYYY
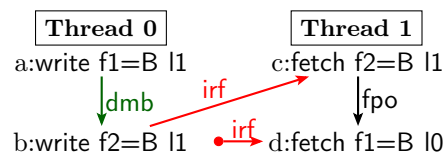
Figure 3.11: Code listing and execution diagram for MP.FF+dmb+fpo.

Without the full cache synchronisation sequence on Thread 0, this is allowed. Inserting that sequence gives MP.FF+cachesync+fpo (Figure 3.12, p44), a forbidden variant of the previous test.

MP.FF+cachesync+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]	BL f2
DC CVAU,X1	MOV X0,X10
DSB ISH	BL f1
IC IVAU,X1	MOV X1,X10
DSB ISH	
STR W2,[X3]	
Forbidden: 1:X0=2, 1:X1=1	

hw-refs: NNNNN
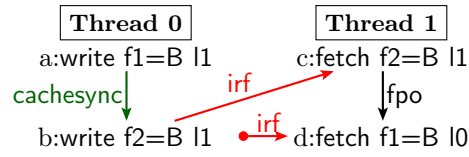
Figure 3.12: Code listing and execution diagram for MP.FF+cachesync+fpo.

At first, this may be surprising as there is no synchronisation on the right-hand side (Thread 1), but the architectural intent is for fetches to appear to be satisfied *in-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 prose at the time of the work, but, concurrent with this work, Arm were independently strengthening their definition to make it so.

Software thread migration The cache maintenance sequence need not be contiguous, it may be split up over many threads. This can be seen in the ISA2.F+dc-dmb+dsb-ic-dsb+ctrl-isb test (Figure 3.13), 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 and IC combinations with their surrounding barriers. The DC does not need a barrier preceding it, because it is ordered w.r.t. the preceding store to the same cache line.

ISA2.F+dc-dmb+dsb-ic-dsb+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]	LDR X0,[X1]	LDR X0,[X2]
DC CVAU, X1	DSB ISH	CBZ X0,l
DMB SY	IC IVAU, X4	l:ISB
STR X2,[X3]	DSB ISH	BL f
	STR X2,[X3]	MOV X1,X10
Forbidden: 1:X0=1, 1:X1=1		

hw-refs: --N-N
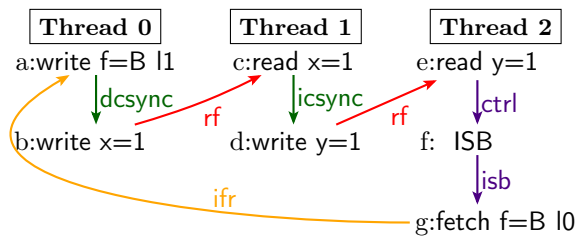
Figure 3.13: Code listing and execution diagram for ISA2.F+dc-dmb+dsb-ic-dsb+ctrl-isb.

This works because the IC IVAU is broadcast to all threads [67, B2.2.5p3]. Therefore the IC happening on a different thread to the DC does not break the sequence, so long as there is ordering between the IC and DC. Additionally, the DC need not happen on the same thread as the initial store, so long as the DC is ordered after the store.

The migration and context-switching code needs only contain a DSB and context-synchronisation (such as an ISB, although usually this is performed implicitly by the exception return mechanism itself) to ensure sufficient synchronisation exists for the sequence to be migrated at any point.

3.6.3 Completion of cache maintenance

Recall we have an asymmetry between the required synchronisation for DC instructions and IC instructions: IC instructions must have a preceding DSB to order with earlier accesses, whereas DC instructions do not necessarily need one; DC instructions are ordered by DMB with surrounding memory accesses, whereas IC is not.

This is because the DC is ordered much like a read (see §3.11.1). However, both the DC and IC are not guaranteed to have completed their effect until after the subsequent execution of a DSB instruction on the same thread [68, pp. 5790-5791], and an IC instruction always requires an DSB before it [68, p. 5791].

3.7 Dependencies

Reads, including implicit reads due to an instruction fetch, must have their address become known before the value can be used. This is a general principle Arm have, that values from reads generally cannot be speculated. For instruction fetches, this address is the program counter.

This means that computations which are used in the calculation of that address give rise to *dependencies* in the program. Sometimes these dependencies are hard and must be preserved, and other times, not.

3.7.1 Address dependencies

If the destination of a branch is passed as a register, with the BR (branch-register) or BLR (branch-and-link-register) instructions, then the instruction fetch of the target cannot go ahead until after the address is resolved.

This can be seen in the MP.RF+cachesync+addr test (Figure 3.14), where the target of the branch is dependent on the value of register X2 which comes from the earlier load of x.

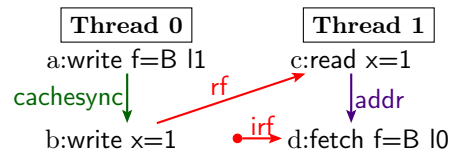| MP.RF+cachesync+addr	AArch64
Initial state: 0:W0="B l1", 0:X1=f, 0:X2=1, 0:X3=x, 1:X2=x, [x]=0	
Thread 0	**Thread 1**

STR W0,[X1]
DC CVAU,X1
DSB ISH
IC IVAU,X1
DSB ISH
STR X2,[X3]
	LDR X0,[X2]
	EOR X2,X0,X0
	ADD X2,X2,f
	BLR X2
	MOV X1,X10

Forbid?: 1:X0=1, 1:X1=1	

hw-refs: -----

Figure 3.14: Code listing and execution diagram for MP.RF+cachesync+addr.

3.7.2 Control dependencies

For branches where the destination is known, but where it is not yet known if the branch will be taken, then it is permitted for the instruction to be fetched and executed speculatively.

MP.RF+cachesync+ctrl   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]
 DC CVAU,X1
 DSB ISH
 IC IVAU,X1
 DSB ISH
 STR X2,[X3]
	LDR X0,[X2]
	 CBNZ X0,l
	 l:
	   BL f
	   MOV X1,X10

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

hw-refs: YYYYY

Figure 3.15: Code listing and execution diagram for MP.RF+cachesync+ctrl.

3.8 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, Armv9-A, and RISC-V are multi-copy atomic (or "other multi-copy-atomic" in Arm terminology) [7, 6, 67]: the programmer can assume there is a single shared memory, with all data-access relaxed-memory effects due to thread-local out-of-order execution.

One again has to ask whether writes are multi-copy atomic when observed by instruction fetches. However, the lack of any fetch atomicity guarantee for most instructions (§3.3), and the lack of coherent fetches for the others (§3.4), means the question of multi-copy atomicity for instruction fetching is not particularly interesting. Tests are either trivially forbidden (by data-to-instruction coherence, as in test WRC.F.RR+po+dmb (Figure 3.16)) or are allowed, but only the full cache synchronisation sequence provides enough guarantees to forbid it, and this sequence ensures all cores will share the same consistent view of memory.

WRC.F.RR+po+dmb   AArch64

Initial state: 0:W0="NOP", 0:X1=f, 1:X1=1, 1:X2=x, [x]=0, 2:X1=x, 2:X3=f		
Thread 0	**Thread 1**	**Thread 2**

STR W0,[X1]	BL f
	 MOV X0,X10
	 STR X1, [X2]
		LDR X0,[X1]
		 DMB SY
		 LDR X2,[X3]

Forbidden: 1:X0=1, 2:X0=2, 2:X2="B l0"		

hw-refs: -NN-N
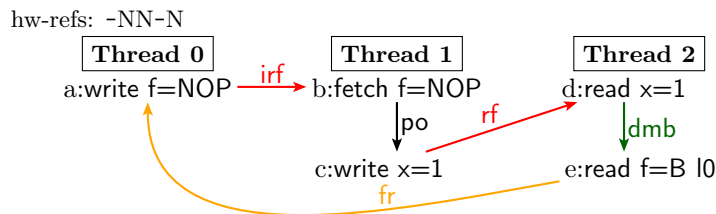
Figure 3.16: Code listing and execution diagram for WRC.F.RR+po+dmb.

3.9 More on instruction caches

Test CoFF (Figure 3.3, p39) showed that fetches can see "old" writes. In principle, there is no limit to the number of distinct values within the instruction cache: 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 MP.RFF+dc-dsb+ctrl-isb-isb test (Figure 3.17, p47) illustrates this, with Thread 0 writing two distinct new opcodes for g, and Thread 1 able to see all three (both of the new, and the initial) values for g.

Figure 3.17: Code listing and execution diagram for MP.RFF+dc-dsb+ctrl-isb-isb.

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.

3.10 Points of unification and coherence

Cleaning the data cache, using the DC instruction, forces a write to become visible to instruction fetch, but does restrict the set of values that could be in the instruction cache. It does this by pushing the write past the Point of Unification (the point where the instruction and data caches become unified). However, there may be multiple Points of Unification: one for each individual core, where its own instruction and data memory become unified, and one for the entire system (or shareability domain) where all the caches eventually 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 test SM.F+ic (Figure 3.18).

Figure 3.18: Code listing and execution diagram for SM.F+ic.

In SM.F+ic, Thread 0 modifies f, and Thread 1 fetches the new value and performs just an IC and DSB, before signalling Thread 0 which also fetches f. The IC (without its sibling DC) 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 at the time of writing, 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 [75]. 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 in theory, even in multi-copy atomic machines, although we have never observed it in practice (suggesting that modern machines either do not do DDI, at least before the Point of Unification, or that instruction fetches are not as weak as permitted).

With insufficient synchronisation of the data caches, there is theoretically no limit to how far back we can fetch from. Recall the MP.RF+dmb+ctrl-isb test (Figure 3.6, p41), it required the full cachesync sequence to forbid the 'bad' behaviour. Test FOW (Figure 3.19) is similar to that MP-shaped test, but writes two new values to the data consecutively rather than one, and has two threads reading the flag before fetching that address. Here, both threads can see the updated flag, but can execute different instructions on the instruction fetch of g, even after invalidating the instruction cache.

Figure 3.19: Code listing and execution diagram for FOW.

This is not clear in the existing architecture text at the time of writing. 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, before the *point of coherence*, 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.

In theory, once a write passes the Point of Coherency (the point where all data and unified caches eventually unify) then any writes coherence before that write cannot be seen at all by instruction fetches any more. While we do not set out to attempt to model this, as a general notion of a point of coherency is not required in the models as we do not model device memory or DMA, our operational model does capture it.

3.11 Cleans and invalidates are like reads and writes

Cache maintenance operations can generally be split into one of two kinds:

▷ Invalidations mark cache lines as *invalid*, meaning they can no longer be read from.

▷ Cleans force a write-back of a cache line further down the cache hierarchy.

For instruction cache maintenance only invalidation is provided, but for data cache maintenance the programmer can choose whether to do a clean, an invalidate, or both; and whether the maintenance takes effect to the Point of Unification or the Point of Coherency.

If one data cache operation is sufficient for the sequence, then a stronger one is also sufficient. All valid promotions from DC CVAU to stronger DC are given below:

Instruction	Clean/Invalidate	Target	Can promote from CVAU?
DC CVAU	Clean	PoU	–
DC IVAC	Invalidate	PoC	No
DC CVAC	Clean	PoC	Yes
DC CIVAC	Clean & Invalidate	PoC	Yes

3.11.1 Cleans are similar to reads

Microarchitecturally, cleans are non-destructive; they push the data further down the cache hierarchy, without causing the data to be lost. In hardware, these clean operations may be propagated around the system in much the same way reads are. This gives clean operations the same feel (memory ordering constraints) as data reads (and in some implementations, may be implemented in that way).

This means that DC CVAUs wait for program-order previous reads and writes (and other DCs) of the same location (or really, within the same cache line of minimum size, see §3.12), and do not require any other explicit barriers or dependencies between them. Cleans may be speculated, but otherwise respect dependencies and fences, even with respect to surrounding non-same-cache-line accesses.

3.11.2 IC invalidates are not quite like writes

Invalidations are destructive: data that was once visible is lost, potentially forever.

Invalidations behave somewhat like writes; they cannot be performed speculatively, and end up existing at some place within the global coherence order of that location: reads reading from writes from before the invalidation can see the value, but after invalidation, they cannot.

IC invalidations behave like this, with some extra details about in-order fetching (see test MP.FF+dmb+fpo (Figure 3.11, p43)), with one major exception: they do not respect dependencies or barriers other than DSB. This means that in practice every IC requires a DSB between it and any program-order earlier or later memory accesses, in order to synchronise with them.

3.11.3 DC and IC address speculation

Normal data load and store instructions (in Arm-A and in other relaxed architectures) respect *address dependencies*: reads cannot be satisfied, and writes cannot be forwarded from or committed, until their addresses are resolved from previous register writes (though those can still be out-of-order or speculative). In other words, the architecture forbids programmer-visible value speculation of such addresses.

The same question arises here for DC CVAU and IC IVAU, which are loosely analogous to loads and stores from the specified addresses. Test MP.R.RF+addr-cachesync+dmb+ctrl-isb (Figure 3.20, p50) illustrates this for DC. Thread 0 writes to g and performs the full cache synchronization sequence. However, the DC's address depends on a detour through Thread 1 which writes an even newer instruction to g. Since the address of the DC cannot be speculated, this address dependency must be preserved and so the final fetch of g after the cache synchronization must observe the branch Thread 1 wrote.

Figure 3.20: Code listing and execution diagram for MP.R.RF+addr-cachesync+dmb+ctrl-isb.

This is unclear in the current prose, but the architectural intent is that it should be forbidden: addresses of cache maintenance instructions should not be visibly value-speculated, and so these instructions must respect their address dependencies.

3.11.4 DC might be to same address

Data loads and stores can be ordered by the fact that they might access the same address [76, §12.5]. Arm made it clear in the architectural text that DC is ordered with respect to loads and stores with addresses in the same cache line, while IC is not [67, D4.4.8]. We therefore have to ask whether DC is subject to a might-access-same-address restriction in the same way as data loads and stores. The MP.RRF+dmb+addr-cachesync-isb test (Figure 3.21, p51) below illustrates this, with a case in which program-order previous load/store addresses may not be determined when the DC executes. The architectural intent (which was not clear from the architectural text at the time of writing) is that DC should be like loads in this respect too, with the aforementioned test architecturally allowed. Microarchitecturally, the DC is not required to wait for those addresses to be determined before executing, but if they end up being to the same address, the DC must be re-issued. Because the read d was not to the same location, the DC need not be re-issued and so may have happened before the write a to f.

MP.RRF+dmb+addr-cachesync-isb   AArch64

Initial state: 0:W0="B l1", 0:X1=f, 0:X2=1, 0:X3=x, [x]=0, 1:X1=x, 1:X4=z, [z]=0, 1:X5=f	
Thread 0	**Thread 1**
STR W0,[X1]	LDR X0,[X1]
DMB SY	EOR X2,X0,X0
STR X2,[X3]	LDR X3,[X4,X2]
	DC CVAU,X5
	DSB ISH
	IC IVAU,X5
	DSB ISH
	ISB
	BL f
	MOV X6,X10
Allowed: 1:X0=1, 1:X6=1	

hw-refs: --NN-
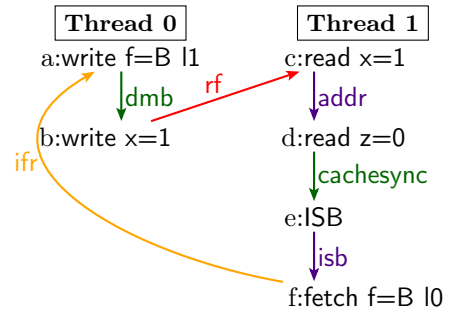
Figure 3.21: Code listing and execution diagram for MP.RRF+dmb+addr-cachesync-isb.

3.11.5 DC ordering with respect to other memory accesses

We saw that the DC instruction is ordered with program-order-previous stores to the same address. Normal 'data' loads are additionally ordered with respect to other same-location accesses in the same thread. Here we ask how far we can extend this to data cache maintenance operations.

po-previous loads   We assume that the DC has the same thread-local same-address ordering constraints as 'data' loads. For example, DCs are ordered with respect to program-order-earlier same-location loads as in CoRF+cachesync-isb (Figure 3.22), and may be re-ordered with respect to program-order-later same-location loads, as in MP+dmb+addr-dc (Figure 3.23, p52).

Note that these have not yet been clarified with architects; the stated results coming from our models.

CoRF+cachesync-isb   AArch64

Initial state: 0:W0="B l1", 0:X1=f 1:X1=f	
Thread 0	**Thread 1**
STR W0,[X1]	LDR W0,[X1]
	DC CVAU, [X1]
	DSB ISH
	IC IVAU, [X1]
	DSB ISH
	ISB
	BL f
Forbid?: 1:X2=1	

hw-refs: -----
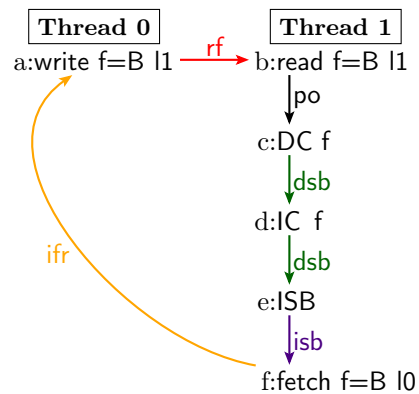
Figure 3.22: Code listing and execution diagram for CoRF+cachesync-isb.

```
MP+dmb+addr-dc                    AArch64
┌─────────────────────────────────────────┐
│ Initial state: 0:X0=1, 0:X1=x            │
│ 0:X2=1, 0:X3=y                           │
│ 1:X1=y, 1:X3=x                           │
├──────────────────────┬───────────────────┤
│      Thread 0        │     Thread 1       │
├──────────────────────┼───────────────────┤
│ STR X0,[X1] // a     │ LDR X0,[X1] // c   │
│ DSB SY               │ EOR X5,X5,X0       │
│ STR X2,[X3] // b     │ ADD X5,X5,X3       │
│                      │ DC CVAU,X5 // d    │
│                      │ LDR X2,[X3] // e   │
├──────────────────────┴───────────────────┤
│ Allow?: 1:X0=1, 1:X2=0                    │
└─────────────────────────────────────────┘
hw-refs: -----
```
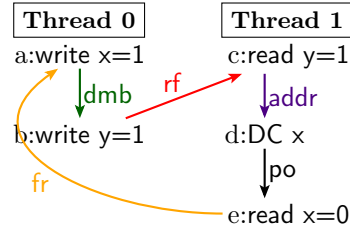

Figure 3.23: Code listing and execution diagram for MP+dmb+addr-dc.

3.12 Same-cache-line ordering

Arm-A has an architected *cache line of minimum size*. There are two cache lines of minimum size, one for the data caches and one for the instruction caches. They are accessible as the DMinLine and IMinLine bitfields of the CTR_EL0 register which encode log_2 the number of (32-bit) words in the smallest cache-line size[1], for the data and instruction caches, respectively.

Accesses being within the same cache line do not impose additional ordering constraints unless one of the accesses is a cache maintenance operation. For example, in the SB+scls test (Figure 3.24), a variation of the classic store buffering example but where the two locations are to the same cache line, the test is still allowed as the reads and writes of different locations (even within the same cache line) are not ordered. Note that in this test, X is an array of size $2^{2+\text{DMinLine}}$ bytes, and X is aligned on a cache boundary, therefore X and X+4 are 32-bit aligned addresses in the same (data) cache line of minimum size.

```
SB+scls                          AArch64
┌─────────────────────────────────────────┐
│ Initial state:                           │
│ uint32_t x[DMinLine];                    │
│ 0:X0=1, 0:X1=x, 0:X3=x                   │
│ 1:X0=1, 1:X1=x, 1:X3=x                   │
│ DMinLine1≥1                              │
├──────────────────────┬───────────────────┤
│      Thread 0        │     Thread 1       │
├──────────────────────┼───────────────────┤
│ STR W0,[X1]          │ STR W0,[X1,#4]     │
│ LDR W2,[X3,#4]       │ LDR W2,[X3]        │
├──────────────────────┴───────────────────┤
│ Allow?: 0:W2=0, 1:W2=0                    │
└─────────────────────────────────────────┘
hw-refs: -----
```
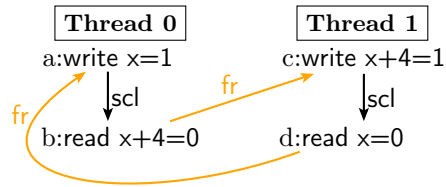

Figure 3.24: Code listing and execution diagram for SB+scls.

DC to same cache line   Given two locations f and g in the same cache line of minimum size, performing the cache clearing sequence for one will also clear the other.

---

[1] Note that, while the encoding allows DMinLine and IMinLine to be zero, this assignment does not make much sense for hardware, and it is likely no implementation exists with either less than the size of the largest implemented single-copy atomic access size.

Figure 3.25: Code listing and execution diagram for SM+sclcachesync-isb.

3.13 Mixed-size instruction fetching

In the tests so far we have always replaced a single instruction with another whole instruction, with a single write. However, it is easy to imagine code that replaces an instruction byte-by-byte, or perhaps even only replacing a single field in the instruction encoding.

It is clear that performing individual per-byte writes and then performing the full cache synchronization sequence, without concurrently attempting to fetch the location, should give the desired result without unpredictable behaviour.

For example, in the following SM8+sclcachesync-isb test (Figure 3.26, p54), a new 32-bit instruction is written byte-by-byte before performing a full cache synchronisation sequence on a single core. Here, it is not a *concurrent* modification of the location, as it is all on a single core and the sequence is complete before the fetch happens, and so the result is a well-defined forbidden outcome. This pattern does occur in practice, as code often gets loaded from some other memory by means of some memcpy call, before being executed.

Note that the 32-bit opcode for B l1 differs from that of B l0 only in the last byte (at f[0] since instructions are always stored little-endian in Arm-A), so deleting the final three STRB instructions (events b–d) from the test would not affect the result (it is still forbidden).

Figure 3.26: Code listing and execution diagram for SM8+sclcachesync-isb.

It is less clear in the architectural prose at the time of writing what happens if one were to *concurrently* modify part of an instruction, either in a single thread without sufficient synchronisation as in SM+mixed (Figure 3.27), or across multiple threads as in W+F+mixed (Figure 3.28, p55). We do not discuss this in detail, and the authors are not aware of any software patterns that rely on it. We leave this question open for the architects to resolve at a later time.

Figure 3.27: Code listing and execution diagram for SM+mixed.
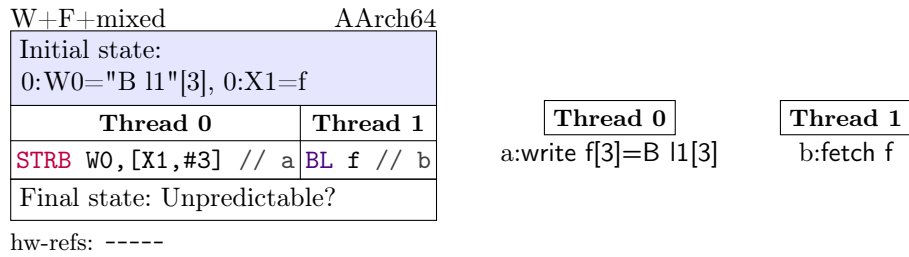
Figure 3.28: Code listing and execution diagram for W+F+mixed.

3.14 Cache type strengthening: IDC and DIC

Where implementations are stronger than the architecture guarantees by default, the architecture provides an additional identification mechanism: the CTR_EL0 register has two fields (named IDC and DIC) to identify where the implementation may have stronger semantics and thus fewer requirements for the use of cache maintenance instructions by software.

Ignoring these fields, and programming 'to the architecture' as set out in the previous sections, will always be safe. If implementations advertise (through the IDC and DIC fields) that particular cache maintenance operations are not required, then those cache maintenance instructions simply become hints or NOPs, and over-protective cleans and invalidations will not be harmful to the program.

As the names suggest, these fields are related to the kinds of coherence introduced in §3.4. IDC is related to instruction-to-data coherence, and requirements on data cache maintenance. DIC is related to data-to-instruction coherence, and the requirement for instruction cache maintenance.

3.14.1 IDC

When CTR_EL0.IDC is 1, the DC instruction is not required as part of the sequence [68, p. 201].

Point of Unification   When the DC instruction is not required, it means that writes must reach the Point-of-Unification before being propagated to other threads. This means, under IDC=1, the earlier SM.F+ic test (Figure 3.18, p47) is forbidden.

3.14.2 DIC

When CTR_EL0.DIC is 1, the IC instruction is not required as part of the sequence [68, p. 201].

In-order fetching   Recall from §3.5 that instruction fetches must either happen in-order, or the IC instruction must touch the internal fetch queues of the individual threads. When DIC=1, the IC instruction is not required, and this forces fetches to be satisfied from the instruction cache in the order they are fetched into the fetch queue. This is exactly how our operational model is implemented (which we shall see in Ch4).

3.15 Related Work

Explicit cache maintenance makes these tests, and the models presented in the next two chapters, quite different to the 'user mode' relaxed memory models discussed in Chapter 2.

Previous work on verification, of operating systems, hypervisors, and JITs, has had to work with idealised models of the underlying hardware.

Myreen's JIT compiler verification [62] models x86 icache behaviour with an abstract cache that can be arbitrarily updated, cleared on a `jmp`.

Cai, Shao, and Vaynberg produce a Hoare-style logic for certifying programs which contain self-modifying patterns [77], extending a version of *Concurrent Abstract Predicates* (CAP) [78] for generalised von-Neumann machines.

Goel et al's work on verification of x86 machine code programs [79, 80] includes a system step relation, based on their x86 instruction implementation models in ACL2. This model fetches instructions from memory, but avoids the complexity of caches and pipelines [81].

Lustig et al describe a framework for concurrent models, with relaxed behaviours, for machine-code x86 programs based on stages of hardware micro-operations [82]. They produce some models in this framework which include instruction fetching and the (data and TLB, not instruction) caches of a specific hardware implementation. These models explain behaviours seen based on knowledge of the underlying microarchitecture, but are not intended to be architectural models.

The verification of seL4 [52] included self-modifying patterns, but assumed the correctness of the required cache maintenance, without producing tight architectural models of the individual instructions.

CertiKOS [53, 54] verifies an assortment of safety and security properties (no code injection, no buffer overflows, no data races, and so on) for a custom-written kernel, with respect to an underlying concurrent (but not relaxed) x86 hardware machine model ('x86mc'), without self-modifying code.

SeKVM [83] similarly verified a custom-written (in this case, for Arm) micro-kernel, with respect to an underlying concurrent, and somewhat relaxed, hardware model. This model is far less idealised than those used in earlier verification efforts (but still not an architectural definition by any means), such as those in the seL4 and CertiKOS projects. The *KCore* kernel itself does not require self-modifying code, and the contextual refinement did not consider programs with concurrent or self-modifying code, and the underlying hardware model did not support data or instruction cache maintenance operations.

For architectural models which include cache maintenance, the closest is Raad et al.'s work on non-volatile memory. They model the required cache maintenance for persistent storage in ARMv8-A [84], as an extension to the ARMv8-A axiomatic model, and for Intel x86 [85] as an operational model.

There is also some work on address translation and TLB maintenance, which has a very similar flavour to cache maintenance. We explain the related work on TLBs in more detail in §8.10.

At the time of doing this work, Arm informally confirmed they would adopt the model [Private communication]. We are aware of independent work by Arm happening at the time of writing, extending the herdtools suite of tools, models, and tests, for instruction fetching and cache maintenance. This work has not yet been published, nor any documents describing the models or tests released. It is therefore difficult for now to say whether, and if so, in which ways, this new Arm model has evolved from what is presented here.

Chapter 4

Operational instruction fetching

4.1 An Operational Semantics for Instruction Fetch

Previous work on operational models for IBM Power and Arm 'user-mode' concurrency (see Chapter 2) has shown, perhaps surprisingly, that for the programmer-visible behaviours one can abstract from almost all hardware implementation details of the memory system (store queues, the cache hierarchy, the cache protocol, and so on). For Arm-A, following their 2018 shift to a multicopy-atomic architecture, one can do so completely: the Flat model has a shared flat memory, with a per-thread out-of-order thread subsystem. This out-of-order thread subsystem abstractly models pipeline effects which are alone sufficient to explain all the observable relaxed behaviours.

For instruction fetch, and the required cache maintenance, it is no longer possible to abstract completely from the data and instruction cache hierarchy. But, we can still abstract from some of its complexity. Flat has a fixed instruction memory, with a single transition which would fetch from that fixed instruction memory an instruction and decode it. This transition could be taken at any time, for any in-flight (non-finished) instruction, for any address of a potential (even speculative) program-order successor of that in-flight instruction. We now extend Flat by removing that fixed instruction memory, enabling instructions to be fetched from the flat memory, with values written by normal 'data' writes, along with adding the additional instruction-fetch related structures shown in Figure 4.1. We call this extended model *iFlat*. The remainder of this chapter will describe these new structures in detail, and enumerate the transitions of iFlat.
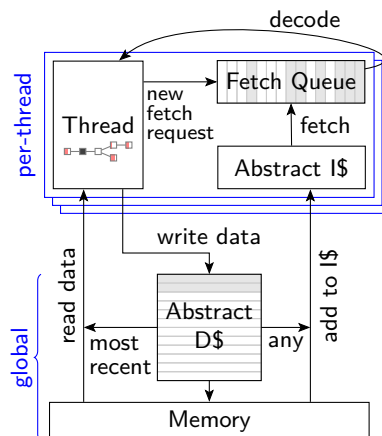
Figure 4.1: Structure of the iFlat state: per-thread fetch queues and instruction caches, with a global abstracted data cache.

4.2 The iFlat operational state

We extend the original Flat state with per-thread instruction caches and fetch queues with a global abstracted data cache, all of unbounded size, leaving the global flat memory unchanged.

4.2.1 Fetch queues (per-thread)

The per-thread fetch queues are ordered buffers of fetch request entries, waiting to be decoded and begin execution. Entries are either a fetched 32-bit opcode, or a yet unsatisfied ('*unfetched*') request.

Fetch queues allow the model to speculate and pre-fetch instructions ahead of where the thread is currently executing. Fetch requests are placed into the fetch queue in-order, entries are removed from the fetch queue to be decoded in-order, but the values may be satisfied out-of-order.

In this way the fetch queues abstract from multiple real-hardware structures: instruction queues, line-fill buffers, loop buffers, slots objects, and others. We believe the out-of-order satisfaction of instruction fetches are not observable on real hardware (in part due to the general lack of coherence in instruction caches subsuming this behaviour, see §3.5), and the model is equivalent to one that fetches in order, but this presentation of the model is more consistent with the description in the Arm reference manuals, and we believe has a closer correspondence with the underlying microarchitecture.

4.2.2 Abstract instruction caches (per-thread)

Each thread has an abstract instruction cache, which are just sets of writes.

When a new fetch request happens, and that request is added to the fetch queue, it will be satisfied from that thread's abstract instruction cache, either immediately if the instruction cache contains an entry for it (called a *hit*) or at some later point in time otherwise (a *miss*).

The instruction cache can contain many possible writes for each location (§3.9), and can be spontaneously updated with new writes in the system at any time ([67, B2.4.4]), or have entries spontaneously be dropped from the cache.

Unlike the flat memory, the instruction caches are not updated on a write. There is no guarantee values are ever dropped from the instruction cache, unless an explicit instruction cache maintenance operation is performed. Therefore, the instruction cache may contain values which are arbitrarily stale.

Instruction cache invalidation operations do not propagate atomically. Instead, an IC requests all threads invalidate their caches, and waits for them to have done so. To handle this, each thread keeps a list of addresses yet to be invalidated by any in-flight ICs.

4.2.3 Abstract data cache (global)

Before the single shared flat memory for the entire system, we insert a shared buffer (a list of writes) abstracting from the many possible coherent data cache hierarchies. Explicit reads (e.g. those from load instructions) must be coherent, reading from the most recent write to the same address in the buffer or memory. Instruction fetches may read from any write of the same location from the buffer or memory (§3.4).

On propagation, writes are initially placed into the abstract data cache buffer, before eventually flowing into the shared flat memory (in coherence order). In this model, the shared flat memory acts as the system-wide Point of Unification; writes before that point may or may not be seen by the threads, but once they reach the shared flat memory an instruction cache fill must see that write, or something coherence newer.

4.2.4 Outcome types

To link the model transitions to the execution of the instructions in the program the interface's outcome types (described in §2.2) must be extended to cope with the new instructions. Namely, we must add outcomes for the two cache maintenance operations, where instruction cache invalidation must be be further split in two due to its non-atomic nature (as discussed earlier). The full list of outcomes for the iFlat model can be found in Figure 4.2.

4.2.5 Pseudocode states

We add new pseudocode states, for the fetch queue states, and for pending IC instructions (again, as they do not happen atomically). Figure 4.3 lists all the pseudocode states in iFlat, with the new ones highlighted.

Read_mem(read_kind, address, size, read_continuation)	Read request
Perform_IC(address, res_continuation)	Propagate an `ic ivau`
Wait_IC(address, res_continuation)	Wait for an `ic ivau` to complete
Perform_DC(address, res_continuation)	Propagate a `dc cvau`
Write_ea(write_kind, address, size, next_state)	Write effective address
Write_memv(memory_value, write_continuation)	Write value
Barrier(barrier_kind, next_state)	Barrier
Read_reg(reg_name, read_continuation)	Register read request
Write_reg(reg_name, register_value, next_state)	Write register
Internal(next_state)	Pseudocode internal step
Done	End of pseudocode

Figure 4.2: iFlat outcomes (new outcomes highlighted in blue).

Plain(next_state)	Ready to make a pseudocode step
Unfetched(pc)	Placed into fetch queue but pending satisfaction of the fetch itself
Fetched(opcode)	Fetch satisfied but not yet begun pseudocode execution
Pending_mem_reads(read_cont)	Performing the read(s) from memory of a load
Pending_mem_writes(write_cont)	Performing the write(s) to memory of a store
Pending_IC(ic_cont)	Performing an `IC IVAU` to some address and waiting for the result

Figure 4.3: iFlat pseudocode states (new states highlighted in blue).

4.3 iFlat's transitions

This section is an extract from the full iFlat prose description document, which can be found in the appendix of our published ESOP'20 paper [32].

To accommodate instruction fetch and cache maintenance, we introduce the following 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
- ▷ Add to instruction cache for thread

In addition to these transitions, we modify some existing ones:

- ▷ Commit barrier
- ▷ Satisfy memory read by forwarding from writes
- ▷ Satisfy memory read from memory
- ▷ Commit store instruction
- ▷ Propagate memory write
- ▷ Complete store instruction (when its writes are all propagated)

Together, 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.

4.3.1 New transitions

Transitions for all instructions:

- ○ Fetch request: This transition (perhaps speculatively) requests the next address as a po-successor of a previous instruction.

1630   ▷ Fetch instruction: Satisfy the fetch request from the instruction cache.
1631   ○ Decode instruction: Decode the instruction.

1632 Cache maintenance instructions:

1633   ▷ Begin IC: Initiate instruction cache maintenance.
1634   ○ Propagate IC to thread: Do instruction cache maintenance for a specific thread.
1635   ▷ Perform DC: Clean the abstract data cache for a specific cache line.

1636 Instruction cache updates:

1637   ▷ Add to instruction cache for thread: Update instruction cache for thread with write.

Fetch request   For some instruction `i`, any possible next fetch address `loc` can be requested, adding it to the fetch queue, if:

1. it has not already been requested, i.e., none of the immediate successors of `i` in the thread's `instruction_tree` are from `loc`; and
2. either `i` is not decoded, or, if it has been, `loc` is a possible next fetch address for `i`:
   (a) for a non-branch/jump instruction, the successor instruction address (`i.program_loc+4`);
   (b) for a conditional branch, either the successor address or the branch target address[1]; or
   (c) for a jump to an address which is not yet determined, any address (this is approximated in our tool implementation, necessarily).

Note that this allows speculation past conditional branches and calculated jumps. Action: add an unfetched entry for `loc` to the fetch queue for `i`'s thread.

Fetch instruction   For any fetch-queue entry in the UNFETCHED state, its fetch can be satisfied from the instruction cache, from write-slices `ws`, if:

1. the write-slices (parts of writes) `ws` have the 4-byte footprint of the entry and can be constructed from a write in the instruction cache.

Action: change the fetch-queue entry's state to FETCHED(`ws`).

Fetch instruction (unpredictable)   For any fetch-queue entry in the UNFETCHED state, its fetch can be satisfied from the instructiion cache in a constrained-unpredictable way, if:

1. there exists a set of sets of write-slices, each of which can be constructed in the same way as above;
2. that set contains multiple values, and at least one of those values corresponds to an instruction that is not `B.cond` or one of {`B`, `BL`, `BRK`, `HVC`, `SMC`, `SVC`, `ISB`, `NOP`}, and they are not all `B.cond` instructions.

Action: record the fetch-queue entry as CONSTRAINED_UNPREDICTABLE. When this has reached decode and the corresponding point in the instruction tree becomes non-speculative, the entire thread state will become CONSTRAINED_UNPREDICTABLE.

Fetch instruction (B.cond)   For any fetch-queue entry in the UNFETCHED state, its fetch can be satisfied from the instruction cache, from write-slices `ws` and `ws'`, with value `ws''`, if:

1. there exists write-slices `ws` and `ws'`, each of which can be constructed in the same way as above;
2. `ws` and `ws'` correspond to the encoding of two conditional branch instructions `b` and `b'`;
3. the write-slices `ws''` can be constructed as the combination of `ws` and `ws'` such that `ws''` is the encoding of the branch instruction with `b`'s condition and `b'`'s target.

Action: record the fetch-queue entry as FETCHED(`ws''`).

Decode instruction   If the last entry in the fetch queue is in FETCHED(`ws`) state, it can be removed from the queue, decoded, and begin execution, if all po-previous `ISB` instructions in the instruction tree have finished. Action:

1. Construct a new instruction instance `i` with the correct instruction kind and state, for `i`'s program location, and add it to the instruction tree.

---

[1] In AArch64, all the conditional branch instructions have statically determined addresses.

2. Discard all speculative entries in the instruction tree that are successors of i that are now known to be incorrect speculations.

Begin IC   An instruction i (with unique instruction instance ID iiid) in state PERFORM_IC(address, state_cont) can begin performing the IC behaviour if all po-previous DSB ISH instructions have finished. Action:

1. For each thread tid' (including this one), add (iiid, address) to that thread's ic_writes;
2. Set the state of i to PROPAGATE_IC(address, state_cont).

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.

Complete IC   An instruction i (with ID iiid) in the state WAIT_IC(address, state_cont) can complete if there exists no entry for iiid in any thread's ic_writes. Action: set the state of i to PLAIN(state_cont).

Perform DC   An instruction i in the state PERFORM_DC(address, state_cont) can complete if all po-previous DMB ISH and DSB ISH instructions have finished. Action:

1. For the most recent write slices wss which are in the same data cache line of minimum size in the abstract data cache as address, update the memory with wss;
2. Remove all those writes from the abstract data cache.
3. Set the state of i to PLAIN(state_cont).

Add to instruction cache for thread   A thread tid's instruction cache can become spontaneously updated with a write w from the storage subsystem, if this write (as a complete slice) does not already exist in the instruction cache. Action: Add this write (as a complete slice) to thread tid's instruction cache.

4.3.2 Updated transitions

For those transitions which we update the guard or action, sometimes the change is minor but the full text of the transition is reproduced here, with the delta highlighted.

Commit barrier   A barrier instruction i in state PLAIN(next_state) where next_state is BARRIER(barrier_kind, next_state') can be committed if:

1. all po-previous conditional branch instructions are finished;
2. all po-previous dmb sy barriers are finished;
3. [ifetch] all po-previous dsb sy barriers are finished;
4. if i is an isb instruction, all po-previous memory access instructions have fully determined memory footprints; and
5. if i is a dmb sy instruction, all po-previous memory access instructions and barriers are finished;
6. [ifetch] if i is a dsb sy instruction, all po-previous memory access instructions, barriers and cache maintenance instructions have finished.

Note that this differs from the previous Flowing and POP models: there, barriers committed in program-order and potentially re-ordered in the storage subsystem. Here the thread subsystem is weakened to subsume the re-ordering of Flowing's (and POP's) storage subsystem.

Action:

1. update the state of i to PLAIN(next_state');
2. [ifetch] if i is an isb instruction, for all threads instruction tree's, for any instruction instance i in the FETCHED state, set it to the UNFETCHED state.

Satisfy memory read by forwarding from writes   For a load instruction instance i in state PENDING_MEM_READS(read_cont), and a read request, r in i.mem_reads that has unsatisfied slices, the read request can be partially or entirely satisfied by forwarding from unpropagated writes by store instruction instances that are po-before i, if the *read-request-condition* predicate holds. This is if:

1. [ifetch] all po-previous dsb sy instructions are finished;
2. all po-previous dmb sy and isb instructions are finished.

Let wss be the maximal set of unpropagated write slices from store instruction instances po-before i, that overlap with the unsatisfied slices of r, and which are not superseded by intervening stores that are either propagated or read from by this thread. That last condition requires, for each write slice ws in wss from instruction i':

▷ that there is no store instruction po-between i and i' with a write overlapping ws, and
▷ that there is no load instruction po-between i and i' that was satisfied from an overlapping write slice from a different thread.

Action:

1. update r to indicate that it was satisfied by wss; and
2. restart any speculative instructions which have violated coherence as a result of this, i.e., for every non-finished instruction i' that is a po-successor of i, and every read request r' of i' that was satisfied from wss', if there exists a write slice ws' in wss', and an overlapping write slice from a different write in wss, and ws' is not from an instruction that is a po-successor of i, or if i' was a data-cache maintenance by virtual address to a cache line that overlaps with any of the write slices in wss', restart i' and its data-flow dependents.

Satisfy memory read from memory   For a load instruction instance i in state PENDING_MEM_READS(read_cont), and a read request r in i.mem_reads, that has unsatisfied slices, the read request can be satisfied from memory.

If: the read-request-condition holds (see previous transition).

Action: let wss be the write slices from memory or the data cache network, whichever is newer, covering the unsatisfied slices of r, and apply the action of Satisfy memory read by forwarding from writes.

Note that Satisfy memory read by forwarding from writes might leave some slices of the read request unsatisfied. Satisfy memory read from memory, on the other hand, will always satisfy all the unsatisfied slices of the read request.

Commit store instruction   For an uncommitted store instruction i in state PENDING_MEM_WRITES(write_cont), i can commit if:

1. i has fully determined data (i.e., the register reads cannot change);
2. all po-previous conditional branch instructions are finished;
3. all po-previous dmb sy and isb instructions are finished;
4. [ifetch] all po-previous dsb sy instructions are finished;
5. all po-previous store instructionshave initiated and so have non-empty mem_writes;
6. all po-previous memory access instructions have a fully determined memory footprint; and
7. all po-previous load instructions have initiated and so have non-empty mem_reads.

Action: record i as committed.

Propagate memory write   For an instruction i in state PENDING_MEM_WRITES(write_cont), and an unpropagated write, w in i.mem_writes, the write can be propagated if:

1. all memory writes of po-previous store instructions that overlap w have already propagated
2. all read requests of po-previous load instructions that overlap with w have already been satisfied, and the load instruction is non-restartable ; and
3. all read requests satisfied by forwarding w are entirely satisfied.

Action:

1. restart any speculative instructions which have violated coherence as a result of this, i.e., for every non-finished instruction i' po-after i and every read request r' of i' that was satisfied from wss', if there exists a write slice ws' in wss' that overlaps with w and is not from w, and ws' is not from a po-successor of i, or if i' is a data-cache maintenance instruction to a cache line whose footprint overlaps with w, restart i' and its data-flow dependents;

2. record w as propagated; and

3. add w as a complete slice to the data cache network.

Complete store instruction (when its writes are all propagated)   A store instruction i in state PEND-ING_MEM_WRITES(write_cont), for which all the memory writes in i.mem_writes have been propagated, can be completed. Action: update the state of i to PLAIN(write_cont(true)).

4.3.3 Auxiliary definition – cache line of minimum size

Cache maintenance operations work over entire cache lines, not individual addresses (§3.12). 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 smallest possible cache line, with one for each of data and instruction caches. The CTR_EL0.{DMinLine, IMinLine} register fields describe the cache lines of minimum size for the data and instruction caches as $\log_2$ of the number of words in the cache line.

Caches lines are always aligned on their minimum size, and we define a write slice *overlapping* with a cache line if the footprint of the write slice overlaps with the $2^{2+\text{DMinLine}}$ (or $2^{2+\text{IMinLine}}$ for instruction cache lines) byte slice starting from the beginning of the aligned cache line region.

4.3.4 Handling cache type strengthenings

When CTR_EL0.DIC is 1, and therefore the IC instruction is not required, the following transitions are modified:

▷ Fetch instruction:
  – Instead of satisfying from the instruction cache, the request must be satisfied from composing combinations of writes from the abstract data cache buffer and flat memory.
  – Fetch requests may be only be satisfied if all po-previous in-flight fetch requests are also satisfied (no out-of-order satisfaction).
▷ Fetch instruction (unpredictable) (same modification as previous).
▷ Fetch instruction (B.cond) (same modification as previous).
▷ Begin IC:
  – Replace action with that of Complete IC.
▷ Add to instruction cache for thread (removed).

Together these effectively remove the instruction cache from the model, forcing in-order fetching, and satisfaction of fetch requests from memory (or the abstract data cache).

When CTR_EL0.IDC is 1, and therefore the DC instruction is not required, the following transitions are modified:

▷ Propagate memory write:
  – Update Action (3) to add w to the flat memory, instead of the abstract data cache buffer.

This effectively removes the abstract data cache buffer from the model, causing all writes to immediately reach the system-wide Point of Unification on propagation.

An axiomatic instruction fetch model

Based on the operational model, we develop an axiomatic semantics, as an extension of the Arm-A axiomatic model [50, 7] (described earlier in §2.4).

The existing axiomatic model is given as a predicate on *candidate executions*, hypothetical complete executions of the given program which satisfy some basic well-formedness conditions, defining the set of *valid* executions to be those satisfying its axioms.

We now extend this model, both extending the base events and candidate relations, as well as modifying the axioms over those events.

We will do this in a way that tries to retain the original model events, relations, and axioms, as unchanged as is reasonable to do so.

5.1 Candidates for self-modifying programs

We add new events:

> ▷ *instruction-fetch* (IF) events for each executed instruction, representing the read of the 32-bit opcode from memory.
> ▷ DC events, for the propagation of a DC CVAU instruction.
> ▷ IC events, for the propagation of a IC IVAU or IC IALLU instruction.
> ▷ DSB events for the data synchronization barrier instruction.

5.1.1 Program order

We keep program order (po) as just between the explicit memory events and barriers, only adding the cache operations (DC,IC) and the new barrier (DSB) to this set.

By adding an instruction fetch event, we now potentially have multiple events per instruction, as well as events for instructions with no associated explicit memory or barrier events at all. To keep track of the order of events within a single instruction, and between multiple instructions of the same thread, we add two new relations:

> ▷ *fetch-to-execute* (fe) which relates the instruction fetch (IF) event with the intra-instruction-ordered-later explicit memory or barrier (or cache op) event.

> ▷ *fetch-program-order* (fpo) relates each instruction-fetch (IF) event with all IF events of program-order later instructions.

We make fpo the fundamental candidate relation, and make po derived:

$$\text{po} = \text{fe}^{-1};\ \text{fpo+};\ \text{fe}$$

Figure 5.1 shows an example execution graph from a program with three instructions, a load, a mov, and a store, with the fpo and fe relations highlighted.
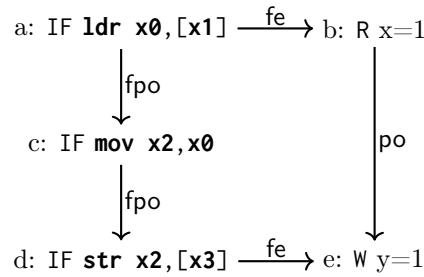
Figure 5.1: `fpo`,`fe` example, showing derived `po` relation from fundamental `fpo` and `fe` relations.

5.1.2 Same-location

We extend `loc` to relate same-address reads, writes, instruction fetches and IC/DC events.

Cache maintenance operations which affect all addresses, for example the `IC IALLU` instruction, are related to all memory and ifetch events.

Same-cache-line   Many of the operations now operate not over a single location but for an entire cache line. To handle these operations, we add to the candidate relations a *same-cache-line* relation, relating reads, writes, fetches, DC, and IC events to addresses in the same cache line of minimum size.

The DC and IC instructions operate over different cache line sizes. To handle this we split same-cache-line into two relations, `same-dcache-line` and `same-icache-line`, to relate events in the same data or instruction cache line of minimum size. Note that the same-icache-line and same-dcache-line relations also relate non-cache-op events.

We combine these relations to get a `scl` (same cache line), between memory (including ifetch) events and cache ops, where that memory event is to the same cache line, for that particular cache op:

```
1  scl0 = [DC]; same-dcache-line | [IC]; same-icache-line | [W]; loc
2  scl = scl0 | scl0 ⁻¹
```

5.1.3 Generalised Coherence

We add an acyclic, transitively closed, relation `wco`. This `wco` relation is a kind of *generalised-coherence-order*. It is an extension of `co`, with orderings for cache maintenance (DC and IC) events: it includes an ordering (e, e') or (e', e) for any cache maintenance event e and same-dcache-line event e' if e' is a write or another cache maintenance event.

Since `wco` relates events in the same cache line, and is transitively closed, it may end up relating writes that are not the same location. So `[a:W];wco;[b:W]` does not imply `[a:W];co;[b:W]` (although `co` does imply `wco`).

5.1.4 Dependencies

We extend the control dependency relation `ctrl` to include cache operations, but not instruction fetches. This ensures that `ctrl` remains a subset of `po`, and `[a]; ctrl; [b]; po; [c]` implies `[a]; ctrl; [c]`.

We extend `addr` to include cache operations: for (e, e') where e is a read and e' is a cache operation (DC or IC) whose address (cache line) is determined by the value read by e.

Since cache operations do not have any data associated with them, the `data` relation is left unchanged.

5.1.5 Reads-from

We add *instruction-read-from* (`irf`), which relates a write to any IF event that fetches from it, and *instruction-from-reads* (`ifr`), the analogue of `fr` for instruction fetches, relating a fetch to all writes coherence-after the one it fetched from:

$$\text{ifr} = \text{irf}^{-1}; \text{co}$$

```
1  include "cos.cat"
2  include "arm-common.cat"  (*5.2.1*)
3
4  (* might-be speculatively executed *)
5  let speculative =
6      ctrl
7    | addr; po
8
9  (* Fetch-ordered-before *)
10 let fob =
11     [IF]; fpo; [IF] (*5.2.4*)
12   | [IF]; fe (*5.2.4*)
13   | [ISB]; fe⁻¹; fpo (*5.2.5*)
14
15 (* Cache-op-ordered-before *)
16 let cob = (*5.2.8*)
17     [R|W]; (po & scl); [DC]
18   | [DC]; (po & scl); [DC]
19
20 (* DC synchronised required after a write *)
21 let dcsync =
22   if IDC
23     then id
24     else [W]; (wco & same-dcache-line); [DC]
25
26 (* IC sync required after a write or DC *)
27 let icsync =
28   if DIC
29     then id
30     else (
31       [W]; (wco & same-icache-line); [IC]
32     | [DC]; wco; [IC]
33     )
34
35 let cachesync =
36   dcsync; icsync
37
38 (* instruction synchronised ordered before
     *)
39 let isyncob = (*5.2.2*)
40     (ifr; cachesync) & scl⁻¹
```

```
1  (* observed by *)
2  let obs = rfe | fr | wco | irf
3
4  (* dependency-ordered-before *)
5  let dob =
6      addr | data
7    | speculative; [W]
8    | speculative; [ISB]
9    | (addr | data); rfi
10
11 (* atomic-ordered-before *)
12 let aob =
13     rmw
14   | [range(rmw)]; rfi; [A|Q]
15
16 (* barrier-ordered-before *)
17 let bob =
18     [R]; po; [dmbld]
19   | [W]; po; [dmbst]
20   | [dmbst]; po; [W]
21   | [dmbld]; po; [R|W]
22   | [L]; po; [A]
23   | [A|Q]; po; [R|W]
24   | [R|W]; po; [L]
25   | [F|C]; po; [dsbsy] (*5.2.6*)
26   | [dsb]; po (*5.2.6*)
27   | [dmbsy]; po; [DC] (*5.2.7*)
28
29 (* Ordered-before *)
30 let ob1 =
31     obs | dob | aob | bob
32   | fob | cob | isyncob
33 let ob = ob1⁺
34
35 (* Internal visibility
      requirement *)
36 acyclic po-loc | fr | co | rf as
       internal
37
38 (* External visibility
      requirement *)
39 irreflexive ob as external
40
41 (* Atomic *)
42 empty rmw & (fre; coe) as atomic
```

Figure 5.2: Ifetch Axiomatic model

5.2 Axioms and auxiliary relations

We now make the following changes and additions to the model. The full model is shown in Figure 5.2, with comments referring to the items in the following explanation.

5.2.1 Arm interface

The arm-common.cat file contains definitions for all the base event and relation sets, as well as some Arm-specific ones. Figure 5.3 lists the events and relations defined by arm-common.cat, we elide the full isla-cat definition of these relations here.

Events		Relations	
R	Reads	po,fpo	program-order and fetch-program-order
IF	Instruction-fetch	id,loc	identity and same-location
W	Writes	fe	fetch-to-execute
M	All explicit memory events (R | W)	po-loc	program-order same-location (po & loc)
A	Read-acquire	addr,ctrl,data	dependencies
L	Write-release	wco,irf,rf	existentially-quantified (candidate) relations
Q	Weak read-acquire	rfe,rfi	rf-external (rf & ext), rf-internal (rf & ~ext)
F	All fences (barriers)	coe,coi	co-external, co-internal
C	All cache-ops (DC | IC)	co	coherence-order ([W];wco&loc;[W])
DC	Data cache clean	ifr	instruction-from-reads (irf^{-1};co)
IC	Instruction cache invalidate	rmw	read-modify-write
ISB	Instruction sychronisation barrier		
dmbXY	Barrier, with strength at least XY={st,ld,sy}		
dsbXY	DSB Barrier, with strength at least XY={st,ld,sy}		
		scl	same-cache-line
	same-dcache-line,same-icache-line	same data/instruction cache line

Variants	
DIC,IDC	Boolean flags for CTR_EL0.{DIC,IDC} identity

Figure 5.3: Arm interface for ifetch. New events and relations highlighted in blue.

5.2.2 Cache maintenance

We define the relation isyncob, relating some instruction fetch f, in the most general case, to an IC which completes a cache synchronisation sequence (not necessarily on a single thread) which affects the location written to. Precisely: f reads-from a write w_0 which was coherence before some other write w, and w is wco-followed by a DC event d to some same-dcache-line address A_{dc}, which is in turn wco-followed by an IC event i to some address A_{ic} which was same-icache-line as the original f. This general isyncob shape is shown in Figure 5.4. In operational model terms, this captures traces that propagated w to memory, then subsequently performed a same-cache-line DC, and then a began an IC (and eagerly propagated the IC to all threads). In any state after this sequence it is guaranteed that w, or a coherence-newer same-address write, is in the instruction cache of all threads: performing the DC has cleared the abstract data cache of writes to x, and the subsequent IC has removed old instructions for location x from the instruction caches, so that any subsequent updates to the instruction caches have been with w, or co-newer writes. Therefore, the fetch f must have happened before the IC had completed, otherwise it would have been required to have read from w or something coherence after it.
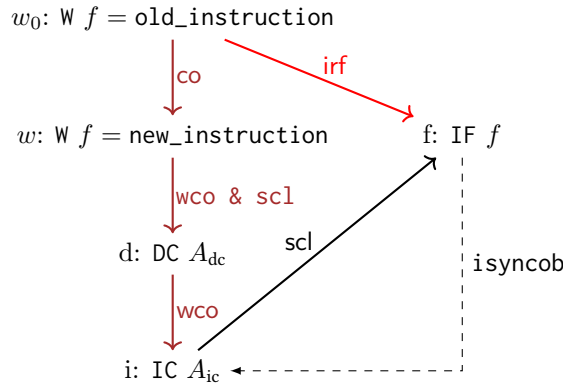
Figure 5.4: General isyncob shape.

This corresponds to the operational model in the following way: because w_0 was coherence-before w, w_0 was propagated before w was propagated in the trace, and because w was wco-earlier than the cache synchronisation

sequence, w was propagated before any of the cache maintenance transitions in the trace. If the fetch transition corresponding to f were to satisfy its fetch in a subsequent state, it would be guaranteed that w (or a coherence-newer write) would be in the instruction cache, and i would not be able to fetch from w. Hence, f must have happened before the IC completing the cache synchronisation sequence.

Cache type strengthening If the IDC or DIC variants are set, then either the DC or IC instruction is not required. This affects the isyncob in the following way:

> ▷ If DIC, then the IC instruction is not required, and therefore f must be ordered before the propagation of the DC, see Figure 5.5 (top left).
> ▷ If IDC, then the DC instruction is not required, and therefore f must be ordered before the propagation of the IC, without the need of an intervening DC, see Figure 5.5 (top right).
> ▷ If both, then f must be ordered before any coherence-later same-location write than w_0, as in Figure 5.5 (below).
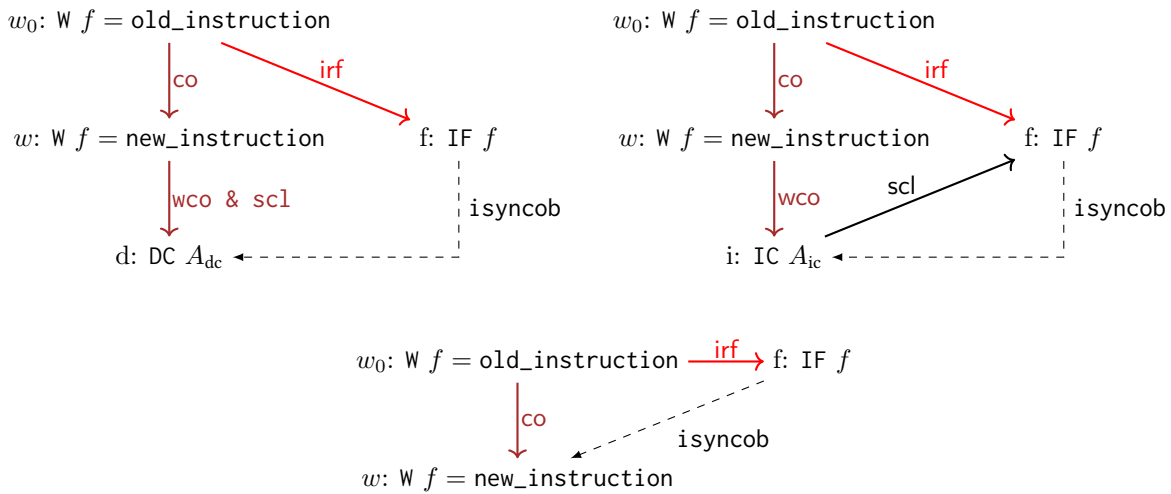
Figure 5.5: Modified isyncob shape, for variants DIC (above left), IDC (above right), and both (below).

To achieve this, the isyncob relation is split into two:

> ▷ dcsync, which broadly captures the 'data cache' requirements. Either from a write to a same cache line DC if not IDC, otherwise, from a write to itself, capturing that with IDC that a write is past the PoU the moment it has propagated.

> ▷ icsync, which captures the 'instruction cache' requirements. Either from a DC (or same-icache-line write), to a wco-later IC, or, if DIC, back to the DC or write itself.

The sequential composition of these two relations (called cachesync) captures the synchronisation required from a write to the point sufficient cache maintenance has been performed to ensure a same-cache-line instruction fetch would see it.

5.2.3 Coherence

The original model includes co in obs; we instead include the relation wco. Including wco in ordered-before corresponds to the intuition that wco records the ordering of the Propagate memory write, Begin IC (and eagerly taking all Propagate IC to thread transitions), and Perform DC transitions in the matching trace.

We also include irf in obs: informally, for an instruction to be fetched from a write, the write has to have been done before. Correspondingly, in the operational model, a write has to have been propagated before it can satisfy fetches in the storage subsystem.

5.2.4 Program order

We add a relation *fetch-ordered-before* (fob), which is included in *ordered-before*.

The relation fob includes fpo, informally requiring fetches to be ordered according to their order in the control-flow unfolding of the execution. Correspondingly in the operational model: fetch requests for instructions within the same thread appear to be satisfied in program order.

We also include the fe *fetch-to-execute* relation in fob, capturing the idea that an instruction must be fetched before it can execute. In the operational model, a read can only satisfy/a write can only propagate/a barrier can only commit/etc. after its instruction's fetch is satisfied.

5.2.5 Instruction synchronisation (ISB)

We include the edge $[\text{ISB}];\text{fe}^{-1};\text{fpo}$ in fob, ordering the fetch of any instruction program-order-succeeding an ISB instruction after the ISB event.

Operationally, a decoded ISB instruction prevents any program-order-later instructions from being removed from the fetch queue and decoded, and when an ISB is executed, it returns all entries in this thread's fetch queue (so any program-order-later instructions) to the UNFETCHED state, forcing the satisfaction of the instruction fetch for those instructions to happen after the ISB completes.

The rule $[\text{ISB}];\text{po};[\text{R}]$ in dob is no longer required, as the combination of rules in fob (in particular $[\text{ISB}];\text{fe}^{-1};$ fpo and $[\text{IF}];\text{fe}$) subsume it.

5.2.6 Data synchronisation (DSB)

For DSB ISH instructions we include po to and from DSB in bob.

We do this in three ways: (1) by extending the barrier hierarchy relations dmbst and dmbld to cover the memory barrier effects of a DSB; (2) by adding $[\text{F}|\text{C}];\text{po};[\text{dsbsy}]$ to capture DSBs waiting for the completion of fences and cache-ops, when using DSBs affecting both reads and writes; and (3) by adding $[\text{dsb}];\text{po}$ to capture the remaining completion fence properties that program-order later events cannot go ahead until the DSB is complete.

Importantly, DSB events do not order IF (ifetch) events in either direction.

5.2.7 Data cache maintenance (DC) is ordered like a read

Barrier-ordered-before (bob) also includes the relation $[\text{dmbsy}];\text{po};[\text{DC}]$, ordering DC events after program-order-preceding DMB SYs. Correspondingly, in the operational model, a DC can only be performed when all preceding DMB SY are finished.

5.2.8 Cache maintenance operations and cache lines

We include the relation *cache-op-ordered-before* (cob) in ob. This relation contains the edge $[\text{R}|\text{W}];(\text{po\&scl});[\text{DC}]$, ordering DC events after program-order-preceding same-dcache-line read and write events.

Operationally, a DC will be restarted by a program-order-preceding same-cache-line load if it was performed before the load was satisfied, and by a program-order-preceding same-cache-line store if it was performed before the store propagated its write.

Moreover, cob contains the edge $[\text{DC}];(\text{po\&scl});[\text{DC}]$, ordering two same-cache-line, same-thread DC events in program-order. In the operational model, a DC can only be performed when program-order-preceding same-cache-line DC instructions have been performed.

5.2.9 Constrained Unpredictable

We do not give precise semantics to programs that exhibit constrained unpredictable behaviour. Instead, we add a mechanism to flag such programs.

```
1  (* include base ifetch model *)
2  include "aarch64_ifetch.cat"
3
4  (* could-fetch-from *)
5  let cff =
6    ([W]; loc; [IF])
7      \ ob^{-1}
8      \ (isyncob^{-1}; ob)
9
10 (* cmodx(opcode) is True
11  * if it is in the list of
      concurrently modifiable
      instructions
12  *)
13 define cmodx(v: bits(32)): bool =
14     ...
```

```
1  define cff_bad(
2      ev1: Event,
3      ev2: Event,
4      ev3: Event
5    ): bool =
6      W(ev1) & IF(ev2) & W(ev3)
7      & ~(ev1 == ev3)
8      & cff(ev1, ev2) & cff(ev3, ev2
       )
9      & (~cmodx(ev1.value)
10       |~cmodx(ev3.value))
11
12 (* assert CU *)
13 assert exists
14     ev1: Event,
15     ev2: Event,
16     ev3: Event
17   =>
18     cff_bad(ev1, ev2, ev3) :named
      CU
```

Figure 5.6: Constrained unpredictable check model (ifetch).

We do this through the definition of an auxiliary *could-fetch-from* (cff) relation, capturing, for each fetch i, the writes it could have fetched from (including the one it did fetch from), as the set of same-address writes that are not ordered-after i, and which are not overwritten by coherence-newer writes that were followed by a cachesync sequence ordered-before i. Operationally, this captures writes that could have been in the instruction cache of i's thread: writes that did not happen *after* i in the trace, and excluding writes cleared by earlier cache synchronisation sequences.

We then add an axiom, asserting the existence of a bad pair of writes (w_1, w_2) which i could have fetched from, where at least one of w_1 and w_2 are not in the list of concurrently-modifiable instructions (as described in §3.2). We identify these (i, w_1, w_2) triples with a ternary relation (cff_bad(w1,i,w2)), whose non-emptiness implies the existence of such a triple. This gives us an extended 'checker' model, where executions which are allowed in the checker model, are also allowed in the original ifetch model, but also exhibit constrained unpredictable behaviour, and so the test should be flagged and any results discarded.

Chapter 6

Validating the ifetch models

We gain confidence in the models presented in the previous chapters by validating those models against the Arm architectural intent, against each other, and against a selection of real hardware.

6.1 The models correctly captures the architectural intent

This property is an important one, but not one that can be objectively demonstrated.

We ensure that the models do reflect the architecture, to the best of our understanding, by engaging in detailed and robust discussions with Arm architects (including their chief architect), as well as the microarchitects involved in the design of individual processors.

This process is an iterative one, where we produce litmus tests, discuss whether they are allowed or forbidden (and by which mechanisms), build models that capture those described mechanisms, and produce more litmus tests that show edge cases or interactions. This process is not obviously terminating, but usually results in reaching a natural fixed point for a core set of architectural features.

The structure of the operational model presented in Ch4 is derived from our discussions with Arm, it carefully includes structures which capture the behaviour they described, and has limitations where the architects decided no reasonable hardware could explore.

The axiomatic model, presented in Ch5, is also a product of the discussions with Arm, who have tentatively accepted it.

6.2 Correspondence between the models

We experimentally test the equivalence of the operational and axiomatic models by making executable-as-a-test-oracle versions of both and running each on a suite of hand-written, and automatically generated, litmus tests, checking that both give the same answer in all cases.

To automatically generate families of interesting instruction-fetch tests, we extended the diy test generation tool [65] 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, (but no data), 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.

6.2.1 Making the operational model executable as a test oracle

To make the operational model presented in Ch4 executable, 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) [7] 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: Propagate IC to thread, Complete IC, Fetch request, and Add to instruction cache for thread.

Eagerly taking Add to instruction cache for thread is ok, as this always increases the visible behaviours: adding a write to an instruction cache does not hide writes that were visible before. Complete IC and Fetch request are also safe to take eagerly, as these advance thread-local state making further transitions available without preventing any others.

Taking Propagate IC to thread eagerly is more subtle, this transition updates the state of another thread and potentially removes transitions it had available to it. If we take an arbitrary trace, containing a propagation of an IC to some thread, then it is safe (by the aforementioned logic) to immediately fill that icache back in. If we have a trace with two IC propagations, to separate threads, from the same instruction, with propagations of writes and DCs in between, then we know that the second Propagate IC to thread must have been an available transition when taking those write and DC propagation transitions, and therefore there must have been another trace where those write and DC propagations happened *after* the second IC propagation, and where the icache is filled immediately after each of those writes.

```
    …
    Propagate IC to X on Thread 1
      Write to X
       Propagate DC to X
        Write to X
         Propagate IC to X on Thread 2
            …

⇒
    …
    Propagate IC to X on Thread 1
      Propagate IC to X on Thread 2
       Write to X
        Eagerly fill icache
         Propagate DC to X
          Write to X
           Eagerly fill icache
              …
```

This new trace groups the propagation of the instruction cache invalidations together as early as possible, maximising the visible behaviour. Therefore, it is safe to always perform all the icache invalidates at once, atomically.

6.2.2 Making the axiomatic model executable as a test oracle

The axiomatic model is given in the `isla-cat` memory modelling language (see §2.4.2).

As `isla-axiomatic` already executes a fetch-decode-execute loop there is little work required, but to simply create fetch events for each read outcome during the fetch part of the cycle.

This is sufficient for making the test executable, but to make the exhaustive enumeration tractable we must make one further optimisation. Allowing the *fetch* part of the loop to be totally symbolic (in location and opcode) would lead to far too many candidate executions, even if the vast majority of them would be dismissed quickly (with trivially unsatisfiable `irf` constraints) they would still take up computing time to generate, and then discard, them. To avoid this, we instead require the user to provide the program-counter addresses and the sets of opcodes those locations' values can be. This ensures that while generating candidates we only need to generate those that actually contain the control-flow and instruction opcodes that are interesting for the test.

Figure 6.1 contains the `isla-axiomatic`-compatible sources for the earlier SM.F+ic test (Figure 3.18, p47) as an example. Lines 7-13[1] define the self-modifiable locations used in the test (for this test that is only label 'f:'), and the fully-concrete opcodes those locations may be; recall that all `isla` traces are a single control-flow path with fully concrete opcodes for each instruction.

6.2.3 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.

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

We experimentally test the equivalence of the operational and axiomatic models on the hand-written and the 1456 diy-generated tests, checking that the models give the same sets of allowed final states.

6.3 Validating against hardware

To run instruction-fetch tests on hardware, we extended the litmus tool [64]. 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, which are, in effect, C functions whose contents 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) 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.

We then ran the diy-generated test suite on a range of hardware implementations, to collect a substantial sample of actual hardware behaviour, checking that the extant hardware does not allow behaviours forbidden by either model.

6.3.1 Results from hardware

Then, for the key hand-written tests described in Ch. 3, together with some others (that have also been discussed with Arm, but not included here), we ran them on various hardware implementations.

Our testing revealed a hardware bug in a Snapdragon 820 (4 Qualcomm Kryo cores). MP.RF+cachesync+ctrl-isb test (Figure 3.10, p43) exhibited an illegal outcome in 84/1.1G runs (not shown in the table), which we have reported. We have also seen an anomaly for MP.FF+cachesync+fpo (Figure 3.12, p44), on an Arm-designed core, although this core had (in previous work) **TODO: ... No reference to cite?** been discovered to suffer a read/read coherence violation. Apart from these, the hardware observations are all allowed by the models. As usual, specific hardware implementations are sometimes stronger, and there are a number of tests which we did not observe on any hardware despite the architecture allowing them.

Finally, we ran the 1456 new instruction-fetch diy tests on a variety of hardware, for around 10M iterations each. The models are sound with respect to the observed hardware behaviour, except for that same Snapdragon 820 device with known coherence violations.

---

[1]Note the use of the array-of-tables feature of TOML here, which allows the user to specify multiple `[[self_modify]]` blocks if they wish [https://toml.io/en/v1.0.0#array-of-tables].

```
1   arch = "AArch64"
2   name = "SM.F+ic"
3   hash = "de102a920be43ce10482e59700a7c976"
4   stable = "X10"
5   symbolic = ["x"]
6
7   [[self_modify]]
8   address = "f:"
9   bytes = 4
10  values = [
11    "0x14000001",
12    "0x14000003"
13  ]
14
15  [thread.0]
16  init = { X3 = "x", X4 = "f:", X0 = "0x14000001" }
17  code = """
18          STR W0,[X4]
19          LDR W2,[X3]
20          CBZ W2, l
21  l:
22          ISB
23          BL f
24          MOV W1,W10
25          B Lout
26  f:
27          B l0
28  l1:
29          MOV W10,#2
30          RET
31  l0:
32          MOV W10,#1
33          RET
34  Lout:
35  """
36
37  [thread.1]
38  init = { X3 = "x", X2 = "1", X1 = "f:" }
39  code = """
40          BLR X1
41          MOV W0,W10
42          IC IVAU, X1
43          DSB SY
44          STR W2,[X3]
45  """
46
47  [final]
48  expect = "sat"
49  assertion = "1:X0 = 2 & 0:X2 = 1 & 0:X1 = 1"
```

Figure 6.1: Test SM.F+ic isla-axiomatic compatible version.

Test	Arch. Intent	H/W 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	[U]0/12.9G
FOW	allow	[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.]

Pagetables and the VMSA

These chapters are based, in part, on: Chapter D5 of the Arm Architecture Reference Manual DDI 0487H.a; and, Relaxed virtual memory in Armv8-A [34] 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); and, finally, give an overview of the tooling and validation of the presented models (§10).

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), and the walk performed by hardware (§7.4); an overview of the multiple stages of translation (§7.5), and the different translation regimes (§7.6); a detailed explanation of the official Arm translation table walk pseudocode (§7.7); and finally a discussion on the existence and purpose of translation lookaside buffers (§7.8).

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* (the *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 [12].

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:

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

Note that this is a simplification. See The Arm translation table walk (§7.4) for a more detailed description of the access permissions and memory attributes.

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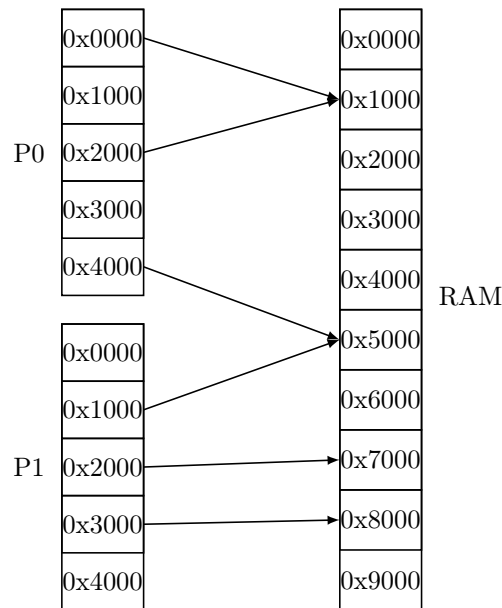
Figure 7.1: Example virtual and physical address spaces for two processes.

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.

7.3 Arm Translation Tables

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

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

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

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

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

Figure 7.2 shows a view of an example set of translation tables. Each rectangular block represents one contiguous block of memory, made up of 512 64-bit entries, each drawn as a dotted box. The base register points to the start of the level 0 table (the 'root' table). The second, seventh, and eleventh, indexes in the root table contain pointers to subsequent (level 1) tables, and so on. Blank cells represent invalid entries (for which the input virtual addresses are unmapped), and otherwise contain a pointer to a nested table or some output physical address. The exact format of these entries is described in the next section (see §7.3.1).
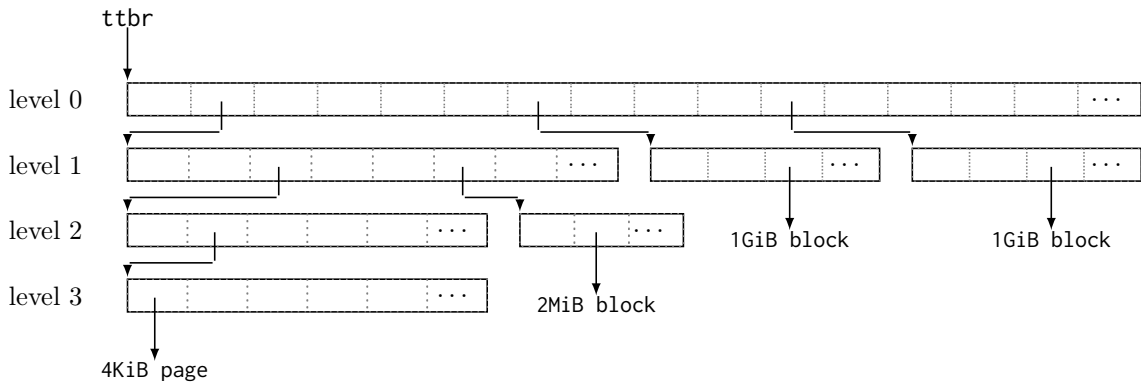
Figure 7.2: Schematic view for an extract of an example tree of translation tables, made of seven individual translation tables over four levels, which defines an address space that maps four separate regions of virtual addresses.

7.3.1 Translation table format

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

Each node is a table of 512 64-bit entries, bound as one 4096-byte block of memory. Each table controls the mapping of a fixed range of the virtual address space. This range is split into 512 equal slices, with each entry 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.
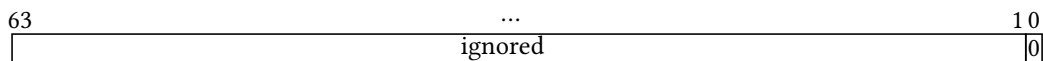
Block or page entries   Block and page entries are similar to each other; both create a mapping for a contiguous slice of the domain mapped by the entry, encoded as an output address (OA) with some metadata (including access permissions, memory type, and some software-defined bits).

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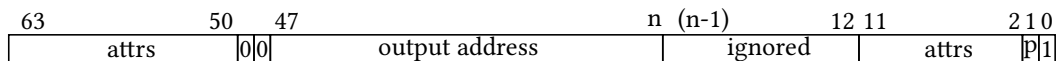
Table entries   A table entry contains a page-aligned pointer to a child table, but can also contain similar metadata as the block or page entry, including access permissions (read/write/execute), which are combined with any permissions from the child table.

Table entries are allowed only at levels 0–2.
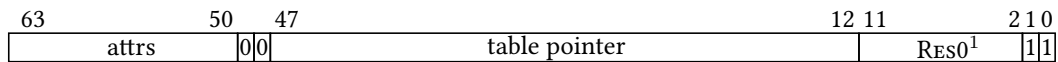
7.4 The Arm translation table walk

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 §7.6); 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.
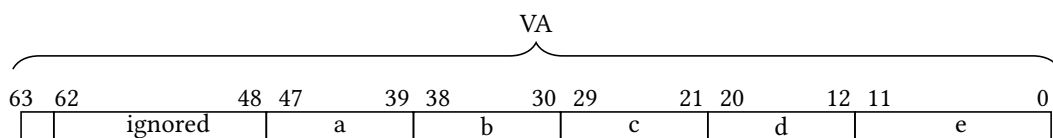
The walk then proceeds, with the MMU taking the following steps:

1 Read the base address from the TTBR register (See Reading the TTBR).
2 Perform a 64-bit single-copy atomic read of `Mem[baddr+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 (See Faults).
   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 Access permissions), report a fault to the processor.
      ii Otherwise, return the output address (See 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.

---

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

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.
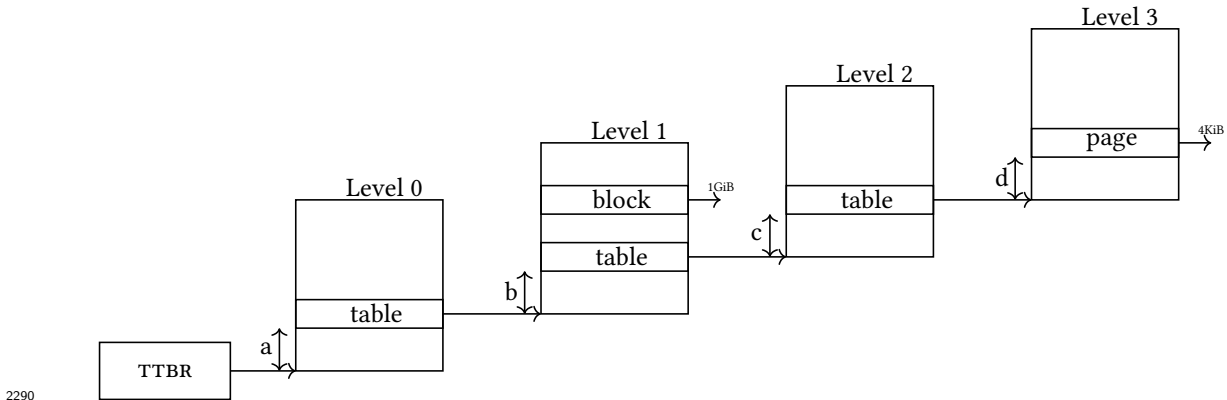
Reading the TTBR The base address register contains three fields: the higher bits store the ASID (see §7.8), or the VMID if for the second stage of a two-stage regime (see §7.5,§7.6); bits 47-1 contain bits 47-1 of the physical address of the root of the translation tables; the final bit is the "Common not Private" (CnP) bit, which is used to indicate when a cluster of processors share the same address space and base address to enable special performance optimisations.
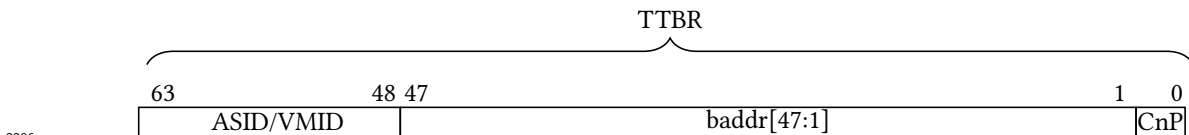
Computing the final output address The output address (OA) of the final descriptor is the start of the range mapped by the entry. The low order bits are all 0 in the output address, and need to be added on to compute the final output address of the translation.

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:

---

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

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 7.3 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, 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.   THE ARM TRANSLATION TABLE WALK   83

7.5 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:

$$\text{translate\_stage1} : \text{VirtualAddress} \rightharpoonup \text{IPA} \times \text{Permissions} \times \text{MemoryType}$$
$$\text{translate\_stage2} : \text{IPA} \rightharpoonup \text{PhysicalAddress} \times \text{Permissions} \times \text{MemoryType}$$

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 7.4 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 7.5.
- ▷ 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 [12, D5-4874] for all possible encodings):
    - − `0b0000`: Device memory.
    - − `0b0101`: Normal non-cacheable.
    - − `0b1111`: Normal write-back inner&outer cacheable.

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

Second-stage translations during a first-stage walk   There is a complication with the story so far. The stage 1 tables are created by the operating system, which is using an intermediate physical address space, not a physical one. The writes the OS does to the tables will be translated, as they are normal data writes. But, the tables themselves contain references to other tables, and those entries will be intermediate physical addresses, and so, they must also be translated, including the value of the TTBR itself.
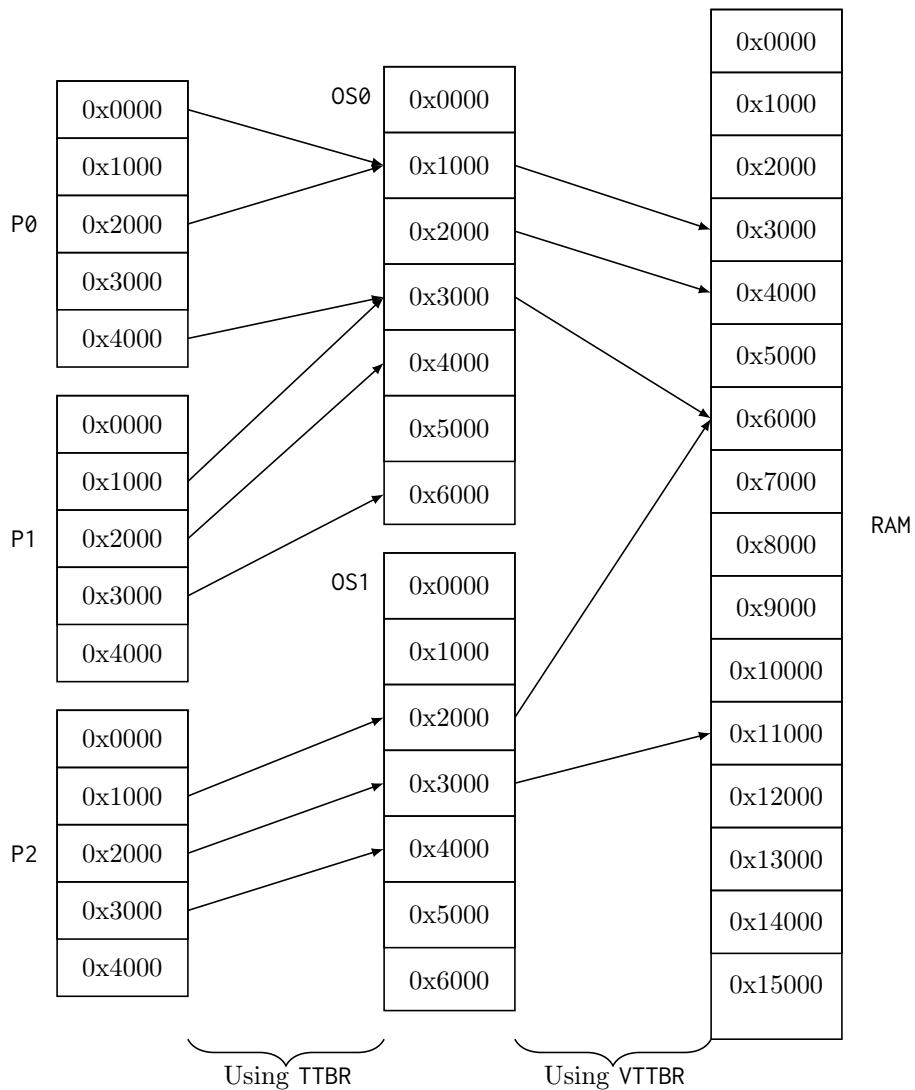
Figure 7.4: Example virtual, intermediate physical, and physical address spaces for three processes running on two operating systems.

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

Figure 7.5: S2AP field encoding.

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

7.6 Translation regimes

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

Arm define a set of these translation regimes. Figure 7.6 gives an overview of three of the most common regimes, which are:

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

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

▷ Translations at EL1 or EL0 use one of the EL1&0 regimes.
▷ Translations at EL2 use the EL2 regime.
▷ TCR_EL2 (set at EL2) determines whether the EL1&0 is a single-stage or two-stage regime.
▷ TTBR0_EL1, TTBR1_EL1 determine the stage 1 of the EL1&0 regimes, and can be set at EL1 or higher.
▷ TTBR0_EL2 determines the stage 1 of the EL2 regime, and can only be set at EL2 or higher.
▷ VTTBR_EL2 determines the stage 2 of the EL1&0 regime, and can only be set at EL2 or higher.

Arm define a wide range of other regimes, see the Arm ARM **TODO: ?REF?**. For simplicity, we ignore secure modes, including all of EL3.
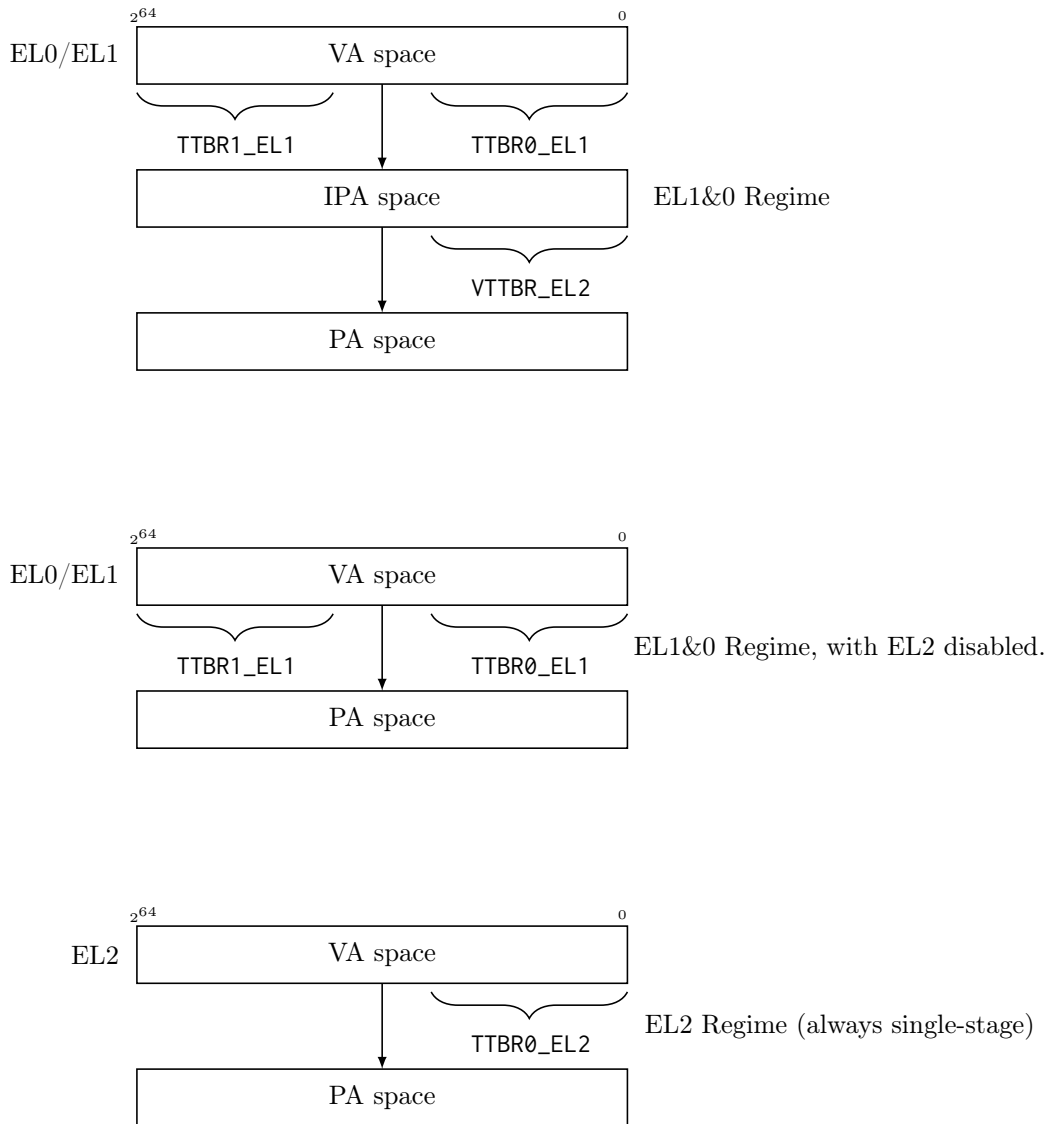
Figure 7.6: Translation regimes that apply to EL0,EL1, and EL2.

7.7 Arm pseudocode

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

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

7.7.1 The lifecycle of a store

Arm give precise executable semantics for every instruction in their domain-specific Architecture Specification Language (ASL).

Recall from §2.2 that the ASL code defines the sequential (intra-instruction) behaviour of each instruction, including memory accesses, and any translation table walks they perform.

Figure 7.7 shows the Arm ASL for the "STR (Immediate)" instruction: STR Xt,[Xn]. This instruction writes the value contained in register Xt into the memory location stored in register Xn. The figure has some uninteresting

```
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 7.7: Arm "STR (immediate)" ASL code.

(for this thesis) parts greyed out: those parts that deal with optional extensions such as memory tagging; unknown register values; register writeback; and, the load and prefetch instructions which use the same ASL code.

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

7.7.2 Writes to memory

The Mem[] function is responsible for checking alignment and performing each memory access the instruction does. The ASL for Mem[] can be found in Figure 7.8.

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

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

```
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
```

The `MemSingle[]` call then performs the translation, and (if successful), the actual write to memory. Its ASL can be found in Figure 7.9, with parts for extensions and store pair greyed out. On line 12, it calls `AArch64.TranslateAddress` to do the translation table walk. If the translation succeeds, then the code calls `PhysMemWrite` (on line 40), an uninterpreted function with no behaviour in ASL, which represents the actual write to memory. After perhaps handling any external aborts from the write, the function returns.

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

7.7.3 Translation table walks

It is the AArch64.TranslateAddress function which begins the process that performs the actual translation table walk, converting the input virtual address to the physical one. The full ASL code is too much to contain in a single figure, and so it can be found in §7.9 at the end of this chapter. This section will reference the relevant lines from the translation table walk ASL.

Figure 7.10 is an example trace of the execution of the STR Xt,[Xn] instruction, as it would happen if we were to execute it from EL1 in the EL1&0 two-stage regime. Each node represents an event in the trace (a memory or register access), and the arrows between them represent control flow. **TODO: Generate from an actual isla trace rather than by hand? at least to be proper... TODO: Give labels to each event?**

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

The events drawn inside the dotted box come from accesses during the call to the translation table walk functions. It first calls FullTranslate (in AArch64.TranslateAddress, page 95, line 2), which calls S1Translate (in AArch64.FullTranslate, page 96, line 12), which calls S1Walk (in AArch64.S1Translate, page 97, line 29) to do the actual first-stage translation table walk. It begins by reading the relevant TTBR register to get the root table address (in AArch64.S1Walk, page 100, line 9). This is stored in a walkstate struct, which the ASL code uses to keep track of the state that changes as the walk progresses, notably, the next-level table address and any accumulated permissions. It then begins the loop to do the walk, starting from the table address read from the TTBR. On each iteration of the loop, the intermediate-physical address of the entry to be read is computed (in AArch64.S1Walk, page 100, line 38), and passed through a second stage of translation (in AArch64.S1Walk, page 100, line 47).

This second stage translation calls S2Walk, which behaves similarly to the S1Walk function, taking the following steps: it reads the VTTBR (in AArch64.S2Walk, page 104, line 11); computes the (now) physical address of the entry to read (in AArch64.S2Walk, page 104, line 41); and reads it (in AArch64.S2Walk, page 104, line 44), eventually calling PhysMemRead (in AArch64.FetchDescriptor, page 106, line 23), which appears as the first R S2 L0 node in Figure 7.10.

S2Walk continues to loop, each time updating the running walkstate with the next-level table address from the decoded descriptor (in AArch64.S2Walk, page 104, line 53), until a leaf entry is found. It is either invalid (in AArch64.S2Walk, page 105, line 65), or, a valid page or block entry (in AArch64.S2Walk, page 105, line 70). These correspond to the next three R S2 Ln events in the figure.

Assuming the walk did not fail with a fault, the S2Translate function returns with the physical address of the stage 1 level 0 table. S1Walk can continue, performing a read of the physical memory in the table (in AArch64.S1Walk, page 100, line 52). From there, S1Walk continues in much the same way as the stage 2 walk did: computing the current table intermediate-physical address, translating it to get the physical address, performing the read of memory to get the descriptor, until a leaf entry is found.

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

Finally, FullTranslate calls S2Translate one last time (in AArch64.FullTranslate, page 96, line 22) on the intermediate-physical address, generating the last Rreg(VTTBR) and R S2 Ln events, and producing the final PA of the translation.

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

7.8 Caching in TLBs

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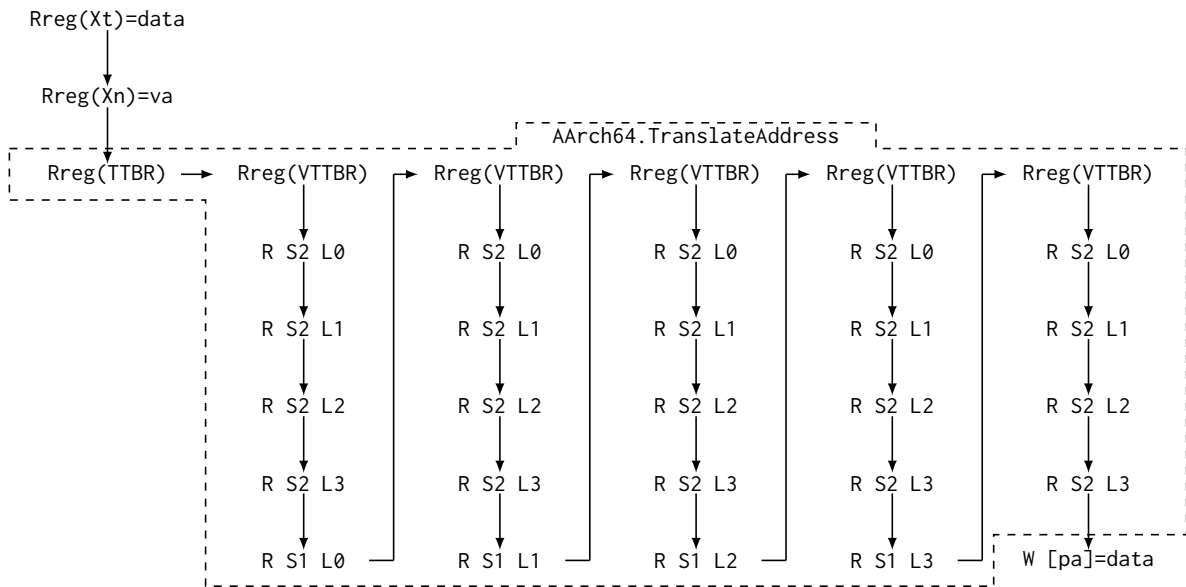
```
Rreg(Xt)=data

    │
    ▼
Rreg(Xn)=va
                                            ┌─────────────────────────────┐
    │                                       │      AArch64.TranslateAddress│
    ▼     ┌─────────────────────────────────┘                             └──────────────────
┌─────────┐
│Rreg(TTBR) ──→   Rreg(VTTBR) ─→  Rreg(VTTBR) ─→  Rreg(VTTBR) ─→  Rreg(VTTBR) ─→  Rreg(VTTBR)
└─────────┘
    │                  │               │               │               │               │
    │                  ▼               ▼               ▼               ▼               ▼
    │               R S2 L0         R S2 L0         R S2 L0         R S2 L0         R S2 L0
    │                  │               │               │               │               │
    │                  ▼               ▼               ▼               ▼               ▼
    │               R S2 L1         R S2 L1         R S2 L1         R S2 L1         R S2 L1
    │                  │               │               │               │               │
    │                  ▼               ▼               ▼               ▼               ▼
    │               R S2 L2         R S2 L2         R S2 L2         R S2 L2         R S2 L2
    │                  │               │               │               │               │
    │                  ▼               ▼               ▼               ▼               ▼
    │               R S2 L3         R S2 L3         R S2 L3         R S2 L3         R S2 L3
    │                  │               │               │               │               │
    │                  ▼               ▼               ▼               ▼               ▼
    └──────────────→R S1 L0 ──→ R S1 L1 ──→ R S1 L2 ──→ R S1 L3 ──→  W [pa]=data
```

Figure 7.10: 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:

```
1    TLBI <type><level><broadcast>{,<reg>}
2
3    <type> =
4      ALL | VMALL | ASID | VA{A|L} | IPAS2
5    <level> =
6      E1 | E2
7    <broadcast> =
8      {IS}
9    <reg> =
10     X0 | X1 | ... | X30
```

Again, see the full description in the Arm manual for a more complete description [12, 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)

7.9 Arm ASL Reference

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

The sources have per-function line numbers and are annotated to direct the reader to those parts highlighted in §7.7.3. Lines which handle out-of-scope features (access flags, dirty bits, shareability domains, debugging, realms, secure states, atomics) are greyed out. Key lines have coloured annotations.

The ASL code listed here (minus the annotations) is copyright 2022 Arm Limited, company 02557590 registered in England. The ASL code is publicly available on Arm's webpage [43], we reproduce here only those parts of the ASL being discussed here (the translation table walk), for the purposes of criticism, review, and quotation [86, s. 30].

7.9.1 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;
```

7.9.2 AArch64.FullTranslate

```
1  AddressDescriptor AArch64.FullTranslate(bits(64) va, AccType acctype, boolean
       iswrite, boolean aligned)
2
3     fault = NoFault();
4     fault.acctype = acctype;
5     fault.write   = iswrite;
6
7     ispriv = PSTATE.EL != EL0 && !(acctype IN {AccType_UNPRIV,
       AccType_UNPRIVSTREAM});
8     regime = TranslationRegime(PSTATE.EL, acctype);
9     ss = SecurityStateAtEL(PSTATE.EL);
10
11    AddressDescriptor ipa;                                    Do the first stage of translation
12    (fault, ipa) = AArch64.S1Translate(fault, regime, ss, va, acctype, aligned,
       iswrite, ispriv);
13
14    if fault.statuscode != Fault_None then    Check for stage 1 translation fault
15      return CreateFaultyAddressDescriptor(va, fault);
16
17    assert (ss == SS_Realm) IMPLIES EL2Enabled();
18    if regime == Regime_EL10 && EL2Enabled() then
19      s1aarch64 = TRUE;
20      s2fs1walk = FALSE;
21      AddressDescriptor pa;                                  Do the second stage of translation
22      (fault, pa) = AArch64.S2Translate(fault, ipa, s1aarch64, ss, s2fs1walk,
       acctype, aligned, iswrite, ispriv);
23
24      if fault.statuscode != Fault_None then    Check for stage 2 translation fault
25        return CreateFaultyAddressDescriptor(va, fault);
26      else
27        return pa;
28    else
29      return ipa;
```

7.9.3 **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      ⟵ Check VA is valid
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;                               Do the translation table walk
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 ⟵—— Check for S1 translation fault
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 ⟵—— Check for permission fault
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
```

```
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

7.9.4 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)
       ;                              Read memory to get descriptor
53     else
```

```
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                                              ╲Extract next level table address
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                                          ╲Extract page start address
73        when DescriptorType_Invalid
74          fault.statuscode = Fault_Translation;
75          return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64)
        UNKNOWN);
76                          └── Return fault if invalid
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);
```

7.9.5 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
```

7.9.6 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);←
42                                                        Get PA of entry to read
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);
66                     ↖ Return fault if invalid
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              ⟵ Check output address is within bounds
79    fault.statuscode = Fault_AddressSize;
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);
```

7.9.7 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);
```

Relaxed virtual memory

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

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

Ideally, we would be able to specify which points belong to which kind. It is, however, not so easy. There is no clean separation between aspects there are clearly defined in the architecture reference, and those that are not; instead, the manual has a shallow covering of many of the behaviours described here. In other places, the reference may have been updated or changed over the course of the work, clarifying parts of the architecture, and while this may have happened concurrently with discussing those and other points with Arm, the reference text itself is solely the responsibility of Arm. In §8.9 we will return to this question, and more directly address the kinds of each point discussed.

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

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

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

Chapter contents

8.1 Virtual memory litmus tests

Virtual memory poses its own challenges, but is fundamentally no different than the other fragments of Arm we have seen, and exploring the architectural intent is best done through the creation, discussion, and evaluation of, small programs which are representative examples of common software patterns.

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, to describe the litmus tests exploring those behaviours, we need a new language for describing the state of the system in a terse way, covering features not supported by the litmus test formats supported by the previous litmus, rmem, herd, and diy tools [64, 49, 39, 87].

The litmus tests here are given in the `isla-axiomatic` test format, extended with a small DSL for describing the initial state (and runtime constraints of) the pagetables. This format is described in detail in the `isla-axiomatic` documentation [88].

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").

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.

AArch64	CoW
Initial State	

```
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
virtual x 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 Handler
```
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 State
```
pa1=1 & pa2=2
```
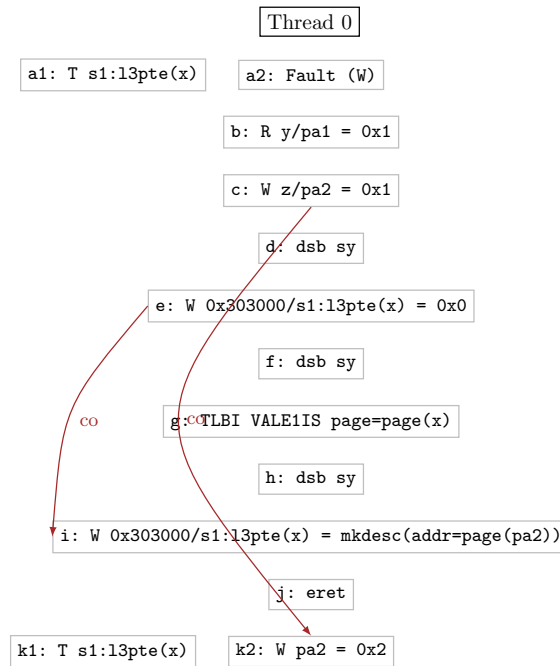
Allow

Figure 8.1: Test CoW: code listing

Figure 8.2: Test CoW: execution diagram

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 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 ≫ 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 [12, B2.3.1 (p157)] [12, 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, p112):

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	

Thread 0

```
a: W x/pa1 = 0x1
      rf   ↓po
b: R y/pa1 = 0x0
```
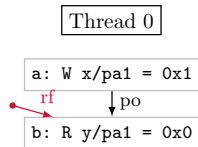
This test is a variation on the standard CoWR test, where the VA is replaced with two distinct VAs, which both alias to the same PA.

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.

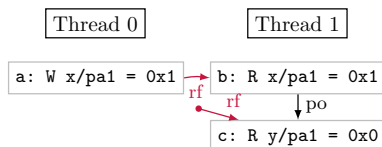While the Armv8-A architecture reference manual describes data caches as being physically-indexed [12, D5.11.1 (p4931)] and so accesses via the same PA are 'fully coherent', further discussions with Arm clarify that this implies not just this coherence test, but that all prior data memory behaviours 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, p113)).

▷ 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, p114), PPOCA.alias (Figure 8.6, p114)).

AArch64		CoRR0.alias+po
Initial State		
0:R0=0b1	1:R1=x	
0:R1=x	1:R3=y	
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		

Thread 0   Thread 1

```
a: W x/pa1 = 0x1  →  b: R x/pa1 = 0x1
               rf      rf   ↓po
                    c: R y/pa1 = 0x0
```

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	1:R0=0b10	2:R1=w	3:R1=y
0:R1=u	1:R1=v	2:R3=x	3:R3=z
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
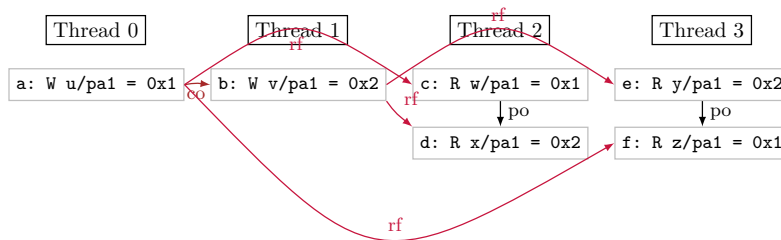
This test is a variation of the data memory CoRR2 test. Here there are many options for adding aliasing, so we choose the maximally aliased version where each individual store and load uses a distinct virtual address but where all those virtual addresses alias to the same physical one.

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	1:R1=y
0:R1=x	1:R3=x
0:R3=z	
0:R5=y	
physical pa1 pa2;	
x |-> pa1;	
y |-> pa2;	
z |-> pa1;	
*pa1 = 0;	
*pa2 = 0;	

Thread 0	Thread 1
STR X0,[X1]	LDR X0,[X1]
LDR X2,[X3]	DMB SY
STR X2,[X5]	LDR X2,[X3]

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

AArch64	PPOCA.alias
Initial State	
0:R0=0x1	1:R1=y
0:R1=z	1:R2=0x1
0:R2=0x1	1:R3=x
0:R3=y	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
	LDR X0,[X1]
	CBNZ X0,L0
STR X0,[X1]	L0:
DMB SY	STR X2,[X3]
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
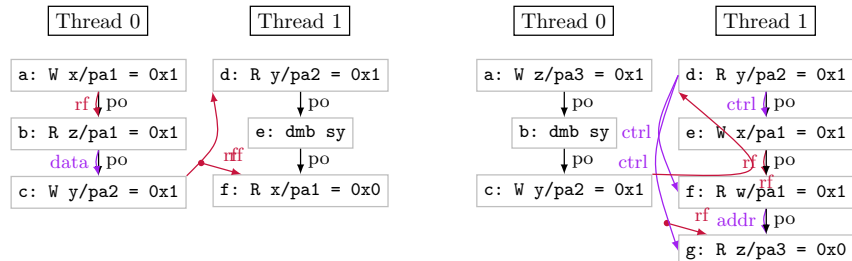
These tests are variations of the standard PPOCA and MP+rfi-data+dmb tests, but with some aliasing. Both are examples of forwarding: a thread-local read of a write before that write has been propagated to memory. These two tests, determined to be allowed architecturally from our discussions with Arm, 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	1:R1=y
0:R1=x	1:R3=x
0:R2=0x1	
0:R3=y	
physical pa1;	
x |-> pa1;	
y |-> pa1;	
*pa1 = 0;	
Thread 0	Thread 1
STR X0,[X1]	LDR X0,[X1]
STR X2,[X3]	LDR X2,[X3]
Final State	
1:X0=1 & 1:X2=0	
Forbid	

Because x and y alias to the same physical address pa1, the two loads (c and d) read the same location, and so cannot read different writes out-of-order.
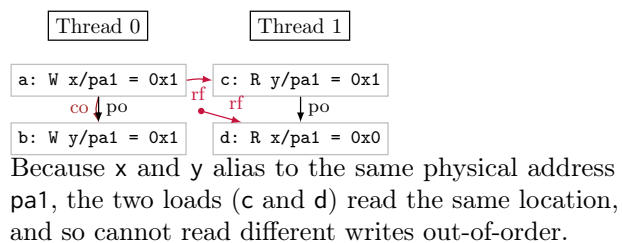
Figure 8.7: Test MP.alias+pos

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 §7.8, 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 [89, 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.
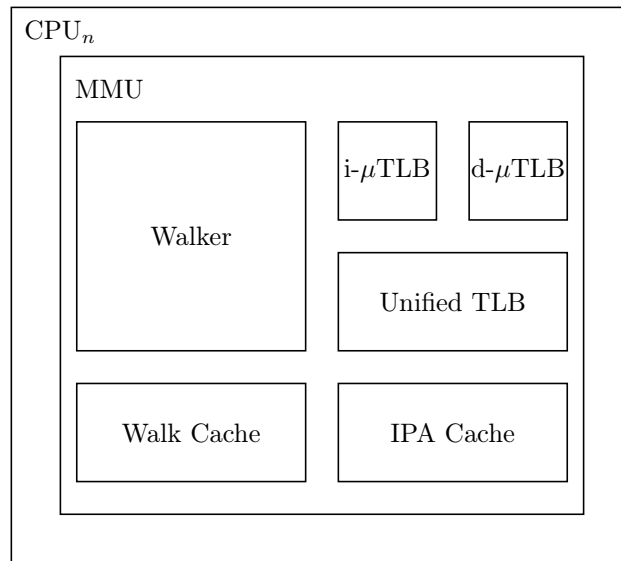
Figure 8.8: A53 Memory Management Block Diagram.

On a TLB miss, the MMU performs a translation table walk using the walker, computing the Arm translation table walk ASL code which we previously explored in §7.7.

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

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

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

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

8.3.2 Model MMU

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

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

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

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

$$\begin{aligned}
\mathrm{TranslationTableEntry} &\equiv \mathrm{u64} \\
\mathrm{Context} &\equiv \mathrm{ArchContext} \times \mathrm{Stage} \times \mathrm{option}\ \mathrm{VA} \times \mathrm{option}\ \mathrm{IPA} \times \mathrm{PA} \times \mathrm{Level} \\
\mathrm{ArchContext} &\equiv \mathrm{VMID} \times \mathrm{ASID} \times \mathrm{Regime} \\
\mathrm{CachedTranslationTableEntry} &\equiv \mathrm{PA} \times \mathrm{TranslationTableEntry} \times \mathrm{Context} \\
\mathrm{TLB} &\equiv \mathrm{set}\ \mathrm{CachedTranslationTableEntry}
\end{aligned}$$

Figure 8.9: Model TLB type definitions.

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

This contextual information includes:

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

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

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

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

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

8.3.3 Invalid entries

It is architecturally forbidden to cache information from attempted translations which result in translation faults, access flag faults, or address size faults (Note that a translation table walk may give rise to other faults as well, as discussed in §7.4, such as permission faults and alignment faults, which do not impose restrictions on TLB caching). More specifically, a TLB entry cannot be a write of a translation table entry which is the *direct* cause of such a fault. In particular, the TLB cannot cache translation table entries whose valid bit is not set.

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

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
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, p120).

AArch64	CoRpteTf.inv+po
Initial State	
0:R0=desc3(y)	1:R1=pte3(x)
0:R1=pte3(x)	1:R3=x
	1:VBAR_EL1=0x1000
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.
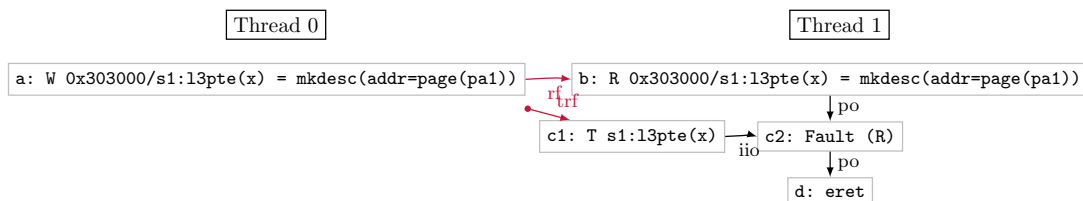

Figure 8.11: Test CoRpteTf.inv+po

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, p121)) with program-order earlier translation table walks, and, translations reading from program-order later writes (CoTW1.inv (Figure 8.13, p121)).

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, p122), which is forbidden by Arm. This is because the non-TLB reads, in the 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.

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
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.
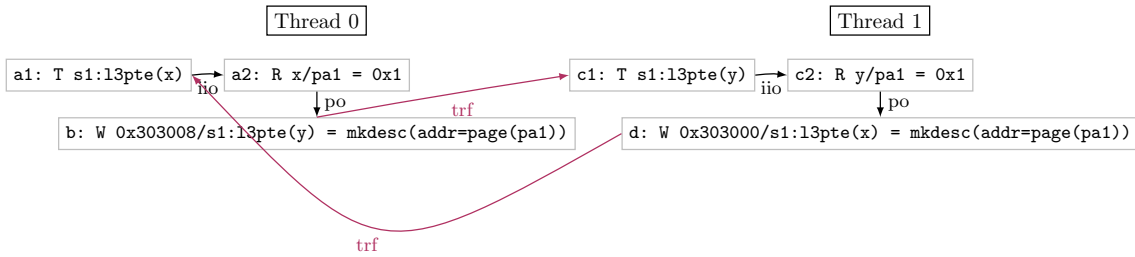

Figure 8.12: Test LB.TT.inv+pos

AArch64	CoTW1.inv
Initial State	
0:R1=x	
0:R2=desc3(y)	
0:R3=pte3(x)	
0:VBAR_EL1=0x1000	
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	
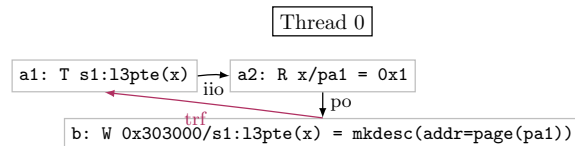
The store to the translation table (b) cannot be reordered with the program-order earlier translation table walk (a), preventing that walk from reading from the store.

Figure 8.13: Test CoTW1.inv

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

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, p123)), or through a normal data load of the translation table (CoRpteTf.inv+dsb-isb (Figure 8.16, p124)), 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, p125)).

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, p126).

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, and memory events program-order after the barrier.

AArch64	CoTTf.inv+dsb-isb
Initial State	
0:R0=desc3(y)	1:R1=x
0:R1=pte3(x)	1:R3=x
	1:VBAR_EL1=0x1000

```
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.
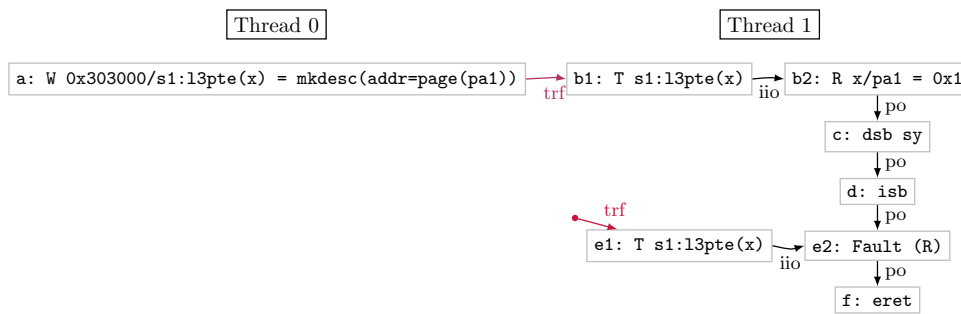
Figure 8.15: Test CoTTf.inv+dsb-isb

AArch64	CoRpteTf.inv+dsb-isb
Initial State	
0:R0=desc3(y)	1:R1=pte3(x)
0:R1=pte3(x)	1:R3=x
	1:VBAR_EL1=0x1000
`option default_tables = true;`	
`physical pa1;`	
`intermediate ipa1;`	
`x |-> invalid;`	
`x ?-> pa1;`	
`y |-> pa1;`	
`identity 0x1000 with code;`	
`*pa1 = 1;`	
Thread 0	Thread 1
	`LDR X0,[X1]`
	`DSB SY`
`STR X0,[X1]`	`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)	1:R1=x
0:R1=pte3(x)	1:R3=x
	1:VBAR_EL1=0x1000
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.
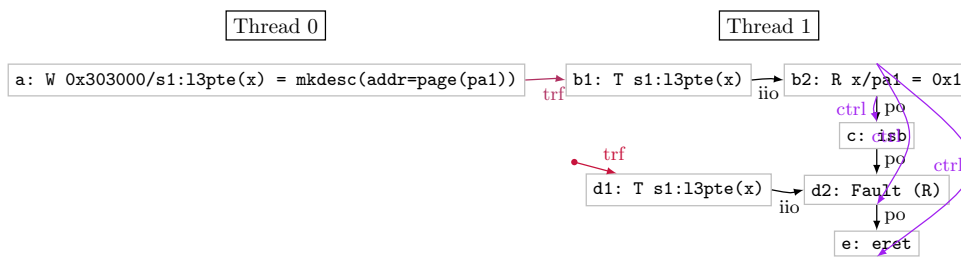
Figure 8.17: Test CoTTf.inv+ctrl-isb

AArch64	CoRpteTf.inv+addr
Initial State	
0:R0=desc3(y)	1:R1=pte3(x)
0:R1=pte3(x)	1:R3=x
	1:VBAR_EL1=0x1000
option default_tables = true;	
physical pa1;	
intermediate ipa1;	
x |-> invalid;	
x ?-> pa1;	
y |-> pa1;	
identity 0x1000 with code;	
*pa1 = 1;	
Thread 0	Thread 1
	LDR X0,[X1]
STR 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.
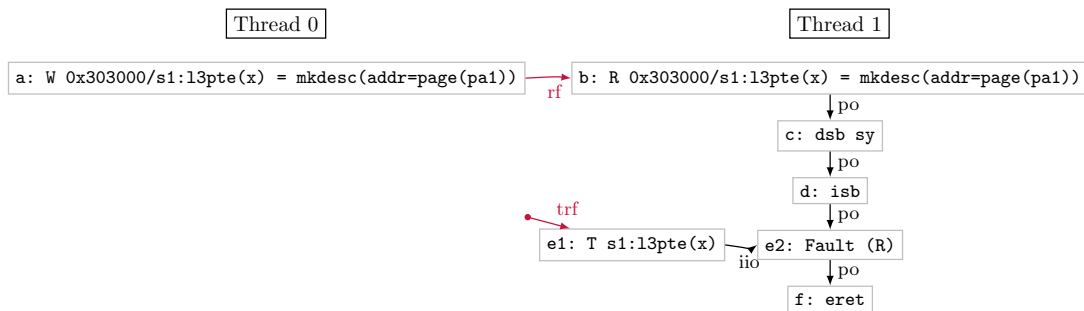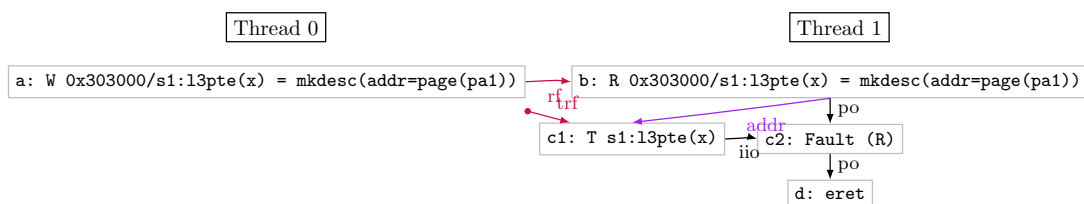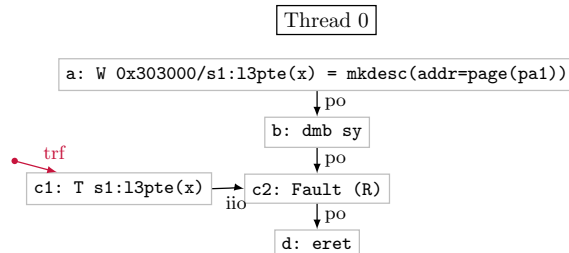

Figure 8.18: Test CoRpteTf.inv+addr

```
AArch64                              CoWTf.inv+dmb
                    Initial State
0:R0=desc3(y)
0:R1=pte3(x)
0:R3=x
0:VBAR_EL1=0x1000
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 principal 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 [12, 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, I_1 and I_2, each either a load or store. Imagine I_1 and I_2 both executing to completion, without generating any translation, address size, or access flag faults. Then, each instruction would
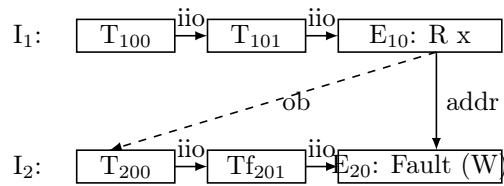
Figure 8.20: ETS Ghost events example: A load instruction (I_1) followed followed (in program order) by a store instruction (I_2), which faults. The address dependency means that the read event E_{10} is syntactically ordered-before the (ghost) write event E_{20}, and so the read event is ordered before the reads of the translation table walk for I_2 read from the TLB or memory (represented by the dashed ob line).

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 E_{ij} for the jth explicit event of instruction I_i.

Each explicit event E_{ij} would have required a translation table walk, generating translation read events which we can call T_{ijk} for the kth translation-table-walk read for the jth explicit memory event for instruction I_i.

Then, if I_2 generates a translation, address size, or access flag fault, and E_{1n} would have been locally-ordered-before E_{2m} in the imagined execution without the fault (and which we can consider a kind of *ghost* event in the real execution), and FEAT_ETS is enabled, then, E_{1n} is locally ordered before any translation table read $T_{2m\_}$ in the execution with the fault. This scenario is described pictographically in Figure 8.20.

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 The architecturally desired effect of FEAT_ETS seems to be that no additional context-synchronisation should be required to prevent spurious aborts, and that simple local orderings (barriers, dependencies) should be enough. To make this so, 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.

AArch64		R.TR.inv+dmb+trfi
Initial State		
0:R0=0x2	1:R0=mkdesc3(oa=pa1)	2:R1=pte3(w)
0:R1=x	1:R1=pte3(w)	
0:R2=0x2	1:R3=w	
0:R3=pte3(w)	1:VBAR_EL1=0x1000	

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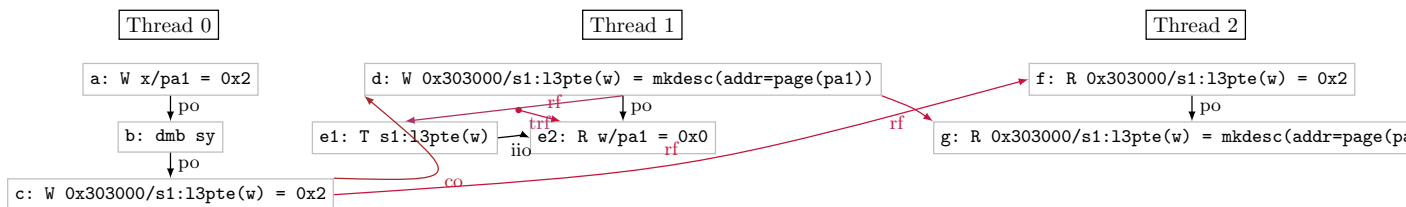

Figure 8.21: Test R.TR.inv+dmb+trfi

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 8.21).

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.

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 8.22, p130) to see an old value for the translation table entry, as the translation can be performed speculatively. **TODO: PS:**

AArch64	MP.RTf.inv+dmb+ctrl
Initial State	
0:R0=desc3(z)	1:R1=y
0:R1=pte3(x)	1:R3=x
0:R2=0b1	1:VBAR_EL1=0x1000
0:R3=y	

```
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.
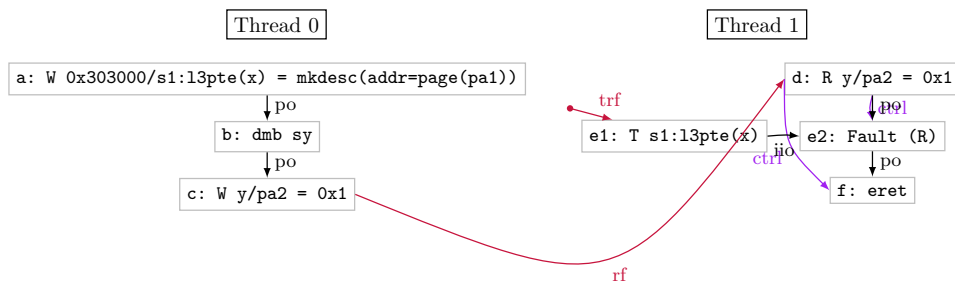

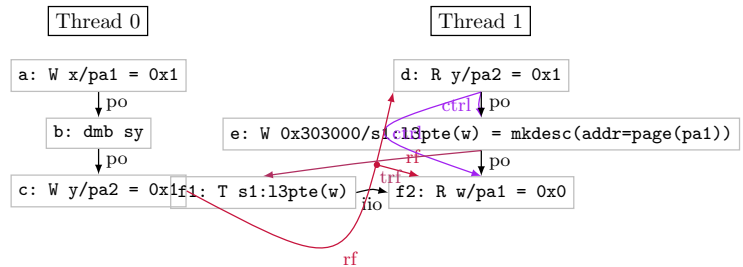**Figure 8.22:** Test MP.RTf.inv+dmb+ctrl

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 translation read to read from a still speculative write. The MP.RT.inv+dmb+ctrl-trfi test (Figure 8.23, p131) 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?**

AArch64	MP.RT.inv+dmb+ctrl-trfi
Initial State	
0:R0=0b1	1:R1=y
0:R1=x	1:R2=mkdesc3(oa=pa1)
0:R2=0b1	1:R3=pte3(w)
0:R3=y	1:R5=w
	1:VBAR_EL1=0x1000
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 8.23: Test MP.RT.inv+dmb+ctrl-trfi

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 8.24, p132) 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 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

```
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
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 8.24: Test CoWroW.inv+dsb-isb

(Figure 8.25, p133). 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 8.26, p134) and WRC.TRR.inv+po+dsb (Figure 8.27, p135) tests, from reading the old value from memory at the end of Thread 2.

TODO: PS: these all need text captions

8.4.8 Translation-table-walk intra-walk ordering

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

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

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

Requiring the translation reads from a translation table walk to be satisfied in translation walk order has an observable effect, for example in the following ROT.inv+dsb test (Figure 8.28, p136) the translation table walk of the read in Thread 1 must see the writes to the translation table done by Thread 0 in the order they were propagated out to memory, and so reading from the old level 3 entry is forbidden.

8.4.9 Multiple translations within a single instruction

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

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

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

TODO: PS: litmus test with misalgined?

AArch64		WRC.TRTf.inv+po+dsb-isb
Initial State		
0:R0=desc3(z)	1:R1=x	2:R1=y
0:R1=pte3(x)	1:R2=0b1	2:R3=x
	1:R3=y	2:VBAR_EL1=0x2000
	1:VBAR_EL1=0x1000	

```
physical pa1 pa2;
x |-> invalid;
x ?-> pa1;
z |-> pa1;
*pa1 = 1;
y |-> pa2;
identity 0x1000 with code;
identity 0x2000 with code;
```

Thread 0	Thread 1	Thread 2

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

	Thread 1 EL1 Handler	Thread 2 EL1 Handler

	0x1400:
	MOV X0,#0
	MRS X13,ELR_EL1
	ADD X13,X13,#4
	MSR ELR_EL1,X13
	ERET
		0x2400:
		MOV X2,#0
		MRS X13,ELR_EL1
		ADD X13,X13,#4
		MSR ELR_EL1,X13
		ERET

Final State		
1:X0=1 & 2:X0=1 & 2:X2=0		
Forbid		

Figure 8.25: Test WRC.TRTf.inv+po+dsb-isb

AArch64		WRC.RRTf.inv+dmb+dsb-isb
	Initial State	
0:R0=desc3(z)	1:R1=pte3(x)	2:R1=y
0:R1=pte3(x)	1:R2=0b1	2:R3=x
	1:R3=y	2:VBAR_EL1=0x2000

```
physical pa1 pa2;
x |-> invalid;
x ?-> pa1;
z |-> pa1;
*pa1 = 1;
y |-> pa2;
identity 0x1000 with code;
identity 0x2000 with code;
```

Thread 0	Thread 1	Thread 2

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

		Thread 2 EL1 Handler

		0x2400:
		MOV X2,#0
		MRS X13,ELR_EL1
		ADD X13,X13,#4
		MSR ELR_EL1,X13
		ERET

	Final State	
1:X0=desc3(z) & 2:X0=1 & 2:X2=0		
Forbid		

TODO: PS: why **DSB** not just any R/R ordering.
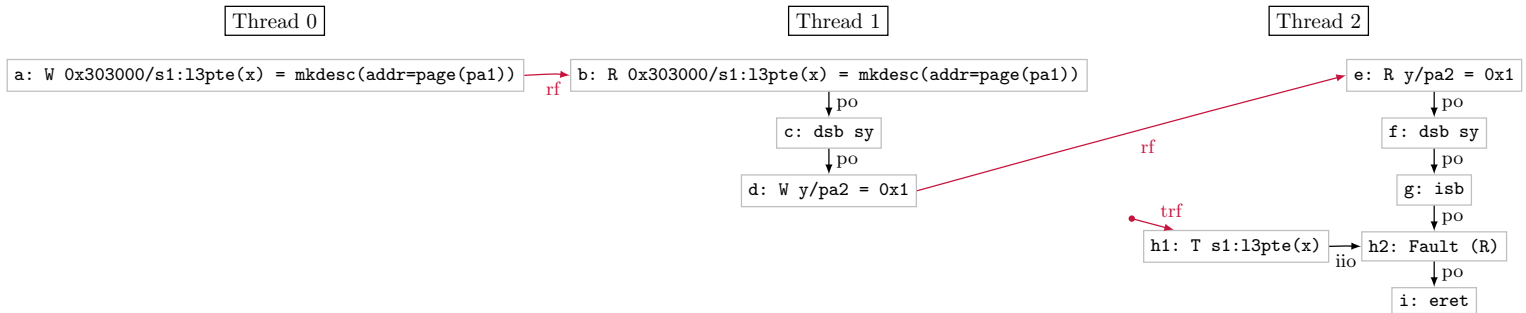
Figure 8.26: Test WRC.RRTf.inv+dmb+dsb-isb

AArch64		WRC.TRR.inv+po+dsb
	Initial State	
0:R0=mkdesc3(oa=pa1)	1:R0=0b0	2:R1=y
0:R1=pte3(x)	1:R1=x	2:R3=pte3(x)
	1:R2=0b1	
	1:R3=y	
	1:VBAR_EL1=0x1000	

```
physical pa1 pa2;
x |-> invalid;
x ?-> pa1;
y |-> pa2;
*pa1 = 1;
identity 0x1000 with code;
```

Thread 0	Thread 1	Thread 2

STR X0,[X1]	LDR X0,[X1]
	STR X2,[X3]
		LDR X0,[X1]
		DSB SY
		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 & 2:X0=1 & ~2:X2=0		
	Allow	

TODO: PS: why **DSB** not just any R/R ordering.

Figure 8.27: Test WRC.TRR.inv+po+dsb

```
AArch64                        ROT.inv+dsb
                    Initial State
 0:R0=mkdesc3(oa=ipa1)       1:R1=x
 0:R1=pte3(x, new_table)    1:VBAR_EL1=0x1000
 0:R2=mkdesc2(table=0x283000)
 0:R3=pte2(x)
 0:PSTATE.EL=0b01
 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 |-> invalid;
   x ?-> ipa1;
 };
 identity 0x1000 with code;
       Thread 0         |        Thread 1
 STR X0,[X1]            |
 DSB SY                 |  LDR X0,[X1]
 STR X2,[X3]            |
                        |  Thread 1 EL1 Handler
                        |  0x1400:
                        |  //  read ESR_EL1.ISS,
                        |      MRS X14,ESR_EL1
                        |      AND X14,X14,#0b11
                        |      CMP X14,#0b111
                        |      MOV X17,#1
                        |  //  if ESR_EL1.ISS.D
                        |      MOV X18,#2
                        |  //  advance ELR
                        |      CSEL X0,X17,X18,
                        |      MRS X13,ELR_EL1
                        |      ADD X13,X13,#4
                        |  //  return
                        |      MSR ELR_EL1,X13
                        |      ERET
                    Final State
 1:X0=1
                      Forbid
```
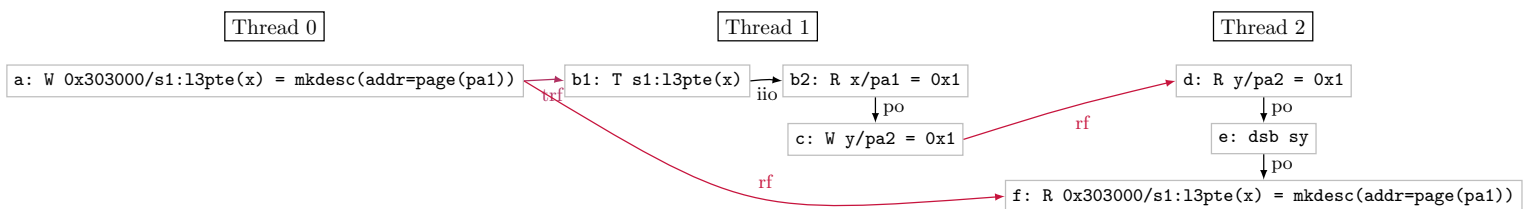
The translation-table walk from the read of x in Thread 1 must perform its translation non-TLB reads in the order they appear in the walk, forbidding reading from the new level 2 table entry in d1, but then reading the stale initial value for that entry from memory.

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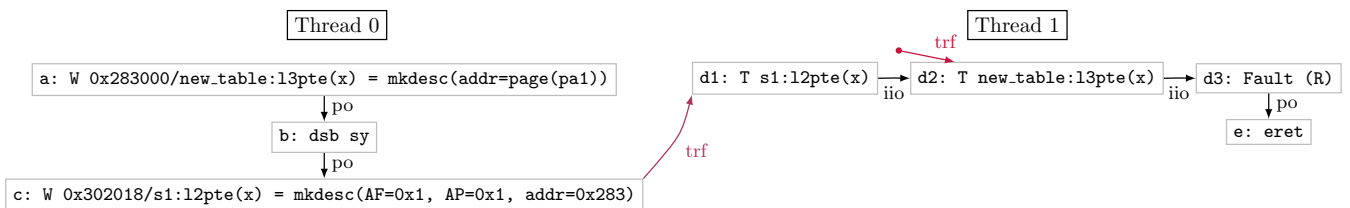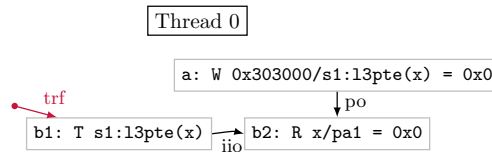

Figure 8.28: Test ROT.inv+dsb

```
AArch64                          CoWinvT+po
                 Initial State
 0:R0=0b0
 0:R1=pte3(x)
 0:R3=x
 0:VBAR_EL1=0x1000
 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
```

Thread 0

a: W 0x303000/s1:l3pte(x) = 0x0

$\downarrow$ po

b1: T s1:l3pte(x) $\xrightarrow{\text{iio}}$ b2: R x/pa1 = 0x0

trf

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 8.29: 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.
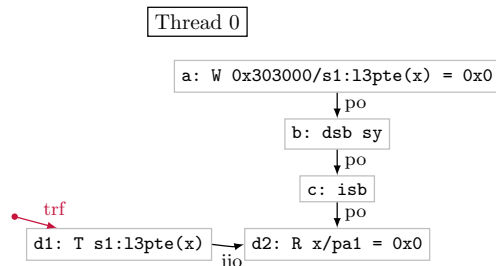
8.5.1 Cached translations

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 8.29) 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 8.30, p138), does not forbid it.

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 8.30: 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 8.31, p139), 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 micro-TLBs

So far we have spoken as if entries are, at any particular moment in time, either present in the TLB or not. Hardware, however, may have multiple *micro*-TLBs for the same thread, each with their own potential cached entry for the same address.

In effect, these micro-TLBs behave as if they were a larger non-deterministic TLB with potentially many values for each entry. 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.

These effects can be seen in the CoTfT+dsb-isb test (Figure 8.32, p140), where the presence of these micro-TLBs (or other distributed caching structures) allow later events (even locally-ordered later) to see old cached entries

AArch64	MP.RT.inv+poloc-dmb+ctrl-isb
Initial State	

```
0:R0=mkdesc3(oa=pa1)     1:R1=y
0:R1=pte3(x)             1:R3=x
0:R2=0b0                 1:VBAR_EL1=0x1000
0:R3=pte3(x)
0:R4=0b1
0:R5=y
```

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

Thread 0	Thread 1
	`LDR X0,[X1]`
	`CBNZ X0,L0`
`STR X0,[X1]`	`L0:`
`STR X2,[X3]`	
`DMB SY`	`ISB`
`STR X4,[X5]`	`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.
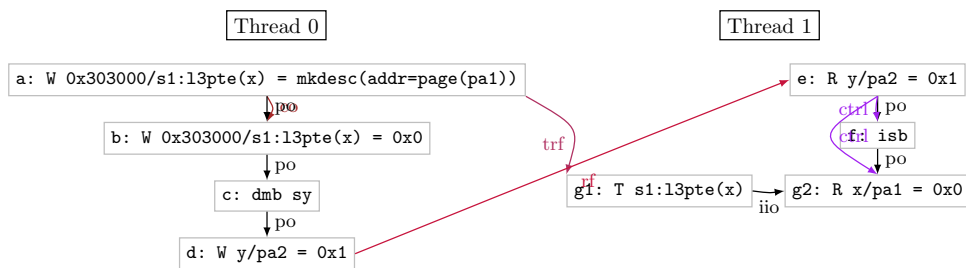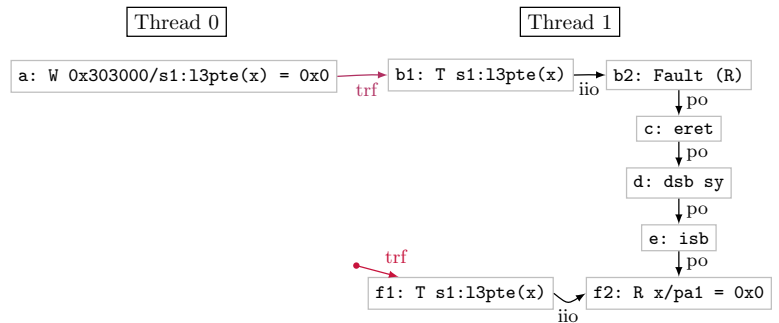

Figure 8.31: Test MP.RT.inv+poloc-dmb+ctrl-isb

Figure 8.32: Test CoTfT+dsb-isb

after earlier events witnessed a TLB miss.

Break-before-make and restrictions We will see later that the ability to have multiple cached entries for a single address causes problems for software managing coherence, and imposes extra restrictions on software practice. This means that, in general, the effects of the micro-TLBs are restricted to only those combinations that do not cause *break-before-make violations* (see §8.6.5).

8.5.4 Partial caching of walks

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

Caching up to last-level table The most common kind of caching structure we are aware of in microarchitecture is the walk cache (see §8.3.1). Traditionally a TLB would store entire virtual to physical mappings, making it fast to lookup the translation (often a single cycle), but there was limited space and induced heavy burden on a TLB miss or TLB invalidation. Walk caches store the last-level table entry, allowing TLB invalidation of leaf entries and TLB misses to re-use a prefix of the walk and perform a minimal number of accesses. This can be seen in the MP.RTT.inv3+dmb-dmb+dsb-isb test (Figure 8.33, p141), where a walk cache could allow the table entry to be cached separately from the last-level entry, allowing the last translation read to read from a much newer value.

Caching of whole translation A common configuration for the TLB is to cache whole translation walks, from virtual to physical. This kind of caching has an important caveat: there is no requirement for the TLB to remember the intermediate physical address of any stage 2 translations that were done during the walk, including the final stage 2 walk of the access address itself. This means that TLB invalidations by IPA might not remove all the cached data associated with a cached entry for that IPA, if there is a whole cached translation which used that entry. **TODO: ?REF?.**

Independent caching of IPAs In a two-stage regime, the virtual addresses are first translated into intermediate 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.

In addition, pre-fetching may perform translations of arbitrary IPAs. This means that any cached translations might not correspond to any valid whole translation table walk, but may still be used during such walks.

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)	
0:R4=0b1	
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]	LDR X0,[X1]
DMB SY	DSB SY
STR X2,[X3]	ISB
DMB SY	MOV X2,#1
STR X4,[X5]	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. TODO: PS: explain the magic numbers and tfr edge a bit.
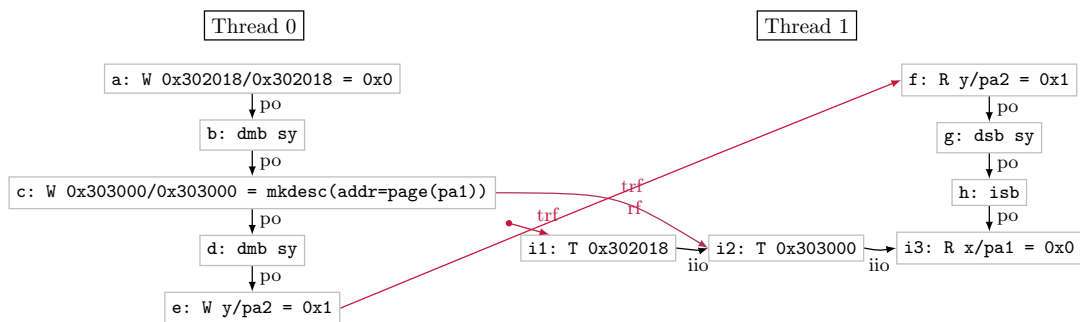

Figure 8.33: Test MP.RTT.inv3+dmb-dmb+dsb-isb

This is most clear in ROT.invs1+dmb2 (Figure 8.34, p143), 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, but there would be overwhelmingly many of these, or even a 'most relaxed' version where every translation table entry comes from different previous translations, but this would be too large to show here. So, for simplicity I show just one of them here, ROT.inv2+dmb (Figure 8.35, p144); 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 may only cache translation table entries which are *reachable* from a translation in the current context. 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, using the current system register state.

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.

This is captured in the RUE+isb (Figure 8.36, p144) ("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 TLB *maintenance* instructions: namely the TLBI ("TLB invalidate" instruction).

TLB maintenance generally causes two microarchitectural effects: erasing stale entries from the TLB, ensuring future TLB fills (for example, due to a translation read) will see the coherent value from memory; and, discarding 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

We saw earlier, in Section 8.5.1, that stale values cached in the TLB can cause coherence violations in the translation, for example, in the CoWinvT+dsb-isb test (Figure 8.30, p138). By inserting the correct TLBI sequence into that test, we produce a new test, CoWinvT.EL1+dsb-tlbi-dsb-isb (Figure 8.37, p145), which is now 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 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.

AArch64	ROT.invs1+dmb2
Initial State	
`0:R0=mkdesc3(oa=pa1)`	`1:R1=x`
`0:R1=pte3(x, s2_table)`	`1:VBAR_EL1=0x1000`
`0:R2=0b0`	`1:VBAR_EL2=0x2000`
`0:R3=pte3(x, s2_table)`	
`0:R4=mkdesc3(oa=ipa1)`	
`0:R5=pte3(x)`	
`0:PSTATE.EL=0b01`	

```
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]`	`MOV X0,#0`
`DMB SY`	`LDR X0,[X1]`
`STR X4,[X5]`	

	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 the write (a) even though it was not reachable by any translation table walk for any VA, as the IPA it maps was not mapped by any stage 1 tables before it was overwritten by (b).

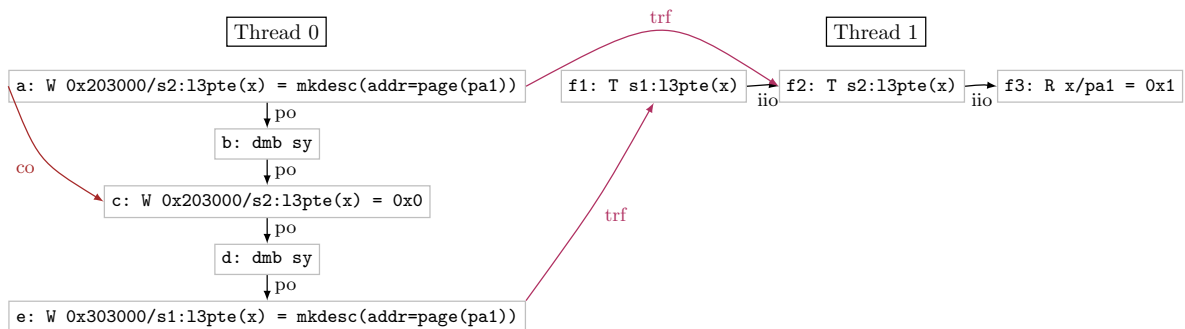This test relies on translation table walks being naturally ordered (by iio), see §8.4.8.

Figure 8.34: Test ROT.invs1+dmb2

```
AArch64                          ROT.inv2+dmb
                  Initial State
 0:R0=0b0                     1:R1=x
 0:R1=pte3(x, new_table)      1:VBAR_EL1=0x1000
 0:R2=mkdesc2(table=0x283000)
 0:R3=pte2(x)
 0:PSTATE.EL=0b01
 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              | MOV X0,#1
 STR X2,[X3]         | 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 8.35: Test ROT.inv2+dmb
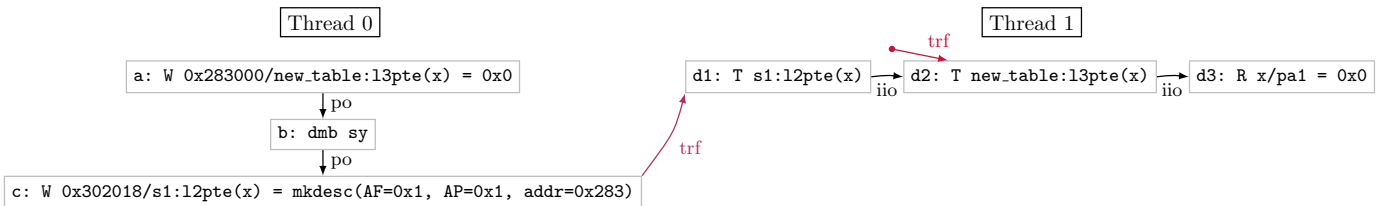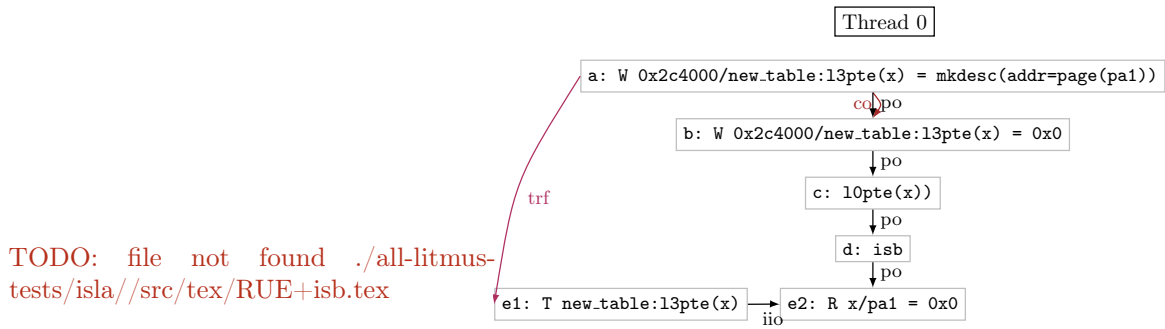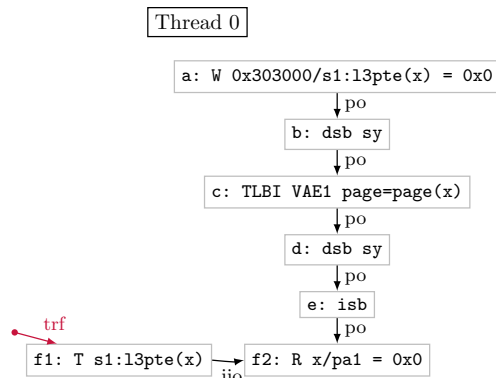

TODO: file not found ./all-litmus-tests/isla//src/tex/RUE+isb.tex

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 8.36: Test RUE+isb

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

FEAT_nTLBPA

Armv8.4-A introduced a new optional Arm feature, FEAT_nTLBPA [12, 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 8.38, p146) 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	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	
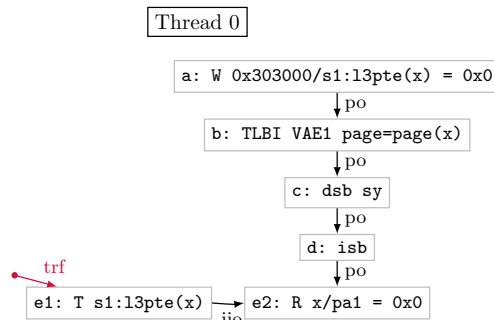
The `TLBI` (b) can be re-ordered with program-order earlier events, due to the lack of `DSB`s ordering it after them, allowing the store (a) to happen later, letting the final translation read (e1) still see the old stale translation.

Figure 8.38: Test CoWinvT.EL1+tlbi-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 `TLBI`s, 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 8.39, p147).

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 `TLBI`s 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 8.40, p147) (**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 8.41, p148), 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 [12, D5.10.2 (p4928)] .

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	
0:R4=page(x)	
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	LDR X0,[X1]
TLBI VAE1IS,X4	DSB SY
DSB SY	ISB
STR X2,[X3]	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	

The broadcast TLBI on Thread 0 (c) ensures that the earlier unmapping (a) is seen by the ordered later translation read on Thread 1 (i1), by ensuring Thread 1's local TLB is cleaned of any stale entries for x.

Figure 8.39: Test MP.RT.EL1+dsb-tlbiis-dsb+dsb-isb

AArch64	RBS+dsb-tlbiis-dsb
Initial State	
0:R0=0b0	1:R1=x
0:R1=pte3(x)	1:VBAR_EL1=0x1000
0:R5=page(x)	
0:R2=0x2	
0:R3=y	
0:PSTATE.EL=0b01	
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	LDR X0,[X1]
DSB SY	
STR X2,[X3]	
	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	
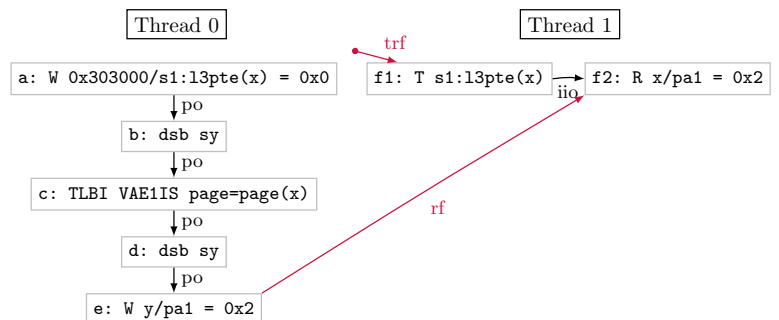
The broadcast TLBI of x (c) ensures that the execution of the load of x in Thread 1 either entirely executes using the old translation and finishes before the TLBI does, or begins execution after the TLBI finishes.

Figure 8.40: Test RBS+dsb-tlbiis-dsb

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

Thread 0

a: W 0x303000/s1:l3pte(x) = 0x0 → (trf) → f1: T s1:l3pte(x) → (iio) → f2: Fault (R)

b: dsb sy (po from a)

c: TLBI VAE1IS page=page(x) (po from b)

d: dsb sy (po from c)

e: W y/pa1 = 0x2 (po from d)

Thread 1

f1: T s1:l3pte(x) → (iio) → f2: Fault (R)

g: eret (po from f2)

h1: T s1:l3pte(x) → (iio) → h2: R x/pa1 = 0x0

(trf to h1) ; (rf to h2) ; (po from g to h2)

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 8.41: Test RBS+dsb-tlbiis-dsb+poloc

Atomic TLBIs In the previous RBS-shaped tests, I describe the behaviour in terms of writes that occur 'before' the TLBI.

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

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

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

8.6.4 Virtualization

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

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

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

This can be seen in MP.RT.EL2+dsb-tlbiipais-dsb+dsb-isb (Figure 8.42, p149), where invalidation of *just* the IPA is not enough. Adding an invalidation of the VA (or all VAs), like in MP.RT.EL2+dsb-tlbiipais-dsb-tlbiis-dsb+dsb-isb (Figure 8.43, p150), ensures that later translations cannot see the stale value anymore.

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

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

```
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]	LDR X0,[X1]
DSB SY	DSB SY
TLBI IPAS2E1IS,X4	ISB
DSB SY	MOV X2,#1
STR X2,[X3]	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.
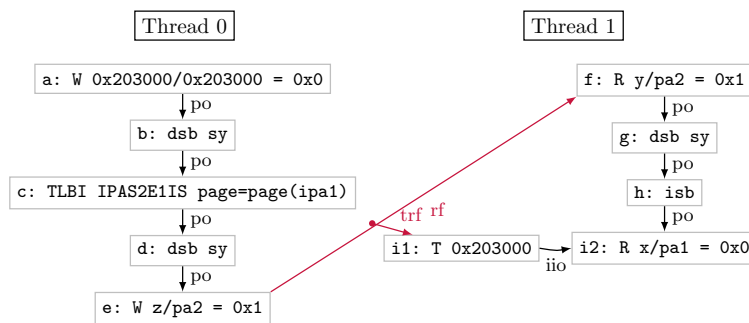
Figure 8.42: Test MP.RT.EL2+dsb-tlbiipais-dsb+dsb-isb

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

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

```
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 8.43: 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.
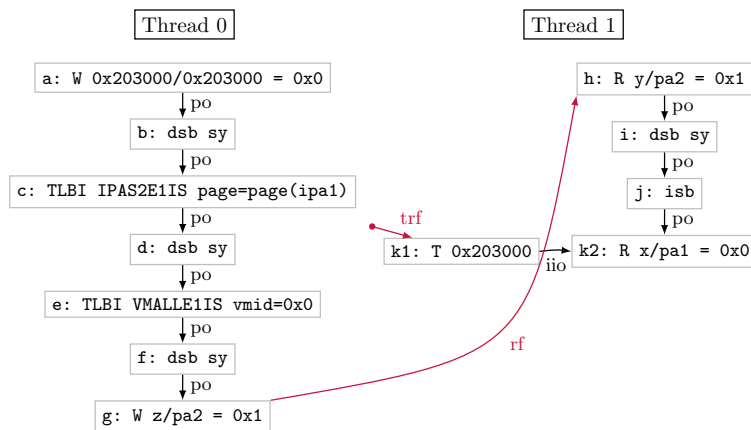
To avoid this possibility, the architecture provides a *break-before-make* sequence, which will ensure that there cannot be two cached translations existing in the TLB at the same time.

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 8.44, p152) 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" [12, 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)	
0:R4=0b1	
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	LDR X0,[X1]
DSB SY	
STR X2,[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.
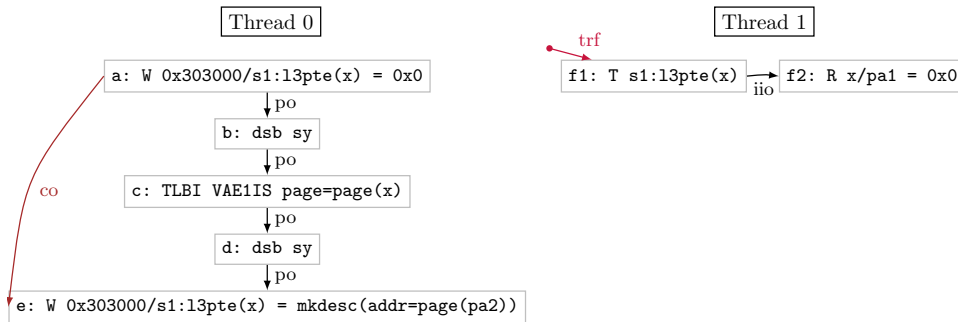

Figure 8.44: Test BBM+dsb-tlbiis-dsb

8.6.7 Access permissions

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

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

Accesses which result in permission faults result in exceptions, much like translation faults do, but may have been read from the TLB. This can clearly be seen in the CoWinvTp.ro+dsb-isb test (Figure 8.45, p154), where ordered after a write to the translation tables a permission failure is experienced, whose descriptor must have come from the TLB.

Multiple cached entries The changing of access permissions not necessarily being break-before-make violations allows us to observe multiple cached entries within the TLB. It is permitted for these entries to exist simultaneously.
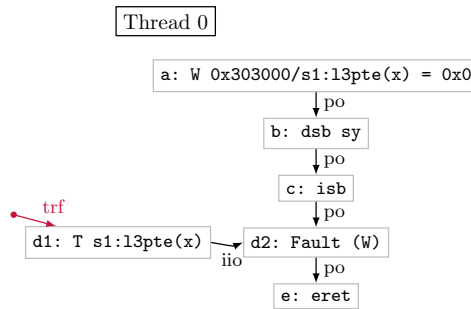
When reading from the TLB, and there existing multiple entries for the same input address, it is allowed for the hardware to generate a *TLB conflict abort*. These aborts are reported as data aborts.

If the hardware does not generate a conflict abort, then translation reads of that address are CONSTRAINEDUN-PREDICTABLE, nondeterministically able to read one or the other or an "amalgamation" of the values [12, K1.2.3 (p11243)] .

Here there seems a contradiction:it is not required to perform break-before-make, but there is no requirement that only one entry be cached in the TLB. We can side step this issue by constructing a test that only changes a single bit of the descriptor, in a way that is not a break-before-make violation, and therefore avoiding any questions about how 'amalgamation' of entries happens. This can be seen with the MP.RTpT.ro+dmb-dmb+dsb-isb-dsb-isb test (Figure 8.46, p155), where the existence of multiple cached entries in the TLB allows multiple translation-reads to read from different stale writes.

Atomic TLB reads Existence of multiple cached translation table entries in the TLB, without break-before-make violations, introduces the question of whether those TLB fills and subsequence TLB reads must read from entire single-copy atomic writes of the original translation table entries (much like a read of memory would) or whether a translation read can read from a mix of different writes. RMD+dmb (Figure 8.47, p156) ("Read-mixed-descriptor") shows that translation reads cannot read partially read from a write, it must read from the entire 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
```

```
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 o|
    MRS X14,ESR_EL1
    AND X14,X14,#0b1111
    CMP X14,#0b1111
//  Permission
    MOV X15,#1
//  Translation
    MOV X16,#2
    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 8.45: Test CoWinvTp.ro+dsb-isb

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

Thread 0	Thread 1
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)	
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
	MRS X14,ESR_EL1
	AND X14,X14,#0b11
	CMP X14,#0b1111
	// Permission
	MOV X15,#1
	// Translation
	MOV X16,#2
	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.
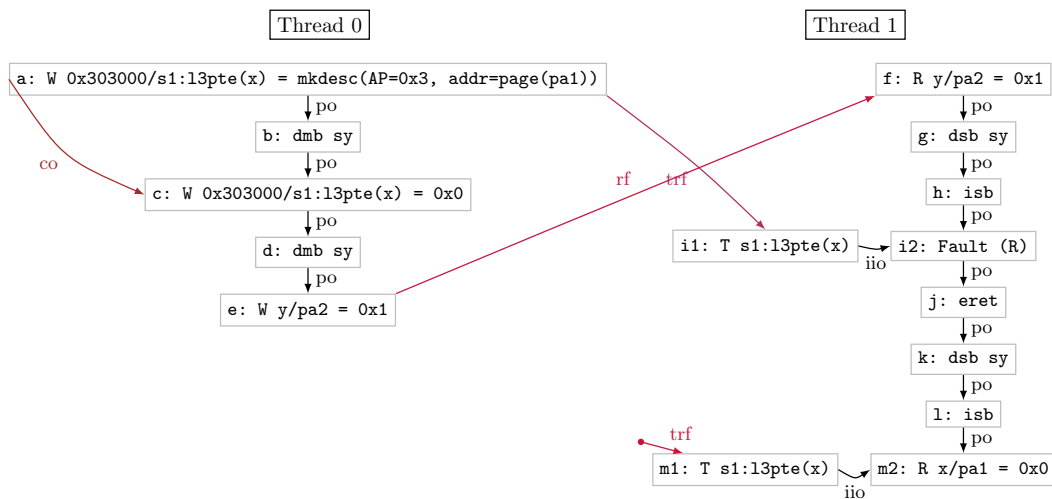

Figure 8.46: Test MP.RTpT.ro+dmb-dmb+dsb-isb-dsb-isb

AArch64	RMD+dmb
Initial State	

```
0:R0=mkdesc3(oa=pa2, AP=0b10)  0:R1=x
0:R1=pte3(x)                  1:VBAR_EL1=0x1000
0:R2=0x1
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]`
`DMB SY`
`STR X2,[X3]`
	`MOV X0,#0`
	`LDR X0,[X1]`

	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.
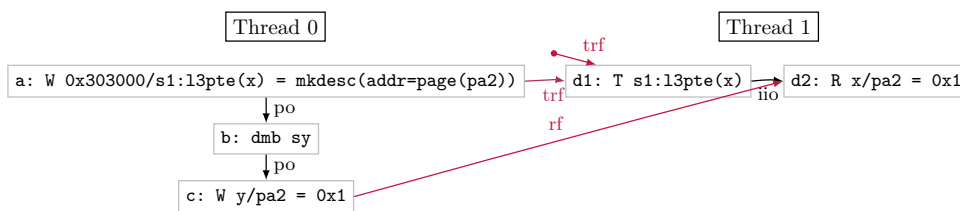

Figure 8.47: Test RMD+dmb

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 [12, 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 Details likely to change

There have been a few places so far I've added words to the effect of 'this is currently under discussion with Arm'. In this section I will summarise those things which we know some things, but also know that it is is likely to change and the ways in which it will.

Caching of entries in the TLB   The biggest change we are aware of to the model is a strengthening in caching of entries in the TLB. We have assumed that the TLB can cache any combination of translation table entries for a walk, and recall any cached combination as well. We are aware that Arm wish to strengthen this, to make the model TLB more in-line with the hardware TLB: essentially requiring the model TLB to behave as a walk cache, caching whole walks (or prefixes of walks) rather than individual entries.

TLB Invalidation   We have primarily considered TLB invalidations of cached level 3 (that is, last level) entries. When invalidating entries higher in the table, they affect more of the address space (as described in §7.3.1), and so the TLB invalidation must affect addresses outside of just the page referenced. The model currently does not support this, this is a simple oversight and we believe not hard to update the model to handle this case.

More complex invalidation patterns, for example, zeroing and invalidating table entries, is still under discussion with Arm.

ETS   We are aware of changes to the architecture regarding ETS. Every attempt has been made to try incorporate those changes into the model as we become aware, however, often they are changes driven by others and we only become aware as they are publically released.

It is very likely the parts of the model dedicated to ETS will gain new strength over the coming weeks and months, it is unfortunately not possible at this time to give a detailed description of what the final state of ETS will look like, partially for confidentiality reasons and partially because Arm have not yet decided.

System registers   As previously described, the current state of relaxed system register reads and writes is unclear. We are in discussion with Arm on this aspect. It is not possible at this time to describe what the final model will look like, or what changes will need to be done to the model presented here.

Exceptions and context-synchronisation   We are in discussion with Arm about the nature of exceptions and their context-synchronising nature and how this interacts with the memory model.

We believe the changes required to the model presented here will be minor, although they will probably be neither a relaxation nor a strengthening of the current model, but rather an incomparable change.

8.9 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 TLBIs, 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.

8.10 Related work

For address translation, the authoritative Arm-internal ASL model [10, 11, 66], and Sail model derived from it [44] cover this, and other features sufficient to boot an OS (Linux), as do the handwritten Sail models for RISC-V (Linux and FreeBSD) and MIPS/CHERI-MIPS (FreeBSD, CheriBSD), but without any cache effects. Goel et al. [80, 90] describe an ACL2 model for much of x86 that covers address translation; and the Forvis [91] and RISCV-PLV [92] Haskell RISC-V ISA models are also complete enough to boot Linux.

Syeda and Klein [93, 94] provide an somewhat idealised model for ARMv7 address translation and TLB maintenance.

Komodo [55] uses a handwritten model for a small part of ARMv7, as do Guanciale et al. [56, 57]. Romanescu et al. [95, 96] do discuss address translation in the concurrent setting, but with respect to idealised models.

Lustig et al. [82] describe a concurrent model for address translation based on the Intel Sandy Bridge microarchitecture, combined with a synopsis of some of the relevant Linux code, but not an architectural semantics for machine-code programs.

Chapter 9

An axiomatic VMSA model

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 Chapter 8, 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 (§7.7) 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 §7.8 Arm have a variety of TLBI instructions, with varying arguments. All of these TLBIs 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 TLBIs.
> ▷ 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 TLBIs 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:

> ▷ DSBSY and DSBISH (here, equivalent as we do not model shareability domains)
> ▷ DSBNSH, for thread-local effects.
> ▷ DSBST, DSBLD, for DSBs affecting only loads or stores.
> ▷ DSBISHST, DSBISHLD, and so on, for all combinations of DSB instruction domain and access types.

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.

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.

```
1  let dsbsy = DSBISH | DSBSY | DSBNSH
2  let dsbst = dsbsy | DSBST | DSBISHST | DSBNSHST
3  let dsbld = dsbsy | DSBLD | DSBISHLD | DSBNSHLD
4  let dsbnsh = DSBNSH
5  let dmbsy = dsbsy | DMBSY
6  let dmbst = dmbsy | dsbst | DMBST | DSBST | DSBISHST |
       DSBNSHST
7  let dmbld = dmbsy | dsbld | DMBLD | DSBISHLD | DSBNSHLD
8  let dmb = dmbsy | dmbst | dmbld
9  let dsb = dsbsy | dsbst | dsbld
```

Figure 9.1: Barrier helper sets.

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 [6, §10.5 p174], and so there is no issue in doing this. Secondly, for broadcast TLBIs, 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 [32, §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 TLBIs, 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.

```
 1  let speculative =
 2      ctrl
 3    | addr; po
 4    | [T]; instruction-order
 5
 6  (* translation-ordered-before *)
 7  let tob =
 8      [T_f]; tfre
 9    | [T]; iio; [R|W]; po; [W]
10    | speculative; trfi
11
12  (* observed by *)
13  let obs = rfe | fr | wco
14    | trfe
15
16  (* ordered-before TLBI and
       translate *)
17  let obtlbi_translate =
18      [T & Stage1] ; tlb_barriered
       ; [TLBI-S1]
19    | ([T & Stage2] ; tlb_barriered
       ; [TLBI-S2])
20       & (same-translation ; [T &
       Stage1] ; trf⁻¹ ; wco⁻¹)
21    | (([T & Stage2] ;
       tlb_barriered ; [TLBI-S2]) ;
       wco? ; [TLBI-S1])
22       & (same-translation ; [T &
       Stage1] ; maybe_TLB_cached)
23
24  (* ordered-before TLBI *)
25  let obtlbi =
26      obtlbi_translate
27    | [R|W|Fault_T]; iio⁻¹; (
       obtlbi_translate & ext); [TLBI
       ]
28
29  (* context-change ordered-before
       *)
30  let ctxob =
31      speculative; [MSR]
32    | [CSE]; instruction-order
33    | [ContextChange]; po; [CSE]
34    | speculative; [CSE]
35    | po; [ERET]; instruction-order
       ; [T]
36
37  (* ordered-before a translation
       fault *)
38  let obfault =
39      data; [Fault_T & FaultFromW]
40    | speculative; [Fault_T &
       FaultFromW]
41    | [dmbst]; po; [Fault_T &
       FaultFromW]
42    | [dmbld]; po; [Fault_T & (
```

```
       FaultFromW| FaultFromR)]
43    | [A|Q]; po; [Fault_T & (
       FaultFromW | FaultFromR)]
44    | [R|W]; po; [Fault_T &
       FaultFromReleaseW]
45
46  (* ETS-ordered-before *)
47  let obETS =
48      (obfault; [Fault_T]); iio⁻¹;
       [T_f]
49    | ([TLBI]; po; [dsb];
       instruction-order; [T]) & tlb-
       affects
50
51  (* dependency-ordered-before *)
52  let dob =
53      addr | data
54    | speculative; [W]
55    | addr; po; [W]
56    | (addr | data); rfi
57    | (addr | data); trfi
58
59  (* atomic-ordered-before *)
60  let aob = rmw
61    | [range(rmw)]; rfi; [A | Q]
62
63  (* barrier-ordered-before *)
64  let bob = [R]; po; [dmbld]
65    | [W]; po; [dmbst]
66    | [dmbst]; po; [W]
67    | [dmbld]; po; [R|W]
68    | [L]; po; [A]
69    | [A | Q]; po; [R | W]
70    | [R | W]; po; [L]
71    | [F | C]; po; [dsbsy]
72    | [dsb]; po
73
74  (* Ordered-before *)
75  let ob = (obs | dob | aob | bob
76    | iio | tob | obtlbi | ctxob |
       obfault | obETS)⁺
77
78  (* Internal visibility
       requirement *)
79  acyclic po-loc | fr | co | rf as
       internal
80  (* External visibility
       requirement *)
81  irreflexive ob as external
82  (* Atomic requirement *)
83  empty rmw & (fre; coe) as atomic
84  (* Writes cannot forward to po-
       future translates *)
85  acyclic (po-pa | trfi) as
       translation-internal
```

Figure 9.2: RVM axioms and relations

```
1  (* Internal visibility requirement *)
2  acyclic po-loc | fr | co | rf as internal
3  (* Writes cannot forward to po-future translates *)
4  acyclic (po-pa | trfi) as translation-internal
```

Unchanged from the original, the `internal` axiom captures the SC-per-location guarantee briefly discussed in §**TODO: ?REF?**. Translations, however, do not have the same per-location guarantees. To account for this, we introduce a second axiom, `translation-internal`, which captures the weaker per-location guarantee for translation table walks. Since translation reads, in the presence of TLB caching and out-of-order pipelines, do not guarantee even coherence, the only behaviour this axiom ends up preventing is translation reads reading from program-order later stores.

External axiom   The `external` axiom asserts acyclicity of the global happens-before ordering for Arm. The happens-before (called ob, 'ordered-before', in Arm) relation is the union of all the ordering relations, given in §9.4.

```
1  (* Ordered-before *)
2  let ob = (obs | dob | aob | bob | iio | tob | obtlbi | ctxob
       | obfault | obETS)⁺
3  (* External visibility requirement *)
4  irreflexive ob as external
```

We choose to include all the pipeline and TLB effects as ordering requirements, rather than introducing new ordering axioms just for translation and TLB invalidation. This produces a model that is more consistent with the previous Arm memory models, and ensures ordering information gained through observing translation table walks are respected by non-translation-table accesses.

Atomic axiom   The `atomic` axiom remains unchanged. In this work we do not consider the interaction of translation with atomic accesses.

```
1  (* Atomic requirement *)
2  empty rmw & (fre; coe) as atomic
```

9.4 Relations

The RVM model modifies some of the original, and introduces some new, ordering relations. This section goes through each in detail, describing the mechanism and justifying the existence or non-existence of particular clauses.

9.4.1 obs

```
1  (* observed by *)
2  let obs = rfe | fr | wco | trfe
```

The 'observed-by' relation. It includes the original `rf` and `fr` (over physical locations), the *generalised coherence order* `wco` (§9.1.2), and the `trfe` (translation-reads-from-external) relation.

Generalised coherence   Including `wco`, which is existentially quantified over the candidates, fixes some global order the writes and TLBIs happen in. Consider, informally, some microarchitectural execution. It would propagate writes to the coherent storage subsystem, and would complete TLBI instructions, and these events would be interleaved in some whole-machine trace. The generalised `wco` relation captures the relative ordering of these events in the axiomatic model, as they would have happened in the traces of machine executions. The model is 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

```
1  let dob =
2      addr | data
3    | speculative; [W]
4    | addr; po; [W]
5    | (addr | data); rfi
6    | (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

```
 1  let bob =
 2      [R]; po; [dmbld]
 3    | [W]; po; [dmbst]
 4    | [dmbst]; po; [W]
 5    | [dmbld]; po; [R|W]
 6    | [L]; po; [A]
 7    | [A | Q]; po; [R | W]
 8    | [R | W]; po; [L]
 9    | [F | C]; po; [dsbsy]
10    | [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, DSBs).

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

```
1  let tob =
2      [T_f]; tfre
3    | [T]; iio; [R|W]; po; [W]
4    | 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). However, it should not be the case that the write is forwarded from a write that is too old or behind a `DSB` if `FEAT_ETS`, except it may be the case that there might be other intervening writes in between. For now, we are unable to give a precise bound on the ordering for thread-local `[T_f];tfr`, and this area is still currently under investigation with Arm.

Speculation   As we saw earlier, speculation interacts with translation in two ways: first, it is forbidden to read-from a still speculative write (§8.4.5), and, secondly, events program-order after an instruction which does a translation table walk are speculative until the translation table walk completes (§8.4.1).

To capture these we first define when one event is considered speculative until another event happens, with a new `speculative` relation, defined as following:

```
1 let speculative = ctrl | addr; po | [T]; instruction-order
```

This captures all the control-flow dependencies that we model here, the classic `ctrl` and `addr; po`, as well as a new general `[T]; instruction-order` which says that all events ordered `(iio|po)+` after a translation read are speculative until the translation read satisfies. We can then include `speculative ; trfi` to succinctly forbid any forwarding of still-speculative writes to translation table walks.

Finally, we include `[T]; iio; [M]; po; [W]` which captures that writes cannot propagate until program-order earlier instructions have their physical address (so, do not fault). Although, this edge is subsumed by the `speculative; [W]` edge in dob, it is kept here for clarity.

9.4.5 ctxob

> NOTE: The model for exceptions and context-synchronising events is currently under revision, and what is presented here is likely to change.

```
1 let ctxob =
2     speculative; [MSR]
3   | [CSE]; instruction-order
4   | [ContextChange]; po; [CSE]
5   | speculative; [CSE]
6   | po; [ERET]; instruction-order; [T]
```

The `ctxob` relation captures the orderings required from context changing and synchronising operations, without trying to capture the full extent of the relaxed behaviours. As such, these orderings are likely to be incomparable to the real semantics.

Speculation   The first guarantee we see is that context changes and synchronisation should not happen speculatively. Speculative context changes may end up creating translation table roots and therefore translation table walks using unreachable writes (§8.5.5). To prevent this we ensure that context changing operations only happen once they are non-speculative, by enforcing `speculative; [MSR]` in ob. Forbidding speculative execution of context synchronisation is done through the inclusion of `speculative; [CSE]` in ob.

Context synchronising   A context synchronisation event (such as an `ISB` or `ERET` instruction) should ensure that program-order earlier context-changing events are seen by program-order later instructions. Microarchitecturally this is achieved by having context-synchronisation events flushing the pipeline, restarting all program-order later instructions. For now this effect seems fixed in the architecture (§8.7), and so we get `[CSE]; instruction-order` in ob subsuming the earlier `ISB` orderings.

To ensure that context changes are seen after the synchronisation we include `[ContextChange]; po; [CSE]`, and the union of these two relations ensures the context change is ordered before any program-order later events.

Exceptions   Taking and returning from exceptions are context synchronising (§8.7), and so those are captured by the previous clauses. However, translation reads of a lower exception level should not satisfy during execution at a higher exception level. We over approximate this with `po; [ERET]; instruction-order; [T]` ensuring all translation reads after an ERET wait.

9.4.6 `obfault` and `obETS`

```
 1  (* ordered-before a translation fault *)
 2  let obfault =
 3      data ; [Fault_T & FaultFromW]
 4    | speculative ; [Fault_T & FaultFromW]
 5    | [dmbst] ; po ; [Fault_T & FaultFromW]
 6    | [dmbld] ; po ; [Fault_T & (FaultFromW | FaultFromR)]
 7    | [A|Q] ; po ; [Fault_T & (FaultFromW | FaultFromR)]
 8    | [R|W] ; po ; [Fault_T & FaultFromW & FaultFromReleaseW]
 9
10  (* ETS-ordered-before *)
11  let obETS =
12      (obfault; [Fault_T]); iio⁻¹; [T_f]
13    | ([TLBI]; po; [dsb]; instruction-order; [T]) & tlb-affects
```

To capture the specific guarantees described by FEAT_ETS (§8.4.3, §8.6.2), we include 'ghost' `Fault` events in the candidate executions. These events sit in the execution (in po order) where the explicit memory event would have been if there was no fault, and tags the fault with the kind of fault it was (translation or permission).

Ordering to a fault   To fully capture the strength of FEAT_ETS we keep track of syntactic dependencies *into* the instruction which faulted, and apply those dependencies to the `Fault` event itself. `obfault` then the syntactic subset of bob and dob where the right-hand side of each clause is substituted with a `Fault_T` (a translation fault).

Using `obfault` we can then keep track of the (syntactic) subset of ob that would have ordered the explicit event after, and associate those relations with the `Fault_T` event instead. `obETS`'s first clause then adds to ob this ordering, but attached to the translation read of the invalid entry itself, as architected by FEAT_ETS.

Note that dependencies and orderings *from* a faulting instruction seem not respected, and so we do not induce orderings out of a `Fault_T`.

FEAT_ETS and TLBI   The second clause of obETS captures the second architected behaviour of FEAT_ETS (§**TODO: TLBI ordering needs ETS explained**), that faults after a thread-local TLBIs do not need context synchronisation to be ordered after the TLBI. Note that one still needs a DSB to complete the TLBI in that case.

9.4.7 `obtlbi`

```
 1  (* ordered-before TLBI *)
 2  let obtlbi =
 3      obtlbi_translate
 4    | [R|W|Fault_T]; iio⁻¹; (obtlbi_translate & ext); [TLBI]
 5
 6  (* translate ordered-before TLBI *)
 7  let obtlbi_translate =
 8    [T & Stage1] ; tlb_barriered ; [TLBI-S1]
 9  | (([T & Stage2] ; tlb_barriered ; [TLBI-S2]) ; wco? ; [TLBI-
       S1])
10    & (same-translation ; [T & Stage1] ; maybe_TLB_cached)
11  | ([T & Stage2] ; tlb_barriered ; [TLBI-S2])
12      & (same-translation ; [T & Stage1] ; trf⁻¹ ; wco⁻¹)
```

Finally, there is the `obtlbi` relation which captures the ordering from translations (and their explicit memory events) and the TLB invalidations which affect them. The relation is split in two: the `obtlbi_translate` clause enforces order between stale translations and the `TLBI`s they are invalidated by, the second clause covers broadcast `TLBI`s.

Capturing stale TLB entries   When a translation read happens, it is allowed for it to read from a stale write (§8.5.1). That is, the translation need not be ordered before writes which come after the write it actually reads from. Consequently the `tfre` relation is not included in `ob`.

We strengthen this, by including some edges from translations to TLBIs, when there is an interposing newer write. The general shape of this ordering is illustrated in Figure 9.3.
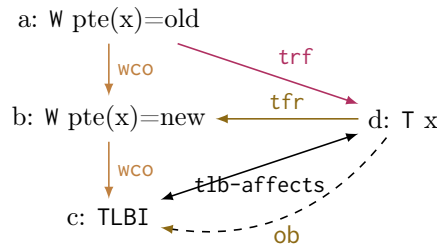

Figure 9.3: General `obtlbi_translate` shape.

This shape is succinctly captured by the `tlb_barriered` auxiliary relation, which relates any translate-read that reads from a write which is `wco` before another write which is `wco` before a TLBI which targets the address, ASID or VMID of the translation:

```
1 let tlb_barriered =
2   ([T] ; tfr ; wco ; [TLBI]) & tlb-affects⁻¹
```

We cannot simply include `tlb_barriered` in `ob`, however. Instead, we must consider the orderings for stage 1 and stage 2 translation reads separately.

Stale stage 1 reads   For stage 1 translation reads, either in single-stage regimes or as part of a two-stage regime, we can include a variant of `tlb_barried` specialised to stage 1 translation-reads and TLBIs which affect stage 1 entries.

Stale stage 2 reads   Stage 2 walks are more subtle. The requirement to perform stage 1 invalidation (§8.6.4) means that, in those instances, we do not get `tlb_barriered` directly.

Instead, we have to case split on the execution: either, (1) the translation table walk does a stage 1 translation read which reads-from an older write, in which case there may have been a whole cached translation that must be invalidated; or, (2) one of the stage 1 translation reads of the translation table walk reads from a write that is newer than the stage 2 TLBI and so there cannot have been any cached whole translation entries in the TLB and so, logically, we only need the stage 2 invalidation. These cases are illustrated in Figure 9.4, and correspond to the two clauses of `obtlbi_translate` which match on stage 2 translation reads.
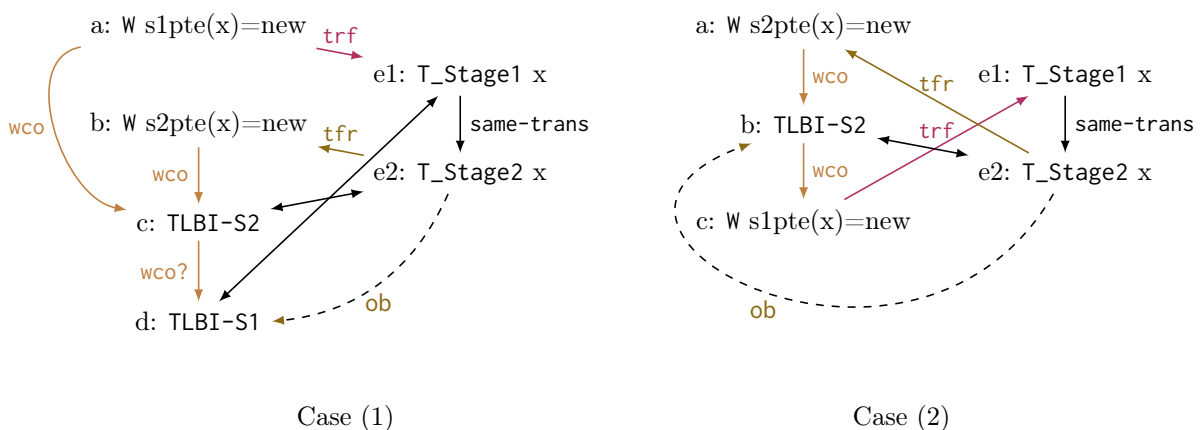

Figure 9.4: `obtlbi` stage 2 scenarios.

We capture the general shape of (1), where a translation-read may have been cached in the TLB, with the following `maybe_TLB_cached` relation:

```
1 let maybe_TLB_cached = ([T]; trf⁻¹; wco; [TLBI]) & tlb-
    affects⁻¹
```

We then use this relation to add ordering from a stage 2 translation-read to the stage 1 TLBI, wco-after a stage 2 TLBI that removed any stale IPA mappings, which would remove any cached whole-translation any stage 1 translation-read might have read from, and after which any fresh translation table walk would be required to not see the stale stage 2 entry the translation-read read from.

Broadcast `TLBIs`   Recall that broadcast TLBIs impose restrictions on other threads (§8.6.3). When a broadcast TLBI's invalidation affects a translation on another core, then it must also affect the explicit memory effect associated with it. This shape is illustrated in Figure 9.5, and corresponds to the final clause of `obtlbi`.
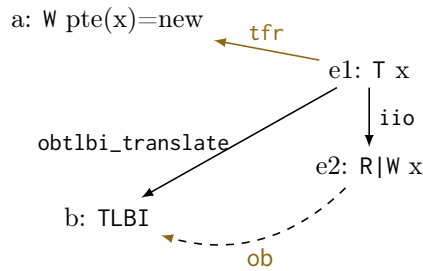

Figure 9.5: `obtlbi` broadcast TLBI shape.

Connecting TLB invalidations to translation reads   The final part of the puzzle is how to relate TLBI events with translations which may be affected by the invalidation. Recall that the TLBIs are grouped into subsets of `TLBI-S1`, `TLBI-VA`, and so on. We define a `tlb_might_affect` that is the cross-product of these with the `same-*` relations:

```
1 let tlb_might_affect =
2   [ TLBI-S1 & ~TLBI-S2 &  TLBI-VA  &  TLBI-ASID & TLBI-VMID]
      ; (same-va-page & same-asid & same-vmid) ; [T & Stage1]
3 | [ TLBI-S1 & ~TLBI-S2 & ~TLBI-VA  &  TLBI-ASID & TLBI-VMID]
      ; (same-asid & same-vmid) ; [T & Stage1]
4 | [ TLBI-S1 & ~TLBI-S2 & ~TLBI-VA  & ~TLBI-ASID & TLBI-VMID]
      ; same-vmid ; [T & Stage1]
5 | [~TLBI-S1 &  TLBI-S2 &  TLBI-IPA & ~TLBI-ASID & TLBI-VMID]
      ; (same-ipa-page & same-vmid) ; [T & Stage2]
6 | [~TLBI-S1 &  TLBI-S2 & ~TLBI-IPA & ~TLBI-ASID & TLBI-VMID]
      ; same-vmid ; [T & Stage2]
7 | [ TLBI-S1 &  TLBI-S2 & ~TLBI-IPA & ~TLBI-ASID & TLBI-VMID]
      ; same-vmid ; [T]
8 | ( TLBI-S1 &            ~TLBI-IPA & ~TLBI-ASID & ~TLBI-VMID)
      * (T & Stage1)
9 | (           TLBI-S2 & ~TLBI-IPA & ~TLBI-ASID & ~TLBI-VMID)
      * (T & Stage2)
```

Finally, we get `tlb-affects` by attaching `tlb_might_affect` to events in the same thread, and if a `TLBI-IS`, to ones in other threads too:

```
1 let tlb-affects =
2   [TLBI-IS]; tlb_might_affect
3 | ([~TLBI-IS]; tlb_might_affect) & int
```

```
1  (* F for all fences *)
2  accessor F: bool = is sail_barrier
3
4  (* from the Arm ASLl, compiled to sail *)
5  union Barrier = {
6    Barrier_DSB : DxB,
7    Barrier_DMB : DxB, // The nXS field is ignored from DMBs
8    Barrier_ISB : unit,
9    Barrier_SSBB : unit,
10   Barrier_PSSBB : unit,
11   Barrier_SB : unit,
12  }
13
14  (* accessors for each relevant constructor *)
15  accessor z_dmb: bool =
16      .match {
17          Barrier_DMB => true,
18          _ => false
19      }
20
21  accessor z_dsb: bool =
22      .match {
23          Barrier_DSB => true,
24          _ => false
25      }
26
27  accessor z_isb: bool =
28      .match {
29          Barrier_ISB => true,
30          _ => false
31      }
32
33
34  (* cat event sets for the different barriers *)
35  define DMB(ev: Event): bool =
36      F(ev) & z_dmb(ev)
37
38  define DSB(ev: Event): bool =
39      F(ev) & z_dsb(ev)
40
41  define ISB(ev: Event): bool =
42      F(ev) & z_isb(ev)
```

Figure 9.6: `isla-cat` accessors for Arm barriers.

9.5 Interface

To support the new Armv9 ISA and the new concurrency interface, we produce architecture-specific definitions using the new `isla-cat` language features in `isla-axiomatic`.

Barriers are instances of the `sail_barrier` outcome. For Arm we instantiate these with the Arm `Barrier` union.

Figure 9.6 contains the `isla-cat` definitions for the Arm barriers. The sail Arm `Barrier` union is reproduced here for the reader's benefit but is not required (nor present) in the source cat file. Similar unions, structs, enums and corresponding accessors and definitions exist for the Arm barrier domains (NSH, ISH, OSH) and access types (ST, LD, SY), elided here for brevity.

We make use of **accessor**s to access fields of the sail structs and unions, both here for barriers, and also for exceptions (faults), and TLBIs, as well as defining the `trf` and `wco` (found in Figure 9.7) relations. The full `isla-cat`-defined interface can be found in the Appendix**TODO: make appendix**.

```
1   declare wco(Event, Event): bool
2
3   (* wco has domain and range of W,CacheOp *)
4   assert forall ev1: Event, ev2: Event =>
5       wco(ev1, ev2) -->
6       (W(ev1) | C(ev1) | (ev1 == IW)) & (W(ev2) | C(ev2))
7
8   (* wco is transitive *)
9   assert forall ev1: Event, ev2: Event, ev3: Event =>
10      wco(ev1, ev2) & wco(ev2, ev3) --> wco(ev1, ev3)
11
12  (* wco is total *)
13  assert forall ev1: Event, ev2: Event, ev3: Event =>
14      wco(ev1, ev3) & wco(ev2, ev3) & ~(ev1 == ev2) -->
15      wco(ev1, ev2) | wco(ev2, ev1)
16
17  (* wco is irreflexive *)
18  assert forall ev1: Event, ev2: Event, ev3: Event =>
19      wco(ev1, ev2) --> ~(ev1 == ev2)
20
21  (* wco is antisymmetric *)
22  assert forall ev1: Event, ev2: Event =>
23      wco(ev1, ev2) --> ~wco(ev2, ev1)
24
25  (* all write/cache-op pairs are wco related *)
26  assert forall ev1: Event, ev2: Event =>
27      W(ev1) & C(ev2) -->
28      wco(ev1, ev2) | wco(ev2, ev1)
29
30  (* wco is consistent with co *)
31  assert forall ev1: Event, ev2: Event =>
32      co(ev1, ev2) --> wco(ev1, ev2)
33
34  (* all C are wco after IW
35   * n.b. all W are wco after IW, because all W are co after IW
         and co => wco
36   *)
37  assert forall ev: Event =>
38      C(ev) --> wco(IW, ev)
```

Figure 9.7: `wco.cat`: isla-cat definition of wco.

Validating the VMSA model

10.1 Validation against the architecture

To ensure the proposed virtual memory model presented in Ch. 9 correctly captures the architectural intent where known we engage in detailed discussions with Arm.

The model is produced through an iterative process, where the production of interesting litmus tests, guided by hardware testing and surveying of software requirements, are presented to, and discussed with, Arm architects. The desired architectural intent for those tests become known, new models are created and those models inform new interesting tests to produce.

Ideally we would run this process to a fixed point, however that is not always possible. Here, we know the model presented in Ch. 9 is incomplete and the litmus tests presented in Ch. 8 are not exhaustive. More work needs to, and is, being done to further update the models.

10.1.1 Clarity of architecture

We claim that the litmus tests presented in Ch. 8 have known architectural intent, and (as will be discussed in the following sections) the presented model correctly captures that intent for those tests.

For some of these behaviours it seems unlikely for the architectural intent to change. Specifically, the guarantess given by the break (e.g. RBS), break-before-make sequences, and general TLB-maintenance shapes, are fundamental to the security and correctness of modern software and so are highly unlikely to weaken over time.

Some of the behaviours arise as consequences of other parts of the design, specifically around TLB fills (§8.5.2) where the strength of the fill itself arises from a historical design of the processors and not a fundamental software requirement. As modern hardware advances, Arm add features to specifically weaken those areas (such as with FEAT_nTLBPA).

Conversely, many of the relaxed behaviours may see changes as the architecture evolves. We already saw how the introduction of FEAT_ETS strengthened some aspects of the architecture, and features such as ETS are still in-flux and there seems no reason to believe that Arm have settled on the final design. Hopefully the questions raised in this thesis have helped guide Arm in that design, towards a more stable architecture.

10.1.2 Remaining questions and updates

There are a number of places where the model as presented lacks the underlying architectural clarity to yet give more precise bounds on the architectural envelope.

There are a few places this is apparent in the model presented here:

▷ CONSTRAINEDUNPREDICTABLE behaviours due to TLB conflicts (break-before-make violations).
▷ Architectural features such as FEAT_nTLBPA, FEAT_ETS2, FEAT_TTL, and FEAT_BBM.
▷ Caching of access permissions, memory types, shareability and so on.
▷ Sharing TLBs betwen PEs.
▷ Caching of non-last-level block entries in the TLB.

The first of those are self-evident, where more discussions with architects are required to be able to present a model with any confidence.

The last one (caching of non-last-level block entries) is more interesting, and represents a gap in the model presented in the previous chapter. When an block entry is cached in the TLB, the hardware has a choice between caching entries per-page or only one for the whole block. The model currently is too weak, allowing separately cached entries per-page, and the architectural intent is now clearly to ensure that TLB invalidations would remove any cached entries for the whole block. **TODO: Say more?**

10.2 Validating against hardware: `system-litmus-harness`

Hardware testing is an important aspect in gaining confidence in any relaxed memory model, and without thorough evaluation of a range of microarchitecture it would not be possible to make strong claims of soundness of such a model.

However, testing systems-level features on hardware is much more challenging than the previous user-level features (including for instruction fetch as the required cache maintenance instructions were all unprivileged). Testing virtual memory requires a setup running at least at EL1, both to be able to run the TLB maintenance instructions and to enable catching of any generated exceptions.

An obvious choice for this would be `klitmus7`, an experimental version of `litmus` **TODO: ?CITE?** which produces a kernel module that runs at EL1. However, kernel modules run as a part of the Linux kernel and any attempts to modify the currently in-use translation tables or exception vectors would interfere with Linux's operations. Using `klitmus7` would therefore require a custom kernel as well as test infrastructure.

Instead, we build a brand new test harness designed for running tests which use systems features such as TLB maintenance and exception handlers: https://github.com/rems-project/system-litmus-harness.

Limitations Some limitations upfront: (1) the harness runs at EL1 and (for now) cannot run tests at EL2; (2) we do not check for known CPU errata for the device being ran on, instead relying on extra-defensive programming; (3) while the harness can run with QEMU/KVM on any device, running it bare metal (without a VM) is supported on only a limited number of devices; and (4) the harness currently uses an ad-hoc litmus test format which is not unified with either `isla-axiomatic` or `litmus7` itself.

We do not believe any of these are fundamental, but are solvable with additional engineering resources dedicated to the project.

10.2.1 Harness overview

`system-litmus-harness` is a relatively simple micro-kernel which runs at EL1. It has built-in a set of litmus tests, with fixed code for each thread and an ad-hoc language for describing the initial state. The user gives the harness (as arguments at boot) the name(s) of litmus tests to run and other run configuration options, and then the harness will run the litmus tests, collect the results, and echo those results back to the user (through the serial output, which can be directed to stdout in QEMU).

The structure of the test runner inside the harness is in a typical *litmus* style. It runs the tests in batches, executing each thread in a loop, where each iteration of the loop operates on a different set of locations making each iteration independent from one another.

Litmus test format Figure 10.1 gives an example litmus test in the `system-litmus-harness` format. Litmus tests are dedicated C files which define a `litmus_test_t` struct containing the litmus test data. The test displayed here can be found at https://github.com/rems-project/system-litmus-harness/blob/master/litmus/litmus_tests/pgtable/CoTR.inv%2Bdsb-isb.c.

The header VARS and REGS define the global variables to allocate (in this case, we want two, named x and y), and the names of output variables (which we usually style after the names of the machine registers which store them) for the final register values to save from the test.

The test then defines two threads with two static functions P0 and P1, both take some data stored in a `litmus_test_run` struct which contain the virtual addresses of each of the global and output variables.

```c
1  #include "lib.h"
2
3  #define VARS x, y
4  #define REGS p1x0, p1x2
5
6  static void P0(litmus_test_run* data)
       {
7    asm volatile (
8      /* setup */
9      "mov x0, %[ydesc]\n\t"
10     "mov x1, %[xpte]\n\t"
11     /* code */
12     "str x0, [x1]\n\t"
13     :
14     : ASM_VARS(data, VARS),
15       ASM_REGS(data, REGS)
16     : "cc", "memory", "x0", "x1"
17     );
18  }
19
20  static void sync_handler(void) {
21    asm volatile (
22      "mov x0, #0\n\t"
23
24      ERET_TO_NEXT(x10)
25     );
26  }
27
28  static void P1(litmus_test_run* data)
       {
29    asm volatile (
30      /* setup */
31      "mov x1, %[x]\n\t"
32      "mov x3, %[xpte]\n\t"
33      /* code */
34      "ldr x0, [x1]\n\t"
35      "dsb sy\n\t"
36      "isb\n\t"
37      "ldr x2, [x3]\n\t"

38      /* teardown */
39      "str x0, [%[outp1r0]]\n\t"
40      "cbz x2, .after\n\t"
41      "mov x2,#1\n\t"
42      ".after:\n\t"
43      "str x2, [%[outp1r2]]\n\t"
44     :
45     : ASM_VARS(data, VARS),
46       ASM_REGS(data, REGS)
47     : "cc", "memory", "x0", "x1",
48       "x2", "x3", "x10"
49     );
50  }
51
52  litmus_test_t CoTRinv_dsbisb = {
53    "CoTR.inv+dsb-isb",
54    MAKE_THREADS(2),
55    MAKE_VARS(VARS),
56    MAKE_REGS(REGS),
57    INIT_STATE(
58      2,
59      INIT_UNMAPPED(x),
60      INIT_VAR(y, 1)
61    ),
62    .interesting_result = (u64[]){
63        /* p0:x0 =*/1,
64        /* p0:x2 =*/0,
65    },
66    .thread_sync_handlers =
67      (u32**[]){
68      (u32*[]){NULL, NULL},
69      (u32*[]){(u32*)sync_handler,
        NULL},
70      },
71    .requires_pgtable = 1,
72    .no_sc_results = 3,
73  };
```

Figure 10.1: CoTR.inv+dsb-isb litmus test, `system-litmus-harness` source.

Taking the code for P1 as an example, it is just an `asm` block which contains the test code sandwiched between some setup and teardown code that moves values from the C code into and out of the machine registers the test uses.

There is an exception handler, `sync_handler` which simply resets `x0` to 0, and then performs an `ERET` to the next instruction address (that is, to ELR+4).

The final block of the test is the `litmus_test_t` struct definition for the test. It gives the name, the number of threads, the global and output variables, which exception handlers to install for each thread, the particular relaxed result to mark, and the initial machine state to run the test from. In this case, the initial state says that x starts unmapped (invalid at level 3) and y is mapped to a location that contains the value 1. Implicitly global variables have virtual addresses in distinct pages.

Litmus test format reference

The test format supports writing a variety of kinds of pagetable tests, through both the initial state setup and the data passed from the harness allocator via the `litmus_test_run` data struct. Appendix ?? describes the test format in full.

As an example, take the INIT_STATE from the ROT1+dsb-dsb-tlbi-dsb test[1], which defines three variables x, y and z. Its initial state is reproduced in Figure 10.2 for convenience. It says that they all start out mapped with initial

---

[1]which can be found at https://github.com/rems-project/system-litmus-harness/blob/master/litmus/litmus_tests/

values 0, 1 and 2 respectively (L13-15). Next it tells the allocator that x should be allocated in its own 2MiB region (L16), but to nevertheless place y in that region too (L17) with the same page offset (overlaps in the last 12 bits) as x (L18). Finally, it tells the allocator to place z in its own 2MiB region, with the same PMD offset (bits 20-12) as x has (L20). This ensures that bits 12-0 overlap for x and y, and bits 20-12 overlap for x and z, and therefore the table containing the entry for y can be assigned to the level 2 entry for x, as required by the ROT test shape (see §8.4.8).

```
1   #define VARS x, y, z
2   #define REGS p0x4
3
4   /* see source for full test */
5
6   litmus_test_t ROT1_dsbtlbidsb = {
7     "ROT1+dsb-dsb-tlbi-dsb",
8     MAKE_THREADS(1),
9     MAKE_VARS(VARS),
10    MAKE_REGS(REGS),
11    INIT_STATE(
12      8,
13      INIT_VAR(x, 0),
14      INIT_VAR(y, 1),
15      INIT_VAR(z, 2),
16      INIT_REGION_OWN(x, REGION_OWN_PMD),
17      INIT_REGION_PIN(y, x, REGION_SAME_PMD),
18      INIT_REGION_OFFSET(y, x, REGION_SAME_PAGE_OFFSET),
19      INIT_REGION_OWN(z, REGION_OWN_PMD),
20      INIT_REGION_OFFSET(z, x, REGION_SAME_PMD_OFFSET),
21    ),
22    .interesting_result = (u64[]){
23        /* p0:x2 =*/1,
24    },
25    .start_els = (int[]){1},
26    .requires_pgtable = 1,
27    .no_sc_results = 2,
28  };
```

Figure 10.2: system-litmus-harness initial state for an ROT-shaped test.

10.2.2 Results from hardware

We ran a collection of hand-written litmus tests on three hardware devices using system-litmus-harness running inside KVM: a Raspberry Pi 4; a Raspberry Pi 3B+; and an AWS `m6g-metal` instance (claiming to be an A76). Note that the hardware tests are an overlapping set of tests than those presented in Ch. 8, some contain BBM violations, and some tests are not reproduced on hardware, and ones that are may appear with slightly different names (for example, CoWTf.inv+dmb test (Figure 8.19, p127) appears in the table as CoWT.inv+dmb). Tables 10.1 and 10.2 list the total results for all the tests from all three devices. **TODO: Go through and add backlinks to tests previously discussed**

Our testing revealed some incompatibilities between the architectural intent and the current implementations. For some break-before-make sequences, such as test MP.BBM1+dsb-tlbiis-dsb-dsb-isb+dsb-isb (architecturally forbidden, experimentally observed) there were some very rare violations observed. The related MP.BBM1+[dmb.ld]-tlbiis-dsb-isb-dsb-isb+dsb-isb test (with a detour after the write) is never observed, suggesting it is related to the DSB not fully propagating the store, which is consistent with related CPU errata **TODO: Quote which ones**. These anomalous results have been reported, and are under investigation by Arm.

10.3 Validation by abstraction

We cannot 'prove' that the model is correct. Correctness of a relaxed memory model like this depends on the whims of the architects, and may change as new revisions of the architecture are released. However, we can

---

pgtable/pmds/ROT1%2Bdsb-dsb-tlbi-dsb.c

Table 10.1: system-litmus-harness hardware results from three devices: Part I.

Type	Name	rpi4b	rpi3bp	graviton2
pgtable	CoRT	964.72K/8M	520.06K/3M	2.29M/108M
pgtable	CoRT+dsb-isb	802.86K/8M	327.02K/3M	3.41M/108M
pgtable	CoTR	2.51M/8M	0/3M	21.70M/107.50M
pgtable	CoTR+addr	0/8M	1/3M	0/107.50M
pgtable	CoTR+dmb	1/8M	0/3M	4/107.50M
pgtable	CoTR+dsb	2/8M	0/2.50M	5/107M
pgtable	CoTR+dsb-isb	1/8M	0/2.50M	1/107M
pgtable	CoTR.inv	3.63M/6.50M	0/2.50M	32.28M/43M
pgtable	CoTR.inv+dsb-isb	0/6.50M	0/2.50M	0/43M
pgtable	CoTR1+dsb-dc-dsb-tlbi-dsb-isb	2/6.50M	0/2.50M	4/43M
pgtable	CoTR1+dsb-tlbi-dsb-isb	2/6.50M	0/2.50M	3/43M
pgtable	CoTR1.tlbi+dsb-isb	6/6.50M	1/2.50M	29/43M
pgtable	CoTT	0/6.50M	0/2M	0/43M
pgtable	CoTW	0/1.50M	0/1.50M	0/10.50M
pgtable	CoWT	3.77M/6.50M	1.85M/2M	22.64M/43M
pgtable	CoWT+dsb	3.76M/6.50M	995.06K/2M	21.50M/43M
pgtable	CoWT+dsb-isb	3.78M/6.50M	995.77K/2M	21.50M/43M
pgtable	CoWT+dsb-svc-tlbi-dsb	0/6.50M	0/2M	0/42.50M
pgtable	CoWT.inv	10/6.50M	1.73M/2M	169/42.50M
pgtable	CoWT.inv+dmb	8/6.50M	69.38K/2M	42/42.50M
pgtable	CoWT.inv+dsb	1/6.50M	0/2M	57/42M
pgtable	CoWT.inv+dsb-isb	0/6.50M	0/2M	0/42M
pgtable	CoWT1+dsb-tlbi-dsb	0/6.50M	0/2M	0/42.50M
pgtable	CoWT1+dsb-tlbi-dsb-isb	0/6.50M	0/2M	0/42.50M
pgtable	CoWinvT	4.17M/6.50M	1.79M/2M	26.81M/42M
pgtable	CoWinvT+dsb-isb	4.19M/6.50M	1.83M/2M	26.80M/42M
pgtable	CoWinvT1+dsb-tlbi-dsb	0/6.50M	0/2M	0/42M
pgtable	CoWinvWT1+dsb-tlbi-dsb-dsb-isb	0/6.50M	0/2M	0/42M
pgtable	ISA2.TRR+dmb+po+dmb	0/6.50M	0/2M	0/42M
pgtable	MP.BBM1+[dmb.ld]-dsb-tlbiis-dsb-isb-dsb-isb+dsb-isb	0/108.50M	0/1.50M	0/437.50M
pgtable	MP.BBM1+[dmb.ld]-tlbiis-dsb-isb-dsb-isb+dsb-isb	0/198.50M	0/1.06G	0/129.50M
pgtable	MP.BBM1+[po]-dsb-tlbiis-dsb-isb-dsb-isb+dsb-isb	0/108.50M	0/1.50M	0/145.50M
pgtable	MP.BBM1+dsb-isb-tlbiis-dsb-isb-dsb-isb+dsb-isb	0/6.50M	0/2M	52/135.50M
pgtable	MP.BBM1+dsb-tlbiis-dsb-dsb+dsb	1/6.50M	0/2M	7/42.50M
pgtable	MP.BBM1+dsb-tlbiis-dsb-dsb+dsb-isb	0/6.50M	0/2M	2/42.50M
pgtable	MP.BBM1+dsb-tlbiis-dsb-dsb-isb+dsb	1/6M	0/2M	0/42.50M
pgtable	MP.BBM1+dsb-tlbiis-dsb-dsb-isb+dsb-isb	2/6M	0/2M	3/42.50M
pgtable	MP.BBM1+po-dsb-tlbiis-dsb-isb-dsb-isb+dsb-isb	0/1M	0/1.50M	9/191.50M
pgtable	MP.BBM1.id+dsb-tlbiis-dsb-dsb+dsb-isb	10/6M	2/2M	87/42.50M
pgtable	MP.RT+svc-dsb-tlbi-dsb+dsb-isb	1/6M	0/2M	3/42M
pgtable	MP.RT+svc-dsb-tlbiis-dsb+dsb-isb	1/6M	0/2M	3/42M
pgtable	MP.RT.inv+dmb+addr	0/6M	0/2M	0/42M
pgtable	MP.RT.inv+dmb+po	0/6M	6/1.50M	0/42M
pgtable	MP.RT1+[dmb.ld]-dmb+dsb-isb	7.15K/6M	986/1.50M	1.26K/42M
pgtable	MP.RT1+[dmb.ld]-dsb-isb-tlbiis-dsb-isb+dmb	0/1M	0/1M	0/23M
pgtable	MP.RT1+[dmb.ld]-dsb-isb-tlbiis-dsb-isb+dsb-isb	0/1M	0/1M	0/23M
pgtable	MP.RT1+[dmb.ld]-dsb-tlbiis-dsb-isb+dmb	0/6M	0/1.50M	0/42M
pgtable	MP.RT1+dc-dsb-tlbiall-dsb+dsb-isb	4/6M	1/1.50M	5/41.50M
pgtable	MP.RT1+dc-dsb-tlbiall-dsb-isb+dsb-isb	3/6M	0/1.50M	2/41.50M
pgtable	MP.RT1+dsb-isb-tlbiis-dsb-isb+dsb-isb	0/6M	0/1.50M	4/41M
pgtable	MP.RT1+dsb-tlbi-dsb+dsb-isb	0/6M	0/1.50M	2/41M
pgtable	MP.RT1+dsb-tlbiall-dsb+dsb-isb	5/6M	0/1.50M	6/41M
pgtable	MP.RT1+dsb-tlbiallis-dsb+dsb-isb	3/6M	0/1.50M	2/41M
pgtable	MP.RT1+dsb-tlbiis-dsb+dsb-isb	1/6M	0/1.50M	1/41M

Table 10.2: system-litmus-harness hardware results from three devices: Part II.

Type	Name	rpi4b	rpi3bp	graviton2
pgtable	MP.RT1+dsb-tlbiis-dsb-isb+dmb	0/6M	0/1.50M	1/41M
pgtable	MP.RT1+dsb-tlbiis-dsb-isb+dsb-isb	0/6M	0/1.50M	1/41M
pgtable	MP.RT1+dsb-tlbiis-dsb-tlbiis-dsb+dsb-isb	0/6M	0/1.50M	3/41M
pgtable	MP.TT+Winv-dmb-Winv+tpo	254.83K/6M	114.48K/1.50M	170.96K/41M
pgtable	MP.TT+dmb+dsb-isb	688.65K/5.50M	174.78K/1.50M	492.98K/41M
pgtable	MP.TT+dmb+tpo	843.79K/5.50M	157.80K/1.50M	480.31K/41M
pgtable	MP.TT.inv+dmb+dsb-isb	0/5.50M	0/1.50M	0/41M
pgtable	MP.TT.inv+dmb+tpo	0/5.50M	0/1.50M	0/41M
pgtable	MP.invRT+dsb+dsb-isb	871.53K/5M	101.75K/1.50M	1.78M/40.50M
pgtable	MP.invRT1+dsb-isb-tlbiis-dsb-isb+dsb-isb	0/5.50M	0/1.50M	1/41M
pgtable	MP.invRT1+dsb-tlbiis-dsb+dsb	0/5M	0/1.50M	2/41M
pgtable	MP.invRT1+dsb-tlbiis-dsb+dsb-isb	1/4.50M	0/1.50M	1/41M
pgtable	WRC.AT+ctrl+dsb	128.64K/4.50M	77.36K/1.50M	214.45K/40M
pgtable	WRC.TRR+addr+dmb	0/4.50M	0/1.50M	0/40M
pgtable	WRC.TRR.inv+addrs	0/4.50M	0/1.50M	0/40M
pgtable	WRC.TRT+addr+dmb	35.28K/4.50M	32.50K/1.50M	103.16K/40M
pgtable	WRC.TRT+dmbs	53.60K/4.50M	36.76K/1.50M	171.51K/40M
pgtable	WRC.TRT+dsb-isbs	18.80K/4.50M	30.44K/1.50M	104.62K/39.50M
pgtable	WRC.TRT.inv+addrs	0/4M	0/1.50M	0/38.50M
pgtable	WRC.TRT.inv+dsb-isbs	0/4M	0/1M	0/38M
pgtable	WRC.TRT.inv+po+addr	0/4M	0/1M	0/37.50M
pgtable	WRC.TRT.inv+po+dmb	0/4M	0/1M	0/37M
pgtable	WRC.TRT1+dsb-tlbiis-dsb+dmb	0/4.50M	0/1M	0/38M
pgtable	WRC.TRT1+dsb-tlbiis-dsb+dsb-isb	0/4.50M	0/1M	0/38M
aliasing	CoWR.alias	0/6M	0/1.50M	0/36M
aliasing	MP+dmb-data+dmb	0/5M	0/1.50M	0/36M
aliasing	MP.alias+dmbs	0/5M	0/1.50M	0/36M
aliasing	MP.alias2+dmb-data+dmb	0/5M	0/1.50M	0/36M
aliasing	MP.alias2+dmbs	0/3M	0/1.50M	0/19.50M
aliasing	MP.alias2+po-data+dmb	2.23K/5M	3.17K/1.50M	407.36K/36M
aliasing	MP.alias3+rfi-data+dmb	51/3M	16/1.50M	36.35K/19.50M
aliasing	SB.alias+dmbs	0/5M	0/1M	0/35.50M
aliasing	WRC.alias2+addrs	0/4M	0/43M	0/19M
aliasing	WRC.alias2+dmbs	0/4M	0/43M	0/18.50M
cacheability	MP.NC+dsb-dc-dsb-dmb+dmb	138.80K/8M	364.97K/26M	54.95K/25.50M
cacheability	MP.NC+po-dmb+dmb	345.33K/7.50M	642.90K/25.50M	333.55K/25.50M
cacheability	MP.NC1+dsb-tlbiis-dsb-dc-dsb-dmb+dmb	0/7.50M	0/25.50M	0/25.50M
cacheability	MP.NC1+dsb-tlbiis-dsb-dmb+dmb	556/7.50M	482/25.50M	6/25.50M
cacheability	WR.NC+dsb	0/0	0/0	0/0
cacheability	WR.NC+po	0/0	0/0	0/0
cacheability	WR.WARA-NC+dsb	0/0	0/0	0/0
cacheability	WR.WARA-NC+po	0/0	0/0	0/0
cacheability	WWR.NC+po-po	0/0	0/0	0/0
pmds	CoWT.L23+dsb-isb	11.45M/13M	6.73M/13.50M	48.94M/84.50M
pmds	CoWT.L23+po	12.88M/13M	13.39M/13.50M	80.61M/84.50M
pmds	CoWT1.L23+dsb-tlbi-dsb-isb	0/13M	0/13.50M	0/84.50M
pmds	ROT+dsb-dsb	0/13M	0/13.50M	0/84.50M
pmds	ROT+po-po	0/13M	0/13.50M	0/84M
pmds	ROT1+dsb-dsb-tlbi-dsb	0/13M	0/13.50M	0/84M
pmds	ROT1+dsb-dsb-tlbivaa-dsb	0/13M	0/13.50M	0/84M
same_page	CoTT+dsb-popage	0/35.50M	0/31M	0/1.12G
same_page	CoTT+po-popage	1/47M	0/43.50M	0/1.20G
sysreg	WR.MAIR1+dsb-isb-dc-dsb	0/0	0/0	0/0
sysreg	WR.MAIR1+dsb-isb-po	0/0	0/0	0/0
sysreg	WR.MAIR1+dsb-tlbi-dsb-isb-dc-dsb	0/0	0/0	0/0
sysreg	WR.MAIR1+dsb-tlbi-dsb-isb-po	0/0	0/0	0/0
sysreg	WR.MAIR1+po-po	0/0	0/0	0/0

10.3. VALIDATION BY ABSTRACTION

identify properties we believe *any* sound model would have, and check that the model presented here has those properties.

The key property that we will prove the presented model has is a 'virtual memory abstraction'. There is no single definition of what such an abstraction is defined to be, but we give one intuitive and informal definition: a program with a fixed injective translation table mapping behaves as if executing above physical memory directly.

10.3.1 Precise statement

We can state the virtual memory abstraction as a property over candidate executions. If given a full (with all the translation table walk events) well-formed (consistent with the intra-instruction semantics) candidate C, with no TLBI events, no T_f events, and no W events to any pagetable location, then, the candidate is consistent in the VMSA model if and only if the translation-erased candidate $C^{\sim T}$ (see below) is consistent in the base model.

Translation-erasure   We define a *translation erasure* operation over candidate executions.

Given a candidate C the translation-erased candidate $C^{\sim T}$ is C but where all TLBI, T, and T_f events are erased, and any edge containing such events as source or target are removed from the candidate relations, and including in $C^{\sim T}$ the derived relations addr and po from C.

10.3.2 Proof

Informally the proof is a straight-forward inclusion proof by relation algebra. The internal (and translation-internal) and atomic axioms are trivially subset inclusions of one another under translation erasure. For external, we show that ob in the base model implies ob in the VMSA model and that ob in the VMSA model implies the same ob in the base model. Therefore they must forbid the same cycles.

See **TODO: Appendix A** for the full proof. **TODO: Include JPP proof**

Conclusion

We presented models for two key parts of the Arm architecture required for systems software, for instruction fetch and required cache maintenance instructions, and virtual memory and its required TLB maintenance instructions. We have produced a corpus of hand-written litmus tests for these architectural aspects, covering a range of interesting hardware optimisations and software requirements. We've clarified the architecture by extracting the architectural intent for those tests, especially where that intent was not clear beforehand, and produced models that capture that intent.

We produced axiomatic-style declarative semantics, based on the herdtools 'cat' language, for both aspects of the architecture. Additionally we produced a microarchitectural-style operational semantics for the instruction fetch fragmnet intended equivalent to the axiomatic one.

These models were validated against hardware implementations, finding places where modern microprocessors deviate from the desired architectural intent. For instruction fetch we extended the herdtool's suite to be able to generate new litmus tests, and run those tests on hardware. We built a brand new test runner, able to run tests on a variety of hardware at EL1, either bare metal or in KVM.

We made these models executable as a test oracle, allowing the user to experimentally check behaviours manually or even do rudimentary model checking of a larger software pattern, by implementing them in our `isla-axiomatic` or RMEM tools. This allowed us to validate the models against each other where applicable, and against the architectural intent, and comparing the results from hardware test runs against the model's predictions.

Finally, for virtual memory we proved a simple virtual memory abstraction which gives confidence that the model correctly captures a key property the model is intended to have.

11.1 Limitations

While we endeavour to be as faithful to the architectural intent as we can and to produce models that are sound abstractions of that intent, we have had to make tradeoffs in places.

We presented two models for two separate parts of the architecture, instruction fetching and virtual memory, but did not merge them together into a single architectural model. The two models can be unioned together to produce a combined model with all the events and relations from both, although more work is needed to understand the interactions between the architectural features: instruction fetches are memory reads which themselves are translated but where that translation behaves subtley different from the normal translations with different caching rules. We do not imagine this is a hugely complex task, but one that we have not yet done.

We have seen three separate languages for defining litmus tests so far. Ideally we would have one unified language that all tools (litmus, isla-axiomatic and system-litmus-harness) all accepted. As stated earlier, we do not believe there is a fundamental restriction to unifying these languages, as currently they have not diverged so far as to be incompatible.

TODO: Some words about virtual memory model and RG/Thibuat questions TODO: Some words about interface

11.2 Future work

There are many areas where the work presented here is only the start, and where further effort could yield fruit.

For more confidence in the architectural intent more hardware testing (especially for the virtual memory tests) is essential, especially running at EL2 (for stage 2 tests) and over more varied devices.

Capturing more of the architecture is always desireable, we made a start here but this is no means the end, and modern systems software relies on much more of the architecture than just covered here, such as: interrupts and exceptions **TODO: del?**, the variety of Arm features for virtual memory (FEAT_ETS2, FEAT_BBM, FEAT_nTLBPA), access permissions and caceheability and shareability domains, device memory and DMA, and much more.

With the models themselves, they can always be improved to be more performant and the tools more usable. `isla-axiomatic` can run the virtual memory tests, but needs optimisations to be able to run in any reasonable timeframe, and even then still takes hours on a modern high-end machine. This seriously restricts the current usefulness of such tools to the average programmer.

There are now many concurrently existing models for Arm for a variety of features, we present two here (instruction fetching and virtual memory) but also from the wider community for persistent memory, memory tagging, access bits and dirty flags, capabilities and probably many others. Simply gluing these together into a single model is not sound, as there are many interactions that would need to be explored and the architectural intent clarified first. However, it seems a necessity that such work is carried out to enable future verification efforts of complex systems.

Work on relaxed virtual memory, on relaxed instruction fetching, and even further afar, has not ceased at the finalization of this work. We are continuing to improve on the models given here, to engage in fruitful discussions with Arm, to produce new models for more of the architecture, and to build more confidence in the models we have already created.

Hopefully this work enables future researchers, academics, engineers, architects, and hardware designers to better understand the environment as it is today, and to produce clearer and more robust architectures and to take the first steps in verifying the complex systems software that underpins so much of the modern base of computing with respect to the reality of the hardware we run them on.

Test format: `system-litmus-harness`

The test format supports writing a variety of kinds of pagetable tests, through both the initial state setup and the data passed from the harness allocator via the `litmus_test_run` data struct.

The data struct contains, for each global variable (e.g. x): the virtual address (`%[x]`); the initial last-level descriptor (`%[xdesc]`); the address of the last-level entry (`%[xpte]`); the address of the entry at level N (`%[xpteN]`); the page index, e.g. for arguments to TLB maintenance (`%[xpage]`). With some aliases for the different levels to match Linux terminology: `%[xpmd]` for the level 2 entry (`xpte2`); `%[xpud]` for the level 1 entry (`xpte1`).

The initial state enables specifying a rich variety of related machine states, each `INIT_STATE` can include directives for the initial value of the variable:

> ▷ `INIT_UNMAPPED(var)`: that the pagetable entry for `var` starts out invalid.

> ▷ `INIT_VAR(var, value)`: that `var` starts out mapped and the location at its physical address starts out containing `value`.

> ▷ `INIT_ALIAS(var1, var2)`: that `var1` and `var2` should be aliased to the same location.

The programmer can also choose the initial permissions and memory attributes the variables are mapped with:

> ▷ `INIT_PERMISSIONS(var, prot, value)`: that `var` should be mapped with field `prot` set to `value`:

>   — for `prot=PROT_AP`, `value` can be any int, but there are some helpful aliases:

>   ∗ `PROT_AP_RWX_X` (0x0): read-write-execute at EL1, execute only at EL0.

>   ∗ `PROT_AP_RW_RWX` (0x1): read-write at EL1, read-write-execute at EL0.

>   ∗ `PROT_AP_RX_X` (0x2): read-execute at EL1, execute only at EL0.

>   ∗ `PROT_AP_RX_RX` (0x3): read-execute at EL1 and EL0.

>   — for `prot=PROT_ATTRIDX`, `value` defines the memory attributes as the index to the default MAIR value, and can be any of:

>   ∗ `PROT_ATTR_DEVICE_nGnRnE` (0): use strongly-ordered device memory.

>   ∗ `PROT_ATTR_DEVICE_GRE` (1): standard device memory (with re-ordering, gathering and early write acknowledgement).

>   ∗ `PROT_ATTR_NORMAL_NC` (2): normal non-cacheable memory.

>   ∗ `PROT_ATTR_NORMAL_RA_WA` (3): normal cacheable memory.

>   ∗ indexes 4-7 are unused.

> ▷ `INIT_MAIR(value)`: defines the otherwise unused `MemAttr7` field of the MAIR for custom tests.

>   — `MAIR_DEVICE_nGnRnE` (0x00): strongly ordered device memory.

>   — `MAIR_DEVICE_GRE` (0x0c): standard device memory (with re-ordering, gathering and early write acknowledgement).

>   — `MAIR_NORMAL_NC` (0x44): normal non-cacheable memory.

4134 — `MAIR_NORMAL_RA_WA` (`0xff`): normal cacheable memory.

4135 Finally, the harness allocator can be guided to place variables in locations with particular relationships between
4136 them (in the same page or cache line, or at the same offset into their respective regions):

4137 ▷ `INIT_REGION_OWN(var, region)`: that var owns a region of memory larger than the default of a page,
4138 `region` can take values:

4139 — `REGION_OWN_CACHE_LINE`: this variable only takes up a single cache line.

4140 — `REGION_OWN_PAGE`: don't allocate other variables in the same page (the default).

4141 — `REGION_OWN_PMD`: don't allocate other variables in the same 2MiB region.

4142 — `REGION_OWN_PUD`: don't allocate other variables in the same 1GiB region.

4143 ▷ `INIT_REGION_PIN(var1, var2, region)`: place var1 and var2 in the same region, where `region` is one
4144 of:

4145 — `REGION_SAME_CACHE_LINE`: place both in the same cache line.

4146 — `REGION_SAME_PAGE`: place both in same page.

4147 — `REGION_SAME_PMD`: place both same 2MiB region.

4148 — `REGION_SAME_PUD`: place both same 1GiB region.

4149 ▷ `INIT_REGION_OFFSET(var1, var2, region)`: ensure that var1 and var2 have the same *offset* into the
4150 region (that is, the least significant bits overlap), where `region` can be one of:

4151 — `REGION_SAME_CACHE_LINE_OFFSET`: ensure both have same lower `CACHE_LINE_SHIFT` bits.

4152 — `REGION_SAME_PAGE_OFFSET`: ensure both have same offset into the page (bits 12-0).

4153 — `REGION_SAME_PMD_OFFSET`: ensure both have same offset into the 2MiB region (bits 20-12).

4154 — `REGION_SAME_PUD_OFFSET`: ensure both have same offset into the 1GiB region (bits 29-20).

Bibliography

[1] 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.

[2] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *Proceedings of POPL 2009: the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 379–391, January 2009.

[3] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *Proceedings of TPHOLs 2009: Theorem Proving in Higher Order Logics, LNCS 5674*, pages 391–407, 2009.

[4] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010. (Research Highlights).

[5] Anthony C. J. Fox and Magnus O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, pages 243–258, 2010.

[6] Christopher Pulte. *The Semantics of Multicopy Atomic ARMv8 and RISC-V*. PhD thesis, University of Cambridge, 2019. https://doi.org/10.17863/CAM.39379.

[7] 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.

[8] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *Proceedings of PLDI 2011: the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 175–186, 2011.

[9] The RISC-V Foundation. Sail RISC-V model. https://github.com/riscv/sail-riscv (accessed 2023-05-24).

[10] Alastair Reid. Trustworthy specifications of ARM v8-A and v8-M system level architecture. In *FMCAD 2016*, pages 161–168, October 2016.

[11] Alastair Reid. ARM releases machine readable architecture specification. https://alastairreid.github.io/ARM-v8a-xml-release/, April 2017.

[12] 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.

[13] William W. Collier. *Reasoning about parallel architectures.* Prentice Hall, Englewood Cliffs, 1992.

[14] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In *Proc. DAMP 2009*, January 2009.

[15] 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

multiprocessors. In *Proc. MICRO-48, the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2015.

[16] 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.

[17] Azalea Raad and Viktor Vafeiadis. Persistence semantics for weak memory: Integrating epoch persistency with the tso memory model. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.

[18] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *ESOP*, 2007.

[19] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and Compiling C/C++ Concurrency: from C++11 to POWER. In *Proceedings of POPL 2012: The 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Philadelphia)*, pages 509–520, 2012.

[20] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising C/C++ and POWER. In *Proceedings of PLDI 2012, the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (Beijing)*, pages 311–322, 2012.

[21] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Prentice Hall Press, USA, 4th edition, 2014.

[22] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, Inc., USA, 3rd edition, 2021.

[23] S.V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996.

[24] William Pugh. The java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(6):445–455, 2000.

[25] RISC-V Foundation. The RISC-V instruction set manual, volume i: User-level ISA, document version 20191213. https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf, December 2019.

[26] *Power ISA^{TM} Version 2.07*. IBM, 2013.

[27] P. Becker, editor. *Programming Languages — C++*. 2011. ISO/IEC 14882:2011. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf.

[28] Mark John Batty. *The C11 and C++11 Concurrency Model*. PhD thesis, University of Cambridge Computer Laboratory, November 2014. Winner of 2015 ACM SIGPLAN John C. Reynolds Doctoral Dissertation Award and 2015 CPHC/BCS Distinguished Dissertation competition.

[29] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. The java language specification. https://docs.oracle.com/javase/specs/jls/se20/jls20.pdf, March 2023.

[30] std::sync::atomic - rust. https://doc.rust-lang.org/std/sync/atomic/index.html (accessed 2023-05-30).

[31] Data.ioref - hackage.haskell.org. https://hackage.haskell.org/package/base-4.18.0.0/docs/Data-IORef.html (accessed 2023-05-30).

[32] Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell. ARMv8-A system semantics: instruction fetch in relaxed architectures. In *Proceedings of the 29th European Symposium on Programming, ESOP 2020*, 2020.

[33] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. In *Proceedings of the 33rd International Conference on Computer Aided Verification, CAV 2021*, July 2021.

[34] Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. Relaxed virtual memory in Armv8-A. In *Proceedings of the 31st European Symposium on Programming, ESOP 2022*, April 2022.

BIBLIOGRAPHY

[35] Ben Simner, Ohad Kammar, Jean Pichon-Pharabod, and Peter Sewell. Relaxed exceptions in arm-a (pre-publication), May 2023.

[36] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models (extended version). *Formal Methods in System Design*, TODO(TODO):TODO, May 2023.

[37] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the ARM and POWER relaxed memory models, October 2012. Draft.

[38] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C-28(9):690–691, 1979.

[39] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM TOPLAS*, 36(2):7:1–7:74, July 2014.

[40] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. Armed cats: Formal concurrency modelling at arm. *ACM Trans. Program. Lang. Syst.*, 43(2), jul 2021.

[41] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[42] William W. Collier. *Reasoning about parallel architectures.* Prentice Hall, 1992.

[43] Arm Limited. Exploration Tools – Arm Developer. https://developer.arm.com/downloads/-/exploration-tools, 2022. Accessed May 2022.

[44] 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.

[45] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. In *In Proc. 33rd International Conference on Computer-Aided Verification*, July 2021. Extended version available at https://www.cl.cam.ac.uk/~pes20/isla/isla-cav2021-extended.pdf.

[46] 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.

[47] Scott Owens, Peter Böhm, Francesco Zappa Nardelli, and Peter Sewell. Lem: A lightweight tool for heavy-weight semantics. In *Proceedings of Interactive Theorem Proving – Second International Conference (previously TPHOLs) (Berg en Dal), LNCS 6898*, pages 363–369, 2011. (Rough Diamond).

[48] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: Reusable engineering of real-world semantics. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 175–188, New York, NY, USA, September 2014. ACM.

[49] Shaked Flur, Jon French, Kathryn Gray, Christopher Pulte, Susmit Sarkar, and Peter Sewell. rmem. www.cl.cam.ac.uk/~pes20/rmem/, 2017.

[50] Will Deacon. The ARMv8 application level memory model. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat (accessed 2019-07-01), 2016.

[51] Arm. The Armv9 application level memory model. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat (accessed 2023-12-21), 2016.

[52] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014.

[53] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 646–661, 2018.

[54] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 653–669, 2016.

[55] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 287–305, 2017.

[56] Roberto Guanciale, Hamed Nemati, Mads Dam, and Christoph Baumann. Provably secure memory isolation for linux on ARM. *J. Comput. Secur.*, 24(6):793–837, 2016.

[57] Christoph Baumann, Oliver Schwarz, and Mads Dam. Compositional verification of security properties for embedded execution platforms. In *PROOFS@CHES 2017, 6th International Workshop on Security Proofs for Embedded Systems, Taipei, Taiwan, Friday September 29th, 2017*, pages 1–16, 2017.

[58] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *J. Funct. Program.*, 29:e2, 2019.

[59] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192, 2014.

[60] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.

[61] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. *SIGPLAN Not.*, 42(6):66–77, jun 2007.

[62] Magnus O. Myreen. Verified just-in-time compiler on x86. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 107–118, New York, NY, USA, 2010. ACM.

[63] Jade Alglave, Luc Maranget, Kate Deplaix, Keryan Didier, and Susmit Sarkar. The litmus7 tool. http://diy.inria.fr/doc/litmus.html/, 2019. Accessed 2019-07-08.

[64] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: running tests against hardware. In *Proceedings of TACAS 2011: the 17th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 41–44, Berlin, Heidelberg, 2011. Springer-Verlag.

[65] Jade Alglave and Luc Maranget. The diy7 tool. http://diy.inria.fr/, 2019. Accessed 2021-07-01.

[66] 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.

[67] ARM Limited. ARM architecture reference manual. ARMv8, for ARMv8-A architecture profile, October 2018. v8.4. ARM DDI 0487D.a (ID103018).

[68] Arm Limited. Arm architecture reference manual. https://developer.arm.com/documentation/ddi0487/ja/?lang=en, April 2023. J.a Armv9 EAC. ARM DDI 0487J.a (ID042523). 12940pp.

[69] Arm Limited. Arm architecture reference manual. Armv8, for Armv8-A architecture profile. https://developer.arm.com/documentation/ddi0487/da/?lang=en, October 2017. D.a Armv8.4 EAC. ARM DDI 0487D.a (ID103018). 7476pp.

[70] Arm Limited. Arm architecture reference manual. Armv8, for Armv8-A architecture profile. https://developer.arm.com/documentation/ddi0487/ea/?lang=en, July 2019. E.a Armv8.5 EAC. ARM DDI 0487E.a (ID070919). 7900pp.

[71] Arm Limited. Arm architecture reference manual. Armv8, for Armv8-A architecture profile. https://developer.arm.com/documentation/ddi0487/gb/?lang=en, 2021. G.b Armv8.7 EAC. ARM DDI 0487G.b (ID072021). 8696pp.

[72] Arm Limited. Arm architecture reference manual. Armv8, for Armv8-A architecture profile. https://developer.arm.com/documentation/ddi0487/fa/?lang=en, February 2020. F.a Armv8.6 Beta. ARM DDI 0487F.a (ID021920). 8128pp.

[73] src/wasm/jump-table-assembler.cc - v8/v8.git - git at google.

[74] ea82d0311b3bd1ab9f92b59e5b5c51bfd5ef87b6 - v8/v8.git - git at google.

[75] Jim Handy. *The cache memory book.* Academic Press Professional, Inc., 1993.

[76] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the ARM and POWER relaxed memory models. Draft available from http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf, 2012.

[77] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 66–77, New York, NY, USA, 2007. Association for Computing Machinery.

[78] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In Theo D'Hondt, editor, *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, pages 504–528. Springer, 2010.

[79] Shilpi Goel and Warren A. Hunt. Automated code proofs on a formal model of the x86. In Ernie Cohen and Andrey Rybalchenko, editors, *Verified Software: Theories, Tools, Experiments*, pages 222–241, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[80] Shilpi Goel, Warren A. Hunt, Matt Kaufmann, and Soumava Ghosh. Simulation and formal verification of x86 machine-code programs that make system calls. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, FMCAD '14, pages 18:91–18:98, Austin, TX, 2014. FMCAD Inc.

[81] Shilpi Goel, Anna Slobodova, Rob Sumners, and Sol Swords. Verifying x86 instruction implementations. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 47–60, New York, NY, USA, 2020. Association for Computing Machinery.

[82] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. COATCheck: Verifying memory ordering at the hardware-OS interface. *SIGOPS Oper. Syst. Rev.*, 50(2):233–247, March 2016.

[83] S. Li, X. Li, R. Gu, J. Nieh, and J. Hui. A secure and formally verified linux KVM hypervisor. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 839–856, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[84] Azalea Raad, John Wickerson, and Viktor Vafeiadis. Weak persistency semantics from the ground up: Formalising the persistency semantics of ARMv8 and transactional models. *Proc. ACM Program. Lang.*, 3(OOPSLA):135:1–135:27, October 2019.

[85] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the Intel-x86 architecture. *PACMPL*, 4(POPL):11:1–11:31, 2020.

[86] HM UK Government. Copyright, Designs and Patents Act. c. 48, 1988.

[87] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In *Proc. CAV*, 2010.

[88] Alasdair Armstrong, Brian Campbell, Ben Simner, Thibaut Pérami, and Peter Sewell. https://github.com/rems-project/isla/blob/master/doc/axiomatic.adoc.

[89] Arm Limited. Arm Cortex-A53 MPCore Processor Technical Reference Manual, 2022. ARM DDI 0500J.

[90] Shilpi Goel. The x86isa books: Features, usage, and future plans. In *Proceedings 14th International Workshop on the ACL2 Theorem Prover and its Applications, Austin, Texas, USA, May 22-23, 2017.*, pages 1–17, 2017. arXiv version: https://arxiv.org/abs/1705.01225.

[91] Rishiyur S. Nikhil and Niraj Nayan Sharma. Forvis: A formal RISC-V ISA specification. https://github.com/rsnikhil/Forvis_RISCV-ISA-Spec, 2019. Accessed 2019-07-01.

[92] Ian J Clester, Thomas Bourgeat, Andy Wright, Samuel Gruetter, and Adam Chlipala. riscv-plv risc-v isa formal specification. https://github.com/mit-plv/riscv-semantics, 2019. Accessed 2019-07-01.

BIBLIOGRAPHY

[93] Hira Syeda and Gerwin Klein. Reasoning about translation lookaside buffers. In *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, pages 490–508, 2017.

[94] Hira Taqdees Syeda and Gerwin Klein. Program verification in the presence of cached address translation. In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, pages 542–559, 2018.

[95] Bogdan F. Romanescu, Alvin R. Lebeck, and Daniel J. Sorin. Specifying and dynamically verifying address translation-aware memory consistency. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 323–334, New York, NY, USA, 2010. ACM.

[96] Bogdan Romanescu, Alvin Lebeck, and Daniel J. Sorin. Address translation aware memory consistency. *IEEE Micro*, 31(1):109–118, January 2011.