Arm systems semantics

Ben Simner

This thesis is submitted for the degree of Doctor of Philosophy

Department of Computer Science and Technology

UNIVERSITY OF CAMBRIDGE

Wolfson College, April 2025

Second edition

Preface

This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the preface and specified in the text. It is not substantially the same as any work that has already been submitted, or is being concurrently submitted, for any degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the preface and specified in the text. It does not exceed the prescribed word limit for the relevant Degree Committee.

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, and exceptions and interrupts, remains mostly obscure, leaving us without a solid foundation for verification of security-critical systems software.

We develop precise mathematical models, for those aspects of the Arm A-class architecture. We implement these models as executable models, in both microarchitectural-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 give a variety of hand-written and machine-generated litmus tests, exercising parts of the architecture previously unexplored.

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

Acknowledgements

My supervisor, Peter Sewell. My examiners, Lars Birkedal and Neel Krishnaswami. Will Deacon, initially as my industrial supervisor at Arm, and then a close collaborator at Google. Simon Moore and Tim Griffin for their guidance in the first few years as my assessors. My co-authors on this work, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, Brian Campbell, Thomas Bauereiss, and Ohad Kammar. My co-authors before and beyond, Matthew Windsor, Mike Dodds, Matthew J Parkinson, Sylvan Clebsch, David Kaloper-Meršinjak, Thibaut Pérami, and Kayvan Memarian. Other collaborators, Yotam Dvir, Yeji Han, Thomas Fourier, Nils Lauermann, and Chung-Kil Hur. The Arm Executive Vice President and Chief Architect, Richard Grisenthwaite, whose unwaivering support, generous donation of time, and direct technical contributions, made this work possible. Derek Williams, for his fount of knowledge on Power and RISC-V, feedback on the various proposals and materials, and general positive support over the last seven years. Various Arm staff who discussed hardware and architecture, here in Cambridge or in Austin, Ian Caulfield, Martin Weidmann, Anthony Fox, Alan Hunter, Magnus Bruce, Dave Martin, Artem Khyzha, Nikos Nikoleris, and Gustavo Petri. Alastair Reid for the many invaluable conversations on ISAs and computer architecture over the years. Google, for their support, and Ben Laurie and Sarah de Haas, for their ceaseless efforts to ensure that support. The Android KVM team, with whom we have worked closely with over the last few years, in particular Keir Fraser, Marc Zyngier, Quentin Perret, Vincent Donnefort, Serban Constantinescu, David Brazdil, Andrew Scull, and Andrew Walbran. Others inside Google who have variously taken time from their day to describe and explain many disprate things from thin-air, to JITs, to real numbers, Hans-J. Boehm, Hong-Seok Kim, and Jaroslav Ševčík. Amazon, and several AWS staff, including Nathan Chong, Hanno Becker, Dominic Mulligan, and Elias El Yandouzi, for inviting me to talk, reciprocating the visit back, and for their detailed analysis of the models. Gregory Malecha, and all those at BlueRock Security, for countless conversations and detailed discussions of models, software, verification, and more. Warren A. Hunt, Anna Slobodova, Rob Sumners, and Shilpi Goel, for their hospitality during my visit to Austin and Centaur Technology. Alejandro Aguirre, Sergei Stepanenko, Giovanna Kobus Conrado, and others who were so welcoming during my visits to Aarhus. My Part II students, Ben Stokes and Jasper Parish. All those who engaged in discussions more broadly, on a wide range of topics from computer architecture, programming languages, and systems, Susmit Sarkar, Zongyuan Liu, Jonathan Woodruff, Peter Rugg, Alexandre Joannou, Daniel Lustig, Simon Cooksey, Mark Batty, Ori Lahav, Conrad Watt, Ilia Shumailov, Stephen Kell, David Chisnall, Anders Alnor Mathiasen, Claudio Russo, and everyone at the Computer Lab. Ross Anderson (d. 2024), who never failed to stop to chat about what I was working on, computer security, or research in general. The Computer Lab staff, especially Joy Rook and Lise Gough. My anonymous reviewers, and all others who provided feedback on papers, talks, posters, and other materials. Particular thanks goes to Shaked Flur, for sharing his TikZ package which generates many of the diagrams in this document.

My master's supervisor at the University of York, Colin Runciman, whose advice then still serves as my guide now. Mike Dodds, who hired me as a summer student, kickstarting my research career. My colleagues from then, Jo Pugh (d. 2023), Matthew Windsor, Michael Walker, and Rudy Matela.

Rachel Berry and Johnny Symonds, whose selfless kindness opened this path to me.

Scott. Robert Cream. Samuel Hand. Chongyang Shi. Sam W. Matt Dawson. Bjørnar Grindheim Meling. Max Aalling. Liam Biddle. My friends, family, and all those who supported me.

This work was supported by an Arm/EPSRC iCASE Studentship Award 2097768 (Simner); an EPSRC grant EP/K008528/1 (REMS); an ERC Advanced Grant 789108 (ELVER); a UK Research and Innovation (UKRI) grant EP/Y035976/1 (SAFER), under the UK government's Horizon Europe funding guarantee ERC-AdG-2022; the UK Government Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme (grant 105694); an ERC-2024-POC grant 101189371 (ELVER-CHECK); an AUFF starter grant (Pichon-Pharabod); two 2023 Amazon Research Awards (Sewell and Simner; Pichon-Pharabod); a Royal Society University Research Fellowship (Kammar); a UK Advanced Research and Innovation Agency (ARIA) project (Qbs4Safety, Kammar); and was part funded by Google. With thanks to the Isaac Newton Institute for Mathematical Sciences (INI), Cambridge, for support and hospitality during the programme Big Specification, where some of the work undertaken. The INI is supported by EPSRC grant EP/Z000580/1.

A lay summary

Modern computing devices have become increasingly complex over time, powered by chips with many interacting components: network controllers; audio and signal processing; graphics processing; memory controllers; and at the heart is the central processing unit, or CPU. The CPU is the principal director of the entire machine, in charge of running all the software and coordinating all the components together. The CPU does calculations, directs data to be read or written, manages connections over the network, and receives all the input from the outside world to decide how to respond.

Historically, 'computers' were mechanical calculators. As time progressed, computers progressed from mechanical to electronic, and became general computation devices and not just simple calculators. They moved from manual to automated, not actioning operations as directed by a human but executing *programs*: pre-determined sequences of *instructions* telling the computer which set of operations to perform.

Machines like *EDSAC* paved the way for the modern computer. Its instructions were few and simple, e.g.: read a number from the input tape, add or subtract two numbers, load a number from memory, store a number into memory, go to a particular instruction, display a number to the user, stop the machine and ring the bell. Computers have advanced much since then, but the interface has not: we still program by giving the machine a sequence of very simple instructions to perform. Today there are many *instruction set architectures* (ISAs), each defining standard collections of instructions. CPUs with Arm, x86 (Intel/AMD), Power (IBM), and RISC-V ISAs now power billions of devices globally. These ISAs are *far* bigger than EDSAC's meagre ∼20 instructions, with the, comparatively small, Arm ISA defining 402 instructions in its base architecture alone.

Modern CPUs have advanced in ways other than just having larger ISAs: caches are placed between the CPU and memory, making data much quicker to access; they have become *multicore*, placing multiple CPUs side-by-side on the same chip; and CPU designs have become *superscalar*, able to perform multiple operations at the same time by executing instructions in a pipelined fashion.

Superscalar and multicore machines can therefore have multiple instructions touching the same shared data simultaneously. If the programmer instructs two CPUs to concurrently access the same location, then the behaviour of the machine is now determined not by the simple description of the individual instructions, but by the complex interaction of these implementation-specific optimisations. Figuring out *what* can happen in that case is the field of *relaxed memory*, essentially: what happens if the user pushes two buttons that touch the same data at the same time? That is the object of study of this thesis.

We are interested in those buttons not for the every day programs, but those which give lower-level control of the machine. These are the instructions used by operating systems to manage many programs at once, and by hypervisors in the cloud to protect and isolate multiple customers virtual machines from one another. Understanding the behaviour of those instructions is crucial in order to be able to develop robust software that uses them and to make credible claims about that software.

In particular, we will investigate three areas of a modern high-performance multiprocessor architecture: the ability to change the program being executed on the fly (*self-modifying code*); the machinery used by systems software to control the access programs have to shared data (*virtual memory*); and the way external events interact with a running program, and, in particular, the switch between a program and its supervisor (*exceptions/interrupts*).

We do this in three steps. We (1) engage in broad technical discussions with architects and hardware designers. We guide such discussions with specific software patterns or hardware optimisations in mind, focusing on small representative experimental programs or *litmus tests*. We (2) reinforce the results of those discussions through empirical analysis of existing hardware by building tools and using them to gather experimental data, which further informs our and the architects' understanding. Finally, we (3) employ the tools and techniques from programming language theory to build robust formal mathematical models giving a clear definition to that interface. All this *clarifies* the architects' intent, gives software engineers a sturdier basis to appeal to, and hopefully will enable richer efforts in certification and verification of those key pieces of software that we all rely on to perform correctly and to keep our data secure.

Contents

Chapter 1

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 hardware: programming languages whose syntax is defined by the binary encoding of the instructions from the *Instruction Set Architecture* (ISA), and whose semantics is the composition of the sequential behaviours of the instructions with a machine execution model. The 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, one might think that this interface would be straightforward to define. One can give precise formal semantics to the individual instructions, as Arm does, in its custom *Architecture Specification Language* (ASL) [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]. 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 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 several reasons to focus on Arm: (1) they are ubiquitous, with over a trillion devices running Arm hardware, (2) Arm has a diverse ecosystem of implementations, meaning that 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, and exceptions.

1.1 Arm-A architecture overview

In this work we primarily focus on Arm. Arm will serve as an example representative modern microprocessor architecture, but many of the behaviours and conclusions will also apply to other architectures including RISC-V, IBM Power, and x86.

Arm produce three major classes of architectures, A-class (Application), R-class (Real-time) and M-class (Microprocessor). Arm principally make *architecture*. While Arm do design several implementations, many of which their partners use in their own designs, a number of partners also design their own implementations which they use in their own chips. This gives us a large surface of interesting designs to test, all implementing the same architecture. In particular, we will focus on the *Application* (A)-class processors.

Arm's A-class architecture is intended to support general-purpose high-performance microprocessors, such as those found in mobile devices, tablets, laptops, and 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 both 32-bit and 64-bit modes, called *execution states*: 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. We will focus here on the 64-bit architecture, AArch64 and its A64 ISA, as found in Armv8-A and Armv9-A. We use the term *Arm-A* to refer to both Armv8-A and Armv9-A interchangeably.

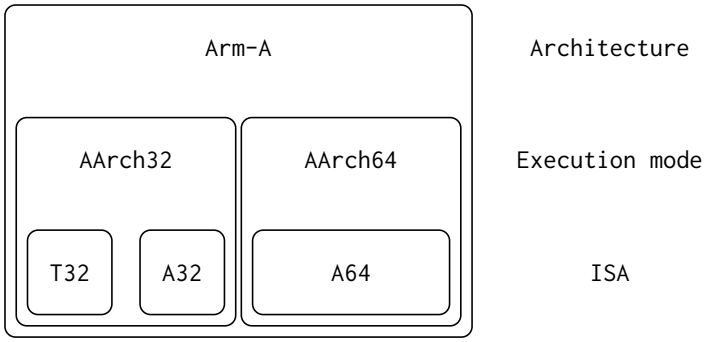


Figure 1.1: Arm-A structure.

At the time of writing, A64 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 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; and is executing at at one of EL0, EL1, EL2 or EL3.

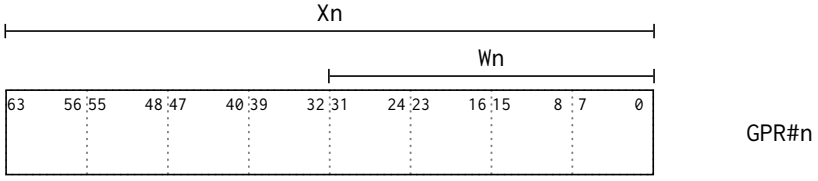


Figure 1.2: Arm-A W and X register views for a general-purpose register.

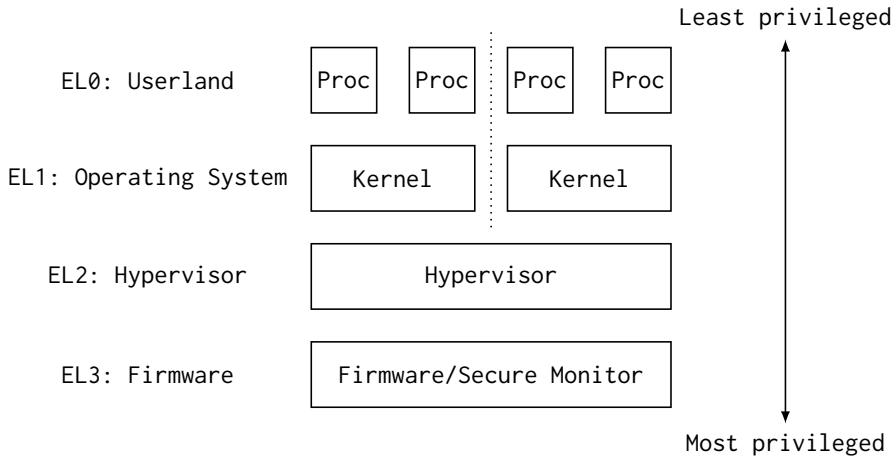


Figure 1.3: Arm-A exception levels.

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] (described in detail in Part II), with each process behaving as if it had direct access to memory, when in fact the operating system (via 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, software 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

Hardware and software are not simply direct implementations of the sequential semantics one might expect of the languages they implement. Whether they are the machine languages implemented by microprocessors or software languages implemented by compilers, as time progressed these implementations have acquired multiple layers of abstraction, made with increasing complexity. Compilers and hardware optimise 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: aside from the timing effects they are designed to cause they 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, with tens of thousands of pages 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. Those optimisations include, but are not limited to, 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], 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 the hardware actually exhibits, by empirically observing what extant hardware does, by talking with architects and hardware designers about what they imagine hardware could do, now or in the future, and by 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 work, we extend the previous relaxed memory work on Arm into the realm of systems software: instruction fetch and cache maintenance (Part I), pagetables and TLB maintenance (Part II), and a start on exception handling (Part III). We produce a microarchitectural-style operational semantics for the first in the style of Flat [7], and herd-style declarative ('axiomatic') semantics [32] for all parts.

In Part I, we produce:

- ▷ A set of litmus tests (Ch.3) which cover a variety of phenomena and architectural features, including self-modifying code and the required cache maintenance, clarifying the architectural intent.
- ▷ A microarchitectural-style structural-operational-semantics (Ch.4) which implements that architectural intent as we understand it.
- ▷ A formulation as an axiomatic-style declarative semantics (Ch.5), intended equivalent to the aforementioned operational model, as an extension to the axiomatic model of [7].
- ▷ An extension of the `litmus7` tool [33], and a set of experimental results from testing against a range of hardware (Ch.6).

The models cover both the base architecture, as well as the implementation-defined choices for stronger cache types. The tests represent the architectural intent as we understand it, modulo mixed-size accesses and open questions where explicitly stated. The base model is thoroughly validated through discussions with architects and experimental testing of hardware, although at the time of the work the hardware did not support the stronger cache types. The two models are intended equivalent, and, although no formal proof-of-equivalence has been done, we have empirically tested their equivalence through the auto-generation of a large suite of tests with an extension to the diy [34] tool. Since publication of the work, Arm have extended the official Arm memory model with instruction fetching and cache maintenance, although we do not believe the architectural intent has changed since this work.

In Part II, we produce:

- ▷ A collection of litmus tests for virtual memory and TLB maintenance, for multi-level translation table walks with both stages (Ch.8), clarifying the architectural intent in many cases or identifying remaining open problems in others.
- ▷ 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).

We have a large collection of hand-written litmus tests, covering a substantial portion of the virtual memory systems architecture of Arm: translation table walks and the respective translation reads, caching in TLBs, TLB maintenance, virtualisation and multi-stage TLB invalidations, and how these all interact with the rest of the memory model. We further clarify the architectural intent in many of these areas, although in some places the architectural intent has further evolved since the original publication of the work; such places are explicitly stated as such, e.g. for forwarding and enhanced-translation-synchronisation. Arm has independent, but adjacent, work on virtual memory covering an overlapping set of features but with a focus on hardware support for access flags and dirty bits as required by KVM; we believe the architectural intent for the tests presented here has no changed, except where explicitly stated, and the models correspond on the tests.

In Part III, we produce:

- ▷ A set of litmus tests (Ch.11), covering precise exceptions, clarifying the architectural intent around precision and synchrony with respect to the weak memory model.
- ▷ An axiomatic-style declarative semantics for precise exceptions in Arm (Ch.12).
- ▷ An extension to the hardware testing harness of Chapter 10 to support hardware testing of exceptions, and experimental results from running tests on hardware (Ch.13).

We cover the base exception machinery, clarifying the ordering guarantees of precise exceptions, and exploring how external errors in the system can effect those guarantees. The model presented, to the best of our knowledge, captures this intent. We have validated the model against some hardware implementations, although much more is required for high confidence. We are aware of independent work by Arm in the same area at the time of writing, but do not know its status.

1.5 Publications and collaborations

The work presented in the three parts were done in collaboration with a variety of other people on different aspects, resulting in 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 [35].
- '**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 [36].
- '**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 [37].
- '**Isla: Integrating full-scale ISA semantics, axiomatic concurrency models (extended version)**', in Formal Methods in System Design (May, 2023), by Alasdair Armstrong, Brian Campbell, **Ben Simner**, Christopher Pulte, and Peter Sewell [38].
- '**Precise exceptions in relaxed architectures**', Accepted for publication at the 52nd International Symposium on Computer Architecture (ISCA 2025), arXiv pre-print, by **Ben Simner**, Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Ohad Kammar, Jean Pichon-Pharabod, and Peter Sewell [39].

Many of the aspects of the work presented in this thesis were done jointly with many of the people listed above. The Isla tooling was primarily written by Alasdair Armstrong. The work on the litmus and diy tools was done by Luc Maranget. The 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 in collaboration with Christopher Pulte and Shaked Flur (for ifetch); Christopher Pulte and Thibaut Pérami (for VMSA); and Jean Pichon-Pharabod and Ohad Kammar (for exceptions). 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. That said, the Arm architecture and its reference text is solely the responsibility of Arm, and the architectural intent can change over time. This thesis describes our current understanding at the time of writing; it is not an Arm publication. We also work closely with the creator of the original Armv8 memory model, Will Deacon, who maintained the official Arm model until 2020, and who is a co-maintainer of the Arm port of the Linux kernel. We had several discussions with Arm's memory modelling team on past and ongoing formal modelling within Arm. They provided us with early access to a version of the Arm VMSA model and tests, and we shared early access to our Arm-related publications with them. We also engaged in various individual discussions with current and former members of Arm staff, including Ian Caulfield, Nikos Nikoleris, Gustavo Petri, Anthony Fox, Nathan Chong, Martin Weidmann, Alastair Reid, and others, For the avoidance of doubt, and except where otherwise stated, all work is my own. These discussions and sharing of materials do not constitute contributions to, or imply approval of, the content of this thesis.

1.6 Thesis overview

This document starts with an introduction to systems software and relaxed memory (Chapter 1), and a discussion of the technical details of the 'usermode' Arm model (Chapter 2). The remainder of the document is then split into three parts:

- ▷ Instruction fetch (Part I comprising chapters 3-6)
- ▷ Virtual memory (Part II comprising chapters 7-10)
- ▷ Exceptions (Part III, comprising chapters 11-13)

We end with a short conclusion (Chapter 14), which includes a discussion of the limitations of the work presented here and some reasonable next steps.

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 instantiations of these for Arm-A.

Part I: Instruction fetching We start with a brief overview of the existing prose text in the Arm-A specification for instruction fetch 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 just-in-time compilers (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 micro-architects. We give 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, and additionally check that they do not forbid behaviours exhibited on hardware by running the test suite on a selection of modern Arm processors.

Part II: Virtual memory Structured similarly to the instruction-fetching chapters, but independently of them, we explore the Arm *Virtual Memory Systems Architecture* or *VMSA*. We begin with some background information in the form of an overview of the sequential aspects (Ch.7), describing the structure and behaviour of the Arm address translation and memory management architecture without considering concurrency or caching effects. Then, we explore the relaxed behaviours of virtual memory (Ch.8) by producing litmus tests and discussing 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.

Part III: Exceptions We finish the trio with a short overview of the initial work on relaxed exceptions in Arm-A. We begin with a discussion on the Arm interpretation of precise exceptions and give litmus tests which capture the key phenomena (Ch.11); we then give an axiomatic model which capture those phenomena (Ch.12), and finally produce some preliminary hardware results to support the models (Ch.13).

Conclusion Finally, Chapter 14 presents a short summary of the presented work, its limitations, and relation to other work in the area. We discuss what was learned, in terms not only of the models produced but also of the process itself, before finally touching on what remains as potential future work.

Chapter 2

Modelling Arm: background

Now we turn our attention to the now well-established methods of precisely and formally modelling relaxed memory behaviours, in the context of 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 succinctly define the validity of whole-program executions.

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 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 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 core pattern (or precisely, the cycle) of events which act as the skeleton of the test. In this case, *message passing* is a common software pattern, where one thread writes some data followed by a flag signalling the data is ready, while another thread concurrently reads the flag in order to further read the data. Thus, the 'MP' shape implies a two-threaded test with two locations (typically named `x` and `y`), with one thread (typically 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
STR X0,[X1]
MOV X2,#1
STR X2,[X3]
	LDR X0,[X1]
	LDR X2,[X3]

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

Figure 2.1: MP test code listing.

The code listing given is totally standard [40]: 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 a conjectured outcome (plus its architectural intent, if known) given as a constraint on the final register and memory state. On Arm, the outcome given in the listing in Figure 2.1 is allowed.

On a sequentially consistent (*SC*) machine, whose executions are simply the interleaving of the instructions of all threads [41], there are many executions of the listed code, each giving rise to (potentially distinct) final states. 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 that 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 to the allowed outcome can be found in Figure 2.2.

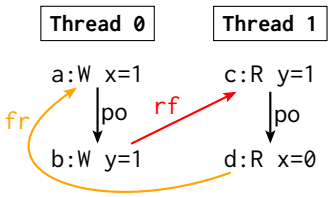


Figure 2.2: MP test execution diagram.

The nodes on the left, below the Thread 0 label, correspond to events from executing Thread 0 of the program. The event labelled `a` 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 write of the second store being propagated. They are related by program-order (`po`), indicating that the instruction `a` came from is earlier than that of `b` in the instruction stream of the processor; that is, `a` comes before `b` in the control flow of that thread, as determined by the order the processor fetched and decoded the instructions in. Similarly, below the Thread 1 label we see the event labelled `c`: the read event corresponding to the first load, reading the address `y` and getting the value `1`. The value read came from the write event `b`, therefore `b` is related to `c` 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. In this case, that write is the initial write 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 most modern hardware.

Litmus testing 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). This has included obtaining both the architectural intent for those patterns, as well as extensive testing campaigns on a variety of modern hardware. In some cases, some outcome 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 exist) *hardware errata*, more commonly referred to as 'bugs'. 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, 40, 32, 16, 7, 6, 42]. 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.

2.1.1 Thread-local ordering

On Arm, instructions need not execute in the order they appear in the program, as we just saw. 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 this does not mean that the hardware is not allowed to re-order the instructions, but that if it does, it must preserve the illusion of in-order execution to the programmer.

Not all re-orderings are permissible. In particular, 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 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.

Dependencies in Arm arise from the intrinsic control and data flow 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 purely dynamic properties of the execution, but rather they are *syntactic* in that the dependencies an instruction induces is a statically known property, thus there are no so-called 'fake' dependencies: the values read or written at runtime by an instruction does not matter; only the set of registers it accesses.

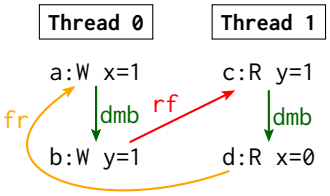
Not all dependencies are equal. On Arm, address and data dependencies enforce both read-to-read and read-to-write ordering, whereas control dependencies enforce read-to-write but not read-to-read ordering. Speculation allows reads to happen 'early', but not writes; this gives an asymmetry where control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

MP+dmbs 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]	DMB SY
DMB SY	LDR X2,[X3]
MOV X2,#1	
STR X2,[X3]	

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



MP+dmb.st+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 ST	LDR X2,[X3,X4]
MOV X2,#1	
STR X2,[X3]	

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

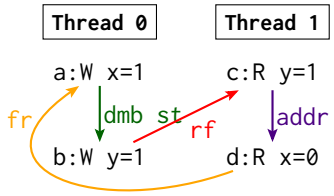


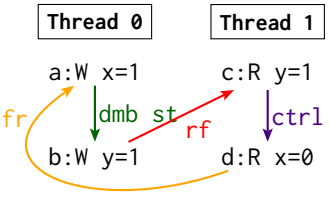
Figure 2.3: Two variants of MP with thread-local ordering.
On the left: MP+dmbs with Arm DMB barrier between instructions.
On the right: MP+dmb.st+addr with an address dependency between the reads.

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

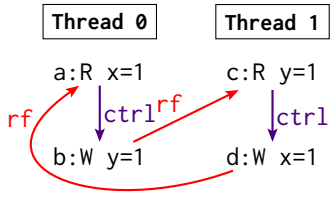


Figure 2.4: Two litmus tests with speculation.
On the left: MP+dmb.st+ctrl with Arm DMB barrier between the writes, but a control dependency between the reads.
On the right: LB+ctrls, a variant of the classic 'load buffering' litmus test, with control dependencies to both writes.

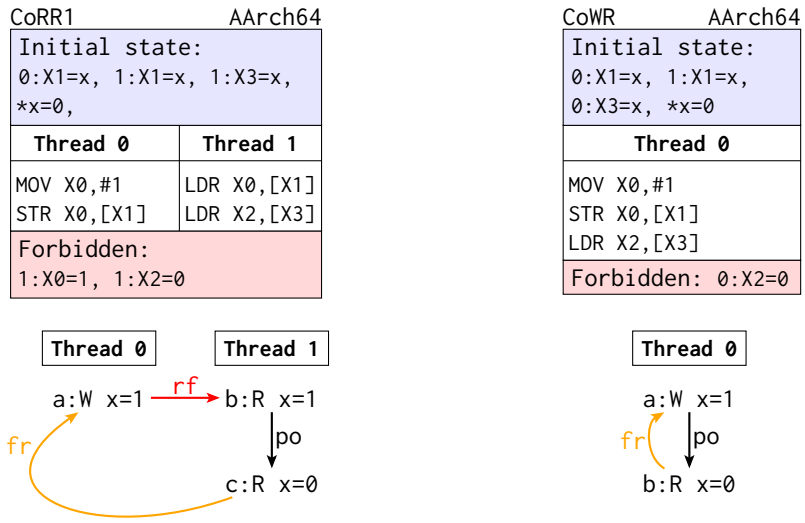


Figure 2.5: Two coherence litmus tests.
On the left: CoRR1, that two subsequent reads of the same location in the same thread should be consistent with the coherence order. On the right: CoWR, that a read of a location cannot skip over a newer program-order earlier write from the same thread.

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 happen in, that all threads agree on [8]. Microarchitecturally, this emerges naturally from the desire to ensure that writes are never dropped by the cache protocol, and since writes may be of sub-cache-line size (down to individual bytes) the cache protocol must ensure consistency over whole cache lines at a time.

This property is one that sets processor consistency models apart from those one would find in databases and other distributed systems, which generally do not require it, such as the classic *causal consistency* model for distributed systems [43].

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

2.1.3 Multi-copy atomicity

Coherence is not sufficient to guarantee that all threads agree on what the most recent write is at the same point in time.

In particular, while all threads will see the same writes to the same location in the same order, at any particular moment some threads may not have caught up to the latest write yet. Architectures that have this property are called *non-multi-copy atomic* [13].

Arm has a kind of partial multi-copy atomicity, which they term *other-multi-copy atomicity*. Other-multi-copy atomicity gives guarantees similar to multi-copy-atomicity, 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 or something newer [7]. The hardware mechanism which motivates this is *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 IRIW (independent-reads independent-writes) litmus test, which demonstrates the latter point, that writes propagate to all threads simultaneously.

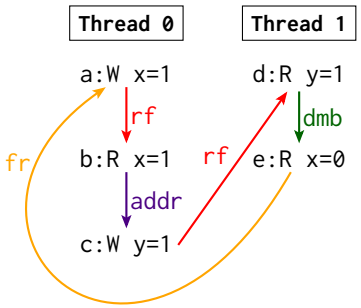
MP+nondep+dmb AArch64

Initial state:
0:X1=x, 0:X5=y,
1:X1=y, 1:X3=x, *x=0, *y=0

Thread 0	Thread 1

MOV X0,#1
STR X0,[X1]
LDR X2,[X1]
EOR X6,X2,X2
MOV X4,#1
STR X4,[X5,X6]
	LDR X0,[X1]
	DMB SY
	LDR X2,[X3]

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



PPOCA AArch64

Initial state:
0:X1=x, 0:X3=y,
1:X1=y, 1:X3=z, 1:X5=z
1:X8=x, *x=0, *y=0

Thread 0	Thread 1

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

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

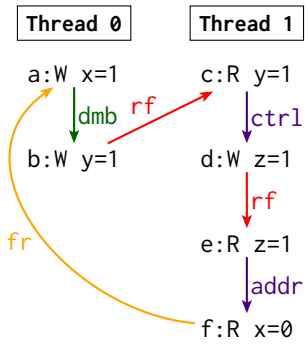


Figure 2.6: Two litmus tests with write forwarding.
On the left: MP+nondep+dmb with write-forwarding down a non-speculative branch.
On the right: PPOCA, with write-forwarding down a speculative branch.

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

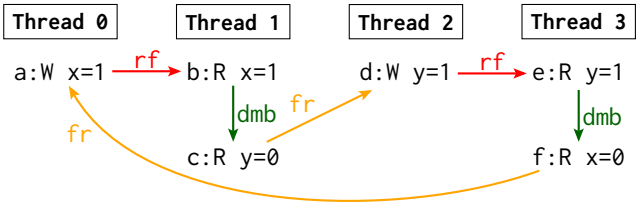


Figure 2.7: IRIW+dmbs: a classic multi-copy atomicity litmus test.

2.2 Intra-instruction semantics

Previous work has, for Arm and RISC-V, established high-fidelity models for the *intra*-instruction behaviour of individual instructions. That is, the sequential behaviour of the register and memory accesses, and any arithmetic over them, the instruction performs.

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 [44].

The ASL and Sail specification languages Although this work is focused on Arm-A and Arm use ASL, the tools we build upon are architecture-agnostic and use the Sail specification language for instruction semantics [45]. We therefore use the automatically generated ASL-to-Sail translation [45, 46] of the official Arm specification for most of the examples presented here, except where specified otherwise. 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 primitive effects, as it is important to the function of the tools we will use later on.

Outcomes Sail programs have effects for the interaction between the instruction semantics and the registers and memory. These effects make Sail programs monadic computations over the Sail effect datatype (called `outcome`). Figure 2.8 lists the outcomes defined by the Sail standard library [15], it contains one pure value (Done), and the other values each represent one step of the intra-instruction semantics suspended at the interface with the environment, containing a continuation to resume the execution with the environment's choice.

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).

An example instruction As an example, take the Arm '`ADD Xd,Xn,Xm`' instruction, whose Sail code can be found in Figure 2.10, as translated 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 generates an effect. Omitting the outcomes for the flag registers, and the exact arithmetic calculation, executing this code results in the trace of outcomes shown in Figure 2.9:

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

Figure 2.9: Trace from the Arm `ADD` instruction.

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

```
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 2.10: Sail pseudocode for the `ADD Xd,Xn,Xm` instruction.

2.3 Arm-A operational model

The state of the art multi-copy-atomic operational semantics for Arm is the *Flat* model [7, 6, 47] by Flur, Pulte, et al. Flat is a small-step operational semantics, with transitions designed to (abstractly) match the kinds of actions we see in hardware. It is implemented as an executable-as-a-test-oracle model in the RMEM tool [48].

RMEM is written in a combination of OCaml, and the Lem [49, 50] 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 [51].

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 instruction instances with explicit out-of-order execution (abstractly modelling modern microprocessor pipelines).

Figure 2.11 illustrates a snapshot of an example instruction tree from a thread with 10 in-flight instruction instances. Some instructions (i_2, filled-inblue) 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, partly-filledblue) 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.

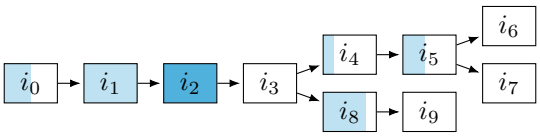


Figure 2.11: A tree of 10 concurrently executing instruction instances.

Flat is therefore composed of two subsystems: the thread subsystem which contains the pool of threads with their instruction trees, a storage subsystem which contains a flat array for memory, as sketched in Figure 2.12.

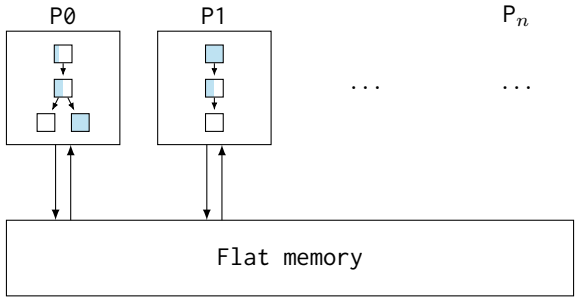


Figure 2.12: Flat state (diagram).

Thread subsystem More precisely, the thread subsystem has a per-thread tree of instruction instances. 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 2.13), 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, which typically update the current pseudocode state without calling the continuation.

Figure 2.14 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 2.13: Operational pseudocode states.

```
type threadSubsystem =
  nat → instruction_tree;
type instruction_tree =
  list (instruction_instance *
    instruction_tree);
type instruction_instance =
  <| id: nat;
     program_loc: address;
     micro_op_state: micro_op_state;
     mem_reads: set address;
     ... |>
type micro_op_state =
  | MOS_plain
      of outcome
  | MOS_pending_mem_read
      of (value → outcome)
  | MOS_potential_mem_write
      of outcome
type thread_trans =
  | T_register_read
      of reg_name * value
  | T_register_write
      of reg_name * value
  | T_satisfy_read
      of value
  | T_mem_write_footprint
      of list write
  | T_mem_potential_write
      of list write
  | T_commit_store
  | T_complete_store
  | T_commit_barrier
      of barrier_kind
  | ...
type sync_trans =
  | T_propagate_write
      of write
  | T_satisfy_read
      of read_request * value
  | T_propagate_barrier
      of barrier_kind
  | ...
```

Figure 2.14: 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 2.15 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 2.15: 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 specific transitions for 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 Store instruction
- ▷Initiate memory writes of store instruction, with their footprints
- ▷Instantiate memory write values of store instruction
- ▷Commit store instruction
- ▷Propagate memory write
- ▷Complete store instruction (when its writes are all propagated)

Transitions on a Barrier
- ▷Commit barrier

Transitions on a Load instruction
- ▷Initiate memory reads of load instruction
- ▷Satisfy memory read by forwarding from writes
- ▷Satisfy memory read from memory
- ▷Complete load instruction (when all its reads are entirely satisfied)

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. See Appendix B for a full rendering of the Flat model.

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, in a so-called *axiomatic* style. Axiomatic models describe the allowed behaviour of programs by a predicate, typically described by a collection of axioms, constraining the event of the *candidate executions* of that program. These candidates 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 such candidates: executions consistent with the intra-instruction semantics, but where the values used in the program are unconstrained. The model then has axioms, generally acyclicity or emptiness of relations over the events, which reject some of these executions as *inconsistent*. Those that remain are the *valid*, or *consistent*, executions of the program.

The model can therefore be used to assert whether some given program can reach a final state satisfying some 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: the set of events of the program, which for Arm these are the memory access and barrier events, labelled with their access type (read or write, or barrier kind); and the candidate relations over those events, such as program-order and address/control/data dependencies, amongst others.

It is often useful to split the candidate execution definition into two steps: first, to define a *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 ($\mathcal{C}_\mathrm{R}$) satisfying some constraints (described in more detail later on), and a candidate witness ($\mathcal{C}_\mathrm{W}$) describing the existentially quantified coherence-order and reads-from relations.

$$\text{Candidate Pre-Execution} \equiv \mathcal{P}(\mathbb{N}) \times (\mathbb{N} \to \text{Label}) \times \mathcal{C}_\mathrm{R}$$
$$\text{Candidate Execution} \equiv \text{Pre-Execution} \times \mathcal{C}_\mathrm{W}$$

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):

$$\xrightarrow{\mathsf{L}} \equiv \mathbb{N} \times \mathbb{N}$$
$$\mathcal{C}_\mathrm{R} \equiv \langle \xrightarrow{\texttt{po}}, \xrightarrow{\texttt{loc}}, \xrightarrow{\texttt{addr}}, \xrightarrow{\texttt{ctrl}}, \xrightarrow{\texttt{data}}, \xrightarrow{\texttt{rmw}}, \xrightarrow{\texttt{ext}} \rangle$$
$$\mathcal{C}_\mathrm{W} \equiv \langle \xrightarrow{\texttt{co}}, \xrightarrow{\texttt{rf}} \rangle$$

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:

1. 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
2. a set of Arm barriers (`DMB`, `ISB`) and their variants.

More precisely, these labels can be described as follows:

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

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 `isla-axiomatic`.

Candidate relations The candidate relations capture the relationships and orderings between the events of the 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).

where R_n is a read event, W_n is a write event, M_n is a read or write, and E_n an event of any type.

Implicit events and ISA-Consistency Candidate executions for Arm contain a limited set of events: reads, writes, and barriers. In reality, there are many more *implicit* events, there in principle but typically excluded from the candidates: register reads and writes, instruction fetches, translation table walks, and so on. We will (in Parts I, II and III) extend the candidate executions to include some of these implicit events. For a candidate execution to be consistent with a given architecture's intra-instruction semantics, as defined by its ISA, there must be a corresponding execution of that intra-instruction semantics, with all its implicit events, which corresponds to the set of explicit events seen in the candidate.

It is uncommon to include such events in the axioms and relations directly, and therefore are typically elided entirely. However, we could imagine that for each candidate execution, one can 'complete' the events to include all the hidden implicit events, such that the events of the candidate correspond exactly to the Outcome type of the intra-instruction semantics (see Figure 2.8).

Let us say that an execution E' is a completion of E, for which we will write $\text{Completion}(E', E)$ if the subgraph of E' restricted to explicit events is equal to E, and where the events within an instruction are ordered by an intra-instruction-causality order (`iico`) consistent with the definition of address/control/data

dependencies, as in Alglave et al. [14]. We can now define what it means for an execution to be consistent with an ISA given some intra-instruction semantics: that, roughly, there are a set of implicit events which complete the candidate such that the groups of events for each instruction correspond to traces from the intra-instruction semantics.

Formally, we can imagine partitioning the completed execution with its implicit events by program-order, to get groups of events corresponding to instructions, with traces of events given by `iico`. Let us call the set of `iico`-traces of an execution IICOTraces(E). We can then define ISA-consistency by asking whether there exist traces in the intra-instruction semantics which correspond to the `iico`-traces of instructions in the execution:

$$\text{SimulatesISA}(E : \text{Execution}) : \forall t.\ t \in \text{IICOTraces}(E) \Rightarrow t \in \text{ISA}$$
$$\text{ISA-Consistent}(E) = \exists E'.\ \text{Completion}(E', E)\ \wedge\ \text{SimulatesISA}(E')$$

In practice, we generally go the other way: tools produce complete traces from the intra-instruction semantics defined by the ISA, and then discard or hide some events to obtain a smaller candidate — thereby producing ISA-Consistent executions by construction.

As an example, take the reader thread of an MP-shaped test, with a barrier between the loads. Figure 2.16 shows a sketch for a completion of that reader thread for Arm, including general-purpose register and instruction fetch events (but still eliding the voluminous configuration register access and translation table walk events for brevity) with introduced implicit eventsin blue.

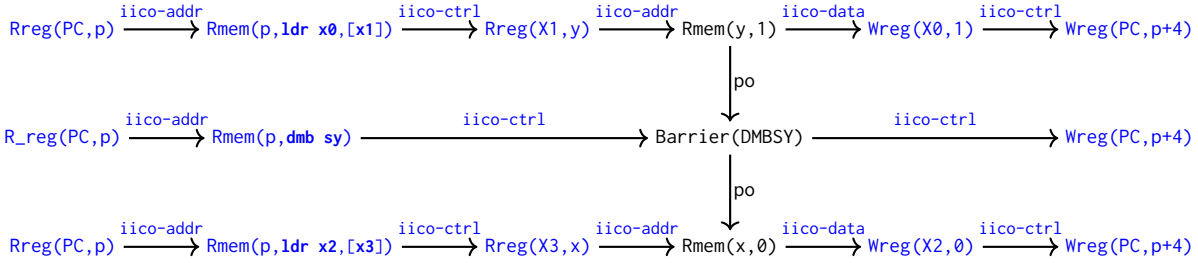


Figure 2.16: Sketch of a completion of reader thread of MP+dmb.sys with implicit events.
Nodes and edges (original in black, implicit in blue) include the implicit register or instruction fetch events which are part of the ISA definition (with implicit events corresponding to system register accesses and translation elided for brevity).

Well-formedness Each of the relations of the candidate relations and witness are subject to some well-formedness constraints. 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 2.17 contains the types and some basic well-formedness properties of the pre-execution relations.

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

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 2.17: 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 2.18.

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

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

Figure 2.18: 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)

We say a candidate execution E is *well-formed*, for which we will write WELL-FORMED(E), if all constraints from Figures 2.17 and 2.18 are satisfied.

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.

$$\begin{aligned}
\text{AXIOM-CONSISTENT}(E : \text{Execution}) &= \text{see §2.4.2} \\
\text{CONSISTENT}(E : \text{Execution}) &= \text{WELL-FORMED}(E) \\
&\quad \wedge\ \text{ISA-CONSISTENT}(E) \\
&\quad \wedge\ \text{AXIOM-CONSISTENT}(E) \\
\text{CONSISTENT}(E : \text{Pre-Execution}) &= \exists \mathtt{co}, \mathtt{rf}.\ \text{CONSISTENT}((E, \langle \mathtt{co}, \mathtt{rf} \rangle))
\end{aligned}$$

Program semantics Architecturally there is no such thing as a 'program'. Instead, there are only whole machine states. The model then allows us to define what set of configurations are reachable from an initial one, i.e. a 'program'. There are primarily two ways of representing the initial state in these models: either (1) by only considering executions which are `co`-prefixed by the set of writes corresponding to the initial memory configuration; or, (2) by including some special initial event which other events can read from. The choice of representation does not matter, so we arbitrarily pick the first.

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

We can then define the outcomes permitted by the model by the set of states reachable from the initial state of the program: an outcome is permitted iff there exists a consistent execution, prefixed with the initial writes from the program, whose final state matches that outcome:

$$\begin{aligned}
\text{State} &\equiv \text{Memory} \times (\text{ThreadId} \to \text{Registers}) \\
\text{FINAL}(E : \text{Execution}) &= \text{`Final register and memory state of } E' \\
\text{PREFIXED}(\mathit{Init} : \text{State}, E : \text{Execution}) &= \text{`E has co-initial writes corresponding to the initial state}' \\
\text{REACHABLE}(\mathit{Init} : \text{State}, S : \text{State}) &= \exists E : \text{Pre-Execution}, \mathtt{co}, \mathtt{rf}. \\
&\quad \text{let } C = (E, \langle \mathtt{co}, \mathtt{rf} \rangle) \text{ in} \\
&\quad \text{PREFIXED}(\mathit{Init}, C) \\
&\quad \wedge\ \text{CONSISTENT}(C) \\
&\quad \wedge\ S = \text{FINAL}(C)
\end{aligned}$$

Giving semantics to an Arm-A program can be done by collecting the set of reachable consistent executions, from an initial machine configuration (program):

$$[\![P : \text{State}]\!] = \{S : \text{State} \mid \text{REACHABLE}(P, S)\}$$

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

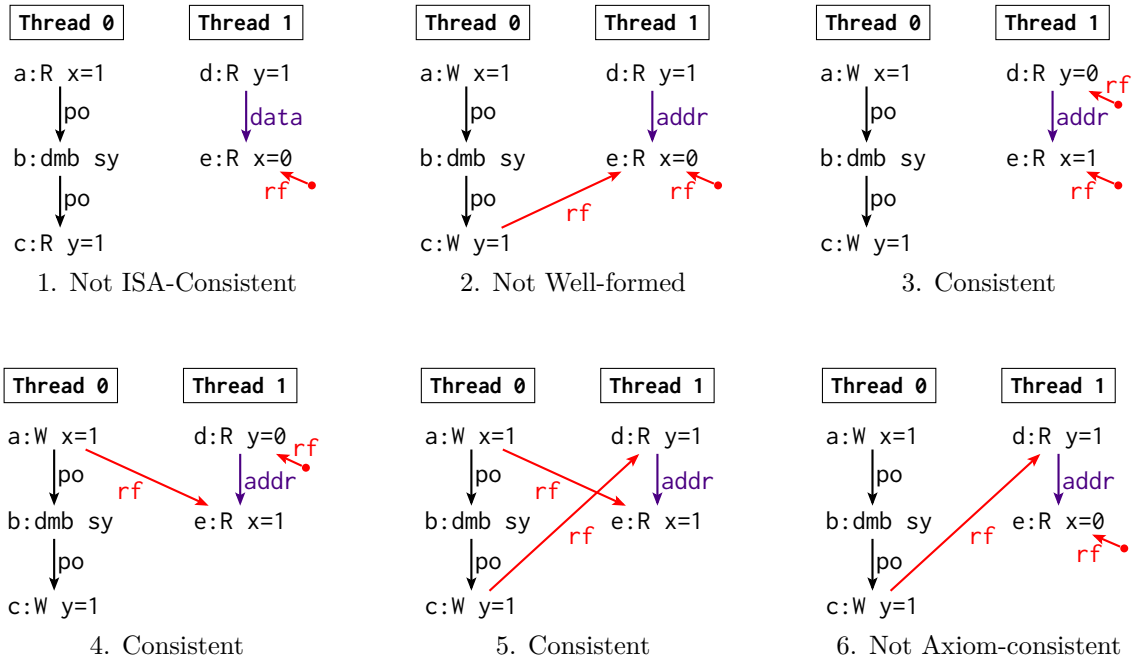


Figure 2.20: Six potential candidate executions for MP+dmb.sy+addr.

An example Consider the classic MP+dmb.sy+addr litmus test, whose code listing can be found in Figure 2.19. The test has two threads, with two store instructions separated by a barrier in the first, and two loads with a syntactic address dependency between them in the second. Thus, it is an instance of the 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 dictate that stores 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 listed 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; those with `co` and `rf` other than those shown would not be well-formed; those 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
STR X0,[X1]
DMB SY
MOV X2,#1
STR X2,[X3]
	LDR X0,[X1]
	EOR X4,X0,X0
	LDR X2,[X3,X4]

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

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

2.4.2 Arm-A axioms

Axiomatic models define *axioms* over candidates, primarily as acyclicity requirements over derived relations over their 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.

Historically, axiomatic models were given as a set of constraints over the derived relations of the model [52, 8]. Contemporary models are described as point-free definitions of relations and acyclicity conditions over them [32, 42]. The derived relations are constructed composing the candidate relations $\mathcal{C}_{\mathrm{R}}$, and the restricted identity relation ($\mathbf{id}_E$, for identity over events with label E), using standard relation operators: union (`|`), intersection (`&`), relation composition (by sequential composition, with `;`), transitive closure ($^+$), and relation inverse ($^{-1}$). The model is then a set of axioms (acyclicity or emptiness) over relations defined in this algebra.

We write these models in the Herd model definition language (often commonly referred to as simply *Cat*), introduced by Alglave et al. [32]. 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 lets the user define either sets of events, or relations over events, using the usual set of set and relational operators, with some custom syntax, reproduced here for quick reference:

- ▷ `R+` for the transitive closure (one-or-more repetitions) of `R`.
- ▷ `[E]` for the identity over events with label `E`, corresponding to the mathematical relation $\mathbf{id}_E$,
- ▷ `[E1|E2|...]` for the set of events with labels which match `E1` or `E2` or so on.
- ▷ `domain(R)` and `range(R)` give the sets of events that are the domain and codomain of a relation `R`.
- ▷ `(E1 * E2)`, is the relation formed by the cartesian product of sets of events with labels `E1` and `E2`, that is, the mathematical relation `range(`$\mathbf{id}_{E1}$`)`×`range(`$\mathbf{id}_{E2}$`)`. `E1` or `E2` can be substituted with an underscore which acts as a wildcard that matches events with any label.
- ▷ `id` for the generalised identity relation over events, which corresponds to $\mathbf{id}_{\_}$;
- ▷ `R?` as a shorthand for relation option, equivalent to `R | id`.

The original `herdtools` Cat language and the `isla-axiomatic` Cat-like model language have diverged over time, but the features described in this section remains common to both.

An Arm-A Cat model A reformulation of the original non-mixed-size multi-copy-atomic Armv8-A model from 2018 [7, 53], can be found in Figure 2.21. The other models presented in this thesis will be extensions 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 [42]. 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.

Ordered-before The main relation, ordered-before (`ob`), is defined on line 27 as the transitive closure of the union of a set of auxiliary ordering relations. These auxiliary relations are:

- ▷ observed-by (`obs`, *l.*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`, *l.*5), which orders events which must not be re-ordered due to syntactic dependencies in the original source program.
- ▷ atomic-ordered-before (`aob`, *l.*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.
- ▷ barrier-ordered-before (`bob`, *l.*17) between events with a memory barrier ordering them.

The axioms The Arm-A model is made up of three axioms: `external` (line 33), which asserts acyclicity of a global ordered-before relation, capturing most of the ordering constraints of the Arm memory model; the `internal` axiom (line 30), 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 36) which asserts that there can be no same-location writes interposing between events of an atomic action.

```
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 (* Ordered-before *)
26 let ob1 = obs | dob | aob | bob
27 let ob = ob1+
28
29 (* Internal visibility
       requirement *)
30 acyclic po-loc | fr | co | rf
       as internal
31
32 (* External visibility
       requirement *)
33 irreflexive ob as external
34
35 (* Atomic:   Basic LDXR/STXR
       constraint   to forbid
       intervening writes. *)
36 empty rmw & (fre; coe) as atomic
```

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

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	po between same-location events
M	Explicit memory events (R|W)	addr/ctrldata	dependencies
A	Read-acquire	co/rf	Witness
L	Write-release	rfi/coi	internal (within thread) rf/co
Q	Weak read-acquire	rfe/coe	external (across threads) rf/co
F	Fences (barriers)	id	identity
ISB	Instruction sychronization barrier		
dmbXY	Memory barrier with kind XY		

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.

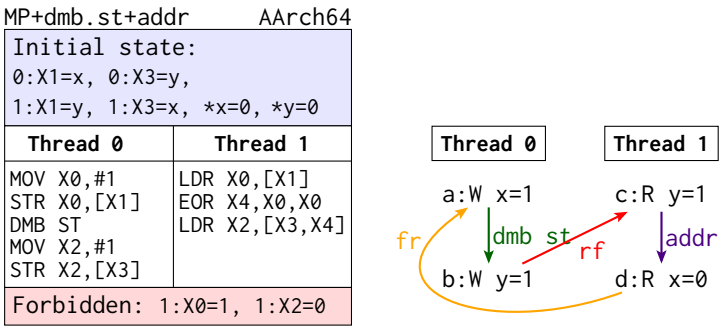


Figure 2.22: MP+dmb.st+addr test code listing and execution diagram.

The interesting candidate execution is the one that has the final state `1:X0=1 ∧ 1:X2=0`. It is this candidate that is shown in Figure 2.22 (right). Being a message-passing (MP) shape, it is characterised by a `po-rfe-po-fre`[1] cycle. In particular, it has the following cycle:

- ▷ `a dmbst 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]; dmbst; [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 axioms 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.

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 [54]. The external axiom usually considers a partially-ordered ordered-before relation 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.

2.5 The isla-axiomatic tool

Throughout this work we will use the `isla-axiomatic` tool [36] to implement executable versions of our axiomatic models. The `isla-axiomatic` tool uses the full ASL specification of the Arm ISA, converted to Sail. The generation of candidates then uses whole machine states, including all instruction fetch and translation table walks as real memory accesses.

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.

[1] A `PodWW Rfe PodRR Fre` cycle in diyone syntax [34].

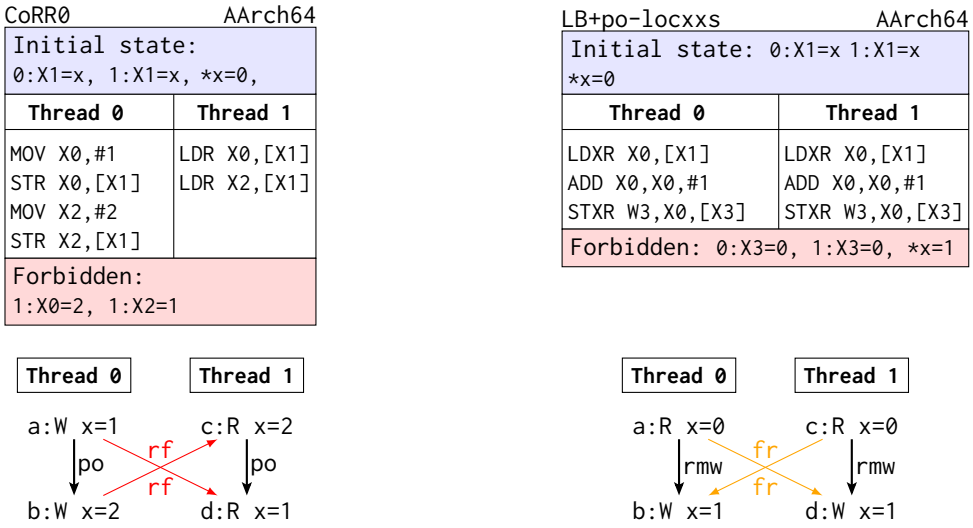


Figure 2.23: Two tests forbidden by the other axioms.
On the left, a variation on coherence which relies on `po-loc` and so is forbidden by the `internal` axiom. On the right, an atomic increment that failed to atomically update the location, forbidden by the `atomic` axiom.

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

Figure 2.24: Extract of the Arm top-level step function.

`isla-axiomatic` candidates Underpinning the `isla-axiomatic` tool is `isla`, a generic symbolic evaluator for Sail programs [36]. `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 (but with symbolic values).

There is a 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 the trace of that instruction. A sketch of the top-level fetch-decode-execute function from our Arm Sail model[1] is given in Figure 2.24.

During symbolic evaluation of the Sail code, when a branch's condition is symbolic and not constrained to one of true or false, the symbolic execution forks. This gives a tree 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

[1] https://github.com/rems-project/sail-arm/blob/master/arm-v9.3-a/src/step.sail#L217

```
enum MBReqDomain = {
    MBReqDomain_Nonshareable,
    MBReqDomain_InnerShareable,
    MBReqDomain_OuterShareable,
    MBReqDomain_FullSystem
}

enum MBReqTypes = {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All}

struct DxB = {
    domain : MBReqDomain,
    types : MBReqTypes,
    nXS : bool
}

union Barrier = {
    Barrier_DSB : DxB,
    Barrier_DMB : DxB, // The nXS field is ignored from DMBs
    Barrier_ISB : unit,
    Barrier_SSBB : unit,
    Barrier_PSSBB : unit,
    Barrier_SB : unit,
}

instantiation sail_barrier with
    'barrier = Barrier
```

Figure 2.25: Arm interface: `isla-cat` definition of barriers.

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 all executions when compiled to SMT are unsatisfiable then 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[1] :

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

Each architecture's Sail specification can then instantiate the `'barrier` type variable with architecture-specific data. For instance, in Armv9-A the `'barrier` type is instantiated with a custom `Barrier` type[2], given in Figure 2.25, which contains a translation of the Arm barrier kind datatype in the ASL/Sail specification.

[1] https://github.com/rems-project/sail/blob/0.18/lib/concurrency_interface/barrier.sail#L75
[2] https://github.com/rems-project/sail-arm/blob/interface-v9/arm-v9.3-a/src/interface.sail#L286

Then the Sail Arm specification can use the `sail_barrier` outcome to generate events in the trace. For example, the `DataSynchronizationBarrier` function is the function which the specification calls on execution of a `DSB` instruction. It is an uninterpreted function in the specification. The Sail model implements that function to output a `sail_barrier` effect[1], which generates a barrier event in the trace when executed:

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

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-cat` does not define a large collection of built-in relations and sets. Instead, it has *accessors*: point-free declarations which define functions over events. Accessors 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:

```
accessor barrier_types: MBReqTypes = .match {
    Barrier_DMB => .types,
    Barrier_DSB => .types,
    _ => default
}
```

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 (`dmb ld`,`dmb sy`, ...) with accessors. An extract of the `isla-cat` definition for Armv9-A[2], for the definition of an example barrier event (`dmbld`, the event set that includes all barrier events that are at least as strong as a `DMB.LD` instruction), is given in Figure 2.26.

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

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

```
accessor F: bool = is sail_barrier

define has_barrier_type(ev: Event, t: MBReqTypes): bool =
    (barrier_types(ev) == t)

accessor is_DxB: bool =
    .match {
        Barrier_DMB => true,
        Barrier_DSB => true,
        _ => false
    }

accessor is_DMB: bool =
    .match {
        Barrier_DMB => true,
        _ => false
    }

define ArmBarrierRM(ev: Event): bool =
    is_DxB(ev) & has_barrier_type(ev, MBReqTypes_Reads)

define DMB(ev: Event): bool =
    F(ev) & is_DMB(ev)

define DMBLD(ev: Event): bool = DMB(ev) & ArmBarrierRM(ev)

define dmbld(ev: Event): bool =
    (* see full code for definitions of dmbsy and dsbld *)
    DMBLD(ev) | dmbsy(ev) | dsbld(ev)
```

Figure 2.26: Arm interface: `isla-cat` definition of barrier accessors for an example barrier kind.

Part I
Instruction fetch

This part is based on: ARMv8-A system semantics: instruction fetch in relaxed architectures [35] 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).

Chapter 3

Relaxed instruction fetching

We now describe the main instruction fetch phenomena and architecture design questions for Arm-A. As usual, this will be done through 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: mostly systems software, such as dynamic loaders, operating system kernels, and hypervisors; and also some usermode, like just-in-time (JIT) compilers. This software forms part of 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.

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 [55], CertiKOS [56, 57], Komodo [58], nor the works of Guanciale et al. [59], nor Baudmann et al. [60], address realistic architecture concurrency, and they use (at best) idealised models of the sequential systems architecture. The CakeML [61, 62] and CompCert [63] 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 [64, 65]. However, those architectures have a very different programmer model than Arm presents, not requiring explicit instruction cache maintenance.

In this part 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 requires specific additional synchronisation to correctly pick up code modifications. Exactly what these are was not entirely clear in the Arm-A architecture text at the time this work was done (up to version D.a [66]).

We clarify this situation, developing precise abstractions that bring the instruction-fetch part of Arm-A system behaviour into the domain of rigorous semantics. 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. At the time of this work Arm intended to officially incorporate a version of the model into their architecture, [67]. Since then, Arm have developed an instruction fetch model as part of the official Arm memory model [68, B2.3]. Detailed comparison of that model with the one presented here will be an important topic for future work, although we do not believe the architectural intent has changed.

Overview In this chapter, we begin by recalling the informal architectural guarantees that the Arm-A architecture provides, and the ways in which real-world software systems such as Linux, the JavaScript and WebAssembly JITs, and other language implementations modify 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 cache maintenance (Ch.4) in an abstract-microarchitectural style (following the Flat model of Flur, Pulte et al. [7], cf. §2.3) capturing the architectural intent as we understand it. We make the operational model executable as a test oracle by integrating it into the RMEM tool [51], with optimisations that make it possible to exhaustively execute example litmus tests.

We give a more concise presentation of the model (Ch.5) in an axiomatic style (in the style of Alglave et al.'s herd models [32], as an extension to the official Armv8 model of Deacon [7], cf. §2.4), and intended to be extensionally equivalent to the aforementioned operational semantics. We give an executable-as-a-test-oracle formulation of the model, in an extension to the `isla-axiomatic` tool, which with Armstrong and Campbell we extend to support instruction fetching litmus tests.

We validate all this (Ch.6), in two ways: by extensive discussion with Arm staff and systems software engineers, 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 [33, 69]. We

then compare hardware and the two models on a suite of 1456 tests, automatically generated with an extension of the diy tool [70]. We also check the operational and axiomatic models for regressions against the sets of previous non-ifetch tests. We found no regressions and no test which distinguishes the models, and all data was consistent with hardware observations, except for one case where our testing uncovered a hardware bug on a Qualcomm device.

Caveats and Limitations Our operational semantics are integrated with a substantial fragment of the Sail Armv8-A ISA (similar to that used for CakeML [71]), but not yet with the full ISA model [45, 10, 11, 72]; this is a matter of additional engineering and is future work. 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 or 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 for performance, including buffering, caching, prediction and prefetching, and pipelining. 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. However, writes to instruction memory are relatively rare, so hardware designers have adopted much more aggressive caching mechanisms specifically for those accesses. The Arm architecture carefully does not mandate exactly what these may be, permitting a wide range of possible hardware implementations. 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. Figure 3.1 shows a typical micro-architectural design: that of the Arm Cortex-A53, with independent per-thread instruction and data caches, which unify into a global cache before memory. Data flows out of the core into the L1 data cache, and then from the data cache to the instruction cache or out to memory.

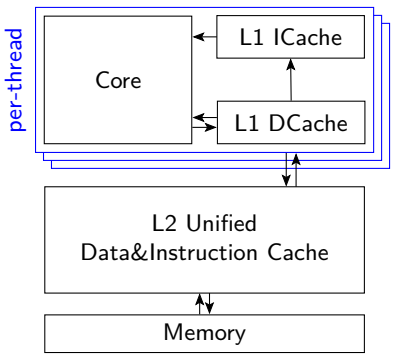


Figure 3.1: Block diagram of the Arm Cortex-A53 [73], with simplified data and instruction flow [74].

Modifying code and maintaining coherence In usermode models the caches are, aside from performance implications, invisible to the programmer, with the hardware cache protocol managing them automatically. In contrast, the caching of instruction data exposes details to the programmer that would otherwise be invisible: such as the cache line size (as caches may cache arbitrarily large 'lines' of memory at a time) and physical hierarchy of the caches.

This is because instruction caching is not necessarily coherent with data memory accesses[1] [66, B2.4.4 (B2-114)]:

> the architecture does not require the hardware to ensure coherency between instruction caches and memory

Hence, programmers must use the explicit cache maintenance instructions [66]:

> If software requires coherency between instruction execution and memory, it must manage this coherency using Context synchronization events and cache maintenance instructions.

The manual further gives a sufficient sequence [66]:

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

At first sight, this may be entirely mysterious. This and the following chapters establish precise semantics for each of the above instructions, explaining why each is required. However, 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 any 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`.
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` waits for the cache invalidation to complete.
5. The final `ISB` flushes this core's pipeline, forcing a re-fetch of all program-order-later instructions.

Central to this sequence is the concept of *cache maintenance*. Caches will fetch data automatically, but may require explicit operations to remove old cached values. Cache maintenance operations can generally be split into one of two kinds:

- ▷ Cleans force a write-back of a cache line, pushing any potential cached copies further down the cache hierarchy.
- ▷ Invalidations remove cached copies of a whole line.

From the programmer's perspective, invalidations are destructive: if the removed cached line's data did not exist further down the cache hierarchy, or in memory, then the data may be lost entirely. However, cleans push data further out thereby making it *more* widely visible. 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 (see §3.10 for an explanation of what the different points mean). Arm therefore provide a large collection of cache maintenance instructions, of which the most relevant for this part are:

[1] Version J.a of the Arm architecture reference manual includes the word 'not' here, which is a typographical error.

Instruction	Operation
DC CVAU	Clean Data&Unified Caches by VA to PoU
DC IVAC	Invalidate Data&Unified Caches by VA to PoC
DC CVAC	Clean Data&Unified Caches by VA to PoC
DC CIVAC	Clean&Invalidate Data&Unified Caches by VA to PoC
IC IVAU	Invalidate Instruction Caches by VA to PoU
IC IVAC	Invalidate Instruction Caches by VA to PoC
IC IALLU	Invalidate Local Instruction Cache to PoU
IC IALLUIS	Invalidate All Instruction Caches to PoU

We discuss more about the relationship between these cache maintenance operations in §3.10.2.

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 in more detail in §3.14.

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 [66, 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 leaves the exact semantics of instruction fetch and those cache maintenance instructions unclear.

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 write new code during execution and modify existing code at runtime. In the JavaScript JIT, this modification happens inside a single thread and so is relatively straightforward. The WebAssembly case is more complex, as one thread is modifying the code being concurrently executed by another.

In practice, software typically does not use the above sequence verbatim. For example, when synchronising a range of addresses all at once, the software may group the DCs and ICs of different addresses together. Additionally, the final ISB may be subsumed by other instruction synchronisation e.g. from exception entry or return. Software threads may also be migrated (by the OS or hypervisor) from one hardware thread to another, potentially interrupting such an instruction cache maintenance sequence. Moreover, for security reasoning, we have to be able to bound the possible behaviour of arbitrary code. For all these reasons, we must consider the effect of each instruction individually and how they compose, and cannot simply assume a canned sequence.

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 *concurrently modifiable*, which is defined as: simple branches, supervisor/hypervisor/secure monitor calls, the ISB (instruction synchronisation) barrier, the BRK (breakpoint) instruction, 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. Stronger constraints for constrained unpredictable is an area under investigation by Arm.

The following W+F test (Figure 3.2, p.54) 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 3.2: 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. However, because at least one of those is not a 'concurrently modifiable' instruction (not in the set of 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 more constrained 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.

Debuggers and breakpoints One challenge in the definition as given by Arm is that it forbids replacing arbitrary instructions with breakpoints concurrently. Other architectures (such as IBM Power) simply require that at least one of the instructions is concurrently modifiable, not both.

In practice, debuggers replace instructions with breakpoints (the `BRK` instruction) regardless. Further work is required to investigate whether a strengthening could be made to the Arm architecture to permit this in general.

Conditional branches In version D.a (and earlier) of the Arm architecture reference manual, it made clear that, for branches with conditions (`B.cond`) which are overwritten by other `B.cond` instructions, 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 [66, B2-94]. In version E.a, this condition was removed entirely, meaning `B.cond` instructions were not permitted to be concurrently updated at all [75, 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 [76, B2-130], with the stronger semantics that one 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 3.3: Code listing for test W+F+branches.

For example, the W+F+branches test (Figure 3.3) 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 'constrained unpredictable' behaviour. Under the G.b and later text, the test has well-defined behaviour, but is now 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 The Arm architecture does not give branch instructions any instruction synchronisation effects. Instead, the architecture relies on explicit synchronisation instructions (see §3.6). This is in contrast to other architectures. For example, x86 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, p.27) 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`.

Instruction fetches are not necessarily coherent: an 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. However, they are not completely incoherent and still must respect some properties, giving rise to three new forms 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 [66, B2.4.4]. This is illustrated by the CoFF test (Figure 3.4), which is a variant of the CoRR1 test (Figure 2.5, p.27) test for coherence discussed earlier, but where the explicit reads of the CoRR shape are replaced by the implicit reads of fetching the instructions.

CoFF AArch64

Initial state: 0:W0="B l1", 0:X1=f		
Thread 0	Thread 1	Common

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

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

hw-refs: YNNNY

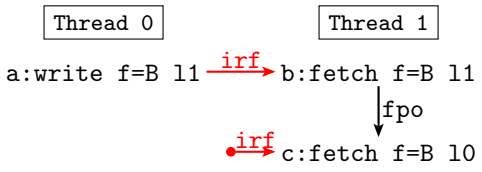


Figure 3.4: Code listing and execution diagram for CoFF.

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`). 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.

The interesting potential execution of this test is the one in which the first call to `f` fetches and executes the newly-written `B l1`, before the second call fetches and executes the original `B l0`. The execution shown in Figure 3.4 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.

As in Chapter 2, we use `po` and `rf` edges for the program-order and reads-from relations, together with new relations:

- `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, for example, the `irf` edge for event `c` in Figure 3.4. We discuss these new candidates in more detail later (Chapter 5).

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, we show only the subgraph of interesting events, as in the CoFF execution diagram in Figure 3.4.

For Arm, this execution is both architecturally allowed and experimentally observed. This is shown in the test listing in Figure 3.4 in the line underneath the final state beginning with `hw-refs`. This line is a condensed table, where each column represents one hardware device and the entry represents 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 `-`). The final `hw-refs` line from CoFF (Figure 3.4, p.55), annotated with the names of the devices (see §6.3 for a more detailed discussion of the hardware testing) is as follows:

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

Where the devices are:

h955-a53	Qualcomm Snapdragon 810 (cluster of 4x Arm Cortex A53)
openq820	Qualcomm Snapdragon 820 (4x Qualcomm Kryo cores)
h955-a57	Qualcomm Snapdragon 810 (cluster of 4x Arm Cortex A57)
nexus9	NVIDIA Tegra K1 (with 2x NVIDIA Denver cores)
s905	Amlogic 905 (with 4x Arm Cortex A53 cores)

3.4.2 Data-to-Instruction coherence

Fetching from a particular write does imply that program-order-later reads of the same address will see that write, or something newer. This is a *data-to-instruction* coherence property, illustrated by CoFR (Figure 3.5). 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

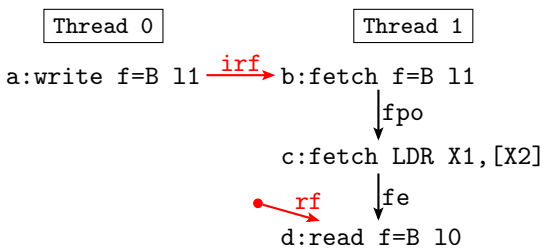


Figure 3.5: Code listing and execution diagram for CoFR.

This ordering guarantee was not clear in the Arm prose specification at the time of this work [66, 77, 76], 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).

This test represents the most fundamental kind of data-to-instruction coherence: that data must become visible to the coherent data side before instruction accesses. However, it alone gives no guarantee when the instruction accesses are guaranteed to see it. We shall see later (§3.6) that instruction cache maintenance will generally be required to guarantee future instruction fetches read from coherence-latest data writes, but that the hardware may announce that it provides a stronger kind of data-to-instruction coherence guarantee rendering such cache maintenance unnecessary (§3.14).

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 3.6).

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: YYYNY

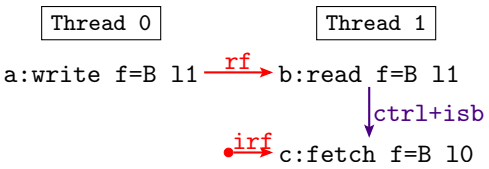


Figure 3.6: Code listing and execution diagram for CoRF+ctrl-isb.

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 was also not explicit in the prose [66, 77, 76], 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 since been evicted or explicitly invalidated, 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.

However, even if the instruction cache is empty (e.g. by manually clearing it with appropriate cache maintenance instructions, see §3.10 and the SM.F+ic test (Figure 3.19, p.64)) the test may still be observed as the instruction fetches and instruction cache fills need not read-from the coherence-latest write.

Software must then use cache maintenance operations to achieve such guarantees (§3.6). However, much like with data-to-instruction coherence, the hardware may announce that it provides a kind of instruction-to-data coherence guarantee, rendering data cache maintenance unnecessary (§3.14).

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, p.24). Here, we replace one or both of the 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 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 3.7).

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

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

hw-refs: -----

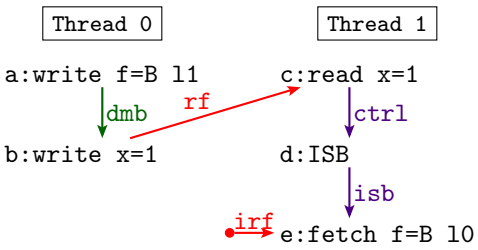


Figure 3.7: Code listing and execution diagram for MP.RF+dmb+ctrl-isb.

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 to perform the entire cache synchronization sequence (§3.2), not just a DMB, to forbid this outcome. Adding that full cache synchronisation sequence gives test MP.RF+cachesync+ctrl-isb (Figure 3.11, p.60), described in more detail later (§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 3.8), in which Thread 0 writes some data (to x) and then writes to the code concurrently being executed by Thread 1, as a kind of message pass. If Thread 1 fetches the new instruction 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]	BL f
DMB ISH	MOV X0,X10
STR W2,[X3]	LDR X1,[X2]

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

hw-refs: NNNN-

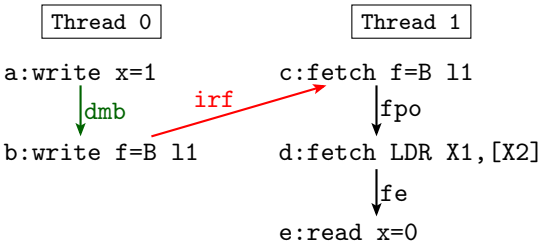


Figure 3.8: Code listing and execution diagram for MP.FR+dmb+fpo-fe.

This was not clear from the architectural prose at the time of the work, but this outcome is intended to be architecturally forbidden. This is for similar reasons as the previous CoFR test (Figure 3.5, p.56): 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 to ensure correct execution of the new instructions.

Test SM (Figure 3.9) 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

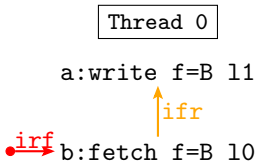


Figure 3.9: Code listing and execution diagram for SM.

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 $\texttt{ifr} = \texttt{irf}^{-1};\texttt{co}$. If the fetch were a data read, this would be a forbidden coherence shape (CoWR). As it is, it is architecturally allowed, as described explicitly by Arm [66, B2.4.4], and it is experimentally observed on all devices we have tested. Microarchitecturally, there are a number of possible explanations, each of which are sufficient to explain the test: fetching b out-of-order with respect to a, fetching b from a stale entry in the instruction cache, or fetching b from memory after a has propagated but before it reaches the point the instruction fetch will see it. We will see that to forbid this test, and guarantee fetching the new instruction, one needs to account for all of those possibilities.

3.6.1 Synchronisation on a single thread

As we described earlier (§3.2), the Arm architecture provides cache maintenance instructions to synchronise the instruction and data streams: the DC data-cache clean instruction, and the IC instruction-cache invalidate instruction. 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

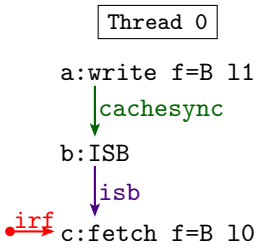


Figure 3.10: Code listing and execution diagram for SM+cachesync-isb.

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 later (Chapter 4).

3.6.2 Broadcast cache maintenance

The hardware threads writing new instructions, performing the necessary cache maintenance, and finally fetching the new instructions, may all be distinct. 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 3.11), 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 its own 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]
DC CVAU,X1
DSB ISH
IC IVAU,X1
DSB ISH
STR X2,[X3]
	LDR X0,[X2]
	CBNZ X0,l
	l:ISB
	BL f
	MOV X1,X10

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

hw-refs: NYNNN

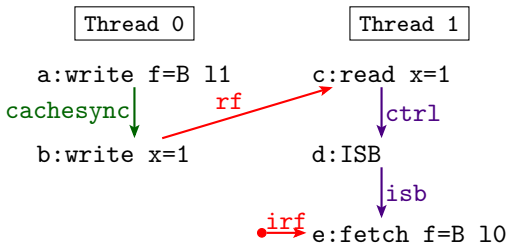


Figure 3.11: Code listing and execution diagram for MP.RF+cachesync+ctrl-isb.

Note the positive observation on this test, despite being forbidden. We return to this in §6.4.1.

In-order fetches One can make both writes of the MP shape 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 [78, 79]. Without the full cache synchronisation sequence on Thread 0, this is allowed as in MP.FF+dmb+fpo (Figure 3.12). Inserting the full cache maintenance sequence on the producer thread forbids the outcome, see the MP.FF+cachesync+fpo test (Figure 3.13, p.61).

MP.FF+dmb+fpo AArch64

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

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

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

hw-refs: YYYNY

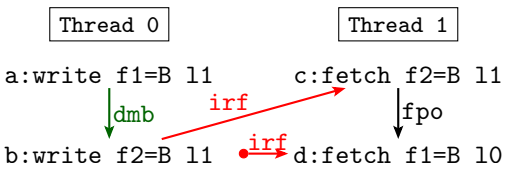


Figure 3.12: Code listing and execution diagram for MP.FF+dmb+fpo.

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]
DC CVAU,X1
DSB ISH
IC IVAU,X1
DSB ISH
STR W2,[X3]
	BL f2
	MOV X0,X10
	BL f1
	MOV X1,X10

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

hw-refs: NNNNN

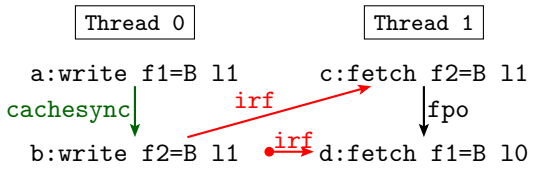


Figure 3.13: Code listing and execution diagram for MP.FF+cachesync+fpo.

This may be surprising at first sight, 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 was not clear in the prose at the time of the work, but, concurrent with this work, Arm were independently strengthening their definition to guarantee this ordering.

Software thread migration The cache maintenance sequence need not be contiguous, or even all on a single thread. In general, it may be split up with many instructions between, and be over multiple threads. This can be seen in the ISA2.F+dc-dmb+dsb-ic-dsb+ctrl-isb test (Figure 3.14), 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-threaded 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.

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

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

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

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

hw-refs: NN---

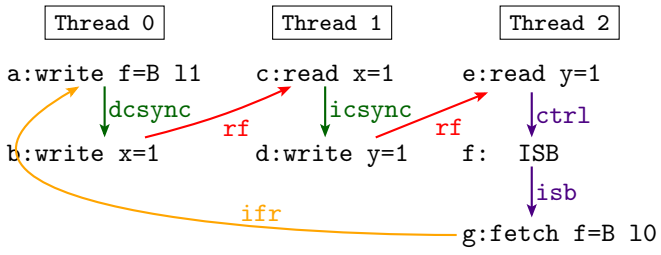


Figure 3.14: Code listing and execution diagram for ISA2.F+dc-dmb+dsb-ic-dsb+ctrl-isb.

This works because the IC IVAU is broadcast to all threads [66, 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 a context-synchronising operation (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.

3.7 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 observably 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

When the destination of a branch is computed, e.g. 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 3.15), 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: -----

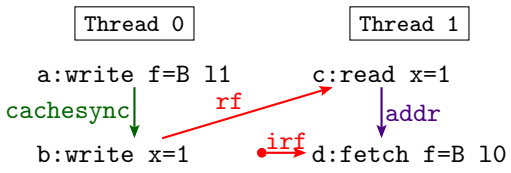


Figure 3.15: Code listing and execution diagram for MP.RF+cachesync+addr.

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

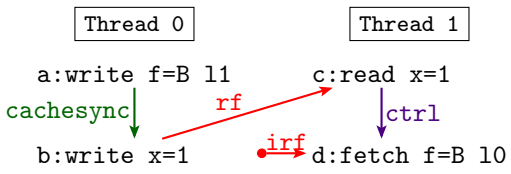


Figure 3.16: Code listing and execution diagram for MP.RF+cachesync+ctrl.

3.8 Multi-Copy Atomicity

For data accesses, the question of whether they are *multi-copy atomic* is a crucial one in relaxed architectures. IBM POWER, ARMv7, and pre-2018 ARMv8-A are *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 all multi-copy atomic (or 'other multi-copy-atomic' in Arm terminology) [7, 6, 66]: 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.

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 3.17)) 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

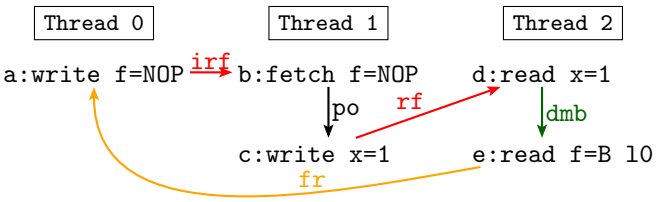


Figure 3.17: Code listing and execution diagram for WRC.F.RR+po+dmb.

3.9 More on instruction caches

Test CoFF (Figure 3.4, p.55) 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 3.18) 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. If the instruction cache could hold at most one value for each location, then after a DC an instruction fetch could read at most two values: one from that and one from data memory. Although it is unlikely that hardware would cache multiple values in the instruction cache, the desire for the simpler and weaker option means the architectural intent is to allow it, and we follow that in our models.

MP.RFF+dc-dsb+ctrl-isb-isb AArch64

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

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

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

hw-refs: NNNNN

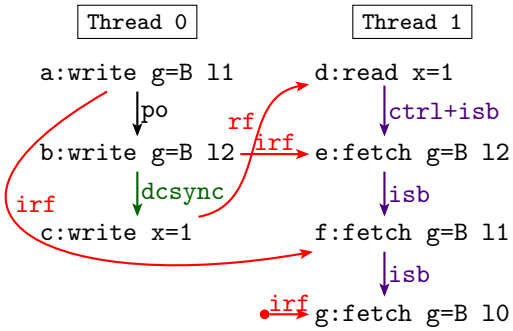


Figure 3.18: Code listing and execution diagram for MP.RFF+dc-dsb+ctrl-isb-isb.

3.10 Points of Unification and Coherency

Although instruction caches behave incoherently, at some point in the memory hierarchy all the caches unify, and all agents agree on the value at that point. Arm define multiple such points:

- ▷ The Point of Unification (for each shareability domain, see below) is where the data and instruction caches unify, and an instruction cache fill is guaranteed to see writes beyond.
- ▷ The Point of Coherency (for the whole system) is where all agents (CPUs, devices, etc.) memory accesses unify, and are all guaranteed to see the same writes beyond that point.

Cleaning the data cache, with the DC instruction, forces previous writes to become visible to instruction fetch, but does not restrict the set of values that could be in the instruction cache. It does this by pushing the writes past the Point of Unification (the point where the instruction and data caches become unified).

There may be multiple Points of Unification, one for each shareability domain. For example, one for each individual core, where its own instruction and data memory become unified, and one for each cluster of CPUs where all the caches eventually unify within that cluster. When a cache maintenance operation is performed by VA to 'the' Point of Unification, the VA is translated and an entry in the translation tables marks the location as either non-shareable or inner-shareable (see §7.3.2). This determines which Point of Unification the cache maintenance operation should be performed *to*.

Fetching a value implies that its write has reached at least that core's PoU, but not necessarily the PoU of any wider domain, even if the write originated from a different core. Consider test SM.F+ic (Figure 3.19).

SM.F+ic AArch64

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

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

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

hw-refs: NNNN-

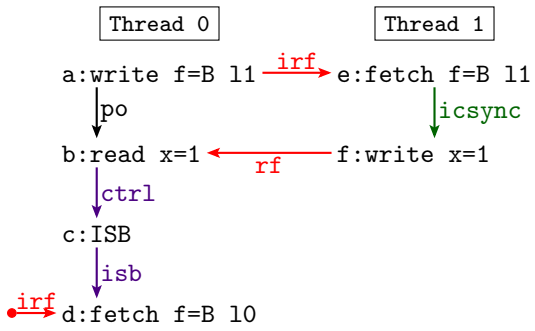


Figure 3.19: Code listing and execution diagram for SM.F+ic.

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.

We have not so far observed this in practice. However, there is a mechanism which gives a compelling argument to permit this behaviour. The Points of Unification of systems are not necessarily hierarchical: 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 [80]. The line could be migrated from Thread 0 to Thread 1, and 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 on multi-copy atomic machines, if they implemented pre-PoU DDI. Therefore, the architectural intent is to allow this behaviour.

With insufficient synchronisation of the data caches, there is theoretically no limit to how far back in time the fetches could read from. Recall that in the MP.RF+dmb+ctrl-isb test (Figure 3.7, p.58), the full cachesync sequence was required to forbid the 'bad' behaviour. Test FOW (Figure 3.20, p.65) 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.

FOW AArch64

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

Thread 0	Thread 1	Thread 2	Common

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

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

hw-refs: NN--N

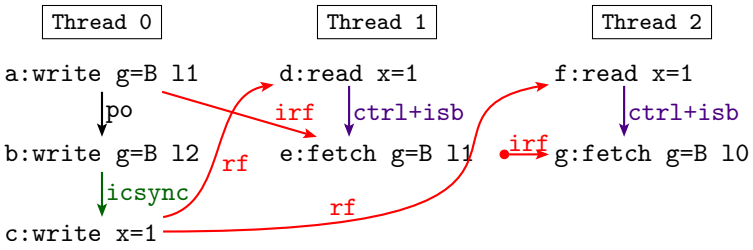


Figure 3.20: Code listing and execution diagram for FOW.

This was not clear in the existing architecture text. It is a case where the architecture design is not very constrained. On the one hand, it has not been observed, and it is thought unlikely that hardware will ever exhibit this behaviour: it would require keeping multiple writes in the coherent part of the data caches, before the *Point of Coherency*, which would require more complex cache coherence protocols, rather than a single dirty line. On the other hand, there does not seem to be any benefit to software from forbidding it. Therefore the architects are forced to make a decision. In this case, the more permissive model is also the simpler one. It makes it easier for programmers to understand and to provides more flexibility for future microarchitectural optimisations. Our models therefore allow the above behaviour.

In theory, once a write passes the Point of Unification for the whole system (the point where all caches eventually unify) then any writes coherence before that write cannot be seen at all by instruction fetches any more, even without explicit DC instructions. We do not attempt to model this since a general notion of a Point of Coherency is not required in the models, as it is only distinguished by device memory or DMA, which we do not model here.

3.10.1 Late unification

There is a final question: can any Point of Unification be *after* the Point of Coherency? The architecture says [81, D7.5.8, p.5777]:

> **Point of Coherency** The point at which all agents that can access memory are guaranteed to see the same copy of a memory location [...]

The question is whether the instruction-fetch-units of a system amount to separate *agents* in the system domain. The definition of shareability domains implies not [81, B2.4.2, p.199]:

> **Shareability** [...] Marking a memory location as shareable [...] requires hardware to ensure [...] the location is coherent for all agents in that domain. [...]

Since the instruction fetch unit is within the same shareability domain, but are not coherent, they must not be *agents* for the purpose of coherency. Therefore, writes passing the Point of Coherency would not guarantee instruct ion fetches to see the writes coherently. We never observed such hardware in practice, and believe it unlikely hardware would be designed in such a way that the PoC would be before a PoU, and therefore leave it open to the architects to clarify.

3.10.2 Promotion

Cache maintenance operations form a partial order, where if one cache operation is sufficient, then a stronger one is also sufficient. Assuming the PoU is before the PoC (see Late unification) that order is:

$$\texttt{DC CVAU} \leq \texttt{DC CVAC} \leq \texttt{DC CIVAC}$$
$$\texttt{DC IVAC} \leq \texttt{DC CIVAC}$$

$$\texttt{IC IVAU} \leq \texttt{IC IVAC}$$
$$\texttt{IC IVAU} \leq \texttt{IC IALLUIS}$$

In litmus tests we will use the least operations in this order, typically `DC CVAU` and `IC IVAU`.

In a program which uses one of these instructions, that instruction can be *promoted*: replaced with a stronger cache maintenance operation. Often software will want to use the least sufficient maintenance as they are typically the most efficient and give the best performance. However, sometimes operating systems and hypervisors will 'trap' cache maintenance operations to emulate or promote them automatically, either for virtualisation or as part of the resolution to CPU errata. In those cases, software must ensure it only promotes cache maintenance consistent with the above ordering.

3.11 Cleans and invalidates are like reads and writes

Recall that we have an asymmetry between the required synchronisation around `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 an `IC` is not.

This is because the clean of the `DC` is ordered much like a read. 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 [81, pp. 5790-5791], and an `IC` instruction always requires an `DSB` to order accesses before it [81, p. 5791].

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 memory ordering constraints as data reads. This, in turn, means that `DC C_VA_` instructions wait for program-order previous reads and writes (and other `DC`s) of the same location just as reads do (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 potentially lost. Data cache 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, such that reads after the invalidation cannot read from writes from before it. `IC` invalidations behave slightly differently, with some extra requirements about in-order fetching (see test MP.FF+dmb+fpo (Figure 3.12, p.60)), and without constraining future *data* reads, and 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.

For `DC` and `IC` instructions, which are loosely analogous to loads and stores from the specified addresses, we similarly have to consider whether or not dependencies from the calculation of their addresses are respected. Test MP.R.RF+addr-cachesync+dmb+ctrl-isb (Figure 3.21) 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.

This was unclear in the prose at the time of this work, but Arm have since decided 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.

MP.R.RF+addr-cachesync+dmb+ctrl-isb AArch64

Initial state: 0:X1=z, 0:W2="B l1", 0:X3=g, 0:X5=1, 0:X6=y, 1:W1="B l2", 1:X2=g, 1:X3=1, 1:X4=z, 2:X2=y, [x]=0, [y]=0

Thread 0	Thread 1	Thread 2	Common

LDR X0,[X1]
STR W2,[X3]
EOR X4,X0,X0
ADD X4,X4,X3
DC CVAU,X4
DSB ISH
IC IVAU,X4
DSB ISH
STR X5,[X6]
	LDR W0,[X2]
	STR W1,[X2]
	DMB SY
	STR X3,[X4]
		LDR X0,[X2]
		CBNZ X0,l
		l: ISB
		BL g
		MOV X1,X10
			g: B l0
			l2: MOV X10, #3
			RET
			l1: MOV X10, #2
			RET
			l0: MOV X10, #1
			RET

Forbidden: 0:X0=1, 1:W0="B l1", 2:X0=1, 2:X1=1

hw-refs: NN--N

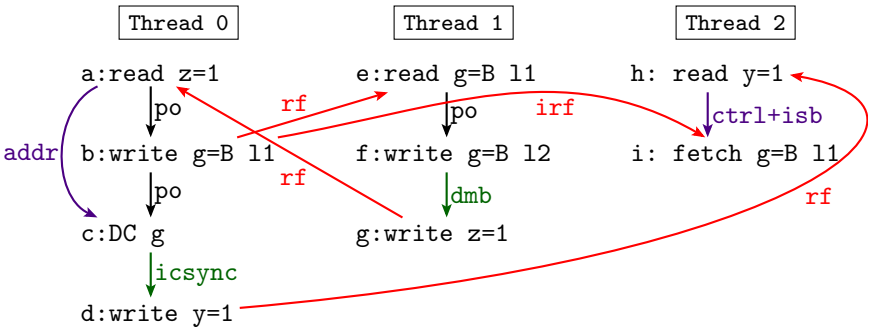


Figure 3.21: Code listing and execution diagram for MP.R.RF+addr-cachesync+dmb+ctrl-isb.

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 [40, §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 [66, 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 [40, §10.5]. The MP.RRF+dmb+addr-cachesync-isb test (Figure 3.22) below illustrates this, in which program-order previous load/store addresses may not be determined when the DC executes. Arm clarified that the architectural intent (which was not clear from the architectural text at the time of this work) 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: N-N--

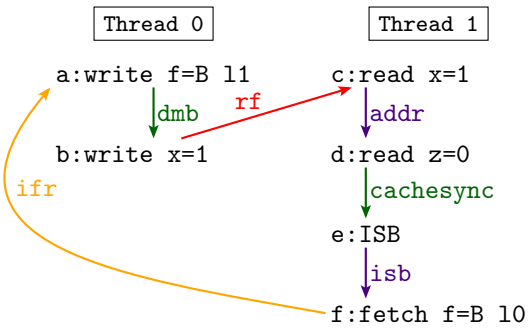


Figure 3.22: Code listing and execution diagram for MP.RRF+dmb+addr-cachesync-isb.

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 extend this to cover all the natural thread-local same-address ordering constraints as normal 'data' loads. For example, DCs are ordered with respect to program-order-earlier same-location loads as in CoRF+cachesync-isb (Figure 3.23), and may be re-ordered with respect to program-order-later same-location loads, as in MP+dmb+addr-dc (Figure 3.24).

Note that these have not yet been confirmed with Arm architects; where the test final state has a question mark, the stated results come from our models and await architectural decision.

AArch64
CoRF+cachesync-isb

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: -----

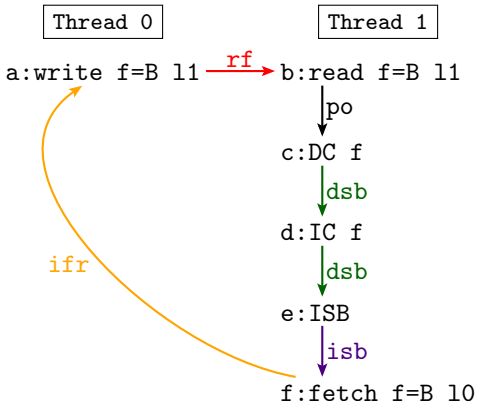


Figure 3.23: Code listing and execution diagram for CoRF+cachesync-isb.

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 DSB SY STR X2,[X3] // b	LDR X0,[X1] // c EOR X5,X5,X0 ADD X5,X5,X3 DC CVAU,X5 // d LDR X2,[X3] // e
Allow?: 1:X0=1, 1:X2=0	

hw-refs: -----

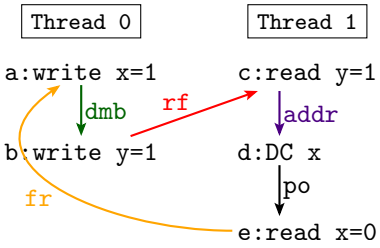


Figure 3.24: Code listing and execution diagram for MP+dmb+addr-dc.

3.12 Same-cache-line ordering

Arm-A has an architected *cache line of minimum size*. There are two cache line minimum sizes: 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 does not impose additional ordering constraints [16], unless one of the accesses is a cache maintenance operation. For example, the SB+scls test (Figure 3.25), which is a variation of the classic store buffering example where the two locations are to the same cache line, is still allowed as the reads and writes of different locations (even within the same cache line) are not ordered.

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.

This is separate to concerns about mixed-size accesses, which we consider in §3.13, where a program writes to the same location with architected writes of different 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] LDR W2,[X3,#4]	STR W0,[X1,#4] LDR W2,[X3]
Allow?: 0:W2=0, 1:W2=0	

hw-refs: -----

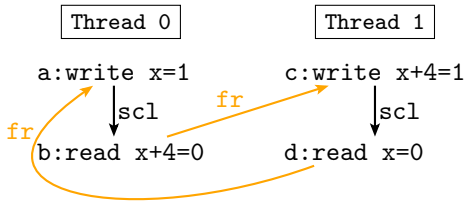


Figure 3.25: Code listing and execution diagram for SB+scls.

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, as in SM+sclcachesync-isb (Figure 3.26)

AArch64
SM+sclcachesync-isb

Initial state: 0:W0="B l1", 0:X1=f, 0:X2=g	
Thread 0	f
STR W0,[X1] DC CVAU, [X2] DSB ISH IC IVAU, [X2] DSB ISH ISB BL f	f: B l0 l1: MOV X0,#2 RET l0: MOV X0,#1 RET
Forbidden: 0:X0=1	

hw-refs: -----

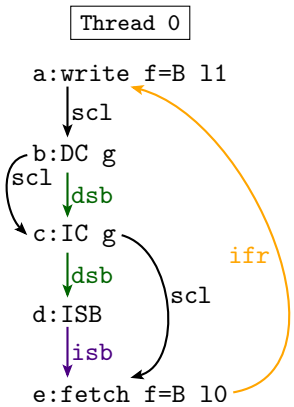


Figure 3.26: Code listing and execution diagram for SM+sclcachesync-isb.

[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.

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 SM8+sclcachesync-isb test (Figure 3.27), 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 can occur in practice, as code often gets loaded from some other memory by means of some memory copying code, which may copy bytes using instructions whose accesses are not naturally instruction-sized, before they are 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 all combinations of the writes correspond to instructions which are in the set of modifiable instructions. One can also delete the final three `STRB` instructions (events `b-d`) from the test, and not affect the result (it is still forbidden).

SM8+sclcachesync-isb AArch64

Initial state:
0:<W0,W1,W2,W3>="B l1"
0:X1=f, 0:X2=g

Thread 0	f
STRB W0,[X4,#0] // a	f: B l0 // h
STRB W1,[X4,#1] // b	l1: MOV X0,#2
STRB W2,[X4,#2] // c	RET
STRB W3,[X4,#3] // d	l0: MOV X0,#1
DC CVAU, [4] // e	RET
DSB ISH	
IC IVAU, [X4] // f	
DSB ISH	
ISB // g	
BL f	

Forbidden: 0:X0=1

hw-refs: -----

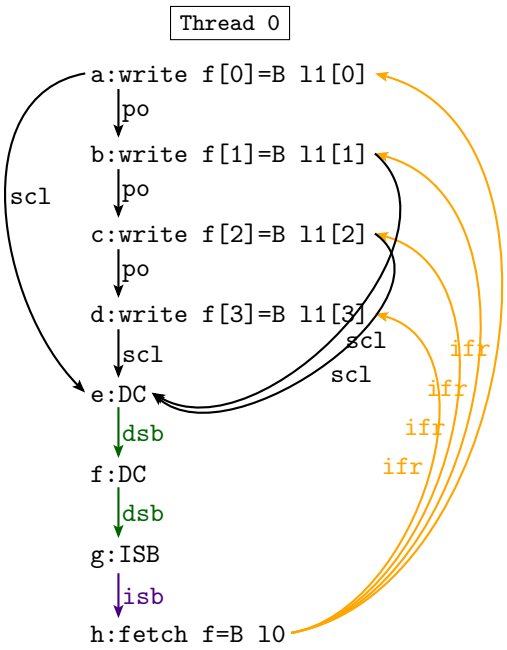


Figure 3.27: Code listing and execution diagram for SM8+sclcachesync-isb.

It is less clear in the architectural prose (even as of the most recent version, J.a [81]) 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 3.28, p.72), or across multiple threads as in W+F+mixed (Figure 3.29, p.72). We do not discuss this in detail, and we 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.

SM+mixed AArch64

Initial state: 0:W0="B l1"[3], 0:X1=f	
Thread 0	f
STRB W0,[X1,#3] // a, b BL f	f: B l0 // c l1: MOV X0,#2 RET l0: MOV X0,#1 RET
Final state: Unpredictable?	

hw-refs: -----

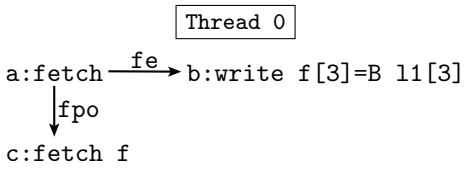


Figure 3.28: Code listing and execution diagram for SM+mixed.

W+F+mixed AArch64

Initial state: 0:W0="B l1"[3], 0:X1=f	
Thread 0	Thread 1
STRB W0,[X1,#3] // a	BL f // b
Final state: Unpredictable?	

hw-refs: -----

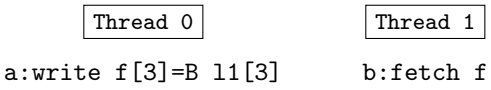


Figure 3.29: Code listing and execution diagram for W+F+mixed.

3.14 Cache type strengthening: IDC and DIC

Implementations may announce that they provide stronger guarantees through two fields in the cache type identification register (`CTR_EL0`). They are the `IDC` and `DIC` fields. The value of these fields then inform software whether each of the cache maintenance instructions are required.

`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. As the names suggest, these fields are related to the kinds of coherence introduced in Section 3.4.

If implementations choose to advertise that one or other of the cache maintenance operations are not required, then those cache maintenance instructions simply become hints or `NOP`s, so defensive cleans and invalidations will not be harmful to the program.

None of the devices we tested had either strengthening enabled.

3.14.1 IDC

When `CTR_EL0.IDC` is 1, the `DC` instruction is not required as part of the sequence [81, 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 3.19, p.64) is forbidden.

3.14.2 DIC

When `CTR_EL0.DIC` is 1, the `IC` instruction is not required as part of the sequence [81, p. 201].

In-order fetching Recall that instruction fetches must either happen in-order, or the `IC` instruction must touch the internal fetch queues of the individual threads (§3.5). 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 expressed (which we shall see in Chapter 4).

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 [65] 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 [82], extending a version of *Concurrent Abstract Predicates* (CAP) [83] for generalised von-Neumann machines.

Goel et al.'s work on verification of x86 machine code programs [84, 85] includes a system step relation, based on their idealised x86 instruction models in ACL2. This model fetches instructions from memory, but avoids the complexity of caches and pipelines [86].

Lustig et al. describe a framework for concurrent models, with relaxed behaviours, for machine code x86 programs based on stages of hardware micro-operations [87]. 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 [55] included self-modifying patterns, but assumed the correctness of the required cache maintenance, without producing tight architectural models of the individual instructions.

CertiKOS [56, 57] 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 [88] 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, 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 [89], as an extension to the ARMv8-A axiomatic model, and for Intel x86 [90] 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 later (§8.10).

During this work, Arm informally confirmed they would adopt the model (subject to necessary updates and changes of architectural intent) [67].

Independent work by Arm, after the conclusion of this work, further extended the herdtools suite of tools, models, and tests, for instruction fetching and cache maintenance. This work has yet to be published, although a prose version of the mathematics appears in the latest manuals [68]. It is difficult to give a comprehensive comparison between the most up-to-date model produced by Arm and the one presented here without further work to do an in-depth comparison. We do not believe the architectural intent has changed.

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 one can capture the architecturally intended envelope of programmer-visible behaviour while abstracting 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 multi-copy-atomic architecture, one can do so completely: the Flat model of Pulte, Flur, et al. [7] 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 — subsuming relaxations which arise from store queues and caches and suchlike.

For instruction fetch, and the required cache maintenance, it is no longer possible to abstract completely from the data and instruction cache hierarchy. However, we can still abstract from some of its complexity. Flat has a fixed instruction memory, and fetches instructions from that fixed instruction memory. This fetch 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: per-thread fetch queues and instruction caches, and a global data cache, as 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. We do so by first describing, informally, iFlat and its transitions, before giving a more detailed, but still in prose, precise description of the model. These descriptions are intended equivalent to a version we implement as an executable test oracle in the RMEM tool which can be found at https://github.com/rems-project/rmem.

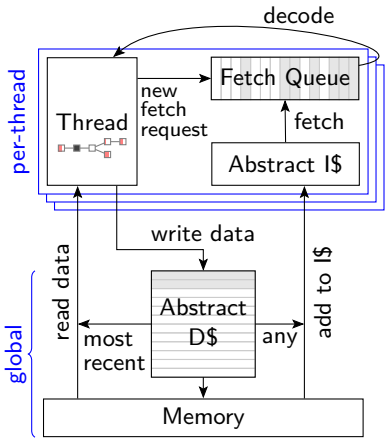


Figure 4.1: Structure of the iFlat state: per-thread fetch queues and instruction caches, with a global abstracted data cache.

4.2 iFlat Transitions: a brief summary

To Flat we add new transitions: splitting the atomic 'fetch next instruction' transition up into three distinct steps (a request, a satisfaction, and a decode); transitions which evolve the abstract cache state; and transitions for the operation of data and instruction cache maintenance.

In particular, these are[1]:

- ○ Fetch request
- ▷ Fetch instruction (ifetch)
- ▷ 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 the ISA definition can be executed to produce intra-instruction traces to continue execution as in original Flat [7].

4.2.1 An example: DC/IC cache synchronisation

Before we describe the iFlat state and transitions in detail, we first give an informal intuition of how it works, with a walkthrough of the previously seen DC/IC sequence.

Figure 4.2 sketches the sequence of updates to the iFlat state over the cartoon sketch of Figure 4.1. It starts from a simple state, with an old cached value in the instruction cache and an already-satisfied element in the fetch queue. It then demonstrates the cache synchronisation sequence of: writing a new instruction, performing data cache maintenance, then instruction cache maintenance, before finally fetching the new instruction, in six steps:

1. A write of a new value is propagated to memory. The new write is placed into the abstract data cache buffer, where it can be seen by data reads but not guaranteed to be seen by instruction fetches.
2. The `DC` 'pushes' the write down, out of the abstract data cache, to memory.
3. The `IC` removes any cached copy of that location from the abstract instruction cache, for all cores. Also, it removes any already-satisfied elements in the fetch queue using those cached copies.
4. A spontaneous fill of the instruction cache is now guaranteed to see the new value, and will cache it in the abstract instruction cache.
5. Program-order later instructions for that location, which have not yet been fetched, can now be satisfied from the new entry in the instruction cache.
6. Once the satisfied fetch request reaches the end of the queue, it is decoded and appended to the end of the instruction tree in the thread to be executed.

Note that we do not include the thread-local effects of barriers (`DMB`, `DSB`, or `ISB`) in this sketch, see the full description of the transitions in §4.4 for further details.

[1] Items marked with ○ may be taken eagerly.

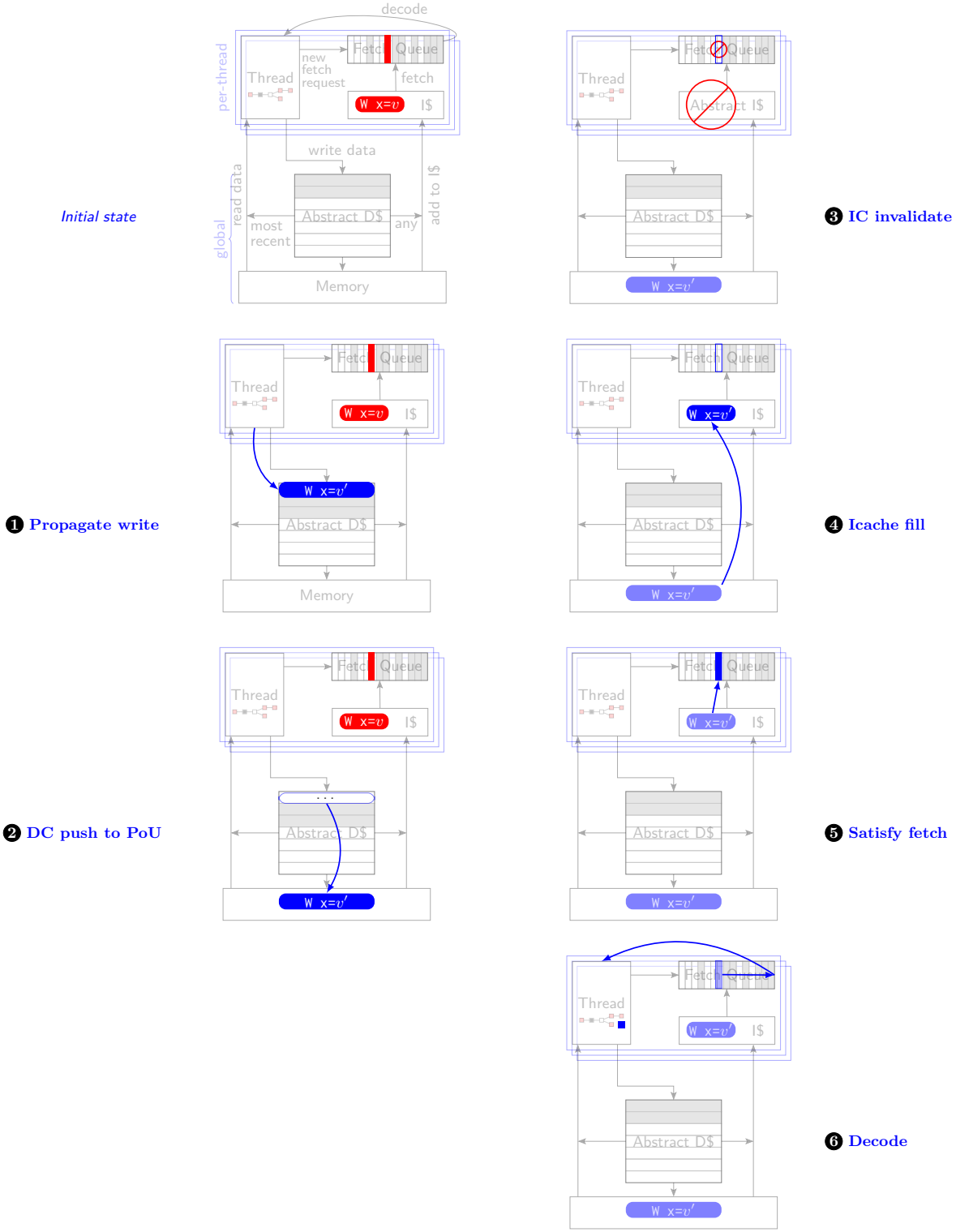


Figure 4.2: Cartoon sketch of DC/IC sequence in iFlat.

4.3 The iFlat state

We extend the original Flat state, to include:

- ▷ Per-thread fetch queues (§4.3.1), which store fetch requests permitting out-of-order satisfaction of instruction fetches.
- ▷ Per-thread instruction caches (§4.3.2), which enable threads to keep old (with respect to memory) copies of instructions which can be used when satisfying instructions in the fetch queue.
- ▷ A global abstracted data cache (§4.3.3), which over-approximates the coherent data cache network.

As is usual for an architectural definition, these are all of unbounded size – abstracting from, and thus overapproximating, the hardware.

The full description of the model state is given in Appendix B.

4.3.1 Fetch queues

We give each thread a dedicated 'fetch queue', which buffers the in-flight instruction fetches. Fetch queues allow the model to speculate and pre-fetch instructions, potentially satisfying them out-of-order.

The thread subsystem fetches instructions by inserting a new entry into the fetch queue. This entry is a *request*, containing the address to be fetched. The entries in the fetch queue can then be satisfied from memory at any point in time, in any order. Entries are removed and decoded in-order from the fetch queue.

Entries are either a yet unsatisfied ('*unfetched*') request, or, a fetched 32-bit opcode.

The model permits entries to be added to the fetch queue for any arbitrary address; as earlier instructions become finished, they will discard successor instructions whose program counter value does not match the one computed from the instruction semantics.

In this way the fetch queues abstract from multiple hardware structures: instruction queues, line-fill buffers, loop buffers, slots objects, and others.

Out-of-order fetching We believe the out-of-order satisfaction of instruction fetches is not observable on real hardware (in part due to the general lack of coherence in instruction caches subsuming this behaviour, see §3.5), and that the model is equivalent to one that fetches in order. However, 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.

Fetch queues and instruction trees Flat already keeps a per-thread tree of in-flight instructions. There is a model design choice between constructing an explicit fetch queue as an independent structure in the iFlat state, or adding a new *unfetched* state to the instruction instances in the tree and interpreting the po-suffix of any unfetched entries in the tree as the fetch queue. The latter has the advantage of allowing model speculation down multiple branches simultaneously, although this does not introduce additional behaviours.

In principle, the two choices are equivalent and are merely presentational. Here, we give the model as if there are explicit separate fetch queues tracked in the state for exposition purposes, but in the executable-as-a-test-oracle implementation in RMEM we simply extend the already existing instruction tree with an unfetched state.

4.3.2 Abstract instruction caches

We give each thread an abstract instruction cache. The abstract instruction cache implements an over-approximation of those that may be found in hardware as a set of writes which the fetch queue entries can be satisfied from. We apply very few restrictions to the abstract instruction cache: it is permitted to cache any set of writes seen since the last instruction cache invalidation. The abstract instruction cache is then used to satisfy fetch requests in the fetch queue.

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 ([66, B2.4.4]), or spontaneously drop entries.

Unlike the flat memory, the instruction caches are not touched on propagating a write. There is no guarantee values are ever dropped from the instruction cache, unless an explicit instruction cache maintenance operation is performed.

Invalidations Instruction caches must be maintained by software, through the use of instruction cache invalidation instructions (`IC`). An `IC` instruction sends messages to each core (including its own), requesting they clear their instruction caches, and then waits for all the cores to reply. Other instructions may execute out-of-order with respect to these messages, except for `DSB`s: the requests are only sent after any program-order earlier `DSB` instructions are complete, and no program-order later `DSB` can complete until all the replies have returned.

To handle this, we keep for each thread a set of addresses yet to be invalidated by in-flight `IC`s. Threads can, at any point in time, spontaneously perform any pending cache invalidation for that thread: removing any writes from the instruction cache or already-satisfied entries in the fetch queue, for that location. Once all pending cache invalidations have been performed on all cores, the `IC` instruction can complete.

4.3.3 Global abstract data cache

Above the single shared flat memory for the entire system, we insert a shared buffer of writes. Implementations may have various topologies of cache hierarchies; we abstract from this by keeping a single large global buffer of all writes which have not yet passed the Point of Unification – over-approximating any individual cache topology.

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 either the buffer or memory (§3.4).

As writes are propagated to memory, they are placed initially into the abstract data cache buffer. At any point in time, the coherence-earliest write in the buffer for any location can spontaneously flow into the shared flat memory, making coherence-earlier writes no longer visible to instruction fetches.

In this way, the shared flat memory acts as a 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.

4.3.4 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, one for the data cache clean, and two for instruction cache invalidation (for the separation of propagation of messages and completion of the whole invalidation). The full list of outcomes for the iFlat model can be found in Figure 4.3.

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 4.3: iFlat outcomes (new outcomes highlighted in blue).

4.3.5 Pseudocode states

We extend the intra-instruction semantics, and associated pseudocode states, to include the fetch-queue fetch states, either fetched or unfetched, and 'pending' `IC` instructions, as they do not happen atomically. Figure 4.4 lists all the pspeudocode states in iFlat, with the new ones highlighted.

PLAIN(`next_state`)	Ready to make a pseudocode step
UNFETCHED(`pc`)	Placed into fetch queue but pending satisfaction of the fetch
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`)	Waiting for completion of an `IC IVAU`

Figure 4.4: iFlat pseudocode states (new outcomes highlighted in blue).

4.4 iFlat Transitions: in full

We now give full descriptions of the new transitions, as an amendment to the Flat model of [7] recalled in Chapter 2. A full transcription of the (i)Flat model can be found in Appendix B.

4.4.1 New transitions

Transitions for all instructions:

- ○ Fetch request: This transition (perhaps speculatively) requests to fetch the next-instruction address, as a po-successor of a previous instruction.
- ▷ Fetch instruction: Satisfy the fetch request from the instruction cache (split into three variants).
- ○ Decode instruction: Decode the instruction.

Cache maintenance instructions:

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

Instruction cache updates:

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

Below gives precise prose descriptions of each of the new transitions' guard and action.

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).

Action: add an unfetched entry for `loc` to the fetch queue for `i`'s thread.

Note that this allows speculation past conditional branches and calculated jumps.

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

Fetch instruction (ifetch) *In ifetch mode this transition replaces the original 'Fetch instruction' transition.*

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 instruction cache in a constrained-unpredictable way, if:

1. there exists a set of write-slices, each of which can be constructed in the same way as above;
2. that set contains write-slices corresponding to distinct opcodes, and at least one of those is 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.
2. Discard all speculative entries in the instruction tree that are successors of `i` that are now known to be incorrect speculations.

Note that this transition is a proxy for the point the instructions will be decoded, but that it is the intra-instruction semantics that actually performs the decoding, with this transition merely starting the execution of the pseudocode.

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 instruction instance 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 be spontaneously updated with a write `w` from the storage subsystem, if this write (as a single slice) does not already exist in the instruction cache.

Action: Add this write (as a single slice) to thread `tid`'s instruction cache.

4.4.2 Updated transitions

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

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; and
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;; and
6. [ifetch] if `i` is a `dsb sy` instruction, all po-previous memory access instructions, barriers, and cache maintenance instructions have finished.

Action:

1. Update the state of `i` to PLAIN(`next_state′`);
2. [ifetch] If `i` is an `isb` instruction, for any instruction instance in this thread's instruction tree, if that instruction instance is in the FETCHED state, set it to the UNFETCHED state.

Note that this corresponds to an `ISB` discarding any already-fetched entries from the fetch queue.

Satisfy memory read by forwarding from writes For a load instruction instance `i` in state PENDING_MEM_READS(`k`), 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; and
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.
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(k), and a read request r in i.mem_reads, that has unsatisfied slices, the read request can be satisfied from memory, if:

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

Action:

let wss be the write slices from memory or the abstract data cache, 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(k), 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(k), 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.
3. Add w as a complete slice to the abstract data cache.

4.4.3 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 data and instruction cache line of minimum size is the smallest possible cache line, for the data or instruction caches respectively. 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 say a write slice *overlaps* 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.

4.4.4 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.

Chapter 5

An axiomatic instruction fetch model

Based on the operational model, we develop an axiomatic semantics, as an extension of the Arm-A axiomatic model [53, 7] described in Chapter 2. Throughout this chapter, references to the base Arm-A axiomatic model refer to the one presented in that chapter.

The base 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, extending both the base events and candidate relations, as well as modifying the axioms over those events. We 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, corresponding to the read of the 32-bit opcode from memory during an instruction fetch.
- ▷ `DC` events, corresponding to the propagation of cache maintenance from a `DC CVAU` instruction.
- ▷ `IC` events, corresponding to the propagation of cache maintenance from an `IC IVAU` or `IC IALLU` instruction.
- ▷ `DSB` events for the data synchronization barrier instruction.

5.1.1 Explicit events and program order

We partition the events into *implicit* events (in this case, just instruction fetches) and *explicit* events. Logically, the explicit events are those that form part of the intensional operation of the instruction: the primary memory events and general-purpose register accesses; whereas the implicit events are those accesses which are indirect, part of the execution model of the machine or configuration but not particular to the behaviour of that instruction. The architecture does not precisely define which events of an execution belong to which category, but the informal notion of an 'implicit' event will still be a useful one.

Program order (`po`) relates explicit events. These are the explicit memory accesses (non-instruction-fetch), barriers (including `DSB`s), and the cache operations (`DC`, `IC`).

By adding an instruction fetch event we now potentially have multiple events per instruction. 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 access, barrier, or cache-op events of the instruction.
- ▷ *fetch-program-order* (`fpo`), which relates each instruction-fetch (`IF`) event with all `IF` events of program-order later instructions.

We make fpo the fundamental relation in candidates, instead of po, which we instead derive:

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

Figure 5.1 shows an example execution graph from a program with three instructions a load, a move, and, a store; with the implicit fetch events highlighted, illustrating the derivation of po.

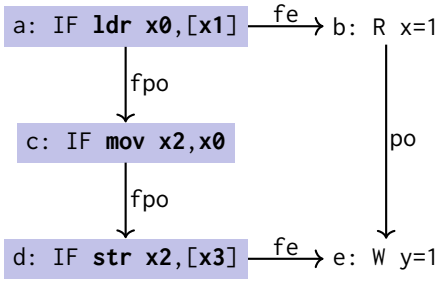


Figure 5.1: An example showing how po is derived from fpo and fe from the implicit fetch events (in blue).

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 an entire cache line. To handle these operations, we add to the candidate relations a pair of *same-cache-line* relations, relating reads, writes, fetches, DC, and IC events to addresses in the same cache line of minimum size.

Since the DC and IC instructions operate over different cache line sizes, we have separate same-dcache-line and same-icache-line relations, 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 single 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 is a generalised coherence-order, an extension of co, with orderings for cache maintenance (DC and IC) events: it includes orderings (e, e') or (e', e) for any cache maintenance event e and same-cache-line write-or-cache-maintenance event e'.

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).

This relation forms part of the witness, and abstractly captures the order that cache maintenance operations and propagation of writes would happen in the operational model.

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 that [a]; ctrl; [b]; po; [c] implies [a]; ctrl; [c].

We extend addr to include cache operations, so that $(e, e') \in$ addr when 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 an *instruction-read-from* (`irf`) relation to the witness. It is the analogue of `rf` for instruction fetches, relating writes to the `IF` event that fetches from it. We derive the analogous from-reads relation, *instruction-from-reads* (`ifr`), from a fetch to all writes coherence-after the one it fetched from:

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

Note the use of `co` not `wco`.

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 ifetch events and relations

The `arm-common.cat` file contains all the Arm-specific event names and relations, as defined in Chapter 2, and can be found in the full isla sources for these models in [91]. Figure 5.3 lists the events and relations defined by that file; 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	Explicit memory event (R|W)	po-loc	program-order same-location (po & loc)
A	Read-acquire	addr,ctrl,data	dependencies
L	Write-release	wco,irf,rf	Witness 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 ($\texttt{irf}^{-1}$;co)
IC	Instruction cache invalidate	rmw	read-modify-write
ISB	Instruction barrier		
dmbXY	Memory Barrier		
dsbXY	DSB Barrier		
		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 ifetch events and relations. New and updated are highlighted in blue.

```
include "cos.cat"
include "arm-common.cat"  (*5.2.1*)

(* might-be speculatively executed *)
let speculative =
    ctrl
  | addr; po

(* Fetch-ordered-before *)
let fob =
    [IF]; fpo; [IF] (*5.2.4*)
  | [IF]; fe (*5.2.4*)
  | [ISB]; fe^-1; fpo (*5.2.5*)

(* Cache-op-ordered-before *)
let cob = (*5.2.8*)
    [R|W]; (po & scl); [DC]
  | [DC]; (po & scl); [DC]

(* DC synchronised required after a write *)
let dcsync =
  if IDC
    then id
    else [W]; (wco & same-dcache-line); [DC]

(* IC sync required after a write or DC *)
let icsync =
  if DIC
    then id
    else (
      [W]; (wco & same-icache-line); [IC]
    | [DC]; wco; [IC]
    )

let cachesync =
  dcsync; icsync

(* instruction-synchronised-ordered-before *)
let isyncob = (*5.2.2*)
    (ifr; cachesync) & scl^-1
```

```
(* observed by *)
let obs = rfe | fr | wco | irf

(* dependency-ordered-before *)
let dob =
    addr | data
  | speculative; [W]
  | speculative; [ISB]
  | (addr | data); rfi

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

(* barrier-ordered-before *)
let bob =
    [R]; po; [dmbld]
  | [W]; po; [dmbst]
  | [dmbst]; po; [W]
  | [dmbld]; po; [R|W]
  | [L]; po; [A]
  | [A|Q]; po; [R|W]
  | [R|W]; po; [L]
  | [F|C]; po; [dsbsy] (*5.2.6*)
  | [dsb]; po (*5.2.6*)
  | [dmbsy]; po; [DC] (*5.2.7*)

(* Ordered-before *)
let ob1 =
    obs | dob | aob | bob
  | fob | cob | isyncob
let ob = ob1^+

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

(* External visibility *)
irreflexive ob as external

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

Figure 5.2: Ifetch Axiomatic model

5.2.2 Cache maintenance

We define a derived relation `isyncob` (*instruction-synchronised-ordered-before*), 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 fetched. Consequently, any instruction fetch must have been satisfied before the completion of any cache maintenance that it is `isyncob`-ordered before. Precisely, f `isyncob` i is defined as:

$$f \texttt{ isyncob } i \iff (f, i) \in (\texttt{ifr};\ \texttt{cachesync}) \cap \texttt{scl}$$

That is, f reads-from some write w_0, which was coherence-before some other write w, and w is `wco`-before by a `DC` event d to some `same-dcache-line` address A_{dc}, which is in turn was `wco`-before 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 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.

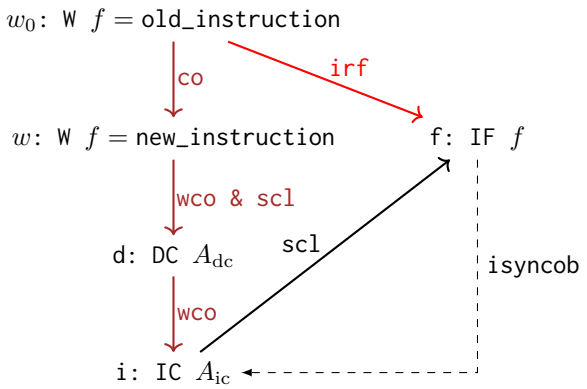


Figure 5.4: General `isyncob` shape.

This corresponds to the operational model in the following way: since w_0 was coherence-before w, w_0 was propagated before w 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).

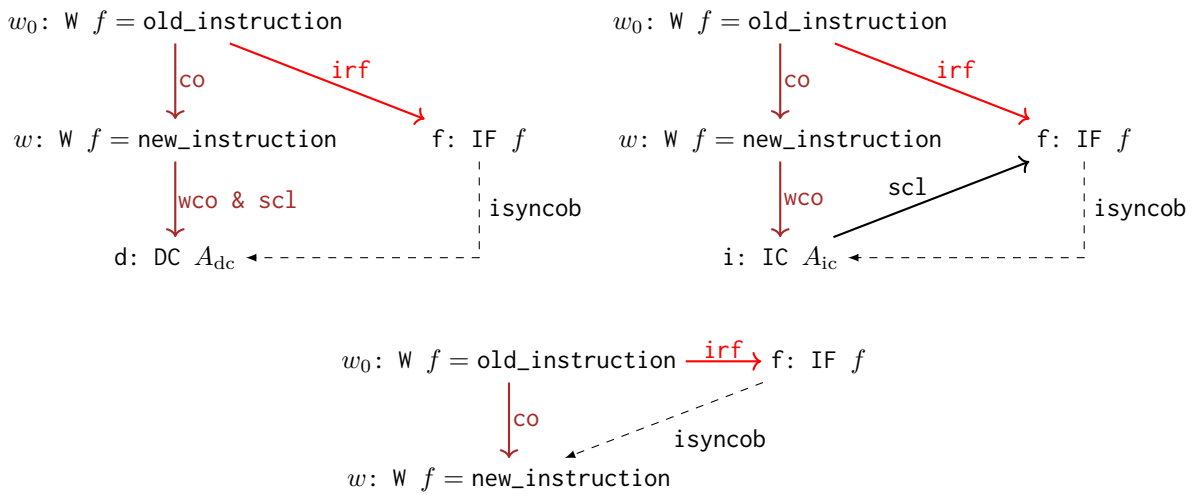


Figure 5.5: Modified isyncob shape, for variants DIC (above left), IDC (above right), and both (below).

To achieve this, the isyncob relation is derived from the composition of two smaller relations:

- ▷ 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. We then finally define isyncob between any instruction fetch, and any cache maintenance operation which is cachesync-after any write coherence-after the one the fetch read from, that is, a write which has had sufficient cache synchronisation to have made earlier writes invisible to the fetch machinery.

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 derived relation *fetch-ordered-before* (fob), which is included in *ordered-before*.

The fob relation 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 be satisfied, a write can only propagate, a barrier can only commit, and so on, after its instruction's fetch is satisfied.

5.2.5 Instruction synchronisation (ISB)

We include the edge [ISB];$\mathtt{fe}^{-1}$;fpo in fetch-ordered-before (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 [ISB];po;[R] in dob is no longer required, as the combination of rules in fob (in particular [ISB];$\mathtt{fe}^{-1}$;fpo and [IF];fe) subsume it.

5.2.6 Data synchronisation (DSB)

For DSB instructions we include po to and from DSB in the standard barrier-ordered-before relation (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 [F|C];po;[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 [dsb];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 also includes the relation [dmbsy];po;[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 [R|W];(po&scl);[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 [DC];(po&scl);[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.

```
(* include base ifetch model *)
include "aarch64_ifetch.cat"

(* could-fetch-from *)
let cff =
  ([W]; loc; [IF])
    \ ob^-1
    \ (isyncob^-1; ob)

(* cmodx(opcode) is True
 * if it is in the list of
    concurrently modifiable
    instructions
 *)
define cmodx(v: bits(32)): bool =
    ...
```

```
define cff_bad(
    ev1: Event,
    ev2: Event,
    ev3: Event
  ): bool =
    W(ev1) & IF(ev2) & W(ev3)
    & ~(ev1 == ev3)
    & cff(ev1, ev2) & cff(ev3, ev2
    )
    & (~cmodx(ev1.value)
      |~cmodx(ev3.value))

(* assert CU *)
assert exists
    ev1: Event,
    ev2: Event,
    ev3: Event
  =>
    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 reflect the architecture, to the best of our understanding, by engaging in detailed and robust discussions with the Arm chief architect, as well as 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 necessarily terminating, but it usually results in reaching a natural fixed point, for a core set of architectural features.

The structure of the operational model presented in Chapter 4 is based on our discussions with Arm; it carefully includes structures which capture the behaviour they described, and has limits where the architects decided no reasonable hardware could exploit.

The axiomatic model, presented in Chapter 5, is also a product of the discussions with Arm.

6.2 Correspondence between the models

We experimentally test the correspondence between the operational and axiomatic models. We do this by making the models executable as test oracles, and running a suite of hand-written and autogenerated litmus tests, checking that both models give the same result in all cases.

To automatically generate families of interesting instruction-fetch tests, Luc Maranget (a co-author of this work) extended the 'diy' test generation tool [70] 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 Chapter 4 executable, that is, 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 the address of a potential successor instruction, 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, which incrementally builds the set of possible branch targets by repeated exhaustive searches [7], to track not only the indirect branch instructions but the successors of *every* program location. 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.

Confluence

We also take some transitions eagerly when they cannot remove behaviour to reduce the search space: 'Propagate IC to thread', 'Complete IC', 'Fetch request', and 'Add to instruction cache for thread'.

Taking 'Add to instruction cache for thread' eagerly 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 in a way that makes 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 `DC`s 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 instruction cache invalidations together as early as possible, maximising the visible behaviour. Therefore, it we perform all the invalidates at once, atomically.

6.2.2 Making the axiomatic model executable as a test oracle

We give the axiomatic model in the `isla-cat` memory modelling language (see §2.4.2).

As `isla-axiomatic` already executes a fetch-decode-execute loop, defined by the Arm intra-instruction semantics, the changes required of the ISA definition are only minor; we need only create outcomes for the fetch memory accesses, and pass them as events to the axiomatic model.

This is sufficient for making the test executable, but exhaustive enumeration becomes intractable, as the fetch events in the candidates should, in theory, be totally unconstrained. To support exhaustive enumeration we must reduce the set of candidates we are required to check. Even permitting the *fetch* part of the loop to be entirely 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 time to generate and discharge. To avoid this, we instead require the user to provide the possible set of program-counter values, 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 3.19, p.64) as an example. Lines 7-13 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.

```
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.

6.3 Equivalence of the models

Ideally, one would have a formal proof that the operational and axiomatic models coincide, or at least a detailed proof of some properties we expect the operational model to have: that the model is equivalent to one that fetches in-order, that the transitions we take eagerly are safe to do so, that the fixed-point calculation is not unsound for the model, and so on. However, this represents a large undertaking, as any detailed proof above the actual definitions of the microarchitectural-flavoured operational semantics have historically been very resource intensive, up to being the subject of entire Ph.D. theses [6]. Therefore, we — sadly — defer such formal proof to future work.

In lieu of such formal proof, we compare the models empirically. 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 also ran 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 tested the equivalence of the operational and axiomatic models on the 52 hand-written and the 1456 diy-generated tests, checking that the models give the same sets of allowed final states.

6.4 Validating against hardware

To run instruction-fetch tests on hardware, we extended the litmus tool [69]. The most significant extension consists in handling code that can be modified, and thus has to be restored between experiments. To that end, we make litmus execute copies of the code, which 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.

6.4.1 Results from hardware

We ran both the hand-written instruction-fetch litmus tests, and the 1456 auto-generated ones, on a variety of hardware implementations. A short table of results can be found in Figure 6.2.

Our testing revealed a hardware bug in the Qualcomm Kryo cores of the Snapdragon 820: an illegal outcome was exhibited by MP.RF+cachesync+ctrl-isb (Figure 3.11, p.60) in 84/1.1G runs (not shown in the table), which we have reported. We have also observed an anomaly for MP.FF+cachesync+fpo (Figure 3.13, p.61) on an Arm-designed core, although this core had (in previous work) been discovered to suffer a read/read coherence violation. Apart from these, the hardware observations are all allowed by our models.

As is expected, hardware does not make full use of the architectural envelope: some tests are architecturally allowed, but never observed on hardware. There are broadly two reasons why this may be: the architecture must be robust to future designs one might imagine but are not typically implemented (such as the direct-data-intervention explanation of SM.F+ic (Figure 3.19, p.64)); in other cases, the architects must make decisions to avoid ambiguity in the model, and where unconstrained by hardware designs or software requirements we choose the simpler programming model (as in FOW (Figure 3.20, p.65)).

We therefore draw high confidence that the presented models correctly capture the architectural intent, and are consistent with existing hardware. There were no existing hardware with either cache type strengthening at the time of the work, and so, while we believe the models consistent with the architectural intent, we are unable to give the same level of confidence in those aspects of the model. However, overall we believe the models are strong enough to forbid the key behaviours guaranteed by hardware, and relied on by software, while still being loose enough to be consistent with expected potential future designs.

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

Figure 6.2: Instruction-fetch hardware results
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.

Part II

Virtual memory

This part is based on: Relaxed virtual memory in Armv8-A [37] 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). The summary of the Arm virtual memory architecture is based on Chapter D5 of the Arm Architecture Reference Manual DDI 0487H.a [12].

Chapter 7

Pagetables and the VMSA

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.

This chapter continues with a brief overview of Arm's virtual memory systems architecture, in enough detail to understand the subsequent chapters, but does not present new contributions or novel research. The rest of this part then describes the relaxed behaviours of virtual memory (Chapter 8), gives an axiomatic semantics that capture these behaviours (Chapter 9), and validates that model against hardware and the architectural intent (Chapter 10).

7.2 Virtual Memory

Arm's *virtual memory system architecture* (*VMSA*) defines the virtual memory and virtualisation features of the Arm architecture. It is described, in detail, in the Arm Architecture Reference Manual [81].

Conventionally, memory is imagined as a flat array of bytes, indexed by *physical addresses*. Larger 'application' class processors rely heavily on virtual memory: interposing one or more layers of indirection between the accesses of a program (using *virtual* addresses) 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 or different permissions e.g. to obfuscate the true allocation of resources or to split permissions of a resource for compartmentalisation.
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.

Typically, operating systems split individual programs into distinct *processes*, where each process is associated with its own virtual to physical mapping. Such a mapping corresponds to a partial function, from that process's own (virtual) addresses to the real hardware physical addresses, with some permissions:

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

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

Typically operating systems create one such mapping for each process, thereby partitioning the physical memory into distinct subsets of physical addresses (which become the *range* of the translate function), and would allocate some convenient numeric values to be the virtual addresses the process interacts with (which become 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, perhaps not present 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.

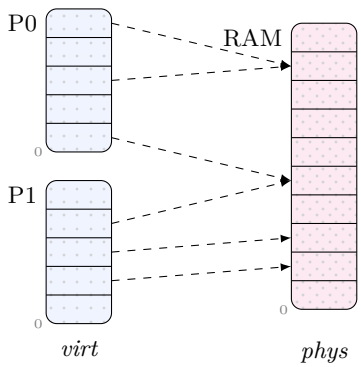


Figure 7.1: Example virtual and physical address spaces for two processes.

The mapping defines an *address space*: the range of virtual addresses a program has access to, and what they correspond to. The diagram in Figure 7.1 illustrates an example for two processes. The diagram represents the mappings:

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

For example, with a page size of 4k, if process `P0` reads or writes the address `0x2305` it will actually access physical location `0x8305`, since virtual page `2` was mapped to physical page `8` in `P0`'s address space, and the offset within a page is preserved.

Each address space corresponds to a distinct translation function. 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 or the range or both.

Large application-class processor architectures provide hardware support in the form of the *memory management unit* (MMU), which, once configured by software, will perform the translation from virtual to physical addresses and any checking of permissions automatically. Software then needs only manage a set of translation functions, in whichever encoding the architecture prescribes (see §7.3 for the encoding used by Arm), switch between translation functions on a context switch, and handle any processor exceptions generated by the MMU.

7.3 Arm Translation Tables

On Arm, as with most architectures, software can configure the MMU through the creation and modification of sets of *translation tables* (also referred to as *page tables*).

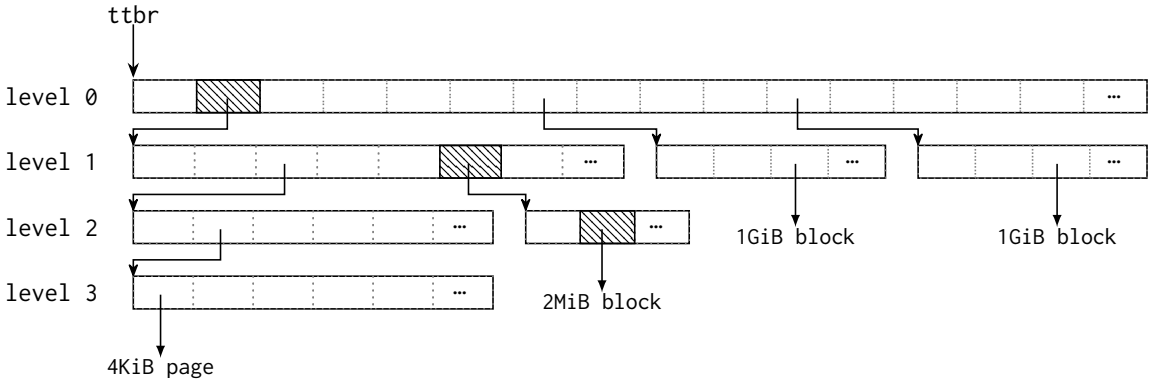


Figure 7.2: Schematic view of an example tree of translation tables. There are seven individual translation tables, over four levels, which defines an address space that maps four separate spans of virtual addresses to spans of physical addresses. In this example, the 2 megabyte block at level 2 encodes the mapping – the output address, permissions, and memory type – for addresses in the range `0x8140200000` up to `0x81403fffff` inclusive, which is determined from the highlighted path in the tree: it is the second level 2 (2M span) entry, for the 6th level 1 (1G span) entry, for the second level 0 (512G span) entry, from the root.

The translation tables form an in-memory tree data structure which encode a translation function. Software creates and maintains these trees, and controls which tree the MMU uses at runtime. On each memory access, the hardware reads from this tree structure to perform the translation, or from one of the various caching structures (described in §7.7).

A pointer to the root of the tree is stored in a `TTBR` ("Translation table base register"), which is one of a family of related registers (see §7.6) that determines which tree of translation tables is currently in use by that processor.

Each node in the tree is a page-aligned chunk of memory whose interpretation is an array of 64-bit entries, where each entry controls the mapping for a particular span of the domain, defining whether the virtual addresses in that span are defined for that process, and, if so, what the output physical address is and what permissions the process has for that memory. The root table controls the entire address space. The tree may recursively split spans into sub-trees. The width of the span mapped by each entry depends on its 'level', which increases with depth. Typically, the root is at level 0, and the tree has maximum depth of 4 (up to level 3) with a page size of 4 KiB. Thus, each pagetable contains 512 entries, with entries in the root table each corresponding to a 512 GiB span. Note that Arm is highly configurable and this merely represents one common configuration.

Figure 7.2 shows a view of an example set of translation tables, with four mapped regions defined in a tree of seven tables. Each rectangular array represents one contiguous page-aligned block of memory, made up of 512 64-bit entries. 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. The exact format of these entries is described in the next section (see §7.3.1).

7.3.1 Translation table format

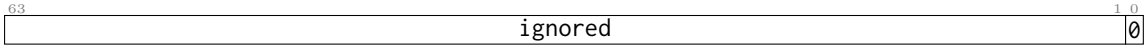
Arm's virtual memory system architecture is highly configurable. Writing to the `SCTLR` ("System control register") and `TCR` ("Translation control register") system registers allow the programmer to configure the processor with a variety of options. To give just a flavour of this configurability, some of those options 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 address space identifiers (ASIDs and VMIDs, used for indexing the caches, see §7.7); 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 the examples we suppress this complexity, and concretise down to just one common instance: the one currently used by the Linux kernel; this gives a tree of translation tables with maximum depth 4, with 4KiB pages, and 48-bit addresses. In this configuration, each node is a table of 512 64-bit entries, in one 4096-byte block of memory.

Each page table entry is 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.
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 the least-significant bit of the entry being `0`. The top 63 bits of an invalid entry are ignored by hardware, and software is free to use those bits to store metadata. 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) of the base address of the slice, with some metadata describing 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, described more in §7.3.2.

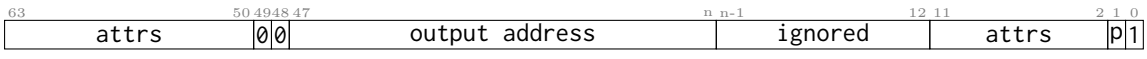
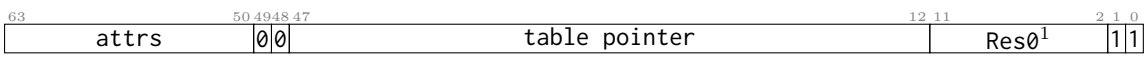


Table entries A table entry contains a page-aligned pointer to a child table, but can also contain similar metadata as the block or page entry, including access permissions (read/write/execute), which are combined with any permissions from the child table.

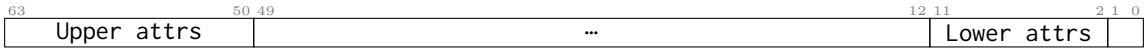
Table entries are allowed only at levels 0–2.



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

7.3.2 Attributes

The encoding of the attributes are split into upper and lower attribute fields:



These fields can be further split (see the Arm ARM D8.3.2 for a more comprehensive breakdown) [81]:

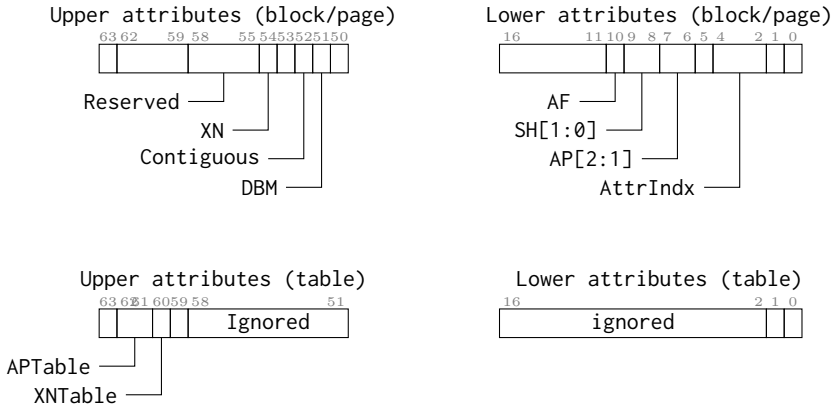


Figure 7.3: Upper and lower attribute encodings for Stage 1 pagetable entries for the 4KiB granule.

Some fields are elided, either because they are for out-of-scope features or otherwise uninteresting, leaving just the following fields of interest:

- ▷ `XN/XNTable`: Execute-Never; when set, this mapping (or child mappings if `XNTable`) does not have execute permissions.
- ▷ `Contiguous`: allows software to inform hardware that a sequence of entries point to contiguous blocks of output memory, to enable more efficient TLB packing.
- ▷ `DBM/AF`: Dirty bit modifier and access flag; these bits allow software to monitor accesses to locations, which are are out-of-scope for this work.
- ▷ `SH`: Shareability; how 'far' into the system the memory must be kept coherent for, e.g. memory marked non-shareable need not be coherent for multiple cores. We do not model shareability domains here, so always assume 'Inner Shareable'.
- ▷ `AP/APTable`: Access permissions; described below in 'Access permissions'.
- ▷ `AttrIndx`: Memory attribute; described below in 'Memory Attributes'.

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 which are 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 an `AP[2:1]` [1] field (`bits[7:6]`). These fields can be decoded using the following table:

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

For executable permissions, which permit or forbid instruction fetching from some region of memory, there are no dedicated encodings of the access permission bits. Instead, all mappings are executable by

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

APTable[1:0]	AP[2]	AP[1]	EL1			EL0		
			R	W	X	R	W	X
00	0	0	✓	✓	✓	×	×	✓
	0	1	✓	✓	×	✓	✓	✓
	1	0	✓	×	×	×	×	✓
	1	1	✓	×	✓	✓	×	×
01	0	0	✓	✓	✓	×	×	✓
	0	1	✓	✓	×	×†	×†	✓
	1	0	✓	×	×	×	×	✓
	1	1	✓	×	✓	×†	×	×
10	0	0	✓	×†	✓	×	×	✓
	0	1	✓	×†	×	✓	×†	✓
	1	0	✓	×	×	×	×	✓
	1	1	✓	×	✓	✓	×	×
11	0	0	✓	×†	✓	×	×	✓
	0	1	✓	×†	×	×†	×†	✓
	1	0	✓	×	×	×	×	✓
	1	1	✓	×	✓	×†	×	×

Figure 7.4: Merging Access Permissions (Stage 1, EL1&0).
Entries with a † highlight differences from the `APTable=00`.

default, unless one of the following applies: the region is mapped writeable at EL0, as writeable 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, we assume the execute-never bits are always disabled.

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 7.4 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.

Memory Attributes

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

To tell the hardware, and to prevent it from performing unsafe optimisations such as speculatively attempting to read from a device, software must 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 used as to index the fields in the `MAIR` register.

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`.

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).

7.4 The Arm translation table walk

When the processor executes an instruction which takes an address, such as a load or store, the (virtual) address is converted to a physical address by the MMU. Without a previously cached result (see §7.7), the MMU must perform a hardware translation table walk.

To do this walk, the MMU reads the relevant `TTBR` to get the currently in-use tree of translation tables, and performs a walk of the tree. The hardware walker first slices up the input virtual address into chunks: the most-significant bit (the sign) is used to determine which base register to use (see §7.6); the next 15 bits are required to be zero; and the rest of the address is split into 9-bit fields which here we call 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 always preserved.

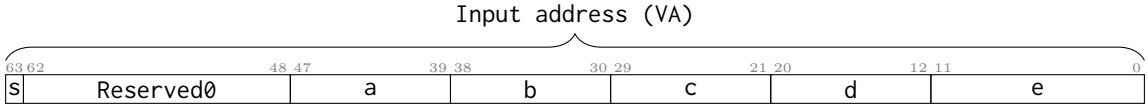
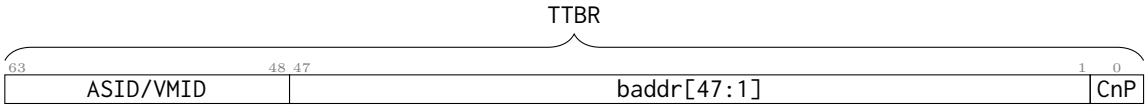


Figure 7.5 gives a simplified algorithm for the hardware walk the MMU does on Arm-A, fixed to the configuration we consider here, eliding the permissions check and hierarchical attribute calculations.

Reading the TTBR The base address register contains three fields: the higher bits store the ASID (see §7.7), 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, which enables further performance optimisations.



```
1:  procedure TranslateAddress(VA, isRWX)        ▷ Input address, and access kind (read/write/execute)
2:      t ← read_TTBR().base_address             ▷ See §7.6, and Reading the TTBR below
3:      attrs ← 0
4:      for i = 0, ..., 3 do                     ▷ Iterate down the levels of the tree
5:          s ← Bitslice(VA, 47 − 9i, 47 − 9i − 9 + 1)   ▷ Slice out fields a—d depending on index
6:          entry ← Mem[t + 8s]                  ▷ Access entry in table
7:          if entry[0] = 0 then                 ▷ Invalid entry
8:              return TranslationFault(VA, Invalid)   ▷ See Faults below
9:          else if entry[1] = 1 ∧ i < 3 then    ▷ Table entry
10:             t ← entry.table_pointer
11:             attrs ← attrs | entry.attrs
12:         else if entry[1] = 0 ∧ (i = 0 ∨ i = 3) then
13:             return TranslationFault(VA, Reserved encoding)
14:         else                                 ▷ Block/page entry
15:             attrs ← attrs | entry.attrs
16:             offset ← Bitslice(VA, 47 − 9i − 9, 0)
17:             OA ← entry.output_address :: offset   ▷ See Computing the final output address below
18:             if !CheckPermissions(attrs, isRWX) then   ▷ See §7.3.2 'Access permissions' above
19:                 return TranslationFault(VA, Permission error)
20:             else
21:                 return OA
22:             end if
23:         end if
24:     end for
25: end procedure
```

Figure 7.5: Simplified single-stage translation table walk for a 4K pagetable.

Computing the final output address The output address (OA) of the final descriptor is the start of the range mapped by the entry. The offset into the range must be added to the start, in order to compute the final output address of the translation.

To compute this address, the MMU takes the OA field 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.

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 generated when the mapping in the translation table is 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 (i.e. the bits reserved-as-zero in the address were set).
- ▷ Permission fault.
  Generated 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.

- ▷ Alignment fault.
  Generated when an operation requires an aligned memory address, but is given a misaligned one.
- ▷ Address size fault.
  Generated 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` ("exception syndrome register") register, in its `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 so on). Exception handling code can read the `ESR` register to determine the fault type and cause, and can read the `FAR` ("fault address register") to determine the virtual address which triggered the fault, and handle the fault appropriately.

7.5 Virtualisation

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 achieve, hardware adds another layer of virtual memory, in addition to the existing one, creating two *stages* of translation. Processes use virtual addresses, which are converted to *intermediate physical* addresses (IPAs, also sometimes known as *guest-physical* addresses) using the operating system's configured translation tables. These then go through another *stage* of translation, typically controlled by the hypervisor, converting those IPAs into physical addresses.

Software manages both sets of translation tables: operating systems manage *Stage 1* tables to convert VAs to IPAs; and hypervisors manage *Stage 2* tables to convert those IPAs to PAs. This gives two separate translation functions, which the hardware composes together at runtime:

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

Hypervisors (running at EL2) configure the second-stage translation in much the same way as operating systems configure the first stage: by creating a tree of translation tables, with an almost identical format as before, and storing a pointer to the root of this tree in the `VTTBR` ("Virtualization translation table base register"). The hardware reads the `VTTBR` 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 7.6 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 physical addresses.

Concretely, if P1 reads from address `0x1001`, it will be translated into the IPA `0x3001`, in OS0's address space. This IPA is then is then translated again into the physical address `0x6001`, by a second stage of translation controlled by the hypervisor, and the processor will actually read from the RAM at location `0x6001`.

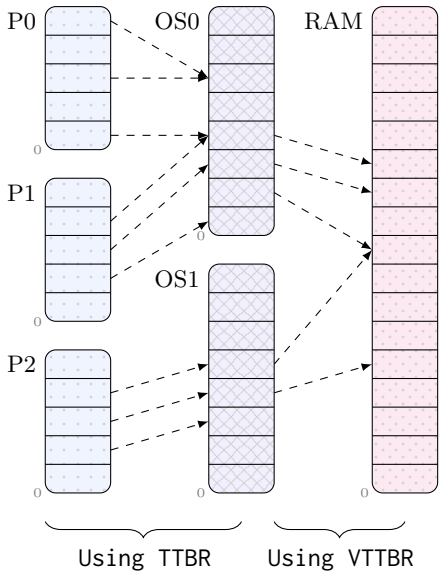


Figure 7.6: Example virtual, intermediate physical, and physical address spaces for three processes running on two operating systems.

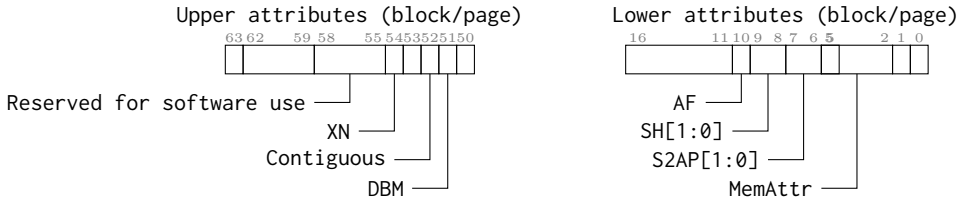


Figure 7.7: Attribute encodings for Stage 2 pagetable entries for the 4KiB granule [81, D8.3].

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

Figure 7.8: S2AP field encoding.

Stage 2 attributes encoding Stage 2 translation table entries are encoded similar to their stage 1 counterparts, as illustrated in Figure 7.7, with some minor differences:

- ▷ Stage 2 table entries do not have any additional attributes, and so do not have an `APTable` field.
- ▷ The Stage 2 `AP` field (called `S2AP`) has a slightly different (and simpler) format, see Figure 7.8.
- ▷ 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.

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. Below we will see an example of the accesses performed during a typical store instruction, as illustrated in Figure 7.9.

Figure 7.10 then gives a simplified algorithm for a two-stage translation-table-walk, with some detail elided: the permissions combining and checking, determining current regime, routing of exceptions, and so on. Arm give a full and precise definition of the translation table walk as part of the ASL defining the intra-instruction semantics.

An example Consider the Arm `STR Xn,[Xt]` instruction. It writes data stored in register `Xn` to an address stored in register `Xt`. Figure 7.9 is an example trace of one execution of the aforementioned store instruction. It is just as the Arm intra-instruction semantics would generate when executed at EL0 in the two-stage EL1&0 regime, in the worst case setting where the address is mapped by last level entries, in both the stage 1 and stage 2 pagetables. Each node represents an event in the trace (a memory or register access), and the arrows between them represent control flow within the intra-instruction semantics. The events in the dotted region come from the translation table walk (calls to the Arm `AArch64.TranslateAddress` pseudocode function).

Translation starts by reading the base address for the stage 1 walk, from the relevant `TTBR`, and performing

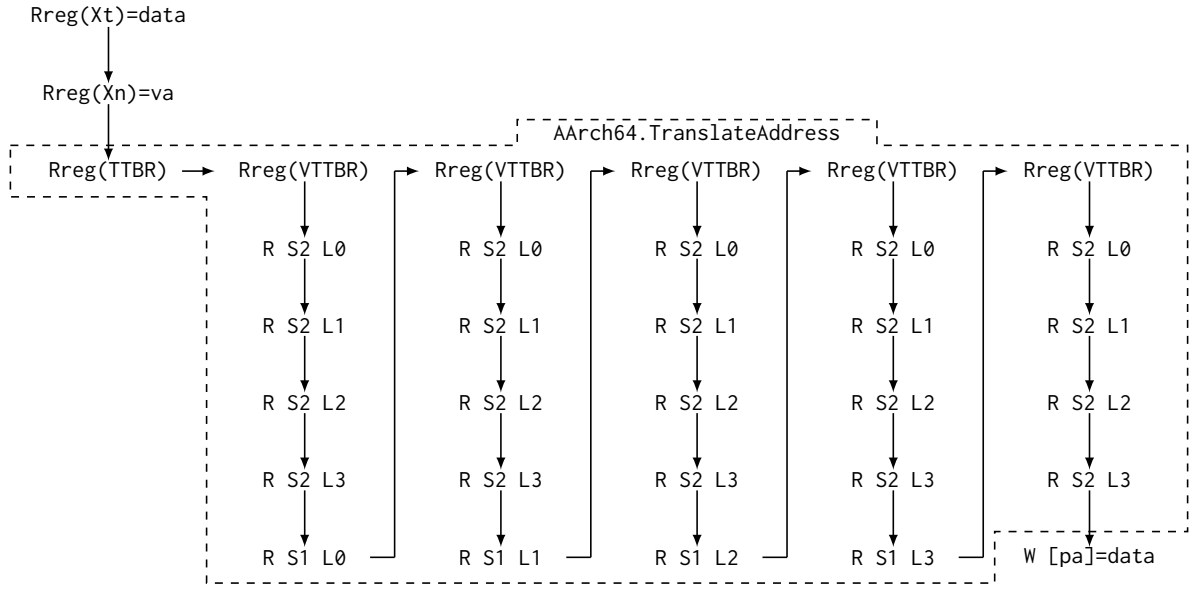


Figure 7.9: Memory and register accesses during a '`STR Xt,[Xn]`' instruction.

a second-stage translation (the events marked as `R S2 L`i) to get the physical address of the stage 0 level 0 table. It proceeds to read from that table (the event `R S1 L0`), repeating the process again, once for each level in the stage 1 table. Once the final result from the stage 1 walk is obtained (from the event `R S1 L3`), the final stage 2 walk is done to calculate the final physical address to be accessed. When the full walk is complete, and the pseudocode returns from the walk, it performs the actual memory access (the `W [pa]=data` event in the diagram).

```
1: procedure WALK(Stage, IA, isRWX)                ▷ IA is now input address, which may be VA or IPA.
2:     if Stage = 1 then
3:         t ← READ_TTBR().base_address                                          ▷ See §7.6
4:     else
5:         t ← VTTBR_EL2.base_address
6:     end if
7:     attrs ← 0
8:     for i = 0, ..., 3 do
9:         s ← BITSLICE(IA, 47 − 9i, 47 − 9i − 9 + 1)        ▷ Slice out fields a—d depending on level
10:        addr ← t + 8s                                          ▷ Address of entry in the table
11:        if ISINTWOSTAGEREGIME() ∧ Stage = 1 then
12:            addr ← WALK(Stage 2, addr, R)          ▷ Do a stage 2 walk to get physical address
13:            if addr is TranslationFault then                          ▷ ... which may fail
14:                return TRANSLATIONFAULT(IA, Stage 2 during Stage 1)
15:            end if
16:        end if
17:        entry ← Mem[addr]
18:        if entry[0] = 0 then                                                ▷ Invalid entry
19:            return TRANSLATIONFAULT(IA, Stage, Invalid)
20:        else if entry[1] = 1 ∧ i < 3 then                                   ▷ Table entry
21:            t ← entry.table_pointer
22:            attrs ← attrs | entry.attrs
23:        else if entry[1] = 0 ∧ (i = 0 ∨ i = 3) then
24:            return TRANSLATIONFAULT(IA, Stage, Reserved encoding)
25:        else                                                              ▷ Block/page entry
26:            attrs ← attrs | entry.attrs
27:            offset ← BITSLICE(IA, 47 − 9i − 9, 0)
28:            OA ← entry.output_address :: offset
29:            if !CHECKPERMISSIONS(Stage, attrs, isRWX) then   ▷ See Stage 2 attributes encoding above
30:                return TRANSLATIONFAULT(IA, Stage, Permission error)
31:            else
32:                return PA
33:            end if
34:        end if
35:    end for
36: end procedure
37:
38: procedure TRANSLATEADDRESS(VA, isRWX)
39:    if ISINSINGLESTAGEREGIME( ) then
40:        PA_or_Fault ← WALK(Stage 1, VA, isRWX)
41:        return PA_or_Fault
42:    else
43:        IPA_or_Fault ← WALK(Stage 1, VA, isRWX)
44:        if IPA_or_Fault is TranslationFault then
45:            return IPA_or_Fault
46:        end if
47:        IPA ← IPA_or_Fault
48:        PA_or_Fault ← WALK(Stage 2, IPA, isRWX)
49:        return PA_or_Fault
50:    end if
51: end procedure
```

Figure 7.10: Simplified two-stage translation table walk for a 4K pagetable.

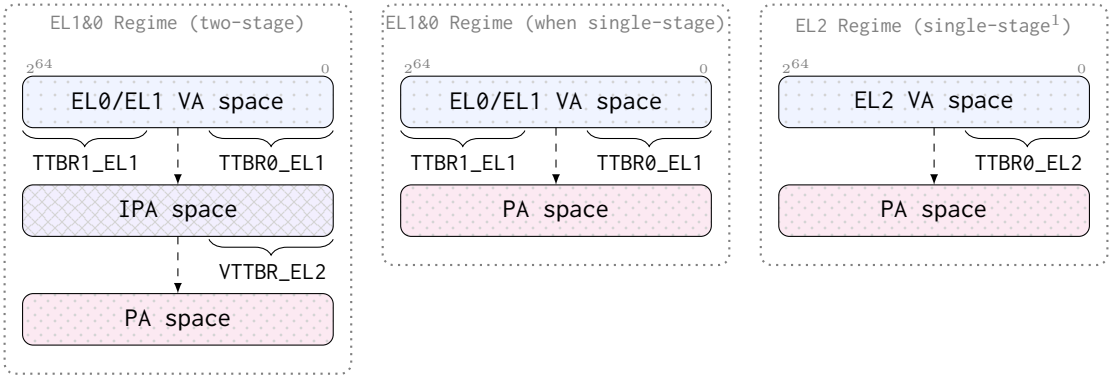


Figure 7.11: Translation regimes that apply to execution at EL0, EL1, and EL2.

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 7.11 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 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 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 only 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 which we do not cover here, including for EL3, secure mode, and the virtualised host extension (`FEAT_VHE`); see the Arm ARM [81, §D8.1.2] for more information.

[1] EL2 is always a single-stage regime. Note that there is a two-stage EL2&0 regime, which is not discussed here.

7.7 Caching in TLBs

It would have an unacceptable performance penalty to simply perform the (up to) 24 additional memory accesses for every instruction-fetch, read, or write. Therefore, the hardware does not do this. Instead, the results of previous translations of the same address are cached in specialised structures called *Translation Lookaside Buffers* (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.

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, unlike normal data caches, TLBs are not kept coherent by the hardware automatically. Therefore, systems code is expected to manage the TLBs manually, 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 maintenance operations, and the TLB cache misses they subsequently create, impose additional performance penalties on the software using them. To reduce this 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* (ASIDs), for Stage 1 address spaces, and *virtual machine identifiers* (VMIDs), for Stage 2 address spaces. Entries in the TLB are tagged with the current ASID and VMID, and only that address space will see entries in the TLB with that combination.

The current identifier 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. It is not necessary to do TLB maintenance, so long as it ensures the identifiers are distinct.

As there are only finitely many identifiers available (typically it is an 8-bit field), eventually TLB maintenance is required in order to re-use a previously allocated identifier, for a new address space. But, this typically happens far less frequently than context switches between pre-existing address spaces. The provided TLB maintenance instructions can target specific ASIDs or VMIDs, 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.

TLB maintenance Recall that on a TLB miss, any translation table entries read during any hardware translation table walk can be cached in the TLB, tagged with the current context: the translation regime, the ASID and VMID, the VA/IPA being translated, and so on. TLB maintenance instructions remove cached copies of those entries in the TLB.

A single TLB maintenance instruction which cleared the entire TLB would be prohibitively expensive, and often unnecessary; instead, Arm define a large family of instructions for targeted removal of particular entries, under the `TLBI` mnemonic.

A simplified format for a `TLBI` mnemonic is a product of fields:

```
1    TLBI <type><regime>{<broadcast>}{,<Xn>}
2
3    <type> =
4      ALL | VMALL | ASID | VA{A}{L} | IPAS2{L}
5    <regime> =
6      E1 | E2
7    <broadcast> =
8      IS
```

The type selects which kinds of cached entries will be affected:

- `ALL` affects all (Stage 1 and 2) entries.
- `VMALL` affects all (Stage 1 and 2) entries, for a particular VMID.
- `ASID` affects all Stage 1 entries, for a particular ASID.
- `VA` affects Stage 1 entries which were cached for translations of the given virtual address. This affects only the ASID provided, or can optionally appended with `A`, to affect entries for any ASID. Appending `L` makes the TLB maintenance only affect cached last-level entries (only page, not block).
- `IPAS2` affects Stage 2 entries which were cached for translations of the given intermediate-physical address, including for cached entries of a Stage 1 walk during a Stage 2 walk. Appending `L` makes the TLB maintenance only affect cached last-level entries (only page, not block).

The regime selects entries by the tagged translation regime:

- `E1` only removes cached entries for walks for the EL1&0 regime.
- `E2` only removes cached entries for the EL2, or the EL2&0 regime, whichever is currently enabled.

Finally, the optional broadcast flag determines which (set) of CPUs the TLB maintenance should affect:

- `IS` ('Inner-Shareable') broadcasts the TLB maintenance to all cores in the same Inner Shareable domain, and the `TLBI` does not finish until all cores have completed their maintenance.

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 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 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 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)

This is not an exhaustive list, see the full description in the Arm manual for a more complete description [12, D5-4915], but covers all those that appear in the following chapters.

Chapter 8

Relaxed virtual memory

Now, we 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 discusses both the relevant microarchitecture as we understand it, and also the behaviours which 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. In practice, things are not so simple. In some places, there is no clean separation between aspects there are clearly defined in the architecture reference, and those that are not; the manual sometimes has a shallow covering, with some but not all, of the key details. In other places, the reference may have been updated or changed over the course of the work, clarifying parts of the architecture. While such updates may have happened concurrently with discussions with Arm, the reference text itself is solely the responsibility of Arm. In §8.9 we will return to this, 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 permits, 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 first discuss (in §8.2) how translation affects the prior usermode tests covered in previous work, primarily for the case where locations are aliased. Then, we explore how translation entries may be cached (§8.3) and the fundamental behaviours which arise from translation and the walk (§8.4). Building upon that, we will see how those caches affect the discussed behaviours (§8.5). Then, we will explore how the various kinds of TLB maintenance interact with those cached translations (§8.6), 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 (§8.7).

Chapter contents

8.1 Virtual memory litmus tests

As we explore deeper into the systems semantics, we are exposed to more and more of the microarchitectural machine state; understanding that state is integral to understanding the behaviour of the machine. Virtual memory poses its own specific challenges, but is fundamentally no different than the other fragments of Arm we have seen. As such, exploring the architectural intent is best done through the creation, discussion, and evaluation of, small test programs which are representative examples of common software patterns or interesting hardware behaviours. Therefore, litmus tests exploring those behaviours must include information about not only the memory locations of the test, but also the setup of the pagetables which map them. This is best demonstrated by an example.

A virtual memory litmus test Much as in usermode (and ifetch, see Chapter 3) we examine litmus tests containing a relatively small amount of code corresponding to some interesting behaviour we wish to investigate. To illustrate this, Figure 8.1 contains the test listing for a simple (but non-trivial) virtual-memory litmus test, MP.RTf.inv+dmb+po.

MP.RTf.inv+dmb+po AArch64

```
Initial state: x |-> invalid, z |-> pa1,
*pa1 = 1, y |-> pa2, 0:X0=desc3(z),
0:X1=pte3(x), 0:X2=1, 0:X3=y, 1:X1=y, 1:X3=x
```

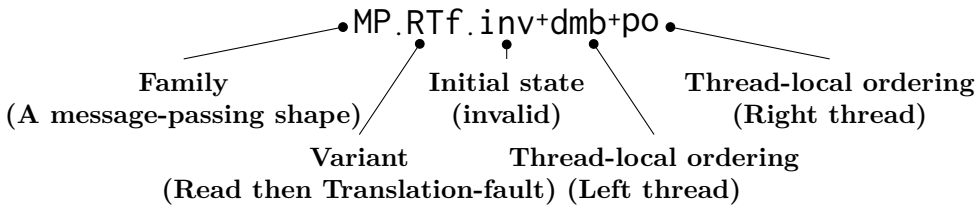
Thread 0	Thread 1	Thread1 El1 Handler

STR X0,[X1]
DMB ST
STR X2,[X3]
	LDR X0,[X1]
	LDR X2,[X3]
		MOV X2,#0
		
		MRS X13,ELR_EL1
		ADD X13,X13,#4
		MSR ELR_EL1,X13
		ERET

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

Figure 8.1: Test MP.RTf.inv+dmb+po: code listing.

This test is a variant of the classic message-passing test, but where one of the reads in the relaxed cycle of events is an implicit read due to a translation table walk. More specifically, the second read in the right-hand thread is the implicit read of the last-level entry of the stage 1 translation table walk, which in this case was initially invalid and so the interesting executions results in a translation fault. The test is explained in more detail below. In general we can take each of the classic usermode litmus test shapes, and re-imagine them in a virtual memory context, replacing one or more of the explicit memory events in the cycle with implicit ones from one or more translation table walks, and typically making some of the writes be writes to pagetables. We can then assign a relatively lightweight naming scheme for such litmus tests: for example, in MP.RTf.inv+dmb+po, the name can be broken down into separate fields representing the shape (family), which of the events are replaced by implicit ones, and whether the initial state for those implicit accesses are valid or invalid:



Not all litmus tests follow this convention: some do not correspond to a shape from the suite of usermode litmus tests, and others are derived from virtual-memory-specific patterns which arise in software or from discussion with architects.

In detail, this test mimics the usual message-passing pattern, with two locations `x` and `y`, with one thread reading the locations sequentially, in the inverse order, reading the 'flag' `y` first, then the 'data' `x` second. However, in this case, the data is not a value in a memory location, but the mapping of the memory location itself. This can be seen in the 'Initial State' part of the code listing (Figure 8.1), which contains

not only the usual initial register and memory location values for the test, but also a terse description of the initial mappings of those locations: `x` is invalid, so any access results in a translation fault; `z` maps to physical address `pa1` which initially holds 1; and `y` maps to `pa2`, which holds zero. The initial register state now also can reference parts of the pagetable: register `X1` in Thread 0 contains the value `pte3(x)` which is the address of the last-level (level 3) entry which is responsible for mapping `x`; and `X0` contains the value `desc3(z)`, which is the initial value of the entry responsible for mapping `z`.

The test then begins in Thread 0 by copying the entry which maps `z` into the entry which maps `x`, effectively making `x` an alias of `z`, before passing a message to Thread 1 via `y`. Thread 1 then reads `y`, and then attempts to read `x`. The second load will either be translated using the new translation, in which case it reads from `pa1` and get `1`, or be translated from the initial value and result in translation fault. In the case where the second load faults, execution jumps to the 'Thread 1 EL1 Handler' block, which writes `0` to `X2` and advances the program counter to the next instruction[1]. The final state corresponds to an execution in which the first load receives the message, and so reads `1`, but the second fails with a translation fault reading from a stale translation table entry.

The interesting relaxed execution can be seen as a set of events with some relations which witness the order the events happened in. This test's events diagram is shown in Figure 8.2.

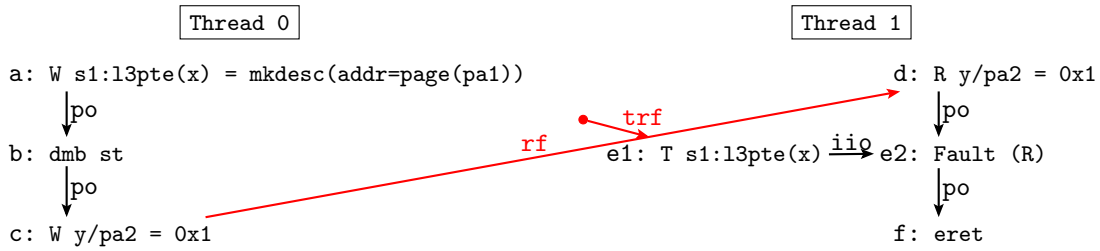


Figure 8.2: Test MP.RTf.inv+dmb+po: execution diagram

These diagrams are much like the ones drawn for usermode tests, but with a few key differences:

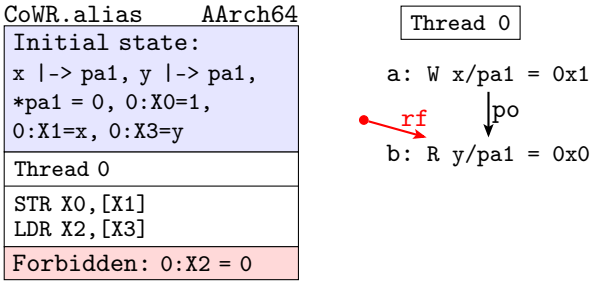
- ▷ The implicit reads due to translation table walks are included in the execution, labelled with `T` (for *T*ranslate), and ordered within an instruction by `iio` (intra-instruction-order), with each other and with the associated explicit events of the instruction.
- ▷ Memory accesses are annotated with both their virtual and physical addresses, e.g. event `d: R y/pa2 = 0x1` denotes the read for a virtual address `y`, and read from the physical address `pa2`.
- ▷ We introduce a notation whereby some addresses and values can be written using symbolic functions, e.g. `a: W s1:l3pte(x) = mkdesc(addr=page(pa1))` denotes the write is to the stage 1 level 3 pagetable entry which maps `x`, with a value that is a 64-bit descriptor whose output-address field is for `pa1`'s page.
- ▷ Accesses which fault generate a `Fault` event, annotated with the access kind (read/write/execute).

We elide translation read events, physical address labels, and other uninteresting and extraneous details.

Symbolic helper functions These symbolic functions are implemented as part of the `isla-cat` language, accepted by `isla-axiomatic`. Here are the helpers used by most of the tests in this section. Entries listed as `f<N>` mean a family of functions `f1`, `f2`, `f3` and so on, where `N` is typically the level.

- ▷ `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 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 64-bit descriptor for a valid table entry at level N where the next-level-table address is given by the `table` parameter.

[1] The `ELR_ELx` (Exception-link-register) defines the return address of an exception to ELx. Translation faults, by default, return to the instruction that generated them.



This test is a variation on the standard write-read coherence test, CoWR, but 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`.
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: Test CoWR.alias

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, in the usermode fragments of the ISA. 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

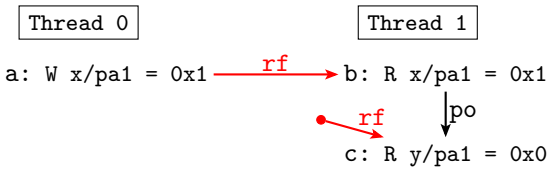
For data memory accesses, one of the most fundamental guarantee that architectures provide is *coherence*: in any execution, for each memory location, there is a total order of the accesses to that location, consistent with the program order of each thread, with reads reading from the most recent write 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 CoWR.alias test (Figure 8.3).
- 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, p.120), CoRR2.alias+po (Figure 8.5, p.120)).
- 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, p.121), PPOCA.alias (Figure 8.6, p.121)).

CoRR0.alias+po AArch64

Initial state: x |-> pa1, y |-> pa1, *pa1 = 0, 0:X0=1, 0:X1=x, 1:X1=x, 1:X3=y	
Thread 0	Thread 1
STR X0,[X1]	LDR X0,[X1] LDR X2,[X3]
Forbidden: 1:X0=1 & 1:X2=0	

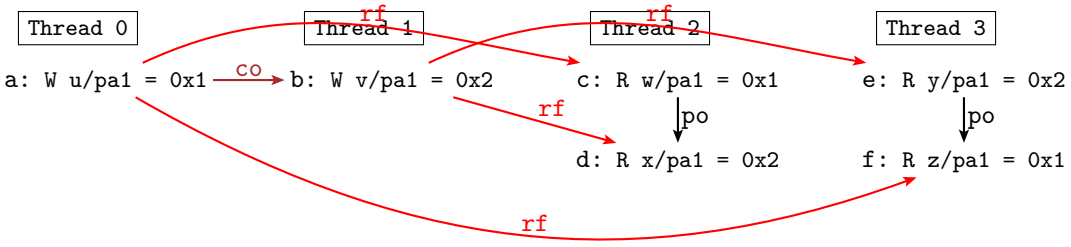


This test is a variation of the data memory CoRR0 test, where one of the loads has been replaced with a load of a distinct virtual address which aliases to the same underlying physical address. Note that, like the original test, it is forbidden to read from the initial state in the later load, as this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

Figure 8.4: Test CoRR0.alias+po

CoRR2.alias+po AArch64

Initial state: u |-> pa1, v |-> pa1, w |-> pa1, x |-> pa1, y |-> pa1, z |-> pa1, *pa1 = 0, 0:X0=1, 0:X1=u, 1:X0=2, 1:X1=v, 2:X1=w, 2:X3=x, 3:X1=y, 3:X3=z			
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]
Forbidden: 2:X0=1 & 2:X2=2 & 3:X0=2 & 3:X2=1			



This test is a variation of the data memory CoRR2 test. Here, there are many options for considering aliasing; we present here 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: Test CoRR2.alias+po

MP.alias3+rfi-data+dmb AArch64

Initial state: x |-> pa1, y |-> pa2, z |-> pa1, *pa1 = 0, *pa2 = 0, 0:X0=1, 0:X1=x, 0:X3=z, 0:X5=y, 1:X1=y, 1:X3=x	
Thread 0	Thread 1

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

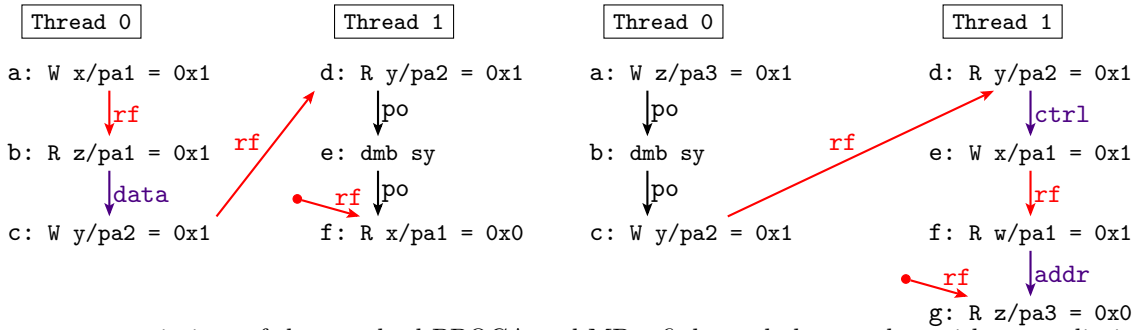
Allowed: 1:X0 = 1 & 1:X2 = 0	

PPOCA.alias AArch64

Initial state: w |-> pa1, x |-> pa1, y |-> pa2, z |-> pa3, *pa1 = 0, *pa2 = 0, *pa3 = 0, 0:X0=1, 0:X1=z, 0:X2=1, 0:X3=y, 1:X1=y, 1:X2=1, 1:X3=x, 1:X5=w, 1:X7=z	
Thread 0	Thread 1

STR X0,[X1]
DMB SY
STR X2,[X3]
	LDR X0,[X1]
	CBNZ X0,L0
	L0:
	STR X2,[X3]
	LDR X4,[X5]
	EOR X8,X4,X4
	LDR X6,[X7,X8]

Allowed: 1:X0 = 1 & 1:X4 = 1 & 1:X6 = 0	



These tests are variations of the standard PPOCA and MP+rfi-data+dmb tests, but with some aliasing. Both are examples of *forwarding*: thread-locally reading from a write before it 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: forwarding tests with aliasing.

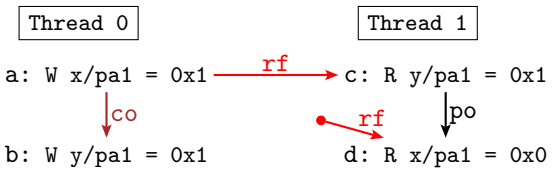
8.2.2 Aliasing different locations

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, starting from the traditional MP+pos test, aliasing the two locations to the same physical address gives 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.

MP.alias+pos AArch64

Initial state: x |-> pa1, y |-> pa1, *pa1 = 0, 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] STR X2,[X3]	LDR X0,[X1] LDR X2,[X3]
Forbidden: 1:X0 = 1 & 1:X2 = 0	



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. In Flat, this may permit some program-order-later instructions to know they operate over disjoint footprints and begin execution early.

With the introduction of address translation, however, the point where instructions know the footprints of previous instructions 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.

8.2.4 Must not be same physical address

One might expect that, when deciding whether to propagate a store, if the offset within the page of virtual memory is distinct 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. This is, perhaps surprisingly, not the case: although the accesses definitely will not access the same physical address, the program-order earlier access may still fault, meaning the write will not be reached. This means that writes must wait for program-order-earlier translations to finish (or at least, be known to not fault) before they can be propagated to other threads.

8.3 What can be cached in TLBs

As was described in §7.7, Arm hardware can have TLBs, caching previously seen translations. This caching is, however, restricted, 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.

We explore this by first examining the structure of a concrete implementation's MMU, before abstracting away details to produce an abstract model, which we then use to explain the behaviours throughout the rest of the chapter.

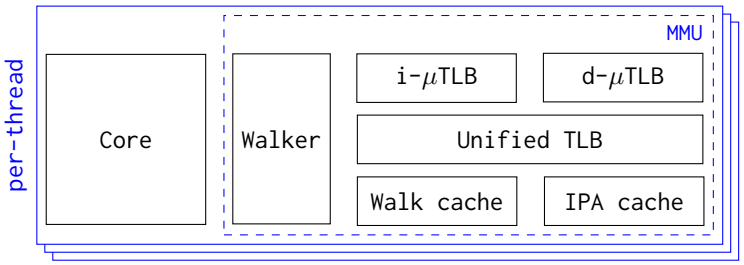


Figure 8.8: Block diagram of the Arm Cortex-A53 memory-management-unit [73].

8.3.1 Microarchitectural TLBs

Here we 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, 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 generally have comparable TLB configurations [92, 93, 94, 95, 96].

The Arm A53 Technical Reference Manual (TRM) describes, in detail, the structure of the memory management unit [73, 5-2] of the A53, and its constituent parts. Figure 8.8 contains a block diagram representing the key structures in the A53's memory management unit.

Each core has its own MMU, and each MMU contains:

- ▷ the walker, which actually does the translation table walk;
- ▷ one instruction micro-TLB (denoted i-μTLB in the diagram);
- ▷ one data micro-TLB (d-μTLB);
- ▷ one unified TLB;
- ▷ one walk cache; and,
- ▷ one IPA cache.

The microarchitectural TLBs store translations: virtual to physical mappings, plus permissions and so-on, tagged with their translation context: the translation regime, ASID, VMID, virtual or intermediate-physical address, and so on. The TLBs are arranged hierarchically, with small, 10-entry, micro TLBs for instruction and data streams separately, and one large 512-entry unified TLB. On a TLB miss, the MMU performs a translation table walk using the walker.

When it begins this walk, the MMU first checks the walk cache. Walk caches store mappings from virtual address to the physical address of the last level translation table – commonly seen in modern MMUs. When the walk cache has an entry, the walker can skip over most of the walk and directly read the leaf entry out the last-level table.

If a second stage of translation is required during the walk, the IPA cache is used (and may be used many times during the same walk). The IPA cache stores mappings from intermediate physical to physical memory — without an 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.

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 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 imagine modelling the MMU as if it were a separate asynchronous unit, one for each thread, each with its own TLB.

Later, we will see tests that justify and ground this particular choice of abstraction, and we will explore the consequences of this model in more rigorous detail. For now, we can imagine this model MMU as a set of (concurrently) executing translation table walks, and a 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 enumerate all the possible shapes of TLBs, and the kinds of information they can cache. Instead, we define a *model TLB*. This model TLB acts as a cache of writes of translation table entries, each tagged with some context. This allows the model to cache any combination of valid entries in a translation table walk: weak enough to allow all currently known TLB implementations, but strong enough to not break any of the guarantees software requires. These guarantees are explored, in detail, in §8.4 and §8.5.

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. Each entry must also be tagged with some contextual information. This contextual information is required for two distinct purposes, and so we split into two parts: an architectural context which defines the scope of a cached translation table entry, that is, for which translations the cached value is permitted to be used; and an extended model context, which serve as tags for TLB maintenance operations to target.

This contextual information is:

- ▷ the architectural context, used for matching entries in the TLB:
  - – the translation regime;
  - – the VMID;
  - – the ASID (or a 'global indicator');
  - – and the virtual address, intermediate physical address, and/or physical address of the translation.
- ▷ the model context, used for targeted TLB maintenance:
  - – 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.

Operationally, one can imagine performing a translation using the model MMU 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. Any behaviour exhibited by a specific micro-architectural MMU and TLB configuration, and therefore all the litmus tests in this chapter, should be consistent with 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 does not enumerate all the possible speculation machinery or prefetchers, so instead we opt for a model that defines a wide envelope of behaviour, weak enough to permit a range of plausible hardware implementations: 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 'Model TLB entries'), 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 context information from the walk. The behaviour of those non-TLB reads is 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 write a new mapping without potentially having multiple entries in the TLB for the same virtual address, as can be seen in the tests in §8.4, which forms a key part of the software pattern for Arm to update a previously-valid pagetable entry: *break-before-make*, discussed in §8.6.5.

8.4 Reads not from TLB

The requirement that invalid entries are not cached in the TLB gives us a way to directly observe the behaviour of non-TLB reads: translation table reads which directly result in a translation fault *must not* have come from a 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 'Enhanced Translation Synchronisation', `FEAT_ETS`, see §8.4.3).

In this section we will explore the properties these reads have, and the guarantees software can rely on. We will see that these reads are affected by thread-local re-ordering, to an even greater extent than data memory reads, and explore the synchronization that recovers the sequential semantics. We will further 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 (non-TLB reads) preserve the original program order. One of the simplest questions one can 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 8.9, p.126). Starting with a VA ('x') initially invalid at level 3, which therefore cannot have its level 3 entry cached in any TLB (directly or indirectly), the test 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`. The question is whether the read of `x` can read-from the old translation table entry, generating a translation fault. This result is architecturally allowed, and readily observed on hardware.

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. 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 8.10, p.126).

A translation table walk read may not, in general, be re-ordered past 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 re-ordering of the propagation of the write to other threads (LB.TT.inv+pos (Figure 8.11, p.127)) with program-order earlier translation table walks, and translations reading from program-order later writes (CoTW1.inv (Figure 8.12, p.127)).

CoWTf.inv+po AArch64

Initial state: x |-> invalid, y |-> pa1, *pa1 = 1, 0:X0=desc3(y), 0:X1=pte3(x), 0:X3=x	
Thread 0	Thread0 El1 Handler

STR X0,[X1]
LDR X2,[X3]
	MOV X2,#0
	MRS X20,ELR_EL1
	ADD X20,X20,#4
	MSR ELR_EL1,X20
	ERET

Allowed: 0:X2 = 0	

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).

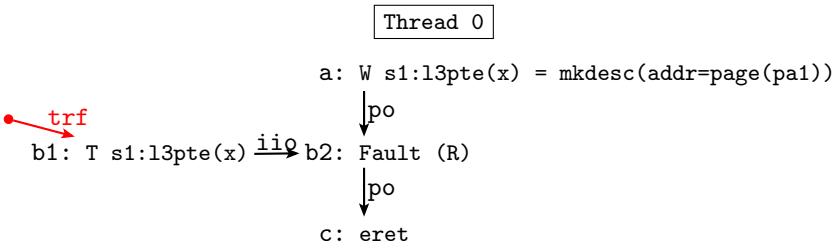


Figure 8.9: Test CoWTf.inv+po

CoRpteTf.inv+po AArch64

Initial state: x |-> invalid, y |-> pa1, *pa1 = 1, 0:X0=desc3(y), 0:X1=pte3(x), 1:X1=pte3(x), 1:X3=x		
Thread 0	Thread 1	Thread1 El1 Handler

STR X0,[X1]	LDR X0,[X1]
	LDR X2,[X3]
		MOV X2,#0
		
		MRS X13,ELR_EL1
		ADD X13,X13,#4
		MSR ELR_EL1,X13
		ERET

Allowed: 1:X0 = desc3(y) & 1:X2=0		

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.

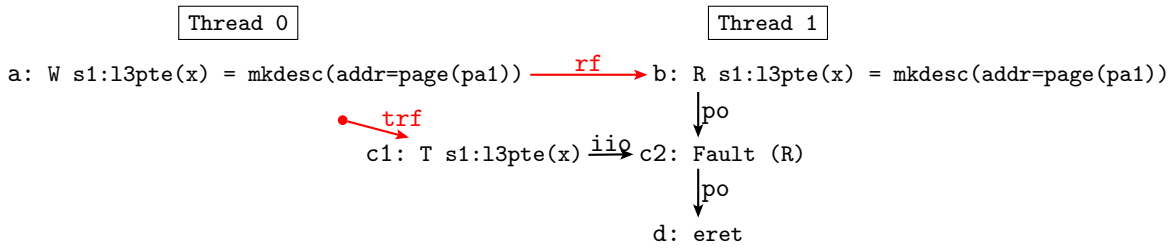


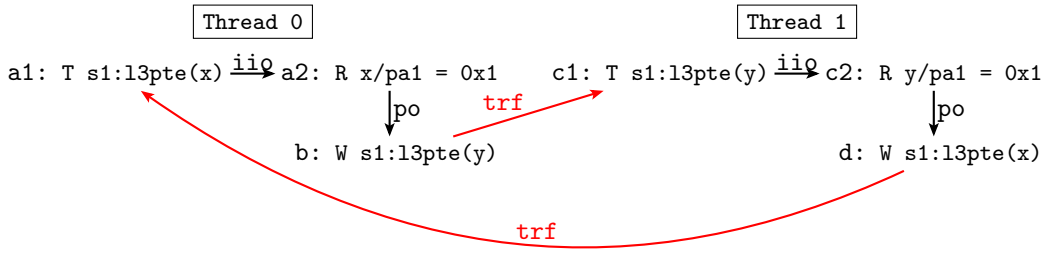
Figure 8.10: Test CoRpteTf.inv+po

LB.TT.inv+pos AArch64

Initial state: x |-> invalid, y |-> invalid, *pa1 = 1, 0:X1=x, 0:X2=mkdesc3(oa=pa1), 0:X3=pte3(y), 1:X1=y, 1:X2=mkdesc3(oa=pa1), 1:X3=pte3(x)			
Thread 0	Thread 1	Thread0 El1 Handler	Thread1 El1 Handler

MOV X0,#0
LDR X0,[X1]
STR X2,[X3]
	MOV X0,#0
	LDR X0,[X1]
	STR X2,[X3]
		MRS X13,ELR_EL1
		ADD X13,X13,#4
		MSR ELR_EL1,X13
		ERET
			MRS X13,ELR_EL1
			ADD X13,X13,#4
			MSR ELR_EL1,X13
			ERET

Forbidden: 0:X0 = 1 & 1:X0=1			



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.

Figure 8.11: Test LB.TT.inv+pos

CoTW1.inv AArch64

Initial state: x |-> invalid, y |-> pa1, *pa1 = 1, 0:X1=x, 0:X2=desc3(y), 0:X3=pte3(x)	
Thread 0	Thread0 El1 Handler

LDR X0,[X1]
STR X2,[X3]
	MOV X0,#0
	
	MRS X13,ELR_EL1
	ADD X13,X13,#4
	MSR ELR_EL1,X13
	ERET

Forbidden: 0:X0 = 1	

The store to the translation table (b) cannot be re-ordered with the program-order earlier translation table walk (a1), preventing that walk from reading from the store.

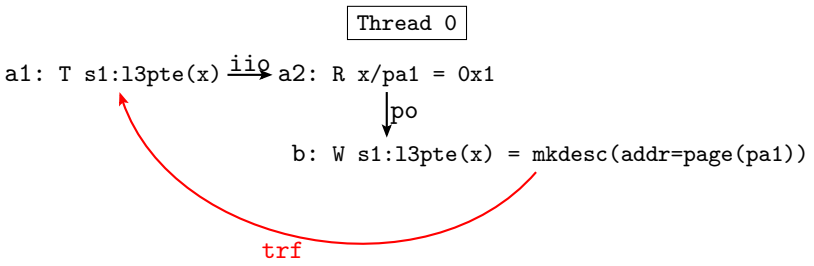


Figure 8.12: Test CoTW1.inv

CoWTf.inv+dsb-isb AArch64

Initial state: x |-> invalid, y |-> pa1, *pa1 = 1, 0:X0=desc3(y), 0:X1=pte3(x), 0:X3=x	
Thread 0	Thread0 El1 Handler

STR X0,[X1]
DSB SY
ISB
LDR X2,[X3]
	MOV X2,#0
	MRS X20,ELR_EL1
	ADD X20,X20,#4
	MSR ELR_EL1,X20
	ERET

Forbidden: 0:X2 = 0	

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.

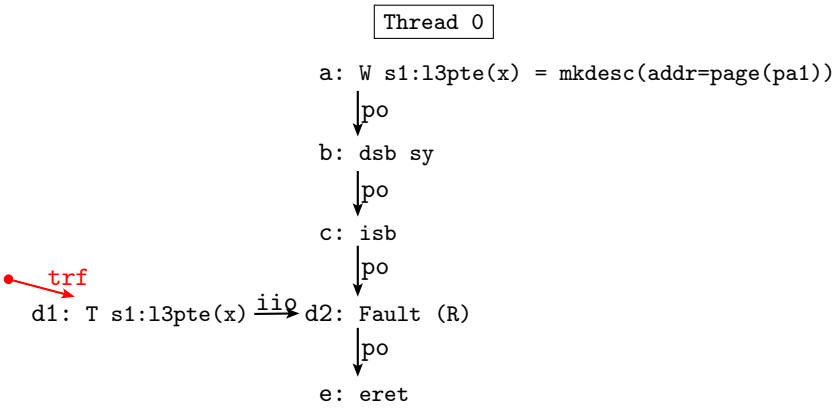


Figure 8.13: Test CoWTf.inv+dsb-isb

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, e.g. to ensure that program-order later accesses use the new translations.

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, then the ISB ('Instruction Synchronization Barrier') ensures that program-order later instructions begin execution (including translation table walks) after the completion of the DSB, and therefore the earlier stores.

Locally-ordered-previous writes If we introduce this sequence into the previous CoWTf.inv+po test (Figure 8.9, p.126), we obtain the CoWTf.inv+dsb-isb test (Figure 8.13), 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.

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 8.14, p.129)), or through a normal data load of the translation table (CoRpteTf.inv+dsb-isb (Figure 8.15, p.129)), 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) suffices.

Microarchitecturally, this is because translation table walkers behave as separate 'observers' [68, p.14726]. Essentially, the MMU is a sibling element to the processor, accessing memory through the same coherent mechanism as the primary CPU. This 'separate observers' principle may seem a reasonable model at first, however, we will see later on in §8.4.4 where it breaks down as an architectural model.

CoTTf.inv+dsb-isb AArch64

Initial state: x |-> invalid, y |-> pa1, *pa1 = 1, 0:X0=desc3(y), 0:X1=pte3(x), 1:X1=x, 1:X3=x		
Thread 0	Thread 1	Thread1 El1 Handler

STR X0,[X1]	LDR X2,[X1]
	MOV X0,X2
	DSB SY
	ISB
	LDR X2,[X3]
		MOV X2,#0
		
		MRS X13,ELR_EL1
		ADD X13,X13,#4
		MSR ELR_EL1,X13
		ERET

Forbidden: 1:X0 = 1 & 1:X2=0		

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.

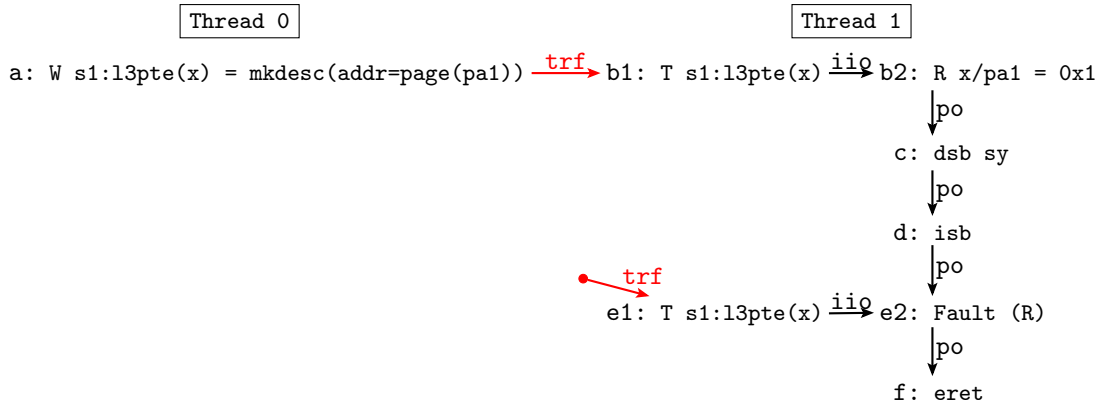


Figure 8.14: Test CoTTf.inv+dsb-isb

CoRpteTf.inv+dsb-isb AArch64

Initial state: x |-> invalid, y |-> pa1, *pa1 = 1, 0:X0=desc3(y), 0:X1=pte3(x), 1:X1=pte3(x), 1:X3=x		
Thread 0	Thread 1	Thread1 El1 Handler

STR X0,[X1]	LDR X0,[X1]
	DSB SY
	ISB
	LDR X2,[X3]
		MOV X2,#0
		
		MRS X13,ELR_EL1
		ADD X13,X13,#4
		MSR ELR_EL1,X13
		ERET

Forbidden: 1:X0 = desc3(y) & 1:X2=0		

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.

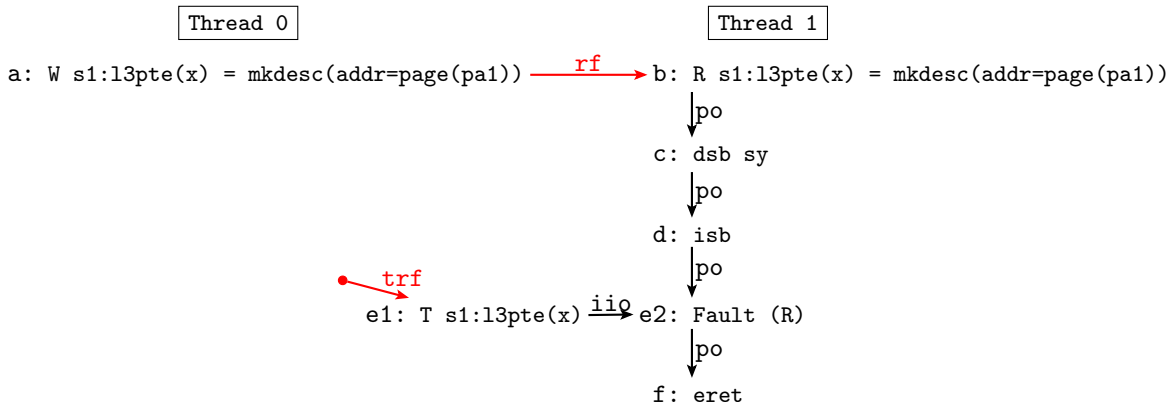


Figure 8.15: Test CoRpteTf.inv+dsb-isb

CoTTf.inv+ctrl-isb AArch64

Initial state: x |-> invalid, y |-> pa1, *pa1 = 1, 0:X0=desc3(y), 0:X1=pte3(x), 1:X1=x, 1:X3=x		
Thread 0	Thread 1	Thread1 El1 Handler

STR X0,[X1]	MOV X0,#0
	LDR X0,[X1]
	EOR X4,X0,X0
	CBNZ X4,LC00
	LC00:
	ISB
	MOV X2,#0
	LDR X2,[X3]
		MRS X13,ELR_EL1
		ADD X13,X13,#4
		MSR ELR_EL1,X13
		ERET

Forbidden: 1:X0 = 1 & 1:X2=0		

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.

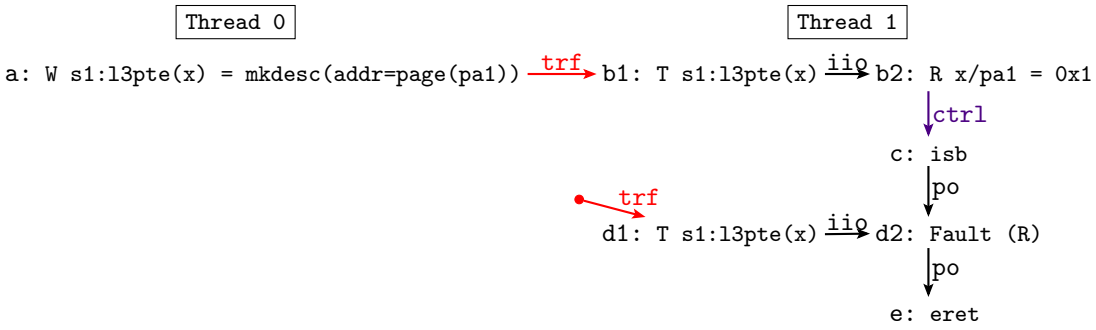


Figure 8.16: Test CoTTf.inv+ctrl-isb

Instruction synchronisation 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 we introduced a DSB in the previous tests, without which they would be allowed. A control-dependency to the ISB would also work (CoTTf.inv+ctrl-isb (Figure 8.16)).

Address dependencies In previous work, address dependencies were assumed fundamental. Now we can define what an address dependency is: dataflow into the translation table walk. Address dependencies remain a strong way to order events. Arm does not permit observable speculation of the values or addresses of 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. Therefore, non-TLB translation reads are locally-ordered-after any read whose value flows into that non-TLB read, as demonstrated in CoRpteTf.inv+addr (Figure 8.17, p.131).

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.

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, see Part I).

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 8.18 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 see that the DMB does not enforce

CoRpteTf.inv+addr AArch64

Initial state: x |-> invalid, y |-> pa1, *pa1 = 1, 0:X0=desc3(y), 0:X1=pte3(x), 1:X1=pte3(x), 1:X3=x		
Thread 0	Thread 1	Thread1 El1 Handler

STR X0,[X1]	LDR X0,[X1]
	EOR X4,X0,X0
	LDR X2,[X3,X4]
		MOV X2,#0
		
		MRS X13,ELR_EL1
		ADD X13,X13,#4
		MSR ELR_EL1,X13
		ERET

Forbidden: 1:X0 = desc3(y) & 1:X2=0		

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

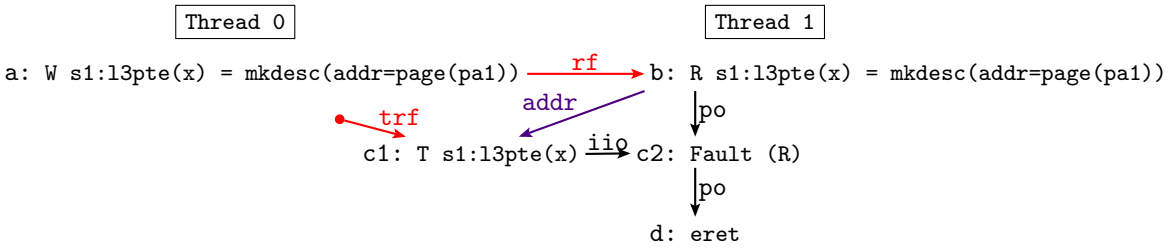


Figure 8.17: Test CoRpteTf.inv+addr

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 Arm provides two memory barrier instructions: DMB ('data memory barrier') and DSB ('data synchronisation barrier'). The base intent is that DMB orders explicit memory accesses, whereas DSB is a strictly stronger barrier also ordering some implicit accesses, and other barriers and cache maintenance (including TLB invalidation). This means that, for any litmus test with a DMB, a DSB of the same access kind could be substituted, and the resulting test is no weaker. Over time, the architectural intent around how barriers order implicit events has changed, and is still subject to change.

CoWTf.inv+dmb	AArch64
Initial state: x |-> invalid, y |-> pa1, *pa1 = 1, 0:X0=desc3(y), 0:X1=pte3(x), 0:X3=x	
Thread 0	Thread0 El1 Handler

STR X0,[X1]
DMB SY
LDR X2,[X3]
	MOV X2,#0
	MRS X20,ELR_EL1
	ADD X20,X20,#4
	MSR ELR_EL1,X20
	ERET

Forbidden if ETS0:X2 = 0	

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

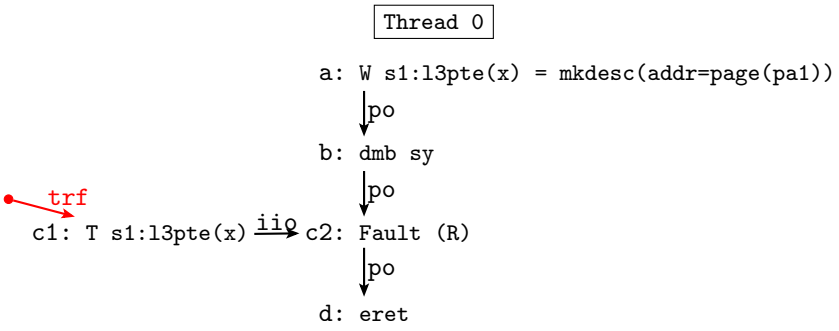


Figure 8.18: Test CoWTf.inv+dmb

8.4.3 Enhanced Translation Synchronization

> Note: since this work there have been further extensions including FEAT_ETS2 and FEAT_ETS3. See the discussion at the end of the section for more details.

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

When an instruction takes a translation fault, it is permitted (without ETS) to commit to taking that fault immediately. However, there may have been some thread-local dependencies or ordering into that instruction with the intent to order any explicit memory access (which do not happen if the instruction faults). The early committal to the fault effectively breaks such dependencies, allowing the handler to be executed before the dependency would have been resolved. This means, for example, if software were to create new translation tables and publish them, 'later' translations may have already committed to taking a fault, which software would observe as a *spurious* fault. This is what ETS tries to resolve.

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 phrased confusingly with reference to a seemingly non-existent event RW2, so requires some clarification: the scenario being described here is a case with two instructions, $\mathtt{I}_1$ and $\mathtt{I}_2$, each either a load or store. Imagine $\mathtt{I}_1$ and $\mathtt{I}_2$ both executing to completion, without generating any translation, address size, or access flag faults. Then, each instruction would have generated one or more explicit memory events. For example, a store might generate up to 8 separate write events (one for each byte). Call those events $\mathtt{E}_{ij}$ for the jth explicit event of instruction $\mathtt{I}_i$.

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

Then, if in some execution: $\mathtt{I}_2$ generates a translation fault, address size fault, or access flag fault; $\mathtt{E}_{1n}$ would have been locally-ordered-before $\mathtt{E}_{2,m}$ in an execution without the fault; and FEAT_ETS is enabled; then, $\mathtt{E}_{1n}$ is locally-ordered-before any translation table read $\mathtt{T}_{2,m,\_}$ in the execution with the fault. That is, we can imagine that in the execution with the fault there is a 'ghost' event corresponding to the fault in the place the explicit event *would have occurred* with the same ordering into it. This scenario is illustrated in Figure 8.19.

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 by ensuring faults cannot come from an out-of-order read of (what is now) a stale value from memory.

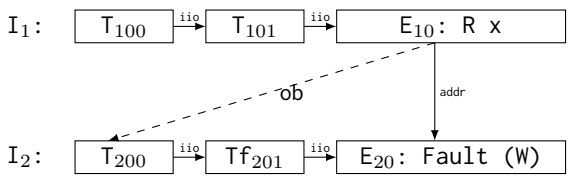


Figure 8.19: ETS Ghost events example: A load instruction ($\mathtt{I}_1$) followed followed (in program order) by a store instruction ($\mathtt{I}_2$), which faults. The address dependency means that the read event $\mathtt{E}_{10}$ is syntactically ordered-before the (ghost) write event $\mathtt{E}_{20}$, and so the read event is ordered before the reads of the translation table walk for $\mathtt{I}_2$ read from the TLB or memory (represented by the dashed ob line).

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 behaviour 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.

In particular, Linux 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. 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. 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.

ETS2, ETS3, and the future of ETS Since this work was done, Arm have further refined the architecture; there are now two further extensions to ETS: FEAT_ETS2, which extends the behaviour described above to include ordering from all instructions program-order before the fault [68, D8.2.6.1]; and FEAT_ETS3, which extends this to all MMU faults [97, D8.2.6.1].

FEAT_ETS, as described in this document, no longer exists. The memory model since developed by Arm, as found in the Arm architecture reference manual, describes the latest state of ETS [97, B2.3]. FEAT_ETS3 is now mandatory from Armv9.5.

8.4.4 Forwarding to the translation table walker

> Note: our understanding of the architectural intent for forwarding to translation table walks has changed over time. See the discussion at the end of the section for more details.

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

Although the translation table walker is described as a 'separate' observer, it is also part of the core that hosts it, and is allowed to read from that core's store buffer, effectively allowing writes to be 'forwarded' to the walker, as shown in the R.TR.inv+dmb+trfi test (Figure 8.20, p.135).

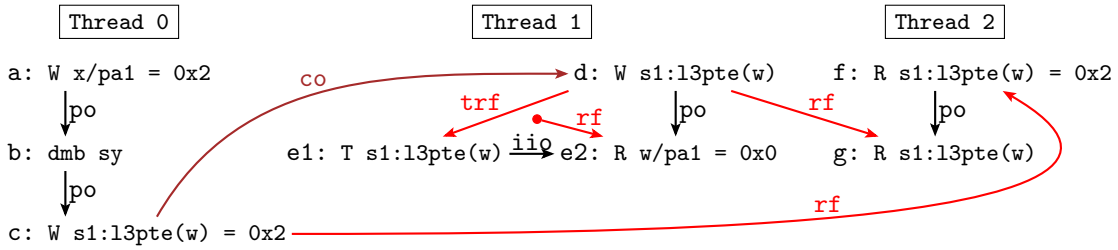
Changes to the architectural intent Since this work was completed, the architectural intent has been further clarified: non-TLB reads, except with FEAT_nTLBPA, must be multicopy atomic with respect to data accesses.

R.TR.inv+dmb+trfi AArch64

Initial state: w |-> invalid, x |-> pa1, *pa1 = 0, 0:X0=2, 0:X1=x, 0:X2=2, 0:X3=pte3(w), 1:X0=mkdesc3(oa=pa1), 1:X1=pte3(w), 1:X3=w, 2:X1=pte3(w)			
Thread 0	Thread 1	Thread 2	Thread1 El1 Handler

STR X0,[X1]
DMB SY
STR X2,[X3]
	STR X0,[X1]
	MOV X2,#1
	LDR X2,[X3]
		LDR X0,[X1]
		LDR X2,[X1]
			MRS X13,ELR_EL1
			ADD X13,X13,#4
			MSR ELR_EL1,X13
			ERET

Allowed: 1:X2=0 & 2:X0=2 & 2:X2=mkdesc3(oa=pa1)			



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. Thread 2 is an observer thread, witnessing that the write d happens after c.

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

8.4.5 Speculative execution

To facilitate out-of-order pipelines, the machine begins fetching and executing the next instruction before earlier instructions are finished. However, those earlier instructions may change the flow of execution through the program. Executing later instructions before those earlier instructions are finished means that those later instructions are being executed *speculatively*: the predicted control flow, or assumptions of independence between instructions, may turn out to be incorrect. When the control flow is mispredicted, or a speculative access leads a coherence violation, the speculated effects must be discarded.

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 reads and writes can be performed speculatively, their associated translations must also be permitted to be performed speculatively. This is what permits tests such as MP.RTf.inv+dmb+ctrl (Figure 8.21, p.136) to see an old value for the translation table entry down a speculative path.

However, forwarding from a speculative write to the translation table walker is disallowed. Reads from normal memory have no side effects, but other locations, such as devices, are *read sensitive*: reads may effect the device state. Software protects such locations by marking them as device memory in the translation tables, or leaving them unmapped altogether. A still-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 still-speculative writes. The MP.RT.inv+dmb+ctrl-trfi test (Figure 8.22, p.136) 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 restarting. In the usermode 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.

MP.RTf.inv+dmb+ctrl AArch64

Initial state: x |-> invalid, z |-> pa1, *pa1 = 1, y |-> pa2, 0:X0=desc3(z), 0:X1=pte3(x), 0:X2=1, 0:X3=y, 1:X1=y, 1:X3=x		
Thread 0	Thread 1	Thread1 El1 Handler

STR X0,[X1]
DMB SY
STR X2,[X3]
	LDR X0,[X1]
	CBNZ X0,L0
	L0:
	LDR X2,[X3]
		MOV X2,#0
		
		MRS X13,ELR_EL1
		ADD X13,X13,#4
		MSR ELR_EL1,X13
		ERET

Allowed: 1:X0 = 1 & 1:X2=0		

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

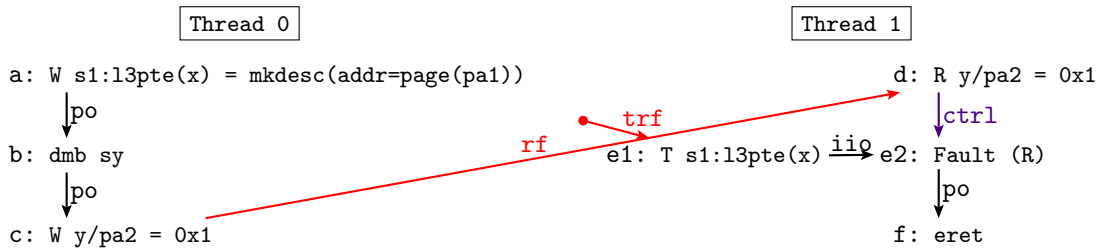


Figure 8.21: Test MP.RTf.inv+dmb+ctrl

MP.RT.inv+dmb+ctrl-trfi AArch64

Initial state: w |-> invalid, x |-> pa1, *pa1 = 0, y |-> pa2, 0:X0=1, 0:X1=x, 0:X2=1, 0:X3=y, 1:X1=y, 1:X2=mkdesc3(oa=pa1), 1:X3=pte3(w), 1:X5=w		
Thread 0	Thread 1	Thread1 El1 Handler

STR X0,[X1]
DMB SY
STR X2,[X3]
	LDR X0,[X1]
	CBZ X0,LC00
	LC00:
	STR X2,[X3]
	LDR X4,[X5]
		MOV X4,#2
		
		MRS X13,ELR_EL1
		ADD X13,X13,#4
		MSR ELR_EL1,X13
		ERET

Forbidden: 1:X0 = 1 & 1:X4=0		

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.

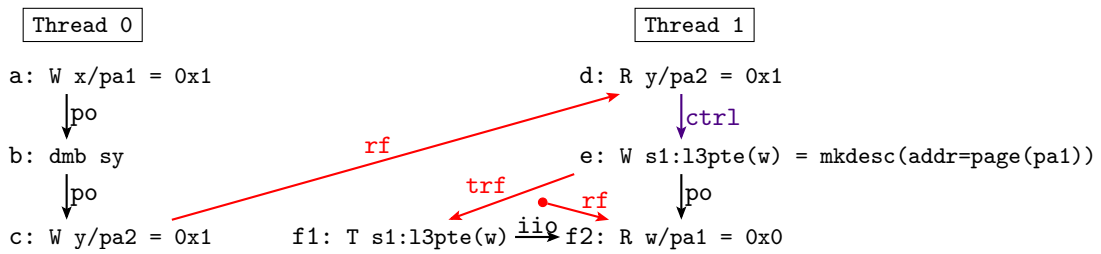


Figure 8.22: Test MP.RT.inv+dmb+ctrl-trfi

CoTTf.inv+po AArch64

Initial state: x |-> invalid, y |-> pa1, *pa1 = 1, 0:X0=desc3(y), 0:X1=pte3(x), 1:X1=x, 1:X3=x		
Thread 0	Thread 1	Thread1 El1 Handler

STR X0,[X1]	LDR X2,[X1]
	MOV X0,X2
	LDR X2,[X3]
		MOV X2,#0
		
		MRS X13,ELR_EL1
		ADD X13,X13,#4
		MSR ELR_EL1,X13
		ERET

Allowed: 1:X0 = 1 & 1:X2=0		

The translation-table-walks of the two same-address loads of x can execute out-of-order, even when the later translation table read (c1) reads from a different write than the program-order-earlier one (b1).

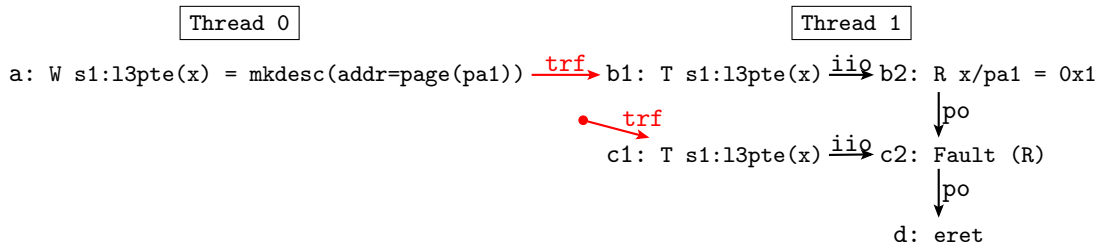


Figure 8.23: Test CoTTf.inv+po

Translation table walk reads, while they are reads, are not required to be restarted to recover coherence, do not do this hazard checking. This is most obvious in the CoTTf.inv+po (Figure 8.23, p.137), where the two translations for the two same-address loads in Thread 1 are performed out-of-order.

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 atomicity and multi-copy atomicity.

Recall that single-copy atomic reads always read the maximum they can from another single-copy atomic write; in particular a 64-bit atomic read 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, p.138) 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 Arm and RISC-V give, but earlier versions of Arm and IBM's Power architectures do not. On Arm, threads can observe their own writes early, through write forwarding, giving a weaker form of multi-copy atomicity referred to as *other-multi-copy atomicity* by Arm.

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 respect other-multi-copy-atomicity: 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.

CoWroW.inv+dsb-isb AArch64

Initial state: x |-> invalid, *pa1 = 0, 0:X0=mkdesc3(oa=pa1, AP=3), 0:X1=pte3(x), 0:X2=1, 0:X3=x	
Thread 0	Thread0 El1 Handler
STR X0,[X1] DSB SY ISB STR X2,[X3]	MRS X20,ELR_EL1 ADD X20,X20,#4 MSR ELR_EL1,X20 ERET
Forbidden: *pa1=1	

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, so there is no generated events diagram for this test.

Figure 8.24: Test CoWroW.inv+dsb-isb

There are three combinations of multi-thread reads of interest, where a weaker architecture (with split pagetable and data memory storage) might have mixed non-multi-copy atomic behaviours. The first of these is the most basic: translation-read to translation-read, that is, the pagetable accesses are multi-copy atomic, and this is what forbids reading the old translation table value in Thread 2 in the WRC.TRTf.inv+po+dsb-isb test (Figure 8.25, p.139). The other two are combinations of read-to-translation-read and translation-read-to-read; these show us that translation accesses and explicit data accesses are architecturally unified: information about the memory state learned through one kind of access applies to accesses of the other. This is what forbids the WRC.RRTf.inv+dmb+dsb-isb (Figure 8.26, p.140) and WRC.TRR.inv+po+dsb (Figure 8.27, p.140) tests from reading the old value from memory at the end.

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 Chapter 7, each translation table walk performs many reads, as many as 24, 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 executed out-of-order, translation-reads within a translation table walk cannot, as this would require the hardware to do value speculation on the next-level table address, but as discussed in §8.4.5, using speculative values in a walk is 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 ROT.inv+dsb test (Figure 8.28, p.141), 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, for example, the ‘load pair’ and ‘store pair’ instructions, misaligned accesses, and 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.

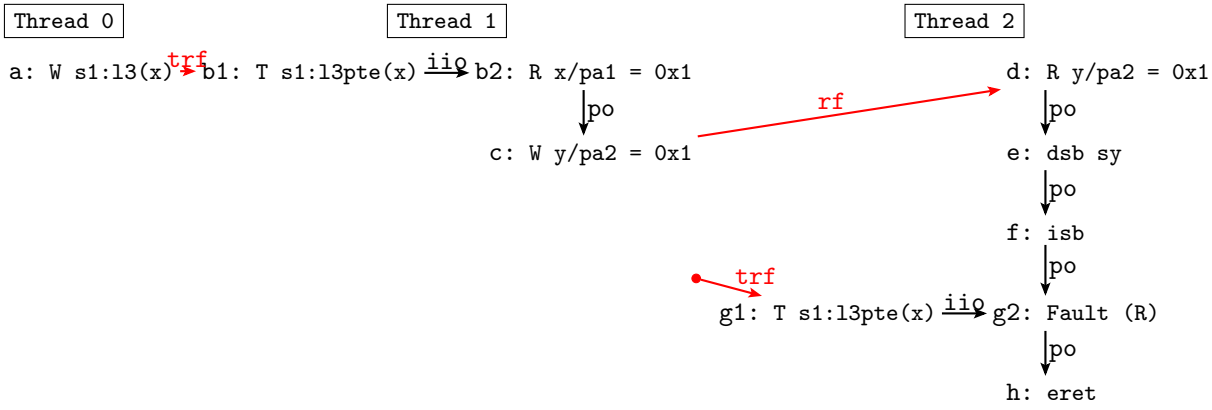
Misaligned accesses, and the load pair and store pair instructions, should generate explicit memory events and associated translations which are unordered with respect to each other.

WRC.TRTf.inv+po+dsb-isb AArch64

Initial state: x |-> invalid, z |-> pa1, *pa1 = 1, y |-> pa2, 0:X0=desc3(z), 0:X1=pte3(x), 1:X1=x, 1:X2=1, 1:X3=y, 2:X1=y, 2:X3=x				
Thread 0	Thread 1	Thread 2	Thread1 El1 Handler	Thread2 El1 Handler

STR X0,[X1]	LDR X0,[X1]
	STR X2,[X3]
		LDR X0,[X1]
		DSB SY
		ISB
		LDR X2,[X3]
			MOV X0,#0
			
			MRS X13,ELR_EL1
			ADD X13,X13,#4
			MSR ELR_EL1,X13
			ERET
				MOV X2,#0
				
				MRS X13,ELR_EL1
				ADD X13,X13,#4
				MSR ELR_EL1,X13
				ERET

Forbidden: 1:X0=1 & 2:X0=1 & 2:X2=0				



The translation-read of x (g1) is ordered after another translation-read of the same address x (b1), so by multi-copy-atomicity g1 may not read from an older write than b1 did.

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

WRC.RRTf.inv+dmb+dsb-isb AArch64

Initial state: x |-> invalid, z |-> pa1, *pa1 = 1, y |-> pa2, 0:X0=desc3(z), 0:X1=pte3(x), 1:X1=pte3(x), 1:X2=1, 1:X3=y, 2:X1=y, 2:X3=x			
Thread 0	Thread 1	Thread 2	Thread2 El1 Handler

STR X0,[X1]	LDR X0,[X1]
	DSB SY
	STR X2,[X3]
		LDR X0,[X1]
		DSB SY
		ISB
		LDR X2,[X3]
			MOV X2,#0
			
			MRS X13,ELR_EL1
			ADD X13,X13,#4
			MSR ELR_EL1,X13
			ERET

Forbidden: 1:X0=desc3(z) & 2:X0=1 & 2:X2=0			

The translation-read of x (h1) is ordered after the read of the translation table entry (b) and so by multi-copy-atomicity it cannot read from an older write than b did. The dsb-isb sequence in Thread 2 ensures the translation-table-walk of g is ordered after the program-order earlier read even without FEAT_ETS (see §8.4.3).

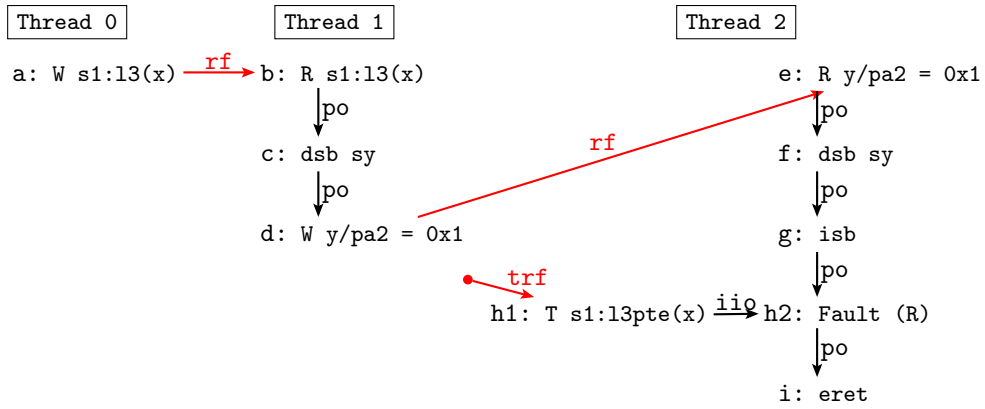


Figure 8.26: Test WRC.RRTf.inv+dmb+dsb-isb

WRC.TRR.inv+po+dsb AArch64

Initial state: x |-> invalid, y |-> pa2, *pa1 = 1, 0:X0=mkdesc3(oa=pa1), 0:X1=pte3(x), 1:X0=0, 1:X1=x, 1:X2=1, 1:X3=y, 2:X1=y, 2:X3=pte3(x)			
Thread 0	Thread 1	Thread 2	Thread1 El1 Handler

STR X0,[X1]	LDR X0,[X1]
	STR X2,[X3]
		LDR X0,[X1]
		DSB SY
		LDR X2,[X3]
			MRS X13,ELR_EL1
			ADD X13,X13,#4
			MSR ELR_EL1,X13
			ERET

Forbidden: 1:X0=1 & 2:X0=1 & 2:X2=0			

The read of the translation table entry for x (f) is ordered after the translation read of x (b1) and so by multi-copy-atomicity it cannot read from an older write than b1 did. The dsb in Thread 2 is sufficient to order the reads, any preserved read-to-read thread-local ordering suffices.

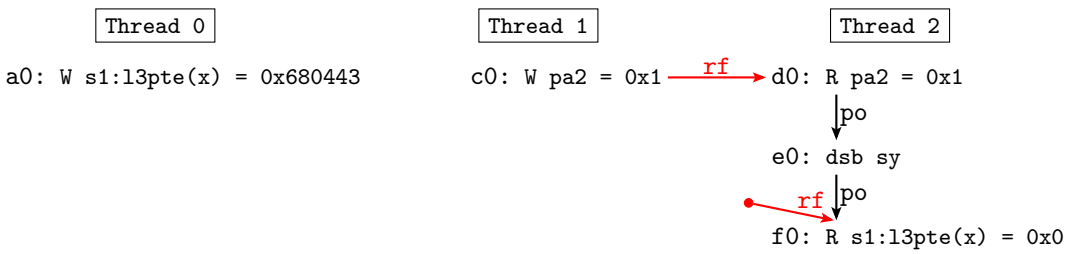


Figure 8.27: Test WRC.TRR.inv+po+dsb

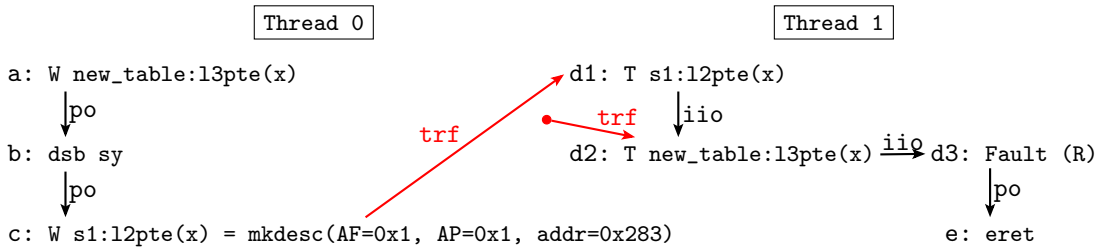
ROT.inv+dsb AArch64

Initial state: ipa1 |-> pa1, x |-> invalid at level 2, s1table new_table 0x280000 {, x |-> invalid, }, 0:X0=mkdesc3(oa=ipa1), 0:X1=pte3(x, new_table), 0:X2=mkdesc2(table=0x283000), 0:X3=pte2(x), 0:PSTATE.EL=1, 1:X1=x

Thread 0	Thread 1	Thread1 El1 Handler

STR X0,[X1]
DSB SY
STR X2,[X3]
	LDR X0,[X1]	// read ESR_EL1.ISS, to see if fault at Level 2 or 3.
		MRS X14,ESR_EL1
		AND X14,X14,#7
		CMP X14,#7
		MOV X17,#1
		MOV X18,#2
		// if ESR_EL1.ISS.DFSC == Translation Level 3 then x0 = 1 else x0 =
		CSEL X0,X17,X18,eq
		
		// advance ELR
		MRS X13,ELR_EL1
		ADD X13,X13,#4
		MSR ELR_EL1,X13
		
		// return
		ERET

Forbidden: 1:X0=1



The translation-table walk from the read of x in Thread 1 must perform its translation non-TLB reads, d1 and d2, 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: the location of the new table is fixed at page 280, so that it is not symbolic; the location of the level 3 table within the new tree will be in page 283, which is known from the fixed isla configuration of page tables. 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 is what the exception handler does in this test.

Figure 8.28: Test ROT.inv+dsb

CoWinvT+po AArch64

Initial state: x |-> pa1, 0:X0=0, 0:X1=pte3(x), 0:X3=x	
Thread 0	Thread0 El1 Handler

STR X0,[X1]
LDR X2,[X3]
	MOV X2,#1
	
	MRS X13,ELR_EL1
	ADD X13,X13,#4
	MSR ELR_EL1,X13
	ERET

Allowed: 0:X2 = 0	

The translation read (b1) of the last-level entry for x can be executed out-of-order with respect to the program-order earlier store (a) of an invalid entry to pte3(x).

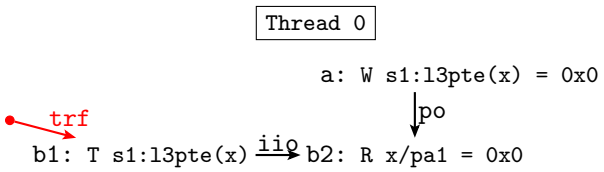


Figure 8.29: Test CoWinvT+po

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 executed out-of-order with respect to program order (§8.4.1), but even inserting thread-local ordering to the translation, such as in test CoWinvT+dsb-isb (Figure 8.30, p.143), does not forbid it.

CoWinvT+dsb-isb AArch64

Initial state: x |-> pa1, 0:X0=0, 0:X1=pte3(x), 0:X3=x	
Thread 0	Thread0 El1 Handler

STR X0,[X1]
DSB SY
ISB
LDR X2,[X3]
	MOV X2,#1
	
	MRS X20,ELR_EL1
	ADD X20,X20,#4
	MSR ELR_EL1,X20
	ERET

Allowed: 0:X2 = 0	

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

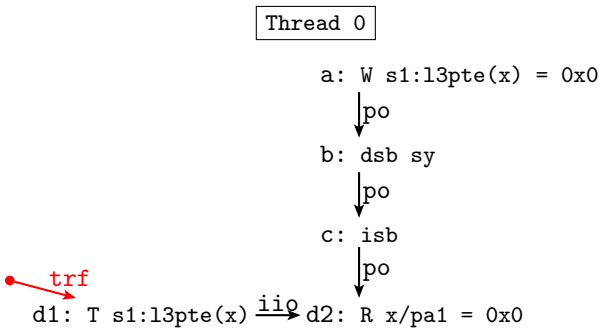


Figure 8.30: Test CoWinvT+dsb-isb

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 TLB 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, prefetch or speculatively read some address. These appear as spontaneous translation table walks. Those spontaneous walks may be cached. We can see this occurring in the following MP.RT.inv+poloc-dmb+ctrl-isb test (Figure 8.31, p.144), 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). 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 microTLBs

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.

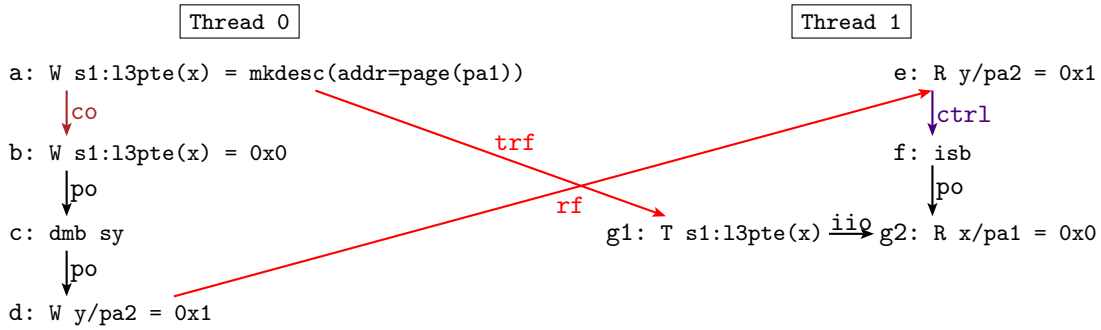
In effect, these microTLBs behave as if they were a larger non-deterministic TLB with potentially many values for each entry. The presence of these smaller caching structures in a superscalar machine means

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

Initial state: x |-> invalid, y |-> pa2, *pa1 = 0, *pa2 = 0, 0:X0=mkdesc3(oa=pa1), 0:X1=pte3(x), 0:X2=0, 0:X3=pte3(x), 0:X4=1, 0:X5=y, 1:X1=y, 1:X3=x		
Thread 0	Thread 1	Thread1 El1 Handler

STR X0,[X1]
STR X2,[X3]
DMB SY
STR X4,[X5]
	LDR X0,[X1]
	CBNZ X0,L0
	L0:
	ISB
	MOV X2,#1
	LDR X2,[X3]
		MRS X13,ELR_EL1
		ADD X13,X13,#4
		MSR ELR_EL1,X13
		ERET

Allowed: 1:X0 = 1 & 1:X2=0		



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 (g1) to see the old cached entry, even though the architectural translation table walk could not have happened while the valid entry was visible.

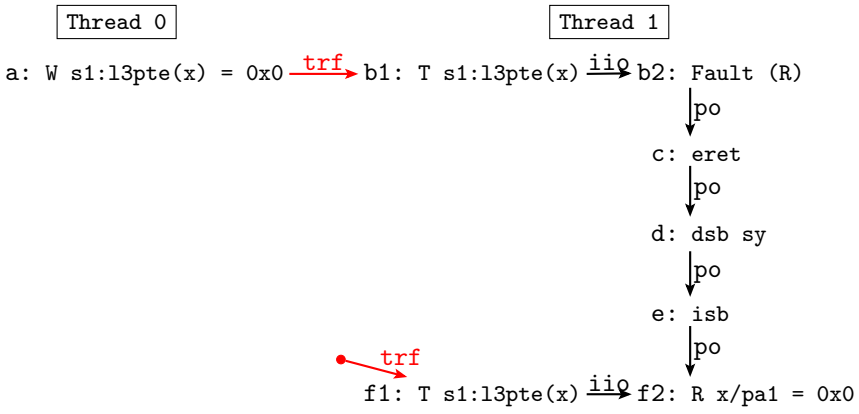
Figure 8.31: Test MP.RT.inv+poloc-dmb+ctrl-isb

CoTfT+dsb-isb AArch64

Initial state: x |-> pa1, y |-> pa1, *pa1 = 0, 0:X0=0, 0:X1=pte3(x), 1:X1=x, 1:X3=x		
Thread 0	Thread 1	Thread1 El1 Handler

STR X0,[X1]	LDR X2,[X1]
	MOV X0,X2
	DSB SY
	ISB
	LDR X2,[X3]
		MOV X2,#1
		
		MRS X13,ELR_EL1
		ADD X13,X13,#4
		MSR ELR_EL1,X13
		ERET

Allowed: 1:X0 = 1 & 1:X2=0		



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

Figure 8.32: Test CoTfT+dsb-isb

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), where the presence of these micro-TLBs (or other distributed caching structures) permits later events (even locally-ordered later) to see old cached entries 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.

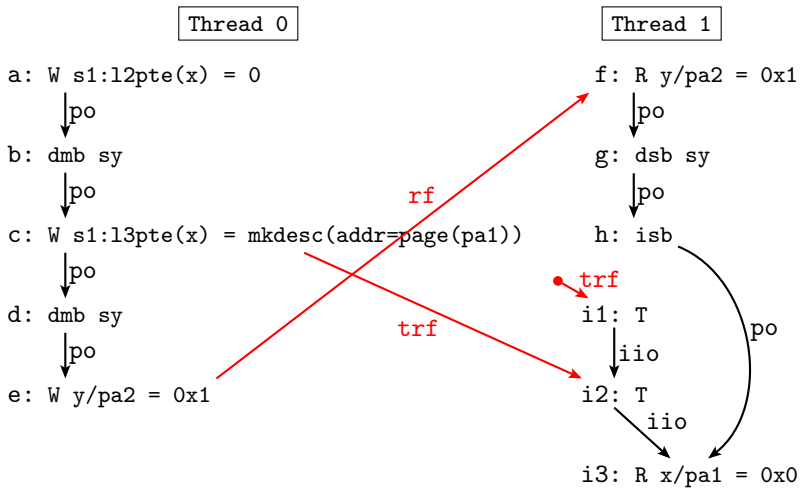
Caching up to last-level table The most common kind of caching structure we are aware of in microarchitecture is the walk cache (see §8.3.1). Traditionally, a TLB would store entire virtual to physical mappings, making it fast to lookup the translation (often a single cycle), but there was limited space, and this 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

MP.RTT.inv3+dmb-dmb+dsb-isb AArch64

Initial state: x |-> invalid, y |-> pa2, *pa1 = 0, *pa2 = 0, 0:X0=0, 0:X1=pte2(x), 0:X2=mkdesc3(oa=pa1), 0:X3=pte3(x), 0:X4=1, 0:X5=y, 1:X1=y, 1:X3=x		
Thread 0	Thread 1	Thread1 El1 Handler

STR X0,[X1]
DMB SY
STR X2,[X3]
DMB SY
STR X4,[X5]
	LDR X0,[X1]
	DSB SY
	ISB
	MOV X2,#1
	LDR X2,[X3]
		MRS X13,ELR_EL1
		ADD X13,X13,#4
		MSR ELR_EL1,X13
		ERET

Allowed: 1:X0 = 1 & 1:X2=0		



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. In the test, x is initially invalid at level 3, and x and y have different level 2 entries (by ensuring they are not in the same 2 MiB region), and writes zero to the level 2 entry for x (a) and then overwrites the previously-zero level 3 entry to point to pa1, such that the final read of x could only see a valid entry if the walk read-from the new level 3 entry, but a stale cached level 2 entry. The magic numbers are concrete instantiations from `isla-axiomatic`'s symbolic evaluation.

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

(Figure 8.33, p.146), where a walk cache allows the table entry to be cached separately from the last-level entry, allowing the last translation read to read from a much newer value.

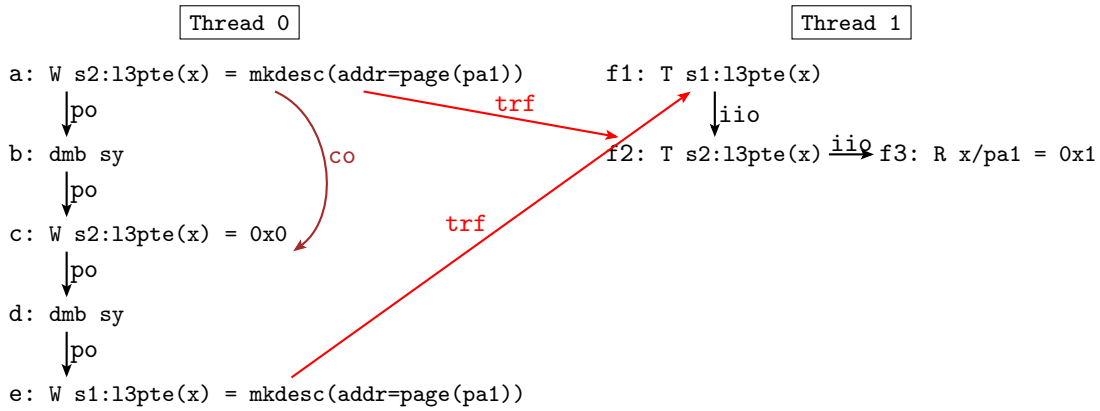
Independent caching of IPAs In a two-stage regime, the virtual addresses are first translated into intermediate physical address. The secondary translations based on the intermediate physical addresses, either of the final output address or of any of the intermediate table addresses, may be cached in the TLB without remembering the originating virtual address. This means that 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. This is most clear in ROT.invs1+dmb2 (Figure 8.34, p.147), 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 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

ROT.invs1+dmb2 AArch64

Initial state: x |-> invalid at level 2, ipa1 |-> pa1, *pa1 = 1, 0:X0=mkdesc3(oa=pa1), 0:X1=pte3(x, s2_page_table_base), 0:X2=0, 0:X3=pte3(x, s2_page_table_base), 0:X4=mkdesc3(oa=ipa1), 0:X5=pte3(x), 0:PSTATE.EL=1, 1:X1=x			
Thread 0	Thread 1	Thread1 El1 Handler	Thread1 El2 Handler
STR X0,[X1] DMB SY STR X2,[X3] DMB SY STR X4,[X5]	MOV X0,#0 LDR X0,[X1]	MRS X13,ELR_EL1 ADD X13,X13,#4 MSR ELR_EL1,X13 ERET	MRS X13,ELR_EL2 ADD X13,X13,#4 MSR ELR_EL2,X13 ERET
Allowed: 1:X0=1			

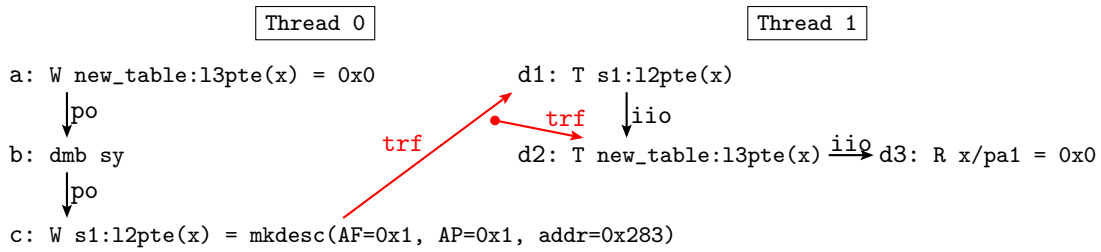


The translation read of the stage 2 leaf entry for x (f2) can read from an old cached version, from the write (a) even though it was not reachable by any translation table walk for any VA (as the IPA it maps was not mapped by any stage 1 tables before it was overwritten by (b)). This test relies on translation table walks being naturally ordered (by iio), see §8.4.8.

Figure 8.34: Test ROT.invs1+dmb2

ROT.inv2+dmb AArch64

Initial state: ipa1 |-> pa1, x |-> invalid at level 2, s1table new_table 0x280000 {, x |-> ipa1, }, 0:X0=0, 0:X1=pte3(x, new_table), 0:X2=mkdesc2(table=0x283000), 0:X3=pte2(x), 0:PSTATE.EL=1, 1:X1=x		
Thread 0	Thread 1	Thread1 El1 Handler
STR X0,[X1] DMB SY STR X2,[X3]	MOV X0,#1 LDR X0,[X1]	MRS X13,ELR_EL1 ADD X13,X13,#4 MSR ELR_EL1,X13 ERET
Allowed: 1:X0=0		



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 §8.8.1 for a discussion on this.

Figure 8.35: Test ROT.inv2+dmb

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 is derived from that entry. See §8.6.4 for more discussion on how this affects requirements on software.

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 we show just one of them here, ROT.inv2+dmb (Figure 8.35), where the last-level entry came from a newer value than the previous levels.

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') family of instructions.

TLB maintenance generally performs 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 the more subtle effect of discarding any partially executed instructions, on other cores, which had already begun execution using a stale entry but had not yet finished executing. We now explore both of these effects, and the subtle interaction with other parts of the VMSA, in more detail.

CoWinvT.EL1+dsb-tlbi-dsb-isb AArch64

Initial state: x |-> pa1, 0:X0=0, 0:X1=pte3(x), 0:X3=x, 0:X5=page(x), 0:PSTATE.EL=1	
Thread 0	Thread0 El1 Handler

STR X0,[X1]
DSB SY
TLBI VAE1,X5
DSB SY
ISB
LDR X2,[X3]
	MOV X2,#1
	
	MRS X13,ELR_EL1
	ADD X13,X13,#4
	MSR ELR_EL1,X13
	ERET

Forbidden: 0:X2 = 0	

The translation-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 or above, so this test starts execution at EL1 rather than the usual default of EL0.

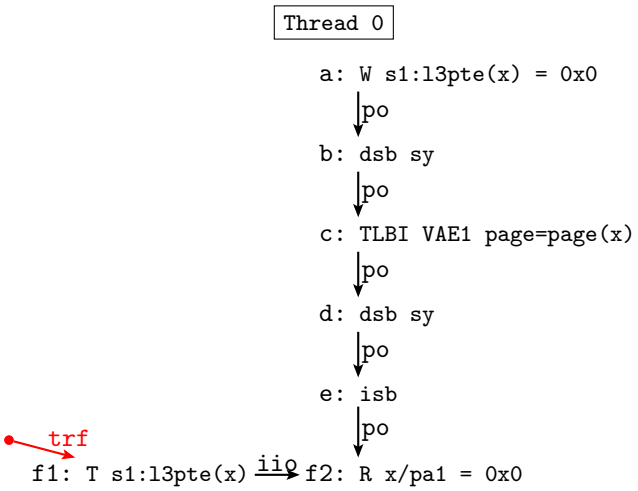


Figure 8.36: Test CoWinvT.EL1+dsb-tlbi-dsb-isb

8.6.1 Recovering coherence

We saw in §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, p.143). By inserting the correct TLBI sequence into that test, we produce a new test, CoWinvT.EL1+dsb-tlbi-dsb-isb (Figure 8.36, p.149), 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: 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 occurring. For example, in the aforementioned CoWinvT.EL1+dsb-tlbi-dsb-isb test (Figure 8.36), if the TLBI had the wrong page then it would have no effect and the test would remain allowed.

FEAT_nTLBPA

Armv8.4-A introduced a new optional Arm feature, FEAT_nTLBPA [12, A2.2.1 (p79)].

This feature adds a field to the memory model feature register (AA64MMFR1_EL1) which identifies whether the current processor's TLB 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.

CoWinvT.EL1+tlbi-dsb-isb AArch64

Initial state: x |-> pa1, 0:X0=0, 0:X1=pte3(x), 0:X3=x, 0:X5=page(x), 0:PSTATE.EL=1	
Thread 0	Thread0 El1 Handler

STR X0,[X1]
TLBI VAE1,X5
DSB SY
ISB
LDR X2,[X3]
	MOV X2,#1
	
	MRS X13,ELR_EL1
	ADD X13,X13,#4
	MSR ELR_EL1,X13
	ERET

Final state: 0:X2 = 0	

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

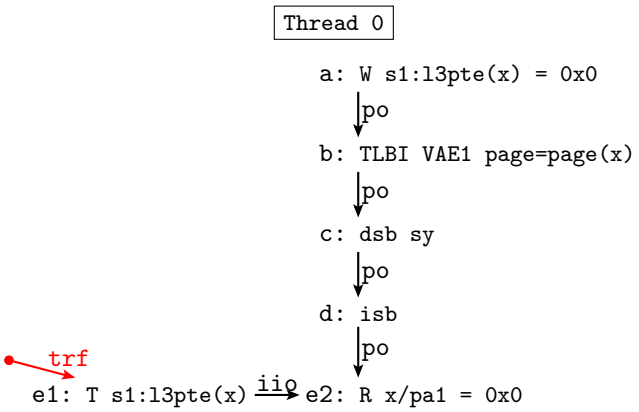


Figure 8.37: Test CoWinvT.EL1+tlbi-dsb-isb

This change adds a field to the register, whose reserved value in Armv8.0 corresponds to non-coherent caches existing. This implies that in processors without FEAT_nTLBPA, one should assume that TLBs may contain non-coherent caching structures, including prior to the introduction of the FEAT_nTLBPA feature entirely: it is not clear to us whether this is intentional. Therefore, some behaviours described here may assume a setting that is too strong, erroneously assuming all non-TLB translation-reads read from the coherent-latest write. Our experimental data did not include any devices with incoherent non-TLB reads.

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 executed out-of-order with respect to other instructions. To ensure they are synchronized with other instructions, the programmer can use the DSB barrier instruction to impose order on the instructions before and after it.

Leaving out one or both of the DSBs around the TLBI leads to insufficient ordering around the TLBI, and allows the invalidation to occur at the wrong time. For example, the CoWinvT.EL1+tlbi-dsb-isb test (Figure 8.37), a version of Figure 8.36 without the DSB b, is allowed as the initial write and TLBI may be re-ordered, negating the architectural effect of the TLBI.

8.6.3 Broadcast

Arm provide broadcast variants of the TLBI instructions. These are generally suffixed with the letters IS ('Inner-shareable') in the mnemonic. Broadcast TLBIs, sometimes referred to as TLB *shootdowns*, allow one processor to perform maintenance on other cores' TLBs. This is in contrast to other systems, such as for x86, and IBM's Power architecture, where maintenance of other cores must be achieved in software through the use of purely thread-local invalidation instructions.

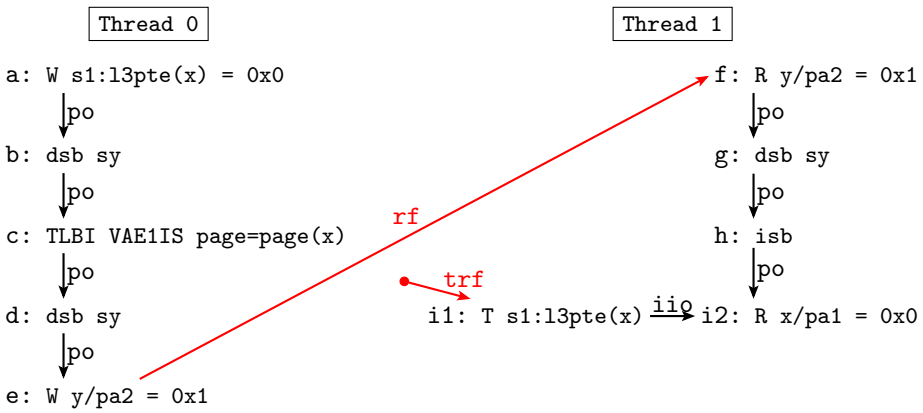
TLB invalidation on another core One of the simplest examples of multi-core invalidation is a message passing invalidation pattern, where the old entry is removed, and a message is sent to another core. This can be seen in the MP.RT.EL1+dsb-tlbiis-dsb+dsb-isb test (Figure 8.38, p.151).

MP.RT.EL1+dsb-tlbiis-dsb+dsb-isb AArch64

Initial state: x |-> pa1, y |-> pa2, 0:X0=0, 0:X1=pte3(x), 0:X2=1, 0:X3=y, 0:X4=page(x), 0:PSTATE.EL=1, 1:X1=y, 1:X3=x		
Thread 0	Thread 1	Thread1 El1 Handler

STR X0,[X1]
DSB SY
TLBI VAE1IS,X4
DSB SY
STR X2,[X3]
	LDR X0,[X1]
	DSB SY
	ISB
	LDR X2,[X3]
		MOV X2,#1
		
		MRS X13,ELR_EL1
		ADD X13,X13,#4
		MSR ELR_EL1,X13
		ERET

Forbidden: 1:X0 = 1 & 1:X2=0		



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.

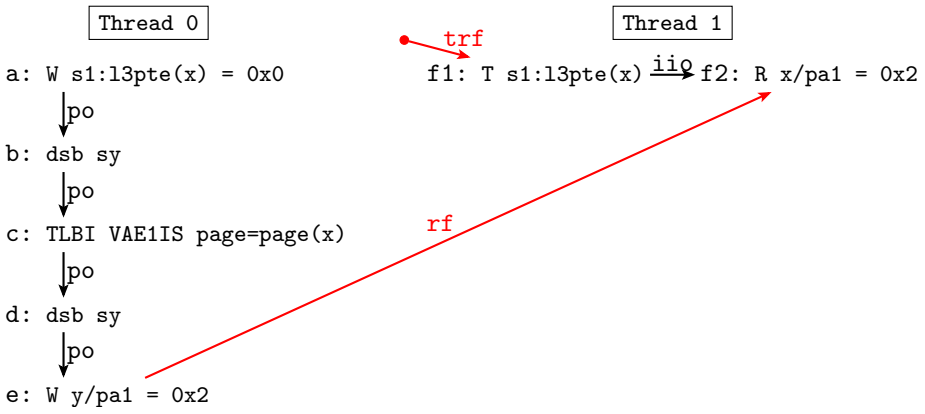
Figure 8.38: Test MP.RT.EL1+dsb-tlbiis-dsb+dsb-isb

RBS+dsb-tlbiis-dsb AArch64

Initial state: x |-> pa1, y |-> pa1, *pa1 = 0, 0:X0=0, 0:X1=pte3(x), 0:X5=page(x), 0:X2=2, 0:X3=y, 0:PSTATE.EL=1, 1:X1=x		
Thread 0	Thread 1	Thread1 El1 Handler

STR X0,[X1]
DSB SY
TLBI VAE1IS,X5
DSB SY
STR X2,[X3]
	LDR X0,[X1]	MOV X0,#1
		
		MRS X13,ELR_EL1
		ADD X13,X13,#4
		MSR ELR_EL1,X13
		ERET

Forbidden: 1:X0 = 2		



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

Figure 8.39: Test RBS+dsb-tlbiis-dsb

Instruction restarts Broadcast TLBIs must do more than touch the other thread's TLB. If another processor has an in-flight instruction which has started but not yet finished execution using a stale translation, then that instruction must be restarted.

This ensures that Arm broadcast TLBIs have the same behaviour as the traditional software IPI-based shootdown (with context synchronization). It 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 any more.

This guarantee is captured in the RBS+dsb-tlbiis-dsb (Figure 8.39) (**Read-Broken-Secret**) test. Once a mapping has been *broken*, and sufficient TLB maintenance performed, any future reads or writes to the original physical location will not be visible through that mapping any more. Note, however, whether this means that instructions which have already completed their execution must be restarted, even if they occur after an earlier restarted instruction, is still under investigation (c.f. RBS+dsb-tlbiis-dsb+poloc test (Figure 8.40, p.153)).

While here we describe things in terms of instruction restarting, these behaviours can be (and presumably sometimes 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.

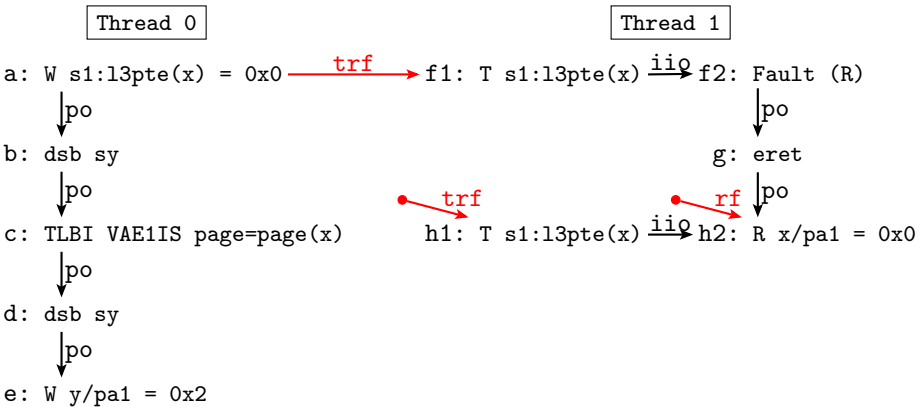
Atomic TLBIs In previous tests, we describe behaviour in terms of writes that occur 'before' or 'after' a TLBI. Microarchitecturally, a TLBI instruction may be non-atomic: it sends messages to all other cores, performs some action on those cores, and sends messages back to the originating core. Program-order-earlier DSBs ensure that instructions program-order-earlier than the DSB are complete before sending the messages. Program-order-later DSBs ensure that all instructions program-order-after the DSB wait for those messages to return before executing.

RBS+dsb-tlbiis-dsb+poloc AArch64

Initial state: x |-> pa1, y |-> pa1, *pa1 = 0, 0:X0=0, 0:X1=pte3(x), 0:X5=page(x), 0:X2=2, 0:X3=y, 0:PSTATE.EL=1, 1:X1=x, 1:X3=x		
Thread 0	Thread 1	Thread1 El1 Handler

STR X0,[X1]
DSB SY
TLBI VAE1IS,X5
DSB SY
STR X2,[X3]
	MOV X0,#1
	LDR X0,[X1]
	MOV X2,#1
	LDR X2,[X3]
		MRS X13,ELR_EL1
		ADD X13,X13,#4
		MSR ELR_EL1,X13
		ERET

Final state: 1:X0 = 1 & 1:X2 = 0		



Even though the broadcast TLBI on Thread 0 (c) ensures that not-yet-completed instructions using the old mapping are restarted, it does not necessarily require that the second load of x in Thread 1 (h) be restarted if it has already satisfied its value, as that value came from a write before the TLBI. Note there is no allowed/forbidden state, as the architectural intent for this test is unknown, requiring further investigation..

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

By ensuring that between any TLBI and potential same-thread access is a DSB ensures that the TLBI's effect happens entirely between the execution of those instructions. 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, as long as the programmer takes care to maintain the required thread-local ordering. This allows us to simplify the model, treating TLBIs as having a single serialisation point at which the invalidation effect happens globally.

8.6.4 Virtualization

Throughout this section, we have considered tests for a single-stage translation with virtual mappings. However, many of these questions and behaviours also apply to the second-stage of a two-stage mapping with intermediate physical addresses, with only a few differences. We now explore how adding a second stage of translation affects the behaviours discussed here.

Virtual to physical and IPA caches The existence of TLBs that cache virtual to physical mappings (§8.5.4) complicates TLB maintenance requirements 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 to invalidate any of the old cached IPA to PA entries; then, perhaps surprisingly, a second invalidation to remove any stale cached end-to-end translations, comprising whole VA to PA mappings (or combinations), as these could have indirectly cached the result of a second stage translation, without remembering the IPA they went through.

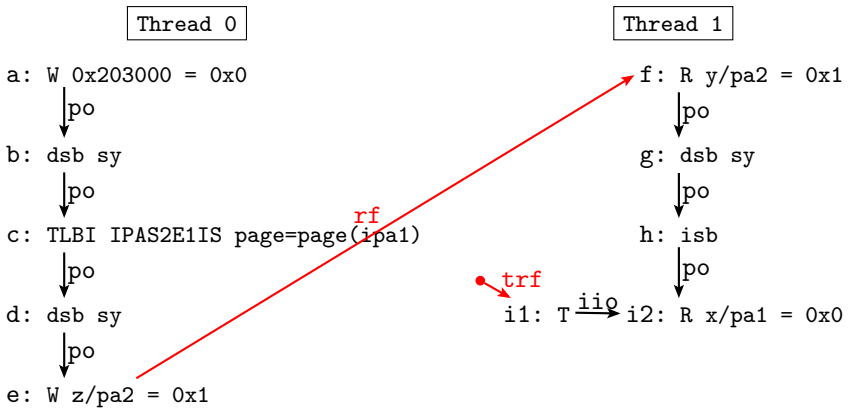
This is illustrated in MP.RT.EL2+dsb-tlbiipais-dsb+dsb-isb (Figure 8.41, p.155), where invalidation of *just* the IPA is not enough to forbid the relaxed behaviour. Adding an invalidation of the VA (or all VAs), like in MP.RT.EL2+dsb-tlbiipais-dsb-tlbiis-dsb+dsb-isb (Figure 8.42, p.156), ensures that later translations cannot see the stale value any more. Note that the invalidations must happen in the specified order, as otherwise the TLB could be immediately refilled from the earlier cached second-stage entries.

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

Initial state: x |-> ipa1, ipa1 |-> pa1, y |-> ipa2, ipa2 |-> pa2, z |-> pa2, *pa1 = 0, *pa2 = 0, 0:X0=0, 0:X1=pte3(ipa1, s2_page_table_base), 0:X2=1, 0:X3=z, 0:X4=page(ipa1), 0:PSTATE.EL=2, 1:X1=y, 1:X3=x		
Thread 0	Thread 1	Thread1 El2 Handler

STR X0,[X1]
DSB SY
TLBI IPAS2E1IS,X4
DSB SY
STR X2,[X3]
	LDR X0,[X1]
	DSB SY
	ISB
	MOV X2,#1
	LDR X2,[X3]
		MRS X13,ELR_EL2
		ADD X13,X13,#4
		MSR ELR_EL2,X13
		ERET

Forbidden if ETS1:X0=1 & 1:X2=0		



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

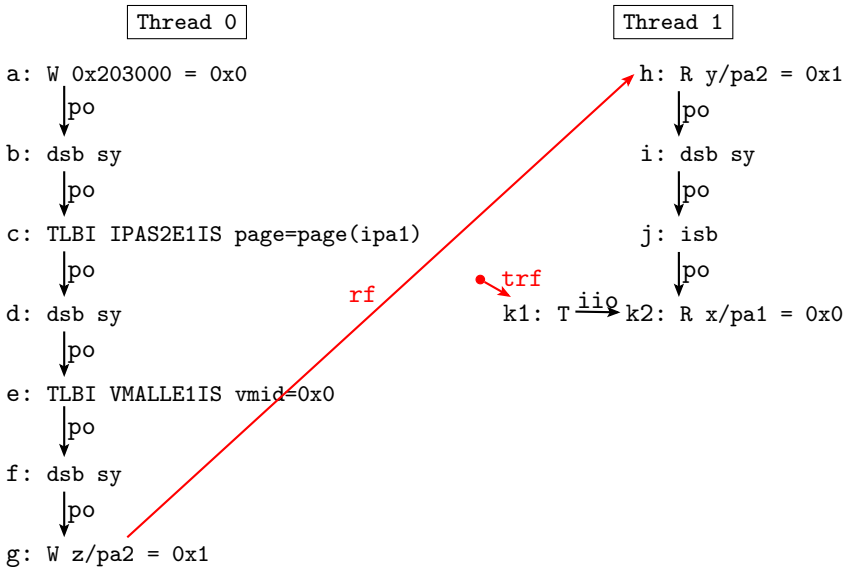
Figure 8.41: Test MP.RT.EL2+dsb-tlbiipais-dsb+dsb-isb

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

Initial state: x |-> ipa1, ipa1 |-> pa1, y |-> ipa2, ipa2 |-> pa2, z |-> pa2, *pa1 = 0, *pa2 = 0, 0:X0=0, 0:X1=pte3(ipa1, s2_page_table_base), 0:X2=1, 0:X3=z, 0:X4=page(ipa1), 0:PSTATE.EL=2, 1:X1=y, 1:X3=x		
Thread 0	Thread 1	Thread1 El2 Handler

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]
		MOV X2,#1
		
		MRS X13,ELR_EL2
		ADD X13,X13,#4
		MSR ELR_EL2,X13
		ERET

Forbidden: 1:X0=1 & 1:X2=0		



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 any more.

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

8.6.5 Break-before-make

TLBs are not required to store only a single cached translation for a given address. There may, in general, be multiple valid translations cached in the TLB. In some cases this is perfectly fine, e.g. for translations which differ only in the permissions. However, if the translations conflict (see 'Conflicting entries') then having those translations both in the TLB simultaneously would be dangerous and may lead to unpredictable behaviour (see 'Violating break-before-make'). To avoid this possibility, the architecture provides a *break-before-make* sequence, which will ensure that there cannot be two cached conflicting translations existing in the TLB at the same time.

Conflicting entries Simply having multiple entries cached at once is not, necessarily, dangerous. Instead, the architecture specifies when such combinations are *conflicting*: 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 cacheability or shareability.
- ▷ A change in block size (e.g. replacing a page of 4KiB leaf with a 2MiB block mapping).

The break-before-make process When updating a translation table entry to a new conflicting entry, the architecture specifies break-before-make is required, and the programmer must:

(1) write an invalid entry to overwrite the currently valid translation table entry in memory (*break* the translation);
(2) perform any TLB maintenance required to sufficiently invalidate the old entry from any TLB(s) required;
(3) write the new valid translation table entry, overwriting the old invalid entry (*make* the new translation).

This sequence must be well-ordered: each step must have been fully completed before the next; in particular, the write of invalid must be visible to the MMU before performing TLB maintenance, and that TLB maintenance must have completed before making the new entry. In practice, this means the sequence requires `DSB SY` barriers (or equivalent) before and after any `TLBI` instructions.

Litmus test For completeness, the BBM+dsb-tlbiis-dsb (Figure 8.43, p.158) gives a simple valid-to-valid concurrent update test. The point is not whether a particular relaxed outcome is allowed, but that the test does does not give rise to unpredictable behaviour.

Violating break-before-make Architecturally, reaching a state where there is a pair of conflicting entries in the TLB leads to a degraded state, defined by ConstrainedUnpredictable behaviour. The only way to avoid this is to use the appropriate break-before-make sequence. The Arm reference manual states that failure to perform break-before-make, when it is 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.

In this work we do not try to give a characterisation of the ConstrainedUnpredictable behaviour arising from TLB conflicts. Understanding unpredictable behaviours in full is left to future work, but a quick summary might be 'any behaviour that an arbitrary program could have performed'. That is, an instantaneous 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.

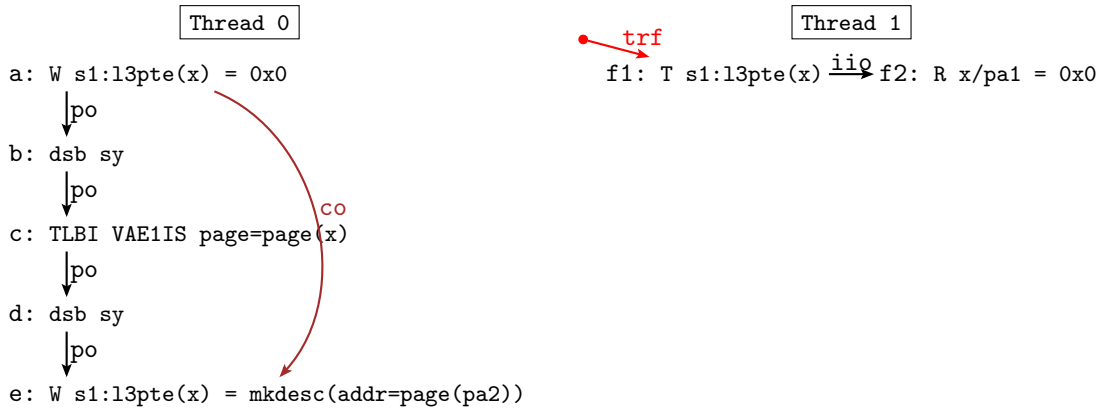
[1] See the Arm ARM 'TLB maintenance requirements and the TLB maintenance instructions' [12, D5.10.1 (p4913)] for the full list of conditions.

BBM+dsb-tlbiis-dsb AArch64

Initial state: x |-> pa1, *pa2 = 2, 0:X0=0, 0:X1=pte3(x), 0:X2=mkdesc3(oa=pa2), 0:X4=1, 0:X6=page(x), 0:PSTATE.EL=1, 1:X1=x		
Thread 0	Thread 1	Thread1 El1 Handler

STR X0,[X1]
DSB SY
TLBI VAE1IS,X6
DSB SY
STR X2,[X1]
	LDR X0,[X1]	MOV X0,#1
		
		MRS X13,ELR_EL1
		ADD X13,X13,#4
		MSR ELR_EL1,X13
		ERET

Allowed: 1:X0=0		



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* a new mapping for x (e). This ensures the concurrent access in Thread 1 is guaranteed to see either the old value, the intermediate broken page (and so a page fault), or the new value. This test is the variant whose final state asserts that the old value was read.

Figure 8.43: Test BBM+dsb-tlbiis-dsb

8.6.6 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: read, write, or execute.

Accesses which result in permission faults result in exceptions, much like translation faults do, but, unlike those, may have been read from the TLB. This can clearly be seen in the CoWinvTp.ro+dsb-isb test (Figure 8.44, p.160), where a permission fault comes from an entry that must have read from the TLB.

Multiple cached entries We can observe multiple cached entries within a TLB by modifying the access permissions of an entry. It is not architecturally required to perform break-before-make when the two entries differ only in permissions, and it is permitted for the TLB to cache them both.

When reading from the TLB where there existing multiple entries for the same input address, it is allowed for the hardware to generate a *TLB conflict abort*.

If the hardware does not generate a conflict abort, then translation reads of that address are ConstrainedUnpredictable as described earlier. However, when there is no requirement for break-before-make, the constraints are tighter: translations are nondeterministically able to read one or the other, (or an 'amalgamation') of the values [12, K1.2.3 (p11243)].

We can avoid the question of 'amalgamation' 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 what amalgamations of entries are allowed. This can be seen with the MP.RTpT.ro+dmb-dmb+dsb-isb-dsb-isb test (Figure 8.45, p.161), where the existence of multiple cached entries in the TLB allows multiple translation-reads to read from different stale writes.

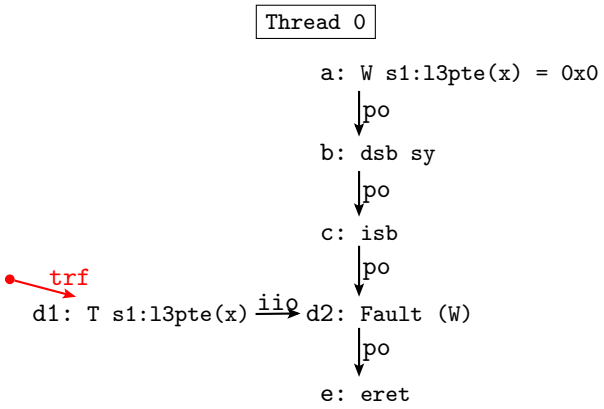
Atomic TLB reads The presence of multiple cached translation table entries in the TLB introduces the question of whether those TLB fills and subsequence TLB reads must read from entire single-copy atomic writes of the original translation table entries (much like a read of memory would) or whether a translation read can read from a mix of different writes. RMD+dmb (Figure 8.46, p.162) ('Read-mixed-descriptor') shows that translation reads cannot partially read from a write: they must read from the entire write or none of it.

CoWinvTp.ro+dsb-isb AArch64

Initial state: x |-> pa1 with [AP = 3] and default, *pa1 = 0, 0:X0=0, 0:X1=pte3(x), 0:X2=1, 0:X3=x	
Thread 0	Thread0 El1 Handler

STR X0,[X1]
DSB SY
ISB
MOV X13,#0
STR X2,[X3]
	// read ESR_EL1.ISS to see if Permission or Translation fau
	MRS X14,ESR_EL1
	AND X14,X14,#0b1111
	CMP X14,#0b1111
	MOV X15,#1 // Permission
	MOV X16,#2 // Translation
	CSEL X13,X15,X16,eq
	MRS X20,ELR_EL1
	ADD X20,X20,#4
	MSR ELR_EL1,X20
	ERET

Allowed: 0:X13=1	



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

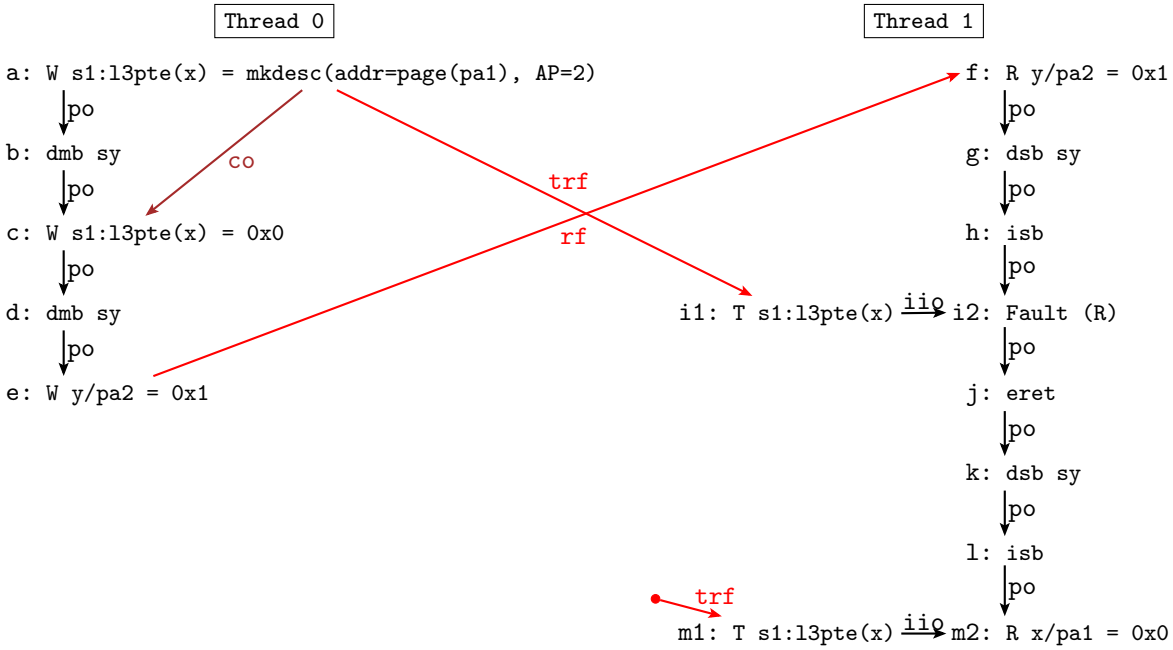
Figure 8.44: Test CoWinvTp.ro+dsb-isb

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

Initial state: x |-> pa1 with [AP = 3] and default, y |-> pa2, *pa1 = 0,
0:X0=mkdesc3(oa=pa1, AP=2), 0:X1=pte3(x), 0:X2=0, 0:X3=pte3(x), 0:X4=1, 0:X5=y,
1:X1=y, 1:X4=x

Thread 0	Thread 1	Thread1 El1 Handler
STR X0,[X1]	LDR X0,[X1]	// read ESR_EL1.ISS to see if Permission or Translation fau
DMB SY	DSB SY	MRS X14,ESR_EL1
STR X2,[X3]	ISB	AND X14,X14,#0b1111
DMB SY	LDR X13,[X4]	CMP X14,#0b1111
STR X4,[X5]	MOV X2,X13	MOV X15,#1 // Permission
	DSB SY	MOV X16,#2 // Translation
	ISB	CSEL X13,X15,X16,eq
	LDR X13,[X4]	
	MOV X3,X13	MRS X20,ELR_EL1
		ADD X20,X20,#4
		MSR ELR_EL1,X20
		ERET

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

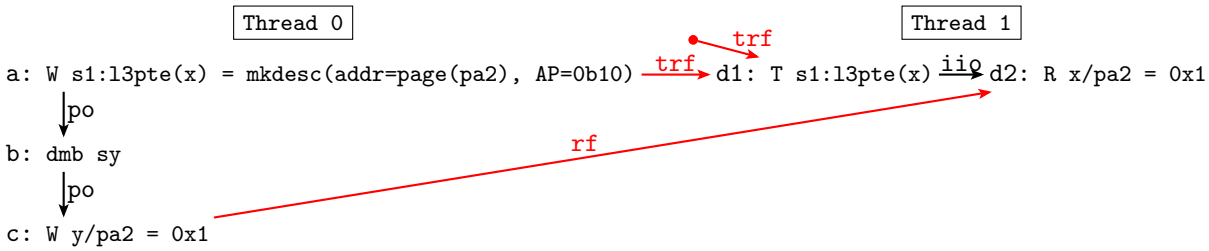


The first translation-read of x (i1) reads from the write that removes read permissions (a) and this write must have come from the TLB because of the intervening invalidation (c), message pass (e-f), and dsb; isb sequence (g-h). The later translation-read of x (m1) can still see an even older value with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

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

RMD+dmb AArch64

Initial state: x |-> pa1 with [AP = 3] and default, y |-> pa2, *pa1 = 0, *pa2 = 0, 0:X0=mkdesc3(oa=pa2, AP=2), 0:X1=pte3(x), 0:X2=1, 0:X3=y, 1:X1=x		
Thread 0	Thread 1	Thread1 El1 Handler
STR X0,[X1] DMB SY STR X2,[X3]	MOV X0,#0 LDR X0,[X1]	MRS X20,ELR_EL1 ADD X20,X20,#4 MSR ELR_EL1,X20 ERET
Forbidden: 1:X0=1		



The translation-read of x (d1) cannot read from both the 64-bit single-copy atomic write 'a' and 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. isla-axiomatic cannot generate such candidates, so the execution diagram shown is hand-written.

Figure 8.46: Test RMD+dmb

8.7 Context synchronisation

There are many operations which change the current system context. We focus 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 the variety of other system configuration state used by the translation table walker.

8.7.1 Relaxed system registers

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

Arm System registers may require the programmer to insert explicit synchronization, as stated 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 must ensure a *Context synchronization* event occurs. Context synchronisation ensures that future instructions are guaranteed to see updates to the context made by program-order-earlier instructions. Microarchitecturally, these can be achieved e.g. by flushing the pipeline, causing future instructions to restart. Some context synchronising operations have already been encountered: the `ISB` instruction, and taking and returning from exceptions.

There are two important considerations: (1) this does not apply to non-System registers, such as the 'Special-purpose' registers, which never require synchronization; and (2) the synchronization required for System registers depends on the *kind* of access.

There are two kinds of accesses to System registers: direct and indirect. Direct accesses are the typical way programmers interact with registers: instructions which explicitly refer to the name in its mnemonic. Indirect accesses happen when an instruction which does not explicitly mention the register by name nevertheless performs an access to it, implicitly during its execution.

Out-of-order execution means 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 A naive interpretation of the relaxed semantics is to allow these reads to read-from the most recent indirect write and any program-order-earlier direct writes since the last context synchronization event.

However, this does not give the correct behaviour. The Arm ASL was not written in a way to accommodate relaxed system register behaviours: sometimes it re-reads the same system register multiple times, or it gets all the fields of a register in one read, or it re-uses the same previously read system register value in multiple places. This leaves open questions about whether these registers can be redundantly 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.

The model we present in the next chapter gives a simple, incomplete and possibly unsound, semantics of system registers with respect to a *pointed set* of writes (see §9.1) which allows the model to permit 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 such TLB-cacheable register: the `MAIR` register. Direct writes to the `MAIR` may fail to be seen by program-order-later translations, even after context synchronization, as the translations may get their value from a stale value in the TLB which was computed using the old `MAIR`. To ensure that an update to a TLB-cacheable register is observed by program-order-later translations, both TLB maintenance and context synchronization are required, in that order.

The registers which can be cached in this way, and the behaviours that arise from this caching, are currently under investigation with Arm.

8.8 Problems

This section describes some in-progress work with Thibaut Pérami.

Some questions, and problems, have arisen after publication of the model in the next section. These fall into two main categories:

1. when a memory location should be considered a pagetable entry by the model (Reachability); and
2. invalidations of block or table entries (Wide invalidations).

8.8.1 Reachability

One important property that the TLB must have is that it may only add new cached translations for translation table entries which are *reachable* by 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 translation table base and other System register state. This means that writes which are coherence-before the most recent write, at the time a translation table entry location becomes reachable, are not visible to the walker, and therefore cannot have been cached in any TLB.

This is captured in the RUE+isb (Figure 8.47, p.165) ('Read-unreachable-entry') test, which is architecturally 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. Currently, the models do not attempt to track reachability, and so erroneously allow this test.

8.8.2 Wide invalidations

In §8.6, we discussed invalidations of entries in the TLB, and investigated how `TLBI` instructions remove cached translations which translate a given page. This raises an important question: does the invalidation apply only to that page, or to all translations mapped by the same translation table entry? On one hand, TLBs can split such 'wide' translations into multiple smaller paged-sized ones, e.g. for when the stage 2 mapping is at a smaller granularity. On the other hand, this would require software to do a very expensive invalidation to clear cached block entries, either iterating over every page in the region, or simply flushing the entire TLB.

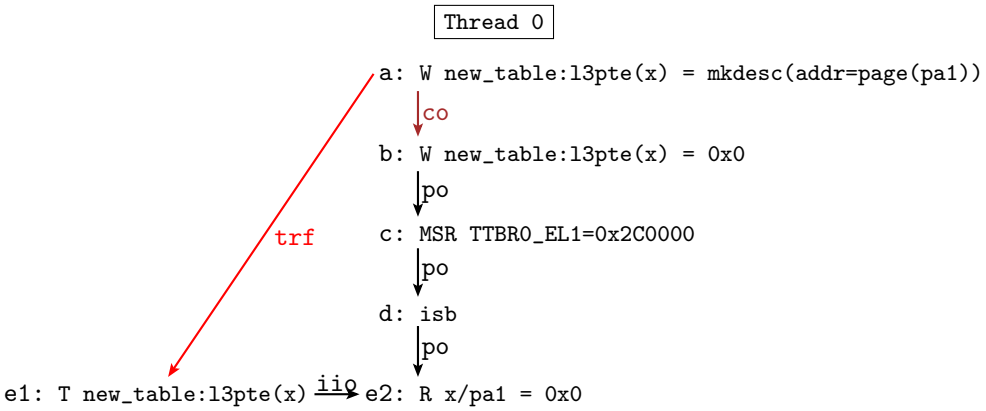
Arm have, tentatively, decided that the architectural intent is that the TLB invalidations should invalidate all mappings which use the same cached translation table entry, see InvalidateWideBlock (Figure 8.48, p.166). However, this does not apply when the original mappings were of smaller granularity. For example, even if writing an invalid entry at level 2 then doing invalidation, the old level 3 entries may still be cached in the TLB, illustrated in InvalidateWide (Figure 8.49, p.167).

RUE+isb AArch64

Initial state: *pa1 = 0, s1table new_table 0x2C0000 {, x |-> invalid, }, 0:X0=mkdesc3(oa=pa1), 0:X1=0, 0:X2=pte3(x, new_table), 0:X3=ttbr(asid=1, base=new_table), 0:X4=x, 0:PSTATE.EL=1, 0:PSTATE.SP=1	
Thread 0	Thread1 El1 Handler

STR X0,[X2]
STR X1,[X2]
MSR TTBR0_EL1,X3
ISB
MOV X1,#1
LDR X3,[X4]
	MRS X13,ELR_EL1
	ADD X13,X13,#4
	MSR ELR_EL1,X13
	ERET

Final state: 0:X1=1	



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-axiomatic currently does not do any kind of reachability analysis, and so does not forbid this test.

Figure 8.47: Test RUE+isb

InvalidateWideBlock AArch64

Initial state: aligned 2097152 virtual x, x |-> pa1 at level 2, 0:X1=pte2(x), 0:X3=x, 0:X4=page(x), 0:PSTATE.EL=1, 0:PSTATE.SP=1	
Thread 0	Thread0 El1 Handler

MOV X2, #0
STR X2, [X1]
DSB SY
TLBI VAE1, X4
DSB SY
ISB
LDR X6, [X3, #0x1000]
	MOV X6,#1
	
	MRS X13,ELR_EL1
	ADD X13,X13,#4
	MSR ELR_EL1,X13
	ERET

Forbidden: 0:X6 = 0	

x is mapped by a 2 MiB block entry at level 2. Breaking it and invalidating the TLB passing x affects all translations in the same 2 MiB block.

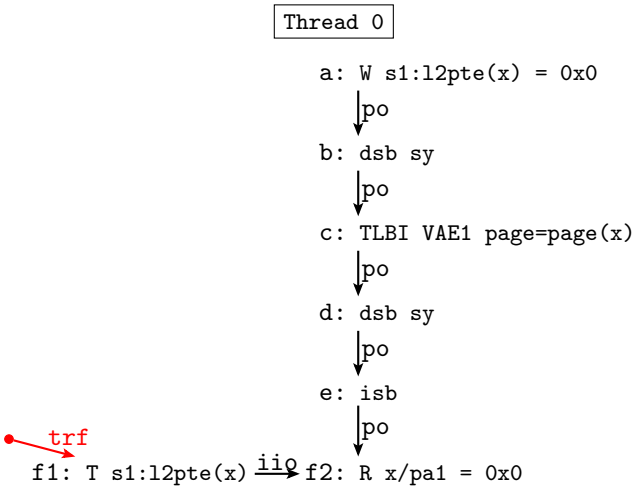


Figure 8.48: Test InvalidateWideBlock

InvalidateWide	AArch64
Initial state: *pa1 = 1, *pa2 = 2, x |-> pa1, z |-> pa2, 0:X1=pte2(x), 0:X3=x, 0:X4=page(x), 0:X5=z, 0:PSTATE.EL=1, 0:PSTATE.SP=1	
Thread 0	Thread0 El1 Handler

MOV X2, #0
STR X2, [X1]
DSB SY
TLBI VAE1, X4
DSB SY
ISB
LDR X6, [X5]
	MOV X6,#0
	
	MRS X13,ELR_EL1
	ADD X13,X13,#4
	MSR ELR_EL1,X13
	ERET

Allowed: 0:X6 = 2	

x and z are two entries mapped at level 3 but within the same 2 MiB region, so are mapped by the same level 2 entry. Breaking the level 2 entry and invalidating with one address does not invalidate the other.

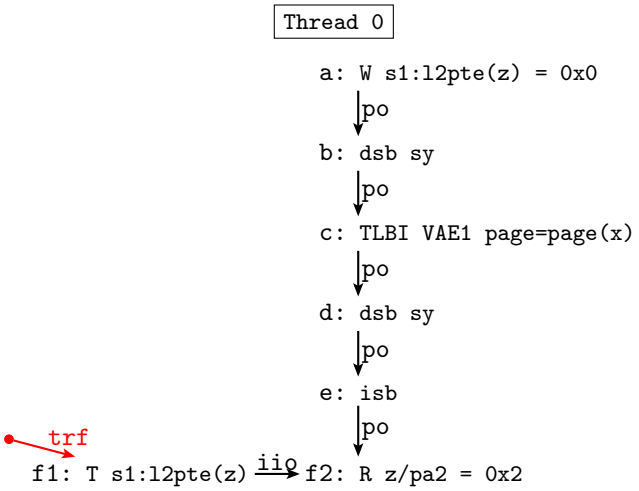


Figure 8.49: Test InvalidateWide

8.9 Contributions

We have now covered all the key relaxed virtual memory behaviours, and will in the next chapter move on to discuss the model which captures 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 it is a manually compiled and curated list of behaviours, there will be corner cases missed and software patterns overlooked. That said, we believe we have covered those patterns which are known 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 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 investigation by Arm.

The first major category are those behaviours which were already clear and 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 involved category, of things that either had some associated reference materials, or was otherwise clear from discussion with architects, but for which further investigation 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, some of which are currently under investigation by Arm. These include further questions about the scope of `TLBI`s, interaction with exceptions and interrupts, changes in cacheability, translations for instruction fetching, and relaxed system register accesses. Those areas will require more work before giving a concrete semantics.

8.10 Related work

The authoritative Arm-internal ASL model [10, 11, 72], and the Sail model derived from it [45] cover address translation, 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. [85, 98] describe an ACL2 model for much of x86 that covers address translation; and the Forvis [99] and RISCV-PLV [100] Haskell RISC-V ISA models are also complete enough to boot Linux.

Syeda and Klein [101, 102] provide a somewhat idealised model for ARMv7 address translation and TLB maintenance.

Komodo [58] uses a handwritten model for a small part of ARMv7, as do Guanciale et al. [59, 60]. Romanescu et al. [103, 104] discuss address translation in the concurrent setting, but with respect to idealised models.

Lustig et al. [87] 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. Hossain [105] continues this line, producing an estimated architectural model for x86, using their *TransForm* framework to automatically synthesise litmus tests from the memory model used in validation.

Tao et al. [106] define a weak virtual memory model, based on what they call wDRF (weak data race freedom), which gives bounds to software when it behaves in accordance with a set of conditions, which they use in the verification of their *SeKVM* hypervisor. These conditions permit some races to the page table code, but do not attempt to capture the full architectural envelope. The model here, we hope, gives a sound underlying base from which one could prove a wDRF-SC property above, connecting the work of Tao to our model.

Independent work by Arm, after the conclusion of this work, further extended the herdtools suite of tools, models, and tests, for some overlapping portion of the VMSA, including covering dirty bits and access flags which we do not consider here [107]. They differ from the model presented here where the architecture has changed (around ETS and write forwarding), and in one other known place: ordering around TLB maintenance with respect to program-order-earlier accesses (see their `CopyOnWrite` test). Further work is required to understand how else the models differ in their intersection and how a model that covers their union could be constructed.

Chapter 9

An axiomatic VMSA model

We now define a semantic model for Arm-A relaxed virtual memory (*RVM*) that, to the best of our knowledge, captures the Arm architectural intent for the questions discussed in Chapter 8, including two-stage translation-table walks and the required TLB maintenance, as an extension to the base usermode Arm-A axiomatic memory model [7] (as recalled in Chapter 2).

In Chapter 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 those specific litmus-test examples and give a general definition of what behaviour is allowed or not. We do this by extending the base model, defining ordering between new translation-read events and other events, without modelling TLBs or out-of-order pipelines directly.

The base Arm 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. We now extend these candidates with new events and new relations over those events, and modify some original relations.

9.1 Candidate events

We extend the events of the candidate executions and the corresponding labelling function, given in Figure 9.1, to contain new events:

- ▷ `T` for the implicit reads of memory originating from architected translation-table walks. These roughly correspond to the point the translation-read was satisfaction from memory, which with TLBs may happen very early[1].
- ▷ `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, annotated with the reason for the exception (not shown here).
- ▷ `MSR` events for writes to relevant system registers, such as the `TTBR`s; and `MRS` for reads.
- ▷ `DSB` events for `DSB` instructions.

Implicit accesses and faults Execution of the translation in the Arm architectural pseudocode performs reads of memory, which would normally generate `R` events in the candidate executions. Instead, when those reads happen during calls to that function, we label them as `T` events. This means that each translation table walk may generate up to 24 `T` events, before the instruction generates the `R` or `W` event. We explored alternative representations, including collecting all reads into a single large translation event, or placing all translations into the standard `R` set. These options have advantages, but we made the choice to keep a 1-to-1 correspondence between the events of the execution and the ISA, and to retain as much of the original 2018 model events and relations unchanged as possible.

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 data memory read events and modelling load buffering, write gathering, and caches.

[1] Analogously to `R` events and load-buffers.

$$
\begin{aligned}
\text{Label} &\equiv \text{Reads} \cup \text{Writes} \cup \text{Barriers} \cup \text{Translations} \cup \text{TLBIs} \cup \text{Exceptions} \cup \text{SysRegs} \\
\text{Reads} &\equiv \{\mathbf{R}, \mathbf{A}, \mathbf{Q}\} \times \text{Loc} \times \text{Val} \\
\text{Writes} &\equiv \{\mathbf{W}, \mathbf{L}\} \times \text{Loc} \times \text{Val} \\
\text{Barriers} &\equiv \{\mathbf{DMB.LD}, \mathbf{DMB.ST}, \mathbf{DMB.SY}, \mathbf{DSB.SY}, \mathbf{ISB}\} \\
\text{Translations} &\equiv \{\mathbf{T}\} \times \text{PA} \times \text{TranslationInfo} \\
\text{TLBIs} &\equiv \{\mathbf{TLBI}\} \times \text{TLBIOp} \times \text{Shareability} \times \text{Regime} \times \text{VMID?} \times \text{ASID?} \times \text{Addr?} \\
\text{Exceptions} &\equiv \{\mathbf{TE}\} \times \text{ExceptionInfo} \cup \{\mathbf{ERET}\} \\
\text{SysRegs} &\equiv \{\mathbf{MSR}, \mathbf{MRS}\} \times \text{SysRegName} \times \text{Val} \\
\text{VA}, \text{IPA} \equiv \text{Addr} &\equiv \text{Bitvec}_{48} \\
\text{Loc} \equiv \text{PA} &\equiv \text{Bitvec}_{64} \\
\text{Val} &\equiv \text{Bitvec}_{64} \\
\text{TranslationInfo} &\equiv \text{VA} \times \text{IPA?} \times \text{Level} \times \text{Stage} \\
\text{TLBIOp} &\equiv \{\mathbf{VA}, \mathbf{IPA}, \mathbf{ALL}, \mathbf{ASID}, \mathbf{VMALL}, \ldots\} \\
\text{ASID}, \text{VMID} &\equiv \text{Bitvec}_{8} \\
\text{Regime} &\equiv \{\mathbf{EL1\&0}, \mathbf{EL2}\} \\
\text{Shareability} &\equiv \{\mathbf{NSH}, \mathbf{ISH}\} \\
\text{SysRegName} &\equiv \{\mathbf{TTBR0\_EL1}, \mathbf{TTBR0\_EL2}, \mathbf{VTTBR\_EL2}, \ldots\} \\
\text{ExceptionInfo} &\equiv \ldots
\end{aligned}
$$

where T? *signifies an optional field of type* T.

Figure 9.1: Definition of candidate event labels for Arm-A RVM candidates. Parts which differ from the original definition are highlighted in blue. Terminal words of the label are given in **bold**.

We add a helper set, `T_f`, for translation reads which read-from a write whose 'valid' 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 the discussion on `obETS` (§9.4.6) for more explanation 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, containing only explicit reads and writes performed by instructions.

TLBIs As described in §7.7, Arm have a variety of `TLBI` instructions, with varying arguments. Each `TLBI` instruction generates a single `TLBI` event, each with a distinct label. To aid in modelling, there are a set of subsets of `TLBI` for various kinds of `TLBI`:

- ▷ `TLBI-S1` for invalidations of Stage 1 entries.
- ▷ `TLBI-S2` for invalidations of Stage 2 entries.
- ▷ `TLBI-IPA` for invalidations by intermediate physical address.
- ▷ `TLBI-VA` for invalidations by virtual address.
- ▷ `TLBI-ASID` for invalidations by ASID.
- ▷ `TLBI-VMID` for invalidations by VMID.
- ▷ `TLBI-ALL` for the `TLBI ALL` instructions.
- ▷ `TLBI-IS` for broadcast `TLBI`s.
- ▷ `TLBI-EL1` for invalidations of the EL1&0 regime.
- ▷ `TLBI-EL2` for invalidations of the EL2 regime.

These do not *cut* the `TLBI` set into partitions. Rather any `TLBI` event may belong to multiple. For example, a `TLBI VAE1IS` event would belong to `TLBI-VA`, `TLBI-VMID`, `TLBI-EL1`, and `TLBI-IS`.

We also include all `TLBI`s in a general `C` ('Cache maintenance') set.

```
let dsbsy = DSBISH | DSBSY | DSBNSH
let dsbst = dsbsy | DSBST | DSBISHST | DSBNSHST
let dsbld = dsbsy | DSBLD | DSBISHLD | DSBNSHLD
let dsbnsh = DSBNSH
let dmbsy = dsbsy | DMBSY
let dmbst = dmbsy | dsbst | DMBST
          | DSBST | DSBISHST | DSBNSHST
let dmbld = dmbsy | DMBLD
          | dsbld | DSBISHLD | DSBNSHLD
let dmb = dmbsy | dmbst | dmbld
let dsb = dsbsy | dsbst | dsbld
```

Figure 9.2: Barrier helper sets.

Exceptions Despite not modelling exceptions in general in this part, 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 behaviours from crossing exception level boundaries.

To support this, we add two new events: `TE` ('Take exception'); and `ERET` ('Exception return').

Barriers The Arm `DSB` ('Data synchronization barrier') instruction is required for TLB maintenance, as was seen in the previous chapter. We include `DSB` events, one for each kind of `DSB` instruction:

- ▷ `DSBSY` and `DSBISH` (which we treat as equivalent, as we do not model shareability domains).
- ▷ `DSBNSH`, for non-shareable (thread-local) `DSB`s.
- ▷ `DSBST`, `DSBLD`, for `DSB`s with `ST` or `LD` kinds.
- ▷ `DSBISHST`, `DSBISHLD`, and so on, for all combinations of `DSB` instruction domain and access types.

Arm define a hierarchy of barriers where, for example:

$$\texttt{DMB.LD} < \texttt{DMB.SY} < \texttt{DSB.SY}$$

That is, any ordering imposed by a `DMB.LD` is also imposed by a `DMB.SY`, and therefore also a `DSB.SY`.

To avoid an explosion in the number of relations as we add the new barrier events, we simplify and update the barrier-ordered-before relation in the Arm model to use a collection of helper sets, which encode this hierarchy. Those helper sets can be found in Figure 9.2.

Context changing and synchronisation Finally, we add events for context-changing and context-synchronising operations. Context changes are updates to system registers which change the current translation regime, which are generated as `MSR` events We add a general context-synchronisation event set `CSE` which includes `ISB`, `TE`, and `ERET`.

Changes to system registers may have relaxed behaviours, as described in §8.7.1, but full relaxation of the system register reads done by the Arm pseudocode 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 pointed set as the new pointed element. 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 same 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.

9.2 Candidate relations

We extend the set of candidate relations and the witness to include the new events: adding the explicit events to program-order, updating address dependencies to take translation into account, and adding new relations to capture intra-instruction ordering and ordering to/from translation-reads and TLBIs. The updated candidate relations are given in Figure 9.3.

Addresses, ASIDs, and VMIDs Each translation table walk will read from general-purpose and system registers to get a value for the input address, the current ASID, current VMID, and the roots of the translation tables. We then relate each `T` with any other `T` where the translation associated with it is for the same virtual 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 `same-*-page` relations, which relate two events when their virtual, intermediate physical, or physical addresses, are in the same page.

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) matches the `T` event's associated translation. 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 add an extended coherence order `wco`, which is an arbitrary linearisation of writes, `DSB` barriers, and cache and TLB maintenance operations, consistent with the usual coherence order. This generalised coherence order captures a global 'commit' order of these operations, consistent with what a hypothetical microarchitectural-style operational semantics would generate.

One might be concerned at the validity of doing this, for two reasons. First, this generalised coherence order will relate all writes, not just same-location ones. However, extending coherence to a total order over all locations is sound [6, §10.5 p174], so this does not cause an issue. Secondly, it enforces a kind of atomicity of `TLBI`s. For broadcast `TLBI`s, microarchitectures 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 §8.6.6, we are able to consider `TLBI` instructions as executing 'atomically', so long as there are no break-before-make violations. This is a similar justification as for including `DC` and `IC` events in a similar generalised coherence order for instruction fetching [35, §5 p29].

The full definition of `wco`, as defined in `isla-axiomatic`, can be found in Figure 9.8, p.183.

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 a special `tdata` (described below) to these.

The intra-instruction-order relation (`iio`) relates two events from the same instruction in the order the intra-instruction semantics 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 investigating 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, with address translation we can now define address dependencies 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 any translation which reads from the register written by that read to compute the address. The traditional `addr` can then be recovered as `tdata; iio*; [M]`.

The Arm-A RVM pre-execution relations are:

- ▷ intra-instruction-order: E_1 `iio` E_2 for events E_1, E_2 in the same instruction where E_1 is generated before E_2 in the intra-instruction trace.
- ▷ program order: E_1 `po` E_2 for explicit events E_1, E_2 such that 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 physical location as used by M_2.
- ▷ same-address (`same-va`, `same-ipa`, `same-pa`): between events E_1, E_2 iff the (virtual/intermediate physical/physical)-address of E_1 is the same as E_2.
- ▷ same-page (`same-va-page`, `same-ipa-page`, `same-pa-page`): between events E_1, E_2 iff the (virtual/intermediate physical/physical) address of E_1 is in the same page (e.g. 4KiB chunk) as E_2.
- ▷ same-address-space (`same-asid`, `same-vmid`): between events E_1, E_2 iff the associated translation of E_1 and E_2 are using the same ASID or VMID.
- ▷ address dependent: R_1 `tdata` T_2 iff the value read by R_1 is used in the calculation of the address which T_2 is a translation of.
- ▷ 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.
- ▷ translation-reads-from (`trf`), from W_1 to T_2 when T_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).

where E_n represents events of any kind, M_n is an explicit memory effect event, T_n is a translation-read event, R_n is a read event, and W_n is a write event.

Figure 9.3: Definition of the candidate relations and witness for Arm-A RVM candidates. Parts which differ from the base definition are highlighted in blue.

9.3 Axioms

The RVM model axioms are, mostly, a syntactic extension to the original Arm-A axiomatic model presented in Chapter 2. This is by design. Although there may be other nicer or more succinct ways of phrasing the model, the variation presented here is designed to be as syntactically 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 equivalence of the axiomatic models in the presence of constant address translation, increasing the confidence we have in the model.

Figure 9.4 contains the axioms and relations for our Arm-A relaxed virtual memory axiomatic model, defined in Cat. Unchanged parts from the original are greyed out. We elide some helper relations, which we will describe in more detail later.

The model has three 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 (* Internal visibility requirement *)
2 acyclic po-loc | fr | co | rf as internal
3
4 (* Writes cannot forward to po-future translates *)
5 acyclic (po-pa | trfi) as translation-internal
```

Unchanged from the original, the `internal` axiom captures the SC-per-location guarantee. 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 even guarantee coherence, the only behaviour that 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 relation (called `ob`, 'ordered-before', in Arm) 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
4 (* External visibility requirement *)
5 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 is 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, although one expects the intra-instruction semantics defined by the ASL already describes the behaviour in enough detail.

```
1 (* Atomic requirement *)
2 empty rmw & (fre; coe) as atomic
```

```
let speculative =
  ctrl
| addr; po
| [T]; instruction-order

(* translation-ordered-before *)
let tob =
  [T_f]; tfre
| [T]; iio; [R|W]; po; [W]
| speculative; trfi

(* observed by *)
let obs =
  rfe | fr | wco
| trfe

(* ordered-before TLBI and translate *)
let obtlbi_translate =
  [T&Stage1]; tlb_barriered; [TLBI-S1]
| ([T&Stage2]; tlb_barriered; [TLBI-S2])
  &
  (same-translation; [T&Stage1]
   ; trf^-1; wco^-1)
| ([T&Stage2]; tlb_barriered; [TLBI-S2]
   ; wco?; [TLBI-S1])
  &
  (same-translation; [T&Stage1]
   ; maybe_TLB_cached)

(* ordered-before TLBI *)
let obtlbi =
  obtlbi_translate
| [R|W|Fault_T]; iio^-1; [T]
  ; (obtlbi_translate & ext); [TLBI]

(* context-change ordered-before *)
let ctxob =
  speculative; [MSR]
| [CSE]; instruction-order
| [ContextChange]; po; [CSE]
| speculative; [CSE]
| po; [ERET]; instruction-order; [T]

(* ordered-before a fault *)
let obfault =
  data; [FaultFromW]
| speculative; [FaultFromW]
| [dmbst]; po; [FaultFromW]
| [dmbld]; po; [FaultFromW|FaultFromR]
| [A|Q]; po; [FaultFromW|FaultFromR]
| [R|W]; po; [FaultFromReleaseW]

(* ETS-ordered-before *)
let obETS =
  (obfault; [Fault_T]); iio^-1; [T_f]
| ([TLBI]; po; [dsb]
   ; instruction-order; [T])
  & tlb-affects

(* dependency-ordered-before *)
let dob =
  addr | data
| speculative; [W]
| addr; po; [W]
| (addr | data); rfi
| (addr | data); trfi

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

(* barrier-ordered-before *)
let bob =
  [R]; po; [dmbld]
| [W]; po; [dmbst]
| [dmbst]; po; [W]
| [dmbld]; po; [R|W]
| [L]; po; [A]
| [A|Q]; po; [R|W]
| [R|W]; po; [L]
| [F|C]; po; [dsbsy]
| [dsb]; po

(* Ordered-before *)
let ob =
  (obs | dob | aob | bob
| iio | tob | ctxob
| obtlbi | obfault | obETS)^+

(* Internal visibility requirement *)
acyclic po-loc | fr | co | rf as internal
(* External visibility requirement *)
irreflexive ob as external
(* Atomic requirement *)
empty rmw & (fre; coe) as atomic
(* Writes cannot forward to po-future
    translates *)
acyclic (po-pa | trfi)
  as translation-internal
```

Figure 9.4: RVM axioms and relations

9.4 Relations

The RVM model modifies some of the original ordering relations, and introduces some new ones. This section goes through each in detail, describing the mechanisms, and justifying the existence or non-existence of particular clauses.

9.4.1 Observed-by

```
1  (* observed by *)
2  let obs = rfe | fr | wco | trfe
```

The 'observed-by' relation includes the original `rf` and `fr` (over physical locations), the '*generalised coherence order*' (`wco`, §9.2), and the translation-reads-from-external (`trfe`) relation.

Generalised coherence Including `wco`, which is existentially quantified over the candidates, fixes some global order in which the writes and `TLBI`s happen in. Consider, informally, some microarchitectural execution. Writes would be propagated to the coherent storage subsystem, and `TLBI`s would be completed, and these would be interleaved within some global-time whole-machine trace. The generalised `wco` relation captures a serialisation of these events. The model is then quantified over all such orderings, accounting for any interleaving of these events.

External translation reads Inclusion of `trfe` enforces that those 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 out-of-order execution (§8.4.1) or TLB caching (§8.5.1). So we do not include the translation analogue of the usual from-reads-external relation in `ob`, `tfre` (translation-from-reads-external), which relates translation-reads with writes coherence-after the ones they read from.

Additionally, writes may be forwarded to its thread's translation walker: translation-reads may be satisfied before the propagation of the write to the coherent storage subsystem (§8.4.4). Therefore we do not include the translation analogue of reads-from-internal in `ob`, `trfi` (translation-reads-from-internal), which relates writes with same-thread translation-reads which read from that write.

9.4.2 Dependency-ordered-before

```
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 forbid thin-air creation of values (§8.4.1, §8.4.2), much like the analogous constraint in the usermode model.

9.4.3 Barrier-ordered-before

```
let bob =
  [R]; po; [dmbld]
| [W]; po; [dmbst]
| [dmbst]; po; [W]
| [dmbld]; po; [R|W]
| [L]; po; [A]
| [A | Q]; po; [R | W]
| [R | W]; po; [L]
| [F | C]; po; [dsbsy]
| [dsb]; po
```

We rewrite the original barrier-ordered-before relation to use the barrier helpers defined in Figure 9.2. The first seven clauses of `bob` capture the same ordering as in the user model, but capturing the barrier hierarchy: imposing the same barrier orderings when using stronger barriers (namely, `DSB`s in place of `DMB`s). We use helper sets `F` for all fences (barriers), and `C` for cache operations (including `TLBI`s).

The Arm `DSB` instruction does have additional ordering over a `DMB`, and these are captured by the two new clauses of `bob` above. First, a `DSB SY` orders `TLBI` instructions and other barriers and cache operations (§8.6.2). Second, all program-order later events must wait for an earlier `DSB` to finish before performing their explicit memory events.

9.4.4 Translation-ordered-before

```
let tob =
  [T_f]; tfre
| [T]; iio; [R|W]; po; [W]
| speculative; trfi
```

Translation table walks themselves impose ordering on the surrounding events, in three possible ways:

- ▷ coherence of translation-reads of invalid entries;
- ▷ might-be-same-address for program-order-later accesses; and
- ▷ non-forwarding of the speculative writes.

Invalid writes Reads of invalid entries must not have come from the TLB (§8.3.3). Therefore, for a translation fault, the translation-read of the invalid entry must have come from the coherence-latest write from memory at the time the translation happened. This is what the first clause of `tob` captures: that any translation-reads which read an invalid entry must happen before any writes coherence after the one it read from.

There is a careful subtlety here: we cannot simply include all of `tfr` after a translation-read of invalid, as a thread-local write may be forwarded to the translation table walker before it has propagated to memory (§8.4.4).

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, second, 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 relation, `speculative`:

```
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 then include `speculative` to same-thread translation-reads-from (`trfi`) in `ob`, forbidding forwarding of still-speculative writes to translation table walks. For now, we are unable to give a precise bound on the ordering for thread-local forwarding, and this area is still currently under investigation with Arm, including potentially being strengthened to forbid this entirely.

Might-be-same-address Finally, we include `[T]; iio; [M]; po; [W]`, which captures that writes cannot propagate until program-order-earlier instructions have determined their physical addresses (and so will not fault). Although this edge is subsumed by the `speculative; [W]` edge in `dob`, it is kept here for clarity.

9.4.5 Contextually-ordered-before

```
1  let ctxob =
2    speculative; [MSR]
3  | [CSE]; instruction-order
4  | [ContextChange]; po; [CSE]
5  | speculative; [CSE]
6  | po; [ERET]; instruction-order; [T]
```

The contextually-ordered-before relation, `ctxob`, captures the orderings required from context-changing and context-synchronising operations, without trying to capture the full extent of the relaxed behaviours. See Part III for a more detailed discussion of the ordering from context-synchronisation on an exception. The `ctxob` relation here is an approximation: capturing the orderings we know the architecture must provide, but perhaps not exploring the full architectural envelope of possible behaviours to its full extent.

Speculation The first guarantee we see is that context changes and synchronisation should not happen speculatively. Speculative context changes may create translation table roots and associated translation table walks from unreachable writes, creating thin-air problems (§8.8.1). To prevent this, we ensure that context-changing operations only happen once they are non-speculative, by enforcing `speculative; [MSR]` in `ob`. Forbidding the speculative execution of context-synchronising operations is achieved by the inclusion of `speculative; [CSE]` in `ob`.

Context synchronising Context-synchronising events (such as from `ISB` and `ERET` instructions) guarantee that program-order-earlier context-changing events are seen by program-order-later instructions. Microarchitecturally, context synchronisation can be achieved by simply 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]` in `ob`. 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). However, translation reads of a lower exception level should not satisfy during execution at a higher exception level. We over-approximate this by including `po; [ERET]; instruction-order; [T]` in `ob`, ensuring all translation-reads after an `ERET` wait.

9.4.6 Fault-ordered-before and ETS

Note: ETS is subject to change, as noted in §8.4.3, p.133.

```
(* ordered-before a fault *)
let obfault =
  data; [FaultFromW]
| speculative; [FaultFromW]
| [dmbst]; po; [FaultFromW]
| [dmbld]; po; [FaultFromW|FaultFromR]
| [A|Q]; po; [FaultFromW|FaultFromR]
| [R|W]; po; [FaultFromReleaseW]

(* ETS-ordered-before *)
let obETS =
  (obfault; [Fault_T]); iio^-1; [T_f]
| ([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 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, tagged 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 is then the syntactic subset of local ordering in ob, where the right-hand side of each clause is substituted with a Fault_T (a translation fault). We use this to construct an obETS relation, whose first clause adds to ob exactly this ordering to the translation read of the invalid entry, as architected by FEAT_ETS.

Note that dependencies and orderings *from* a faulting instruction are not required to be respected, and so we do not induce orderings from a Fault_T.

FEAT_ETS and TLBI The second clause of obETS captures a second architected behaviour of FEAT_ETS: faults after 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 TLBI-ordered-before

```
(* ordered-before TLBI *)
let obtlbi =
  obtlbi_translate
| [R|W|Fault_T]; iio^-1; (obtlbi_translate & ext); [TLBI]
```

Finally, there is the obtlbi relation, which captures the ordering between translations, their explicit memory events, and the TLB invalidations which affect them. The relation is split in two: the first clause enforces order between stale translations and the TLBIs they are invalidated by; the second clause imposes additional ordering on the intra-instruction-later explicit events, caused by broadcast TLBIs (§8.6.3).

Identifying stale TLB entries

```
let tlb_barriered =
  ([T]; tfr; wco; [TLBI]) & tlb-affects^-1
```

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`.

To account for TLB maintenance, we include some edges from translations to `TLBI`s, when there is an interposing newer write. The general shape of this ordering, named `tlb_barriered` in the model, is illustrated in Figure 9.5.

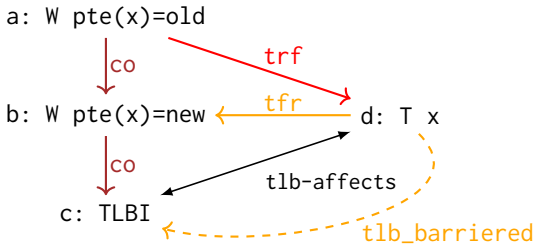


Figure 9.5: General `tlb_barriered` shape.

The `tlb_barriered` auxiliary relation relates a translation read (`d`) to any `TLBI` (`c`) which targets that translations context (ASID, VMID, address, etc) and which is `wco`-after an interposing write (`b`) since the write the translation read read from. Intuitively, 'after' the `TLBI` the stale writes will no longer be in the TLB, and so translation-reads should not read from them any more.

Stale translation reads We can then use this `tlb_barriered` relation to define ordering between translation reads and the TLB maintenance which affect them. For stage 2 translations, `tlb_barriered` alone would be too strong: since invalidations of second-stage entries also require stage 1 invalidations to clear any end-to-end virtual-to-physical mappings (§8.6.4) we must include some additional restrictions, considering the stage 1 and stage 2 cases separately.

```
(* translate ordered-before TLBI *)
let obtlbi_translate =
  [T & Stage1]; tlb_barriered; [TLBI-S1]

| (([T & Stage2]; tlb_barriered; [TLBI-S2])
    ; wco?; [TLBI-S1]
  )
  & (same-translation; [T & Stage1]; maybe_TLB_cached)

| ([T & Stage2]; tlb_barriered; [TLBI-S2])
  & (same-translation; [T & Stage1]; trf^-1; wco^-1)
```

For stage 1 translation reads we can include `tlb_barriered` between stage 1 translation-reads and `TLBI`s directly.

For stage 2 translation reads, 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 we only need the stage 2 invalidation. These cases are illustrated in Figure 9.6, and correspond to the two clauses of `obtlbi_translate` which match on stage 2 translation reads.

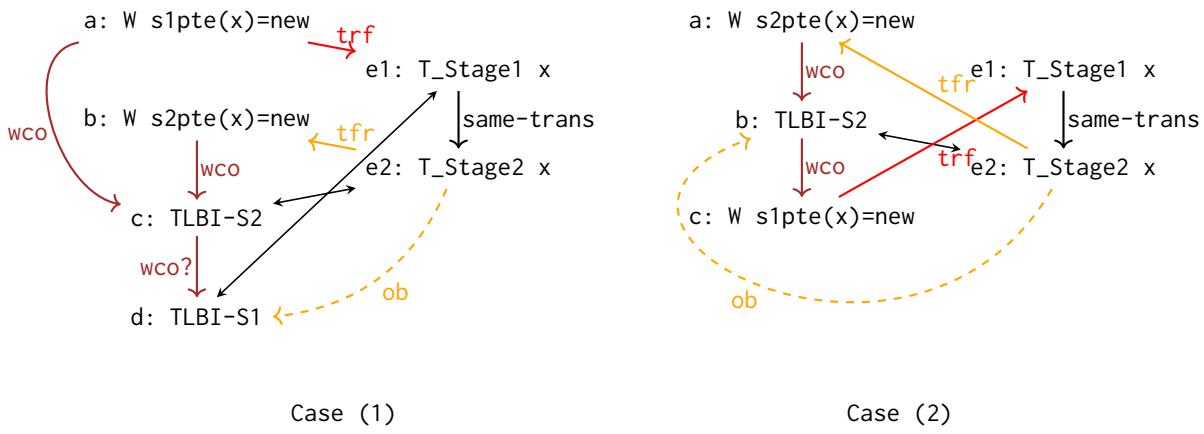


Figure 9.6: `obtlbi` stage 2 scenarios.

The staggered two-step invalidation in case (1), where a translation-read may have been cached in the TLB, is captured with the following `maybe_TLB_cached` relation:

```
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.

We capture the general shape of (2) by ordering the second-stage translation-read with the second-stage `TLBI` using `tlb_barriered` just as we did for Stage 1, but only when one of the same-translation stage 1 walk translation-reads already read from something newer — and therefore there cannot have been a whole-translation cached in the TLB.

Broadcast `TLBI`s Recall that broadcast `TLBI`s 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 9.7, and corresponds to the final clause of `obtlbi`.

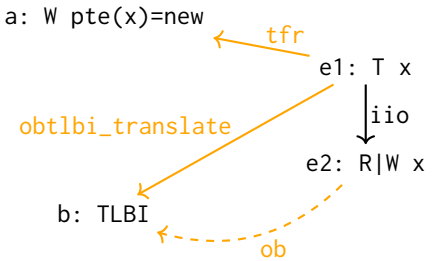


Figure 9.7: `obtlbi` broadcast `TLBI` shape.

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 `TLBI`s 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:

```
let tlb_might_affect =
  [ TLBI-S1 & ~TLBI-S2 &  TLBI-VA  &  TLBI-ASID & TLBI-VMID]
    ; (same-va-page & same-asid & same-vmid) ; [T & Stage1]
| [ TLBI-S1 & ~TLBI-S2 & ~TLBI-VA  &  TLBI-ASID & TLBI-VMID]
    ; (same-asid & same-vmid) ; [T & Stage1]
| [ TLBI-S1 & ~TLBI-S2 & ~TLBI-VA  & ~TLBI-ASID & TLBI-VMID]
    ; same-vmid ; [T & Stage1]
| [~TLBI-S1 &  TLBI-S2 &  TLBI-IPA & ~TLBI-ASID & TLBI-VMID]
    ; (same-ipa-page & same-vmid) ; [T & Stage2]
| [~TLBI-S1 &  TLBI-S2 & ~TLBI-IPA & ~TLBI-ASID & TLBI-VMID]
    ; same-vmid ; [T & Stage2]
| [ TLBI-S1 &  TLBI-S2 & ~TLBI-IPA & ~TLBI-ASID & TLBI-VMID]
    ; same-vmid ; [T]
| ( TLBI-S1 &           ~TLBI-IPA & ~TLBI-ASID & ~TLBI-VMID)
    * (T & Stage1)
| (            TLBI-S2 & ~TLBI-IPA & ~TLBI-ASID & ~TLBI-VMID)
    * (T & Stage2)
```

Finally, we get `tlb-affects` by attaching `tlb_might_affect` to events in the same thread, and if a `TLBI-IS`, to events in other threads too:

```
let tlb-affects =
  ([~TLBI-IS]; tlb_might_affect) & int
|  [TLBI-IS]; tlb_might_affect
```

```
declare wco(Event, Event): bool

(* wco has domain and range of W,CacheOp *)
assert forall ev1: Event, ev2: Event =>
    wco(ev1, ev2) -->
    (W(ev1) | C(ev1) | (ev1 == IW)) & (W(ev2) | C(ev2))

(* wco is transitive *)
assert forall ev1: Event, ev2: Event, ev3: Event =>
    wco(ev1, ev2) & wco(ev2, ev3) --> wco(ev1, ev3)

(* wco is total *)
assert forall ev1: Event, ev2: Event, ev3: Event =>
    wco(ev1, ev3) & wco(ev2, ev3) & ~(ev1 == ev2) -->
    wco(ev1, ev2) | wco(ev2, ev1)

(* wco is irreflexive *)
assert forall ev1: Event, ev2: Event, ev3: Event =>
    wco(ev1, ev2) --> ~(ev1 == ev2)

(* wco is antisymmetric *)
assert forall ev1: Event, ev2: Event =>
    wco(ev1, ev2) --> ~wco(ev2, ev1)

(* all write/cache-op pairs are wco related *)
assert forall ev1: Event, ev2: Event =>
    W(ev1) & C(ev2) -->
    wco(ev1, ev2) | wco(ev2, ev1)

(* wco is consistent with co *)
assert forall ev1: Event, ev2: Event =>
    co(ev1, ev2) --> wco(ev1, ev2)

(* all C are wco after IW
 * n.b. all W are wco after IW, because all W are co after IW
     and co => wco
 *)
assert forall ev: Event =>
    C(ev) --> wco(IW, ev)
```

Figure 9.8: `wco.cat`: `isla-cat` definition of `wco`.

Chapter 10

Validating the RVM model

10.1 Validation against the architecture

To ensure that the proposed virtual memory model presented in Chapter 9 correctly captures the architectural intent (where known), we engaged in detailed discussions with Arm.

Our model is produced through an iterative process: where the production of interesting litmus tests are guided by hardware testing and surveying of software requirements; the resulting tests are presented to, and discussed with, Arm architects; new and updated models are created using any architectural intent learned from those discussions; and, finally, those new models are validated against hardware and software requirements, informing the production of further litmus tests.

Ideally, we would run this process until a fixed point is reached. However, this is not always practical. We know the model presented in Chapter 9 is incomplete, and more work is needed to further update the models with the extensions to the architecture and further clarified intent.

10.1.1 Clarity of architecture

We claim that the litmus tests presented in Chapter 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 improbable that the architectural intent would change. Specifically, the guarantees given by the break, break-before-make, and general TLB-maintenance shapes, are fundamental to the security and correctness of modern software, and so are highly unlikely to be weakened 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 has advanced, Arm have added 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 work have helped guide Arm in that design, and resulted in 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 between PEs.
- ▷ Caching of non-last-level block entries in the TLB.

The first, the constraints on unpredictability, were already discussed earlier (§8.6.5), and more discussion with architects is 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.

10.2 Validating against hardware

Hardware testing is an important aspect in gaining confidence in any relaxed memory model: 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 testing the features covered in previous user-level models (including 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.

One approach would be to use `klitmus7`, an experimental version of `litmus` which produces a kernel module that runs at EL1 [108]. klitmus was primarily designed for the testing of the Linux kernel memory model, with the kernel modules it produces run as part of the Linux kernel. Attempting to modify the currently in-use translation tables or exception vectors would interfere with Linux's operations. Using klitmus 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: `system-litmus-harness`[1].

Limitations `system-litmus-harness` has some limitations, for now: (1) the harness runs at EL1 and cannot run tests at EL2; (2) we do not check for known CPU errata for the device being ran on, instead relying on 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 limitations are fundamental; they should all be solvable with additional engineering resources.

10.2.1 Harness overview

At its core, `system-litmus-harness` is a relatively simple micro-kernel running at EL1. It builds in a set of litmus tests, with fixed code for each thread, and an initial state described in an ad-hoc language. The user gives the harness arguments, at boot, containing the name(s) of litmus tests to run and other run configuration options. The harness then runs the litmus tests, collects the results, and echos those results back to the user through the serial output.

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. This is extended in the obvious way for translation, making each iteration use its own translation tables and ASID.

Litmus test format Figure 10.1 gives an example litmus test, CoTR.inv+dsb-isb, a variation on the straight-forward CoRR coherence shape but for translation walks, 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.

[1] https://github.com/rems-project/system-litmus-harness

```
#include "lib.h"

#define VARS x, y
#define REGS p1x0, p1x2

static void P0(litmus_test_run* data)
{
  asm volatile (
    /* setup */
    "mov x0, %[ydesc]\n\t"
    "mov x1, %[xpte]\n\t"
    /* code */
    "str x0, [x1]\n\t"
  :
  : ASM_VARS(data, VARS),
    ASM_REGS(data, REGS)
  : "cc", "memory", "x0", "x1"
  );
}

static void sync_handler(void)
{
  asm volatile (
    "mov x0, #0\n\t"

    ERET_TO_NEXT(x10)
  );
}

static void P1(litmus_test_run* data)
{
  asm volatile (
    /* setup */
    "mov x1, %[x]\n\t"
    "mov x3, %[xpte]\n\t"
    /* code */
    "ldr x0, [x1]\n\t"
    "dsb sy\n\t"
    "isb\n\t"
    "ldr x2, [x3]\n\t"
    /* teardown */
    "str x0, [%[outp1r0]]\n\t"
    "cbz x2, .after\n\t"
    "mov x2,#1\n\t"
    ".after:\n\t"
    "str x2, [%[outp1r2]]\n\t"
  :
  : ASM_VARS(data, VARS),
    ASM_REGS(data, REGS)
  : "cc", "memory", "x0", "x1",
    "x2", "x3", "x10"
  );
}

litmus_test_t CoTRinv_dsbisb = {
  "CoTR.inv+dsb-isb",
  MAKE_THREADS(2),
  MAKE_VARS(VARS),
  MAKE_REGS(REGS),
  INIT_STATE(
    2,
    INIT_UNMAPPED(x),
    INIT_VAR(y, 1)
  ),
  .interesting_result = (u64[]){
      /* p0:x0 =*/1,
      /* p0:x2 =*/0,
  },
  .thread_sync_handlers =
    (u32**[]){
     (u32*[]){NULL, NULL},
     (u32*[]){(u32*)sync_handler,
    NULL},
    },
  .requires_pgtable = 1,
  .no_sc_results = 3,
};
```

Figure 10.1: CoTR.inv+dsb-isb litmus test, `system-litmus-harness` source.

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`, containing the code of the threads. These functions take a `litmus_test_run` struct, which contains the virtual addresses of each of the global and output variables, and any other initial state required for the test.

Taking the code for Thread 1, in `P1`, as an example, it is given as an `asm` block which contains the test code sandwiched between some setup and teardown code that moves values from the C code into the machine registers the test uses, and back out at the end.

This test has an exception handler for this thread. It is given by the `sync_handler` function and set as the vector for this thread in the initial state. The handler 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, which gives the C definition for the test. It provides 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

Our test format supports writing a variety of kinds of pagetable tests with different initial state setups. Appendix C 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. It says that all three variables start out mapped with initial 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, i.e. it should have the same least significant 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` (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 3B+ (Arm Cortex-A53 r0p4), Raspberry Pi 4B (Arm Cortex-A72 r0p3), and an AWS `m6g-metal` instance (claiming to be an A76).

Note that the hardware tests represent an overlapping set of tests with those presented in Ch.8: some contain BBM violations; some tests are not reproduced on hardware; and some may appear with different names, e.g. CoWTf.inv+dmb test (Figure 8.18, p.132) 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.

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` we

[1] which can be found at https://github.com/rems-project/system-litmus-harness/blob/master/litmus/litmus_tests/pgtable/pmds/ROT1%2Bdsb-dsb-tlbi-dsb.c

did observe some rare violations of the architectural intent on the Arm cortex-based cores in the Raspberry Pi 4B and AWS Graviton cores[1]. The related `MP.BBM1+[dmb.ld]-tlbiis-dsb-isb-dsb-isb+dsb-isb` test (with a detour after the write) was never observed however, suggesting it is related to the `DSB` not fully propagating the store, which implies it may be related to other known CPU errata. 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 architects' intent, and that may change as new revisions of the architecture are released. However, we can identify properties we believe *any* sound model would have, and check that the model presented here has those properties.

The key property is that the presented model has a 'virtual memory abstraction'. While there is no general definition of what such an abstraction is, we give one intuitive and informal definition: a program with a fixed injective translation table mapping behaves as if executing above physical memory directly. We can state this virtual memory abstraction as a property over candidate executions.

To do this, we define a *translation erasure* operation: given a candidate C, the translation-erased candidate $C^{\sim T}$ is C, but where all `TLBI`, `T`, and `T_f` events are erased; any edge containing such events as source or target removed; and extended with the derived relations `addr` and `po` from C.

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 $C^{\sim T}$ is consistent in the base model.

Informally, the proof is a straightforward inclusion proof by relation algebra. The internal and atomic axioms are trivially subset inclusions of one another under translation erasure. Additionally, the translation-internal relation is trivially a subset of the usual internal one with translation events erased. 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 Appendix D for a complete proof, largely due to Jean Pichon-Pharabod.

[1] Further investigation indicates that our `m6g-metal` instance may have in-fact also been a mislabelled Graviton1, also based on the Arm Cortex A72.

Table 10.1: system-litmus-harness hardware results from three devices: Part I.

Name	rpi4b	rpi3bp	graviton2
CoRT	964.72K/8M	520.06K/3M	2.29M/108M
CoRT+dsb-isb	802.86K/8M	327.02K/3M	3.41M/108M
CoTR	2.51M/8M	0/3M	21.70M/107.50M
CoTR+addr	0/8M	1/3M	0/107.50M
CoTR+dmb	1/8M	0/3M	4/107.50M
CoTR+dsb	2/8M	0/2.50M	5/107M
CoTR+dsb-isb	1/8M	0/2.50M	1/107M
CoTR.inv	3.63M/6.50M	0/2.50M	32.28M/43M
CoTR.inv+dsb-isb	0/6.50M	0/2.50M	0/43M
CoTR1+dsb-dc-dsb-tlbi-dsb-isb	2/6.50M	0/2.50M	4/43M
CoTR1+dsb-tlbi-dsb-isb	2/6.50M	0/2.50M	3/43M
CoTR1.tlbi+dsb-isb	6/6.50M	1/2.50M	29/43M
CoTT	0/6.50M	0/2M	0/43M
CoTW	0/1.50M	0/1.50M	0/10.50M
CoWT	3.77M/6.50M	1.85M/2M	22.64M/43M
CoWT+dsb	3.76M/6.50M	995.06K/2M	21.50M/43M
CoWT+dsb-isb	3.78M/6.50M	995.77K/2M	21.50M/43M
CoWT+dsb-svc-tlbi-dsb	0/6.50M	0/2M	0/42.50M
CoWT.inv	10/6.50M	1.73M/2M	169/42.50M
CoWT.inv+dmb	8/6.50M	69.38K/2M	42/42.50M
CoWT.inv+dsb	1/6.50M	0/2M	57/42M
CoWT.inv+dsb-isb	0/6.50M	0/2M	0/42M
CoWT1+dsb-tlbi-dsb	0/6.50M	0/2M	0/42.50M
CoWT1+dsb-tlbi-dsb-isb	0/6.50M	0/2M	0/42.50M
CoWinvT	4.17M/6.50M	1.79M/2M	26.81M/42M
CoWinvT+dsb-isb	4.19M/6.50M	1.83M/2M	26.80M/42M
CoWinvT1+dsb-tlbi-dsb	0/6.50M	0/2M	0/42M
CoWinvWT1+dsb-tlbi-dsb-dsb-isb	0/6.50M	0/2M	0/42M
ISA2.TRR+dmb+po+dmb	0/6.50M	0/2M	0/42M
MP.BBM1+[dmb.ld]-dsb-tlbiis-dsb-isb-dsb-isb+dsb-isb	0/108.50M	0/1.50M	0/437.50M
MP.BBM1+[dmb.ld]-tlbiis-dsb-isb-dsb-isb+dsb-isb	0/198.50M	0/1.06G	0/129.50M
MP.BBM1+[po]-dsb-tlbiis-dsb-isb-dsb-isb+dsb-isb	0/108.50M	0/1.50M	0/145.50M
MP.BBM1+dsb-isb-tlbiis-dsb-isb-dsb-isb+dsb-isb	0/6.50M	0/2M	52/135.50M
MP.BBM1+dsb-tlbiis-dsb-dsb+dsb	1/6.50M	0/2M	7/42.50M
MP.BBM1+dsb-tlbiis-dsb-dsb+dsb-isb	0/6.50M	0/2M	2/42.50M
MP.BBM1+dsb-tlbiis-dsb-dsb-isb+dsb	1/6M	0/2M	0/42.50M
MP.BBM1+dsb-tlbiis-dsb-dsb-isb+dsb-isb	2/6M	0/2M	3/42.50M
MP.BBM1+po-dsb-tlbiis-dsb-isb-dsb-isb+dsb-isb	0/1M	0/1.50M	9/191.50M
MP.BBM1.id+dsb-tlbiis-dsb-dsb+dsb-isb	10/6M	2/2M	87/42.50M
MP.RT+svc-dsb-tlbi-dsb+dsb-isb	1/6M	0/2M	3/42M
MP.RT+svc-dsb-tlbiis-dsb+dsb-isb	1/6M	0/2M	3/42M
MP.RT.inv+dmb+addr	0/6M	0/2M	0/42M
MP.RT.inv+dmb+po	0/6M	6/1.50M	0/42M
MP.RT1+[dmb.ld]-dmb+dsb-isb	7.15K/6M	986/1.50M	1.26K/42M
MP.RT1+[dmb.ld]-dsb-isb-tlbiis-dsb-isb+dmb	0/1M	0/1M	0/23M
MP.RT1+[dmb.ld]-dsb-isb-tlbiis-dsb-isb+dsb-isb	0/1M	0/1M	0/23M
MP.RT1+[dmb.ld]-dsb-tlbiis-dsb-isb+dmb	0/6M	0/1.50M	0/42M
MP.RT1+dc-dsb-tlbiall-dsb+dsb-isb	4/6M	1/1.50M	5/41.50M
MP.RT1+dc-dsb-tlbiall-dsb-isb+dsb-isb	3/6M	0/1.50M	2/41.50M
MP.RT1+dsb-isb-tlbiis-dsb-isb+dsb-isb	0/6M	0/1.50M	4/41M
MP.RT1+dsb-tlbi-dsb+dsb-isb	0/6M	0/1.50M	2/41M
MP.RT1+dsb-tlbiall-dsb+dsb-isb	5/6M	0/1.50M	6/41M
MP.RT1+dsb-tlbiallis-dsb+dsb-isb	3/6M	0/1.50M	2/41M
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.

Name	rpi4b	rpi3bp	graviton2
MP.RT1+dsb-tlbiis-dsb-isb+dmb	0/6M	0/1.50M	1/41M
MP.RT1+dsb-tlbiis-dsb-isb+dsb-isb	0/6M	0/1.50M	1/41M
MP.RT1+dsb-tlbiis-dsb-tlbiis-dsb+dsb-isb	0/6M	0/1.50M	3/41M
MP.TT+Winv-dmb-Winv+tpo	254.83K/6M	114.48K/1.50M	170.96K/41M
MP.TT+dmb+dsb-isb	688.65K/5.50M	174.78K/1.50M	492.98K/41M
MP.TT+dmb+tpo	843.79K/5.50M	157.80K/1.50M	480.31K/41M
MP.TT.inv+dmb+dsb-isb	0/5.50M	0/1.50M	0/41M
MP.TT.inv+dmb+tpo	0/5.50M	0/1.50M	0/41M
MP.invRT+dsb+dsb-isb	871.53K/5M	101.75K/1.50M	1.78M/40.50M
MP.invRT1+dsb-isb-tlbiis-dsb-isb+dsb-isb	0/5.50M	0/1.50M	1/41M
MP.invRT1+dsb-tlbiis-dsb+dsb	0/5M	0/1.50M	2/41M
MP.invRT1+dsb-tlbiis-dsb+dsb-isb	1/4.50M	0/1.50M	1/41M
WRC.AT+ctrl+dsb	128.64K/4.50M	77.36K/1.50M	214.45K/40M
WRC.TRR+addr+dmb	0/4.50M	0/1.50M	0/40M
WRC.TRR.inv+addrs	0/4.50M	0/1.50M	0/40M
WRC.TRT+addr+dmb	35.28K/4.50M	32.50K/1.50M	103.16K/40M
WRC.TRT+dmbs	53.60K/4.50M	36.76K/1.50M	171.51K/40M
WRC.TRT+dsb-isbs	18.80K/4.50M	30.44K/1.50M	104.62K/39.50M
WRC.TRT.inv+addrs	0/4M	0/1.50M	0/38.50M
WRC.TRT.inv+dsb-isbs	0/4M	0/1M	0/38M
WRC.TRT.inv+po+addr	0/4M	0/1M	0/37.50M
WRC.TRT.inv+po+dmb	0/4M	0/1M	0/37M
WRC.TRT1+dsb-tlbiis-dsb+dmb	0/4.50M	0/1M	0/38M
WRC.TRT1+dsb-tlbiis-dsb+dsb-isb	0/4.50M	0/1M	0/38M
CoWR.alias	0/6M	0/1.50M	0/36M
MP+dmb-data+dmb	0/5M	0/1.50M	0/36M
MP.alias+dmbs	0/5M	0/1.50M	0/36M
MP.alias2+dmb-data+dmb	0/5M	0/1.50M	0/36M
MP.alias2+dmbs	0/3M	0/1.50M	0/19.50M
MP.alias2+po-data+dmb	2.23K/5M	3.17K/1.50M	407.36K/36M
MP.alias3+rfi-data+dmb	51/3M	16/1.50M	36.35K/19.50M
SB.alias+dmbs	0/5M	0/1M	0/35.50M
WRC.alias2+addrs	0/4M	0/43M	0/19M
WRC.alias2+dmbs	0/4M	0/43M	0/18.50M
MP.NC+dsb-dc-dsb-dmb+dmb	138.80K/8M	364.97K/26M	54.95K/25.50M
MP.NC+po-dmb+dmb	345.33K/7.50M	642.90K/25.50M	333.55K/25.50M
MP.NC1+dsb-tlbiis-dsb-dc-dsb-dmb+dmb	0/7.50M	0/25.50M	0/25.50M
MP.NC1+dsb-tlbiis-dsb-dmb+dmb	556/7.50M	482/25.50M	6/25.50M
WR.NC+dsb	0/0	0/0	0/0
WR.NC+po	0/0	0/0	0/0
WR.WARA-NC+dsb	0/0	0/0	0/0
WR.WARA-NC+po	0/0	0/0	0/0
WWR.NC+po-po	0/0	0/0	0/0
CoWT.L23+dsb-isb	11.45M/13M	6.73M/13.50M	48.94M/84.50M
CoWT.L23+po	12.88M/13M	13.39M/13.50M	80.61M/84.50M
CoWT1.L23+dsb-tlbi-dsb-isb	0/13M	0/13.50M	0/84.50M
ROT+dsb-dsb	0/13M	0/13.50M	0/84.50M
ROT+po-po	0/13M	0/13.50M	0/84M
ROT1+dsb-dsb-tlbi-dsb	0/13M	0/13.50M	0/84M
ROT1+dsb-dsb-tlbivaa-dsb	0/13M	0/13.50M	0/84M
CoTT+dsb-popage	0/35.50M	0/31M	0/1.12G
CoTT+po-popage	1/47M	0/43.50M	0/1.20G
WR.MAIR1+dsb-isb-dc-dsb	0/0	0/0	0/0
WR.MAIR1+dsb-isb-po	0/0	0/0	0/0
WR.MAIR1+dsb-tlbi-dsb-isb-dc-dsb	0/0	0/0	0/0
WR.MAIR1+dsb-tlbi-dsb-isb-po	0/0	0/0	0/0
WR.MAIR1+po-po	0/0	0/0	0/0

Part III

Exceptions and interrupts

This part is based on: in-progress and under-submission work done in collaboration with Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Ohad Kammar, Jean Pichon-Pharabod, and Peter Sewell.

Chapter 11

Relaxed precise exceptions

We now turn to the final part, and discuss hardware support for exceptions and interrupts.

We do so in the way previous work have made now typical: we describe the main phenomena and architectural design space, through the exploration of litmus tests; we use those litmus tests as a catalyst for discussions about the architectural intent with the architects and for discovery of the current implementations by the surveying of hardware; we produce a formal mathematical model that captures that intent; and, finally, we validate that model by making it executable as a test oracle and execute a suite of litmus tests, comparing the results to hardware and previously collected intent from architects.

Chapter contents

11.1 Introduction

Hardware exceptions (and their many variants: interrupts, traps, faults, aborts, etc.) provide support for many exceptional situations that systems software has to manage. This includes explicit privilege transitions via system calls, implicit privilege transitions from trappable instructions, inter-processor software-generated interrupts, external interrupts from timers or devices, recoverable faults like address translation faults, and non-recoverable faults like memory error correction faults.

To manage exceptions, software relies on a key architectural guarantee, *precision*: that exceptions appear to execute between instructions. To confidently write concurrent systems code that handles exceptions, e.g. mapping on demand at page faults, programmers need a well-defined and well-understood semantics. These modern definitions of precision (e.g. in the current Arm-A documentation) are mostly unchanged over the last 60 years, dating back to at least the IBM System/360. These definitions fundamentally assume a sequential programmers model. For example, Hennessy and Patterson state [109]:

> An exception is imprecise if the processor state when an exception is raised does not look exactly as if the instructions were executed sequentially in strict program order

However, modern architectures with programmer-observable relaxed behaviour, such as Arm-A, make such a naive definition inadequate, and leaves it unclear exactly what guarantees there are on exception entry and exit. On pipelined out-of-order processors with observable speculative execution, exceptions have subtle interactions with relaxed memory behaviour which had not previously been investigated.

Overview In this part, we begin by clarifying the key concepts needed to discuss exceptions in the relaxed-memory setting (§11.1-11.2), through the exploration the basic relaxed behaviour across exception boundaries (§11.3). We extend this by introducing the potential of external aborts and examining how they effect the programmer-visible behaviour (§11.4).

We develop an axiomatic model for precise exceptions on Arm-A, including tooling for executing it as a test oracle, along with a library of tests (Chapter 12).

Finally, we validate this model (Chapter 13) by extending the harness presented in Part II and collecting data from a range of implementations.

11.1.1 Exception taxonomy

Arm-A defines multiple kinds of exception [81, D1.3.1]:

- ▷ Synchronous exceptions. These originate from an instruction, e.g. supervisor/hypervisor calls, traps, data/instruction, page faults, etc.
- ▷ Asynchronous exceptions. These are interrupt requests from other processors/peripherals/timers, or system errors.

In Arm nomenclature, any non-synchronous exception is called an interrupt.

Synchronous exceptions are further broken down into *classes*, for example:

- ▷ PC Alignment, for a misaligned program counter.
- ▷ Instruction abort, for MMU faults on instruction accesses.
- ▷ Undefined instruction encoding.
- ▷ Data abort, for MMU faults on data accesses.
- ▷ Execution of an SVC (supervisor call).
- ▷ Trapped register access, from attempting accessing a register that is not permitted or is configured to trap.

For a complete list of exception classes, and their prioritisation, we refer to the Arm architecture reference manual [81, D1.3.5, p5369].

11.1.2 Exception lifecycle

When an exception is taken, execution jumps to the appropriate *exception vector*. Vectors are predetermined locations which contain code to be executed on the event of an exception. Different kinds of exception jump to different vectors, and so the currently in-use vectors form a *vector table*. Software configures the vectors by setting the base address of the vector table, by writing to the appropriate vector base address register (VBAR).

On taking the exception:

- ▷ The current processor state is saved into the saved program status register (SPSR). This includes the current exception level, status flags and condition bits, and interrupt masking (described in more detail later).
- ▷ The privilege level typically escalates (e.g. from EL0 to EL1).
- ▷ The program-counter to return to (the 'preferred return address') is saved into the appropriate exception link register (ELR).
- ▷ The cause of the exception is saved into either the exception syndrome register (ESR) for synchronous exceptions, telling the programmer the *class* of the exception and other associated data; or into the interrupt status register (ISR), telling the programmer which interrupt(s) are pending.
- ▷ If the exception is a translation-related fault, the faulting address is also saved into the fault address register (FAR).
- ▷ The PC is set to the current VBAR plus the appropriate offset.

The code then executed is termed the *exception handler*. Execution continues in the new state until the processor executes an `ERET` ('exception return') instruction. On executing an `ERET`:

- ▷ The saved processor state (SPSR) is restored.
- ▷ The value saved in the ELR is written to the PC.

Thus, execution jumps back to where the program was executing before the exception was taken, in much the same processor state as it was in at the time.

Preferred return address

The 'preferred return address' of synchronous exceptions has an architecturally defined relationship with the instruction that caused the exception. For most instructions, the preferred return address is the program counter value at the point when the exception is taken, therefore returning back to the same instruction once the exception is handled.

There is one exception to this: the class of *exception generating instructions*, whose sole purpose is to generate a particular kind of exception. The most common of these is the `SVC` ('supervisor call') instruction, which is used to implement system calls. These instructions preferred return address is always the *next* instruction, that is, $PC + 4$.

11.1.3 Vectors and vector tables

The appropriate vector is determined from: the *type* of the exception, either synchronous, interrupt request (IRQ), 'fast' interrupt request, or external abort (which is described in more detail later); the current stack pointer in use; whether the exception originates from a lower exception level; and whether the exception originates from the 32-bit mode or not. As such the vector table contains 16 vectors. Each vector is 128 bytes. The vectors are then located at a given offset from the base address, see Figure 11.1.

Exception from	**Exception type**			
	Synchronous	**IRQ**	**Fast IRQ**	**External abort**
Current EL, using stack pointer SP_EL0	0x000	0x080	0x100	0x180
Current EL, using this EL's stack pointer	0x200	0x280	0x300	0x380
Lower EL, in 64-bit mode	0x400	0x480	0x500	0x580
Lower EL, in 32-bit mode	0x600	0x680	0x700	0x780

Figure 11.1: Arm vector table offsets [81, D1.3.1].

There is not a single active vector table, but one per exception level, with all the exception-related registers (SPSR, ELR, ESR, FAR, etc) appropriately banked (with one per exception level).

Note that in Armv8, fast interrupt requests function identically to normal interrupt requests. However, they have independent routing machinery. Interrupt controllers may freely choose to route different interrupts as different types, but which type the interrupt is has no effect on the execution of the machine.

11.1.4 Precision

Historically, the introduction of pipelined machines raised issues about the behaviour of exceptions: since instructions may have already been partially executed, the resulting interrupts would appear as a discontinuity in the flow of instructions [110]. Since then, hardware has had two kinds of exceptions: imprecise exceptions retain that discontinuity, whereas precise ones take the performance penalty of recovering (e.g. by discarding later instructions and restarting earlier instructions) to guarantee more predictable behaviours that programmers could rely on. Intuitively, for a precise exception one can pinpoint a particular point in the sequence of instructions where the exception happens.

Today, Arm retains imprecise exceptions, but only in some cases: all synchronous exceptions and interrupt requests are precise. Only system errors — errors from the external system reported back the CPU asynchronously — may be imprecise. We discuss external aborts in more detail in §11.4.

11.2 Instruction instances

One often thinks of processors as executing *instructions* in some *instruction sequence*, and common terminology is based on those two concepts. For example, the Arm manual has around 60 instances of *instruction stream* or *execution stream*.

However, exceptions can arise at multiple points within the fetch-decode-execute cycle, including during the fetch and decode, when there is no 'instruction'. For Armv9.4-A, much of this is captured in an Arm top-level function written in the Arm Architecture Specification Language (ASL), which previous work by Armstrong, Campbell et al. [45, 111, 112] translated into Sail. This gives us an executable semantics of the sequential ISA aspects of Armv9.4-A with exceptions (§13.2).

A highly simplified outline of a single-instruction slice of the (400k line) instruction semantics is given in Figure 11.2.

```
function __TopLevel() =
  // in TakePendingInterrupts:
  if IRQ then AArch64_TakePhysicalIRQException()
  if SE then AArch64_TakePhysicalSErrorException(...)
  // in AArch64_CheckPCAlignment:
  if pc[1..0] != 0b00 then AArch64_PCAlignmentFault()
  // in __FetchInstr:
  opcode = AArch64_MemSingle_read(pc, 4) // read memory
  // in __DecodeA64:
  match opcode
    [1,_,1,1,1,0,0,1,0,1,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_] =
      // the semantics for one family of instructions,
      // including loads LDR Xt,[Xn]
      // execute_aarch64_instrs_memory_single_general_
      // immediate_signed_post_idx(n,t,...)
      let address = X_read(n, 64) // read register n
      let data : bits('datasize) = // read memory
        Mem_read(address, DIV(datasize,8))
      // write register t
      X_set(t, regsize) = ZeroExtend(data, regsize)
```

Figure 11.2: Outline of a single-instruction slice of the Arm intra-instruction semantics.

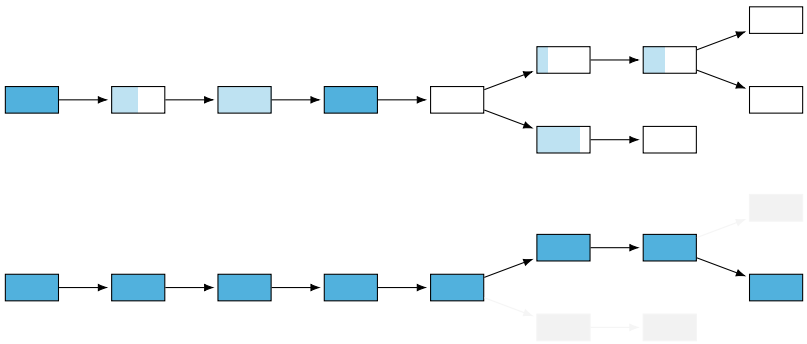


Figure 11.3: Top. The tree of (partially) executed FDX instances at one time, in hardware or operational model execution. **Bottom.** The sequence of architecturally executed FDX instances in a completed execution.

Executing this semantics may lead to one or more kinds of exception, calling the ASL/Sail function `AArch64_TakeException()`. This function writes the appropriate values to registers, e.g. computing the next PC, exception level, etc. and terminates this `__TopLevel()` execution. So instead of 'instruction instances', we refer to *fetch-decode-execute instances* (FDX instances), a single execution of `__TopLevel()`.

11.2.1 Fetch-decode-execute trees and streams

One must relate the out-of-order speculative execution of hardware implementations and the architectural definition of the allowed behaviours.

At any instant, each core may be processing, out-of-order and speculatively, many instructions (really, FDX instances). Partially executed instances are restarted or discarded if they would violate the intended semantics (e.g. on a mispredicted branch).

One can visualise the state of a single core abstractly as a tree of partially and completely executed instances, as in Figure 11.3 (top). Abstract-microarchitectural operational semantics have long made use of this abstraction to implement the thread subsystems [8, 20, 15, 48, 16, 7], see Chapter 2. We now lift this model-specific concept into the domain of architecture.

In the figure, we depict the retired (committed) FDX instances as solid dark green coloured nodes, and partially/tentatively executed in-flight instances as light green coloured nodes. The arrows depict program order. Committed instances can be program-order-after in-flight instances, and non-committed instances may need to be restarted. Eventually all FDX instances for this hardware thread will be either committed or discarded, e.g. as in Figure 11.3 (bottom). These are the *architecturally executed* FDX instances. The architecture definition, and any formal semantics thereof, have to define which such sequences are allowed for each thread. This definition includes the register content; memory read values; and their relationships with other threads, as determined by the relaxed concurrency model. Axiomatic concurrency models, e.g. [13, 113, 114, 115, 116, 117, 4, 2, 34, 118, 69, 42, 32, 46], have candidate executions which contain events just from these architecturally executed instances.

The Arm prose specification, given in Figure 11.4 (top), previously attempted to capture the relationship between implementation execution (out-of-order and speculative) and the architectural definition of allowed behaviour in terms of a notion of a 'simple sequential execution' of the machine. As the prose says, simple sequential execution does not hold for the intended relaxed-memory architecture. We propose a more correct rephrasing that allows for exceptions and other systems phenomena in Figure 11.4 (bottom).

Figure 11.5 depicts a tree of instances involving exception entry (`SVC`) and return (`ERET`). Arm-A allows implementations to execute the exception handler's instruction instances out-of-order with respect to instances program-order-before the exception entry and program-order-after the exception return. The constraints on this freedom is what we now explore.

Architecturally executed An instruction is architecturally executed only if it would be executed in a simple sequential execution of the program. [...]
Simple sequential execution The behavior of an implementation that fetches, decodes and completely executes each instruction before proceeding to the next instruction. Such an implementation performs no speculative accesses to memory, including to instruction memory. The implementation does not pipeline any phase of execution. In practice, this is the theoretical execution model that the architecture is based on, and Arm does not expect this model to correspond to a realistic implementation of the architecture.

Architecturally executed A candidate execution can be architecturally executed if it is composed of a sequence of FDX instances for each thread that together satisfy the Arm concurrency model [extended to cover exceptions, as described here, and other systems phenomena], starting from the machine initial state.

Figure 11.4: Arm prose specification [81, Glossary, p12916] (**top**) and our suggested rephrasing (**bottom**).

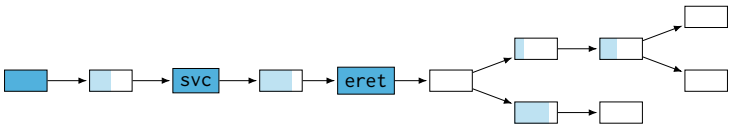


Figure 11.5: The tree of partially and completely executed FDX instances with exceptions, in hardware or operational model execution. Instructions may execute out-of-order across exception boundaries, requiring a modern definition for precision.

11.3 Relaxed behaviour of precise exceptions

Exceptions change the control flow and processor context, that is, the collection of system and special registers which control the execution of the machine. These include the current exception level (`PSTATE.EL`), masking of interrupts (`PSTATE.{D,A,I,F}`), processor flags, and so on. Changes to the context need not take effect immediately; to ensure that program-order-later instructions see such changes, exceptions are *context synchronising*: they ensure updates to the context are seen by later instructions. As a side-effect of that context synchronisation, exception boundaries impose some ordering.

We will see that the context synchronisation performed by the machinery is the primary mechanism that enforces order at the boundary of an exception. In addition to this, different classes of exceptions may come with their own additional ordering constraints: translation faults are bound by the constraints discussed in Chapter 8, interrupts cannot happen before they are generated, and so on. However, we can set a baseline set of behaviours for exceptions by investigating the simplest kind of exception: the unencumbered exception-generating-instructions such as the `SVC` supervisor call. As such, throughout this section we will use exceptions from `SVC` instructions as an exploratory tool, but all behaviours described therein also apply to all other exception types.

In this section, we explain relaxed behaviour of precise exceptions through litmus testing. We start with the baseline out-of-order execution across exception boundaries (§11.3.1), before talking about context synchronisation in detail (§11.3.2). We continue with a collection of potentially interesting edge cases: changing privilege levels (§11.3.3), dependencies through exception machinery (§11.3.4), asynchronous exceptions (§11.3.5), then the stronger behaviour of specific types of exceptions (§11.3.6), before touching on how the instruction semantics needs to be adapted (§11.3.7), and finally we discuss a corner case when disabling context synchronisation (§11.3.8).

11.3.1 Out-of-order execution across exception boundaries

Before discussing the ordering that exception boundaries do impose, we will first see that, in general, exception boundaries do not act as memory barriers. Loads and stores may be executed out-of-order over an exception entry or an exception exit or the composition of both.

Figure 11.6 contains a sample of shapes which show that the reads and writes are able to execute

out-of-order with respect to the various exception boundaries.

As an example, the first of these, MP+dmb.sy+svc, is an MP (message passing) variant. The writer thread is ordered by a full barrier. The reader thread is split in two: after the first read there is an SVC which generates an exception and the exception handler performs the second read. We show this in the execution diagrams with a single svc edge between the two reads.

11.3.2 Context synchronisation and speculation

Updates to the context, such as writes to system registers, need synchronisation to be guaranteed to have an effect. We do not model the behaviour of such context-changing operations when such synchronisation is not performed. Instead, we merely identify when and how exceptions are context-synchronising, and note that this has a knock-on effect on memory accesses.

Architecturally, a context synchronisation event guarantees that no instruction program-order-after the event is observably fetched, decoded, or executed until the context-synchronising event has happened [68, p. 14752, B2.10.1]. A simple microarchitectural implementation for context synchronisation is to flush the pipeline: restarting all program-order-later instances once the context-synchronising effect occurs. More complex implementations may be more clever, as long as they preserve the semantics.

Software can explicitly generate context-synchronising events by issuing an Instruction Synchronisation Barrier (ISB). Context synchronisation can also happen implicitly, for example on exception entry and exit. This is the case in Arm, except in a rare use case we return to in §11.3.8.

The effect of context synchronisation events in exception boundaries is that any instance after the boundary has an ISB-equivalent dependency on the instances before the boundary. This mechanism implies the following fundamental invariant: *context synchronising exception boundaries are never taken speculatively.* This limits speculation of such boundaries to the same well-understood extent as speculation of ISBs. This invariant has interesting interactions with external aborts, which we discuss in §11.4.

The fact that context-synchronising exception boundaries cannot be taken speculatively implies that the code inside an exception handler cannot execute before the exception entry's control-flow is determined (see MP+dmb+ctrlsvc (Figure 11.7, p.203)); and similarly, cannot return before the ERET's control-flow is determined (see MP+dmb+ctrleret (Figure 11.7, p.203)).

11.3.3 Privilege level

MP.EL1+dmb.sy+svc AArch64

Initial state: *x=0, *y=0;
0:X1=x, 0:X3=y; 1:X1=y, 1:X3=x
PSTATE.EL=0b1;

Thread 0	Thread 1	T1 Handler

MOV X0,#1
STR X0,[X1]
DMB SY
MOV X2,#1
STR X2,[X3]
	LDR X0,[X1]
	SVC #0
		LDR X2,[X3]

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

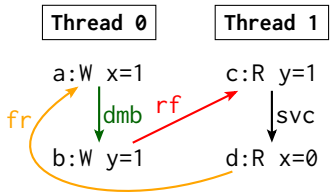


Figure 11.8: Same-exception-level exceptions are no stronger or weaker.

The privilege level (exception level) has little to no additional effect on the behaviours we present: their allowed/forbidden status remains the same whether the privilege goes up/down in entry/exit or remains the same. For example in the MP.EL1+dmb+svc test (Figure 11.8) the exception is taken *from* EL1 and

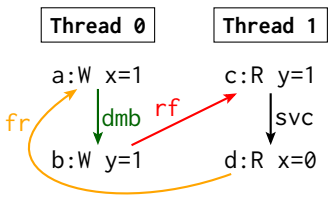
MP+dmb.sy+svc AArch64

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

Thread 0	Thread 1	T1 Handler

MOV X0,#1
STR X0,[X1]
DMB SY
MOV X2,#1
STR X2,[X3]
	LDR X0,[X1]
	SVC #0
		LDR X2,[X3]

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



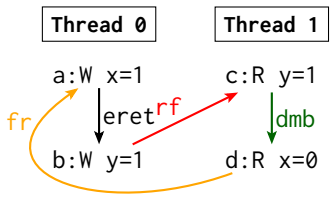
MP+eret+dmb.sy AArch64

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

Thread 0	T0 Handler	Thread 1

SVC #0
MOV X2,#1
STR X2,[X3]
	MOV X0,#1
	STR X0,[X1]
	ERET
		LDR X0,[X1]
		DMB SY
		LDR X2,[X3]

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



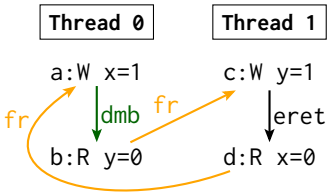
SB+dmb.sy+eret AArch64

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

Thread 0	Thread 1	T1 Handler

MOV X0,#1
STR X0,[X1]
DMB SY
LDR X2,[X3]
	SVC #0
	LDR X2,[X3]
		MOV X0,#1
		STR X0,[X1]
		ERET

Allowed: 0:X2=0, 1:X2=0



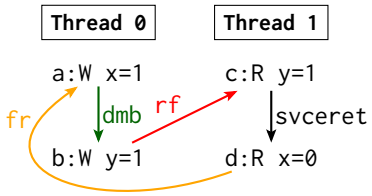
MP+dmb.sy+svceret AArch64

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

Thread 0	Thread 1	T1 Handler

MOV X0,#1
STR X0,[X1]
DMB SY
MOV X2,#1
STR X2,[X3]
	LDR X0,[X1]
	SVC #0
	LDR X2,[X3]
		ERET

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



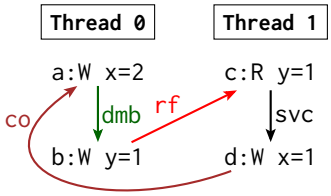
S+dmb.sy+svc AArch64

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

Thread 0	Thread 1	T1 Handler

MOV X0,#2
STR X0,[X1]
DMB SY
MOV X2,#1
STR X2,[X3]
	LDR X0,[X1]
	SVC #0
		MOV X2,#1
		STR X2,[X3]

Allowed: 1:X0=1, *x=2



LB+svc-erets AArch64

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

Thread 0	T0 Handler	Thread 2	T1 Handler

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

Allowed: 0:X0=1, 1:X0=1

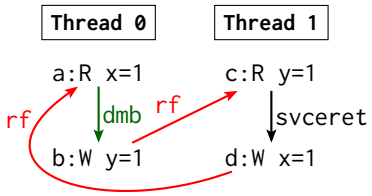


Figure 11.6: Reads and writes may be executed out-of-order across exception entry, exit, or even both.

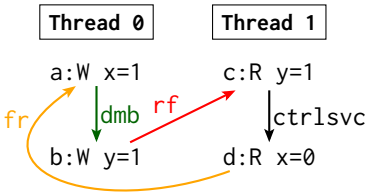
MP+dmb.sy+ctrlsvc AArch64

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

Thread 0	Thread 1	T1 Handler

MOV X0,#1
STR X0,[X1]
DMB SY
MOV X2,#1
STR X2,[X3]
	LDR X0,[X1]
	CBNZ X0,LC00
	LC00:
	SVC #0
		LDR X2,[X3]

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



MP+dmb.sy+ctrleret AArch64

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

Thread 0	Thread 1	T1 Handler

MOV X0,#1
STR X0,[X1]
DMB SY
MOV X2,#1
STR X2,[X3]
	SVC #0
	LDR X2,[X3]
		LDR X0,[X1]
		CBNZ X0,LC00
		LC00:
		ERET

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

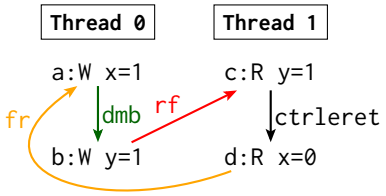


Figure 11.7: Context synchronising exception entry (and returns) are not executed speculatively.

to EL1, but this does not affect any of the machinery (except which vector is used). As before, this is a general statement about the exception machinery, and specific types of exceptions may have additional constraints: e.g. translation faults cannot be caused by out-of-context translations, where the context depends on the exception level (§8.8.1).

Store forwarding It is permitted for writes to be forwarded from a store to a read across exception entry and return. For example in the SB+dmb+rfisvc-addr test (Figure 11.9), the store in Thread 1 is observed by the load in the exception handler (at a higher privilege level) 'early', before it is propagated globally.

SB+dmb.sy+rfisvc-addr AArch64

Initial state: *x=0, *y=0;
0:X1=x, 0:X3=y; 1:X1=y, 1:X3=y, 1:X5=x

Thread 0	Thread 1	T1 Handler

MOV X0,#1
STR X0,[X1]
DMB SY
LDR X2,[X3]
	MOV X0,#1
	STR X0,[X1]
	SVC #0
		LDR X2,[X3]
		EOR X6,X2,X2
		LDR X4,[X5,X6]

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

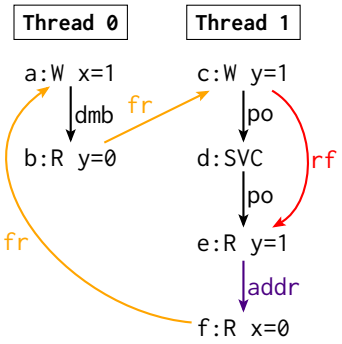


Figure 11.9: Forwarding into a non-speculative handler.

11.3.4 Dependency through system registers

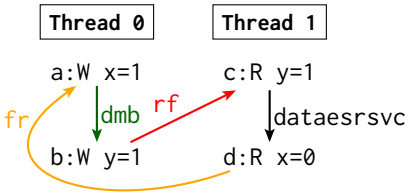
MP.EL1+dmb.sy+dataesrsvc AArch64

Initial state: *x=0, *y=0;
0:X1=x, 0:X3=y; 1:PSTATE.EL=0b1,
1:X1=y, 1:X3=x

Thread 0	Thread 1	T1 Handler

MOV X0,#1
STR X0,[X1]
DMB SY
MOV X2,#1
STR X2,[X3]
	LDR X0,[X1]
	MRS X4,ESR_EL1
	EOR X5,X0,X0
	ADD X5,X4,X5
	MSR ESR_EL1,X5
	SVC #0
		LDR X2,[X3]

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



MP+dmb.sy+ctrlelr AArch64

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

Thread 0	Thread 1	T1 Handler

MOV X0,#1
STR X0,[X1]
DMB SY
MOV X2,#1
STR X2,[X3]
	SVC #0
	LDR X2,[X3]
		LDR X0,[X1]
		MRS X4,ELR_EL1
		EOR X5,X0,X0
		ADD X5,X4,X5
		MSR ELR_EL1,X5
		ERET

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

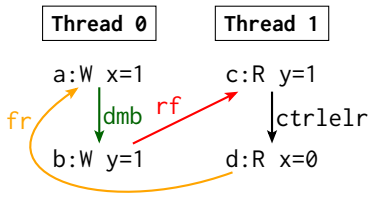


Figure 11.10: System registers and context synchronisation

Where exceptions are taken to and returned to are part of the context, and their respective registers must be read by the exception machinery on taking and returning from the exception. These registers are not read-only; software can write to them. Therefore, they can be involved in register dependency chains. While we do not attempt, in this work, to build a general model of dependencies, we touch on this particular aspect briefly.

Dependencies on system register accesses compose with ordering from context synchronisation events to program-order-later instructions. The MP.EL1+dmb+dataesrsvc test (Figure 11.10) demonstrates that a write to the system register ESR that depends on a read forbids reordering this read across the boundary, even though resolving the dependency does not affect the exception.

The ELR register is a *special-purpose register*, and is therefore 'self-synchronising', unlike system registers [81, D19.1.2, p6331]. Therefore, writes to the ELR do not need context synchronisation to guarantee that they are seen by program-order-later instructions, and this means that dependencies into the ELR are preserved automatically, for example, in the MP+dmb+ctrlelr test (Figure 11.10).

This has two related subtleties, and is currently under investigation by Arm. The Software Thread ID Register (TPIDR) is a system register in which the operating system can store thread identifying information, but has no relevant indirect effects. Further testing and discussions may clarify whether it forbids reordering. While dependencies through special-purpose registers are preserved, context synchronisation does not necessarily need to wait for those writes, and so these dependencies do not necessarily pass to instructions after context synchronisation (in contrast to system register writes).

11.3.5 Ordering from asynchronous exceptions

Asynchronous exceptions cannot be taken speculatively. Therefore, all instructions program-order-after an asynchronous exception happen after that exception.

11.3.6 Exception-specific mechanisms

Some exceptions on some implementations involve additional mechanisms. For example, when an implementation supports Enhanced Translation Synchronisation the translation-table-walks which generate MMU faults gain additional ordering from program-order-previous instances, see §8.4.3. Figure 11.11 compares a message-passing shape involving a translation fault with an asynchronous interrupt.

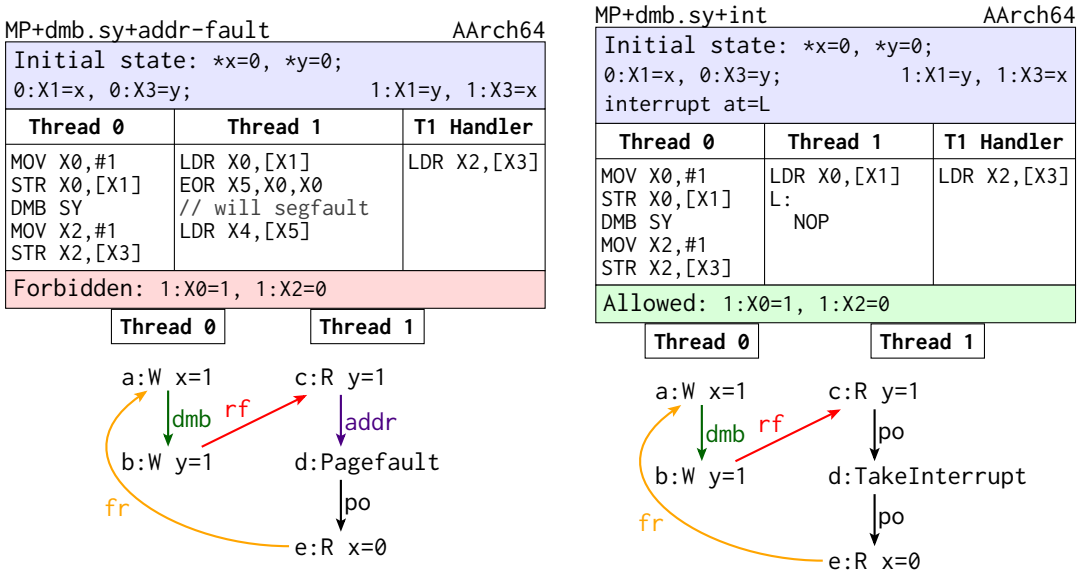


Figure 11.11: Different exception kinds can have different behaviour.

11.3.7 Exceptions and the intra-instruction semantics

Wherever possible, we want to interpret the intra-instruction ASL ordering as preserved, for conceptual simplicity, memory-model tool execution, and reasoning. This has previously been possible except in a few specific cases that are inherently concurrent: instructions that do multiple accesses, and `CSEL`, `CAS`, `SWAP`, etc. Exceptions introduce a new interesting case for instructions that do a register writeback concurrently with a memory access. For example, `STR` (immediate) has 'Post-index' and 'Pre-index' versions [81, C6.2.322, p1996]. The post-index `STR Xt, [Xn], #8`, for example, stores the value in `Xt` to the address initially in register `Xn` and increments `Xn` by `8`. The Arm ARM ASL for `STR` puts that register write at the end, after the memory access has completed.

The architectural intent is that program-order-later instances that depend on `Xn` can go ahead early, e.g. before the data in register `Xt` is available to be written to memory. The related litmus tests have previously been observed on hardware [119].

Previous work captured this allowed by having the register writeback before the memory access in the instruction semantics. However, exceptions require more care: when the memory access generates an exception, the writeback register should appear unchanged to instances after the exception boundary.

Currently, the Armv9.4 ASL, and therefore the Sail derived from it, does not account for the relaxed behaviour of exceptions around these register writebacks. There are a number of proposed solutions, ranging from including code to cleanup after exceptions in the ASL, to treating register writes fully relaxed, like memory writes: with speculative 'forwarding' and discarding on an exception. As was discussed earlier we will need some similar change for relaxed system registers (§8.7.1), and for intra-instruction parallelism (§8.4.9). No solution has yet been implemented, and so the model here does not account for these cases.

11.3.8 Disabling context synchronisation

So far we have assumed exception boundaries are context synchronising. However, Arm has an optional feature, `FEAT_ExS`, which provides two new fields, `EIS` and `EOS`, in the `SCTLR_ELx` system control register. These allow software to disable context synchronisation on exception entry and return, respectively. While the semantics seems clear for these systems, the programming model is unpredictable and hard to program correctly, and so this configuration is rarely encountered in practice.

The result of switching off context synchronisation on exception boundaries is to weaken the previously described speculation tests: permitting speculation of the entry or exit of non-context-synchronising exception boundaries, and all the behaviours associated thereof.

11.4 Synchronous external aborts

The memory system may detect errors such as data corruption independently of the MMU or Debug hardware, e.g. using parity bits or error correcting code. In those cases, it will signal the error by a class of exceptions called *external aborts*. The architecture does not define when implementations report such aborts synchronously [97, D7.6, I_{PJHZS}]. It is implementation defined whether an external abort is reported as a synchronous external abort (under the 'Data abort' class) or asynchronously as a system error, and what errors are reported as external aborts at all [97, R_{FNVVJ}].

Instances program-order-after a potential cause for synchronous external aborts are considered speculative until any such synchronous external abort can be ruled out. This results in stronger behaviour (§11.4.1). In an implementation that always reports external aborts asynchronously, the later instances become non-speculative earlier, allowing them to exhibit weaker behaviours.

In general, systems want to report errors as synchronously as possible. When errors are reported asynchronously, in general, the only recovery is to wind down the aborting process. The Arm Reliability, Availability, and Serviceability (RAS) extension adds some ability for more fine-grained recovery procedures, but this extension is a substantial component of the architecture, far beyond the scope of this work.

11.4.1 Behaviour resulting from synchronous external aborts

There is an asymmetry between reads and writes with respect to speculation: reads can be satisfied speculatively, whereas writes cannot be propagated speculatively. We must therefore consider reads and writes separately.

Instructions program-order-after stores are executing speculatively up until the store has reached the point in its execution where no further synchronous exceptions can occur. If a store may generate a synchronous external abort, then that point is (at least) once it has propagated to memory This means that events program-order-after an unpropagated store must also be considered to be speculative until the store has propagated. For example, this forbids out-of-order propagation of writes (e.g. as in MP+po+addr).

Reads program-order-after writes are permitted to execute early, even speculatively, unless otherwise forbidden by some other part of the model, e.g. with an interposing context-synchronisation-event. For example, when there is the possibility of synchronous external aborts on a store, reads program-order-after `ISB`s program-order-after the store are forbidden from satisfying before the propagation of that store (e.g. as in R+dmb.sy+isb).

Perhaps more interestingly, if a load may generate a synchronous external abort, then program-order-later instances are speculative until the load has completed all its reads, and is non-restartable. This means that writes program-order-after that read are forbidden from executing out-of-order. This forbids interesting tests which would otherwise be allowed, namely load-buffering (LB+pos) and MP with a plain ISB after one load (MP+dmb.sy+isb) [120].

Load buffering and the out-of-thin-air problem This has an important and hitherto not well-understood impact on programming-language concurrency models. Ruling out LB enables substantially simpler design of programming language concurrency models: they can execute instructions in-order and merely keep a history of the writes seen so far, e.g. [121], and thereby avoid the notorious out-of-thin-air problem [122]. These simpler semantics support a line of model checkers for C/C++ and LLVM [123, 124, 125]. In contrast, the presence of LB seems to require significant sophistication [126, 122, 127, 128, 129, 42, 28, 130].

Chapter 12

An axiomatic model for precise exceptions

We now give a formal semantics that describes the concurrent behaviour of precise exceptions on Arm-A. We give it as an extension of the previous model of Deacon et al. [7], as was recalled in Chapter 2. The full model extended with exceptions is given in Figure 12.1.

The model is parameterised along two axes:

- ▷ `FEAT_ExS` corresponds to the feature of the same name being implemented; we do not support runtime changes of the related `SCTLR_ELx.{EIS,EOS}` fields, and so fix them as variants.
- ▷ `SEA_R` and `SEA_W` correspond to the implementation-defined choice of whether loads or stores may generate synchronous external aborts.

Most current hardware does not support `FEAT_ExS`, and moreover, we expect that most software would not use it. However, its semantics is relatively straight-forward as we understand it, and so we include it in our model.

The `SEA` variants in this model are not architecturally-defined identifiers. In fact, in the absence of actually observing a fault directly there appears no architectural way to identify the choice beyond running the litmus tests presented in Chapter 11. These two variants capture whether *any* store or load respectively, *could* generate a synchronous external abort, even though the model does not consider executions in which such aborts actually occur.

12.1 Extended candidates

To support precise exceptions, we add new events to the candidate execution:

- ▷ `TE` (take exception), and `TakeInterrupt`, and `ERET` (exception return). These correspond to the synchronisation points (whether or not they are synchronising) of taking or returning from an exception.
- ▷ `MRS` and `MSR` events for the reading and writing (respectively) of system registers, corresponding to the identically-named Arm instructions.

Exceptions and program-order Program-order includes all the events of the thread, even with interposing exceptions. That is, program-order is not discontinuous, at least for precise exceptions. We therefore include all the new events in program-order. This includes the events from instructions directly before and after taking or returning from an exception.

Interrupts While we do not model inter-processor interrupts or the generic interrupt controller, we do support precise asynchronous exceptions generally (e.g. timers).

Candidates can, at any point in thread, have an instance which does not follow from the natural intra-instruction semantics, but corresponds to pending an interrupt, i.e. setting the appropriate bit in the `ISR`. The intra-instruction semantics then will take the interrupt at the appropriate time.

For performance reasons in the executable-as-a-test-oracle implementation within `isla-axiomatic` we do not allow arbitrary interrupts, see §13.2.

```
"Arm-A exceptions"

include "cos.cat"
include "arm-common.cat"

(* might-be speculatively
   executed *)
let speculative =
  ctrl
| addr; po
| if "SEA_R"
    then [R]; po
    else 0
| if "SEA_W"
    then [W]; po
    else 0

(* context-sync-events *)
let CSE =
  ISB
| if "FEAT_ExS" & ~"EIS"
    then 0
    else TE
| if "FEAT_ExS" & ~"EOS"
    then 0
    else ERET

let ASYNC =
  TakeInterrupt

(* observed by *)
let obs = rfe | fr | co

(* dependency-ordered-before *)
let dob =
  addr | data
| speculative ; [W]
| speculative ; [ISB]
| (addr | data); rfi



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

(* barrier-ordered-before *)
let bob =
  [R] ; po ; [dmbld]
| [W] ; po ; [dmbst]
| [dmbst]; po; [W]
| [dmbld]; po; [R|W]
| [L]; po; [A]
| [A | Q]; po; [R | W]
| [R | W]; po; [L]
| [dsb]; po

(* contextually-ordered-before *)
let ctxob =
  speculative; [MSR|CSE]
| [MSR]; po; [CSE]
| [CSE]; po

(* async-ordered-before *)
let asyncob =
  speculative; [ASYNC]
| [ASYNC]; po

(* Ordered-before *)
let ob = (obs | dob | aob |
  bob | ctxob | asyncob)+

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

(* External visibility
   requirement *)
irreflexive ob as external

(* Atomic: Basic LDXR/STXR
   constraint to forbid
   intervening writes. *)
empty rmw & (fre; coe) as atomic
```

Figure 12.1: Arm-A exceptional model (grayed out parts are unchanged from the original model).

12.2 Extended relations

We expand ordered-before:

- Wherever `ctrl|(addr;po)` was used before, we also include instructions program-order-after reads or writes when in the relevant `SEA` variant. With those variants, the instructions program-order-after those events are speculative up until the memory access has completed.
- The previous model's use of `ISB` was purely for its context synchronisation effect. Accordingly, wherever `[ISB]` was used before, we include exception entry (`TE`) and exit (`ERET`), unless we are in the variant where context synchronisation on those events is disabled.
- We extend barrier-ordered-before with the `DSB` barriers. The barrier event classes are upwards-closed, so that `DSB.SY` is included in all the `dmb` events.
- We add a context-ordered-before (`ctxob`) sub-clause to the ordered-before relation, which captures the ordering of context-changing operations and context-synchronisation: namely, that context-changes and context-synchronisation cannot happen speculatively; that all context-changes are ordered before any context-synchronisation; and that no instruction program-order-after context-synchronisation can be executed until the synchronisation is complete.
- We add an async-ordered-before (`asyncob`) clause to ordered-before, capturing that asynchronous events (such as interrupts) cannot be done speculatively, and instructions program-order-after them may not happen before the asynchronous event which precipitated them.

12.3 Challenges in defining precision

The phenomena we described in §11.3 highlight how the historical definition of precision does not account for relaxed memory. The problem is then *how to adequately define precision in a relaxed-memory setting*. This challenge is hinted at in the way the Arm reference manual [81, D1.3.1, p5355] defines precision as:

> An exception is *precise* if on taking the exception, the hardware thread (aka processing element, PE) state and the memory system state is consistent with the PE having executed all of the instructions up to but not including the point in the instruction stream where the exception was taken from, and none afterwards. [except that in certain specific cases some registers and memory values may be UNKNOWN]

This definition explicitly allows various side effects of an instruction executing when an exception is taken to be visible. The details are intricate, but in outline: registers that would be written by the instruction but which are not used by it (to compute memory access addresses) can become UNKNOWN, and for instructions that involve multiple single-copy-atomic memory writes (e.g. misaligned writes and store-pair instructions), where each write might generate an exception (e.g. a translation fault), the memory locations of the writes that do not generate exceptions become UNKNOWN. These side effects could be observed by the exception handler, and the memory write side effects could be observed by other threads doing racy reads. Hardware updates to page-table access flags and dirty bits, and to performance counters, could also be observable. This means that the abstraction of a stream of instructions executed up to a given point does not account for the relaxed-memory behaviour.

Arm *classify* particular kinds of exceptions as precise or not, but all the above makes it hard to *define* in general what it means for an exception to be precise in a relaxed setting.

The ultimate architectural intent of precision is that it is sufficient to meaningfully resume execution after the exception. For example, for software that does mapping on demand, when an instruction causes a fault by accessing an address which is not currently mapped, the exception handler will map that address and return. This means that re-executing the original instruction will overwrite these UNKNOWNs, and will have ordering properties much like the original instruction would have had if the mapping had already been in place.

Our models are complete enough to reason about such cases in concrete examples. However, a general definition of precision, and the accompanying reasoning principle, would have to capture assumptions

about the exception handler and its concurrent context to ensure that they do not observe the above side effects. More straightforwardly, the above definition of what becomes UNKNOWN would have to be codified, as that is not currently in the ASL architectural pseudocode.

Exceptions may also be *imprecise*, in which case the behaviour is very loosely constrained, and the current architecture does not give well-defined guarantees in the presence of imprecise exceptions.

12.4 Scope and limitations

We do not give semantics to imprecise exceptions. It is unclear how to do so at an architectural level.

We do not define the behaviour of 'constrained unpredictable', and merely flag when it is triggered. Clarifying it will require substantial extensive discussions with Arm architects, likely affecting the wording in the architectural specifications, beyond the scope of this work. We do not model switching between Arm `FEAT_ExS` modes (§11.3.8): they are supported architecturally, but are not commonly implemented. Finally, while we believe our models correctly capture the Arm architectural intent, and that it gives a solid basis for programmers, this is not an authoritative definition of the architecture, and is subject to change.

Chapter 13

Validating the exceptions model

13.1 Validating against hardware

We extend the harness described in Chapter 10, and run a set of 55 hand-written tests on a small collection of devices: Raspberry Pi 3B+, 4B, and 5; an ODROID N2+ with an Amlogic S922X SoC; and an Apple Mac Mini with Apple M2 silicon SoC. The results from that testing can be found in Table 13.1.

We note that many devices do not observably re-order loads with respect to writes or context-synchronisation: of our devices, only the ODROID's S922X SoC had a core (its Arm Cortex-A73) which exhibit those behaviours on usermode tests. For that reason, we did not expect, except on that core, to observe re-ordering loads across exception boundaries.

We find that out-of-order execution of stores across exception boundaries is readily observable over the full range of devices. The Apple M2 is the only device which did not exhibit store-store re-ordering across an exception boundary, but it does not exhibit normal store-store re-ordering across context-synchronisation in usermode tests, so this is not unexpected.

As such, the hardware results for exceptions are disappointingly weak: the hardware we have access to is not aggressive enough in the base usermode behaviours to be expecting to observe the behaviours we seek here, and the number of runs is not substantial enough to draw any inferences on the envelope of behaviours they implement. Further hardware testing will be required to build the necessary confidence, especially around `FEAT_ExS` and the differences in the exception kinds.

13.2 Executable-as-a-test-oracle implementation

We implement the model as an executable-as-a-test-oracle implementation in Isla [46],

To support tests with asynchronous exceptions, we added a construct to specify a label where the exception will occur, so that Isla then pends an interrupt at that program point.

The instruction semantics we use is a translation into the Sail language of the Armv9.4-A ASL specification, including the top-level function provided by Arm [112]. The translation process [45] is mostly automatic, requiring select manual interventions mostly due to differences in the type systems of ASL and Sail. We also added patches to support the integration with Isla, in particular adding hooks to expose information about exceptions being taken in a form that can be readily consumed by Isla. In doing so, we encountered and fixed some bugs in the ASL model related to uses of uninitialised fields in data structures, as well as missing checks for implemented processor features that led to spurious system register accesses.

The results from the model over each of the variants can be found in Table 13.2.

Table 13.1: Exceptions hardware refs. Columns are, respectively, an ODROID N2+ (Amlogic S99X, 'big' cores only, Arm Cortex-A73 r0p2), an Apple M2, and Raspberry Pis 3B+ (Arm Cortex-A53 r0p4), 4B (Arm Cortex-A72 r0p3), and 5 (Arm Cortex-A76 r1p4).

Name	s922x	m2	rpi3b+	rpi4b	rpi5
LB+svc-dmb-erets	0/18M	0/360M	0/1M	0/19M	0/11M
LB+svc-erets	0/18M	0/360M	0/1M	0/19M	0/11M
LB+svcs	388/18M	0/360M	0/1M	0/19M	0/11M
MP+dmb+ctrl-eret	0/18M	0/360M	0/1M	0/19M	0/11M
MP+dmb+ctrl-rfisvceret-addr	0/18M	0/360M	0/22M	0/108M	0/39M
MP+dmb+ctrl-svc	0/18M	0/360M	0/1M	0/19M	0/11M
MP+dmb+ctrlelr	0/18M	0/360M	0/22M	0/108M	0/39M
MP+dmb+data-svc	0/18M	0/360M	0/1M	0/19M	0/11M
MP+dmb+dmb-eret	0/18M	0/360M	0/1M	0/19M	0/11M
MP+dmb+eret	0/18M	0/360M	0/1M	0/19M	0/11M
MP+dmb+eret-dmb	0/18M	0/360M	0/1M	0/19M	0/11M
MP+dmb+eret=addr	0/18M	0/0	0/1M	0/19M	0/11M
MP+dmb+svc	84/18M	0/360M	0/1M	0/19M	0/11M
MP+dmb+svc-addreret	0/18M	0/360M	0/1M	0/19M	0/11M
MP+dmb+svc-dmb	0/18M	0/360M	0/1M	0/19M	0/11M
MP+dmb+svc-dmb-eret	0/18M	0/360M	0/1M	0/19M	0/11M
MP+dmb+svc-eret	0/18M	0/360M	0/33M	0/19M	0/11M
MP+dmb+svcnoeis	23/18M	0/360M	0/1M	0/19M	0/11M
MP+eret+addr	22K/18M	0/360M	63/1M	0/19M	2/11M
MP+eret+dmb	18K/18M	0/360M	1K/33M	262/19M	20/11M
MP+eret+svc	9K/18M	0/360M	244/21M	256K/107M	20/39M
MP+erets	29K/18M	0/360M	30/1M	59/19M	16/11M
MP+svc+addr	17K/18M	0/360M	59/1M	0/19M	8/11M
MP+svc+dmb	18K/17M	0/360M	80/1M	3/19M	876/11M
MP+svc+eret	22K/17M	0/360M	1K/21M	33/107M	77/39M
MP+svc-W-eret-W+addr	14K/13M	0/0	0/0	0/16M	1/6M
MP+svc-dmb+addr	0/17M	0/360M	0/1M	0/19M	0/11M
MP+svc-dmb-eret+addr	0/17M	0/360M	0/1M	0/19M	0/11M
MP+svc-eret+addr	13K/17M	0/360M	52/1M	0/19M	2/11M
MP+svc-erets	3K/17M	0/360M	42/1M	2/19M	8/11M
MP+svcs	8K/17M	0/360M	31/1M	0/19M	20/11M
MP.EL1+dmb+ctrlvbarsvc	0/17M	0/360M	0/21M	0/108M	0/39M
MP.EL1+dmb+svc	29/17M	0/360M	0/33M	0/12M	0/11M
S+dmb+eret	0/17M	0/360M	0/33M	0/12M	0/11M
S+dmb+svc	0/17M	0/360M	0/33M	0/12M	0/11M
S+erets	0/17M	0/360M	0/1M	0/19M	0/11M
S+svc-dmb-erets	0/17M	0/359M	0/1M	0/19M	0/11M
S+svc-erets	0/17M	0/359M	0/1M	0/19M	0/11M
S+svcs	0/17M	0/359M	0/1M	0/19M	0/11M
SB+dmb+eret	38/17M	12K/359M	162K/33M	85K/12M	2K/11M
SB+dmb+rfi-ctrl-eret	17K/17M	10K/359M	10K/1M	42K/19M	46/11M
SB+dmb+rfi-ctrl-svc	13K/17M	8/359M	15K/1M	2K/18M	58K/11M
SB+dmb+rfieret-addr	16K/16M	6K/359M	591K/21M	8K/107M	54/39M
SB+dmb+rfisvc-addr	18K/16M	12/359M	839K/21M	2K/106M	135K/39M
SB+dmb+svc	195/16M	14/359M	351K/33M	458/11M	63K/11M
SB+svc-dmb-erets	0/16M	0/359M	0/1M	0/18M	0/11M
SB+svc-erets	9K/16M	0/359M	22K/1M	7K/18M	38/11M
SB+svcs	2K/16M	0/359M	534K/21M	0/106M	646K/39M
SEA_R_detect	0/16M	359M/359M	0/1M	0/18M	0/10M
SEA_W_detect	0/16M	0/359M	0/1M	0/18M	0/10M
MP+dmb+eret-svc	0/4M	0/360M	0/1M	0/3M	0/5M
MP.EL1+dmb+eret	0/4M	0/360M	0/1M	0/3M	0/5M
MP.EL1+dmb+eret-svc	0/4M	0/360M	0/1M	0/3M	0/5M
MP.EL1+dmb+svc-eret	0/4M	0/360M	0/1M	0/3M	0/5M
SB.EL1+erets	15K/3M	0/359M	25K/1M	148/2M	43/5M
SB.EL1+svc-erets	3K/3M	0/359M	19K/1M	1K/2M	17/5M

Table 13.2: Exceptions model refs

Name	No features	FEAT_ExS	SEA_R	SEA_W	SEA_R&W
LB+svc-dmb-erets	forbid	forbid	forbid	forbid	forbid
LB+svc-erets	allow	allow	forbid	allow	forbid
LB+svcs	allow	allow	forbid	allow	forbid
MP+daifset+dmb	allow	allow	allow	forbid	forbid
MP+dmb+ctrl-eret	forbid	allow	forbid	forbid	forbid
MP+dmb+ctrl-rfisvceret-addr	forbid	allow	forbid	forbid	forbid
MP+dmb+ctrl-svc	forbid	allow	forbid	forbid	forbid
MP+dmb+ctrlelr	forbid	allow	forbid	forbid	forbid
MP+dmb+daifset	allow	allow	allow	allow	allow
MP+dmb+dmb-eret	forbid	forbid	forbid	forbid	forbid
MP+dmb+eret-dmb	forbid	forbid	forbid	forbid	forbid
MP+dmb+eret-svc	allow	allow	forbid	allow	forbid
MP+dmb+eret	allow	allow	forbid	allow	forbid
MP+dmb+eret=addr	forbid	forbid	forbid	forbid	forbid
MP+dmb+svc-addreret	allow	allow	forbid	allow	forbid
MP+dmb+svc-dmb-eret	forbid	forbid	forbid	forbid	forbid
MP+dmb+svc-dmb	forbid	forbid	forbid	forbid	forbid
MP+dmb+svc-eret	allow	allow	forbid	allow	forbid
MP+dmb+svc	allow	allow	forbid	allow	forbid
MP+dmb+svcnoeis	allow	allow	forbid	allow	forbid
MP+eret+addr	allow	allow	allow	forbid	forbid
MP+eret+dmb	allow	allow	allow	forbid	forbid
MP+eret+svc	allow	allow	allow	allow	forbid
MP+erets	allow	allow	allow	allow	forbid
MP+svc+addr	allow	allow	allow	forbid	forbid
MP+svc+dmb	allow	allow	allow	forbid	forbid
MP+svc+eret	allow	allow	allow	allow	forbid
MP+svc-dmb+addr	forbid	forbid	forbid	forbid	forbid
MP+svc-dmb-eret+addr	forbid	forbid	forbid	forbid	forbid
MP+svc-eret+addr	allow	allow	allow	forbid	forbid
MP+svc-erets	allow	allow	allow	allow	forbid
MP+svcs	allow	allow	allow	allow	forbid
MP.EL1+dmb+ctrlvbarsvc	forbid	allow	forbid	forbid	forbid
MP.EL1+dmb+eret-svc	allow	allow	forbid	allow	forbid
MP.EL1+dmb+eret	allow	allow	forbid	allow	forbid
MP.EL1+dmb+svc-eret	allow	allow	forbid	allow	forbid
MP.EL1+dmb+svc	forbid	forbid	forbid	forbid	forbid
S+dmb+eret	allow	allow	forbid	allow	forbid
S+dmb+svc	allow	allow	forbid	allow	forbid
S+erets	allow	allow	allow	allow	forbid
S+svc-dmb-erets	forbid	forbid	forbid	forbid	forbid
S+svc-erets	allow	allow	allow	allow	forbid
S+svcs	allow	allow	allow	allow	forbid
SB+daifsets	allow	allow	allow	allow	allow
SB+dmb+eret	allow	allow	allow	forbid	forbid
SB+dmb+rfi-ctrl-eret	allow	allow	allow	forbid	forbid
SB+dmb+rfi-ctrl-svc	allow	allow	allow	forbid	forbid
SB+dmb+rfieret-addr	allow	allow	allow	forbid	forbid
SB+dmb+rfisvc-addr	allow	allow	allow	forbid	forbid
SB+dmb+svc	allow	allow	allow	forbid	forbid
SB+svc+dmb-erets	forbid	forbid	forbid	forbid	forbid
SB+svcs	allow	allow	allow	forbid	forbid
SB.EL1+erets	allow	allow	allow	forbid	forbid
SB.EL1+svc-erets	allow	allow	allow	forbid	forbid
MP+dmb+ctrl-int	forbid	forbid	forbid	forbid	forbid
MP+dmb+int	allow	allow	allow	allow	allow
MP+int+dmb	allow	allow	allow	allow	allow

Chapter 14

Conclusion

We presented models for three key parts of the Arm architecture required for systems software: instruction fetch and its required cache maintenance instructions; virtual memory and its required TLB maintenance instructions; and the baseline behaviour for precise exceptions. 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 have clarified the architectural intent for those tests, and produced models that capture that intent, as we understand it at the time of writing.

We produced axiomatic-style declarative semantics, in the standard cat language, for all three aspects of the architecture. Additionally we produced a microarchitectural-style operational semantics for the instruction fetch fragment intended equivalent to the axiomatic one.

We validated these models against a variety of hardware implementations, even finding some places where modern microprocessors deviate from the desired architectural intent. For instruction fetch, with Maranget we extended the herdtools suite to be able to generate new litmus tests, and ran those tests on hardware. We built a brand new test harness, `system-litmus-harness`, able to run tests on a variety of hardware at EL1, either bare metal or in KVM. We used this harness to produce experimental data for both the virtual memory and exceptions parts.

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 the `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 that the model is intended to have.

14.1 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 three models for three separate parts of the architecture, but did not merge them together into a single architectural model. The models can be unioned together to produce a combined model with all the events and relations from the models, but 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 subtly different from the normal translations with different caching rules; translation and instruction fetch can both cause exceptions to happen; exceptions cause the control-flow to change and new instructions to be fetched; and so on. We do not imagine this is a particularly arduous or complex task, but one that we have not yet done.

We produced two new separate languages for defining litmus tests. Ideally, we would have one unified language that all tools (litmus, isla-axiomatic, and system-litmus-harness) all accept. 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.

There are some places where it has become known that models presented in this work do not faithfully capture the architectural intent as it is known today. In particular around the reachability of pagetable entries, and invalidation of non-last-level pagetable entries, as was discussed earlier.

14.2 Future work

There are many areas where the work presented here is only the start, and where further effort could bear fruit.

For more confidence in the architectural intent, more hardware testing (especially for the virtual memory tests) is essential. In particular, running at EL2 (for stage 2 tests), and over a more varied collection of devices.

Capturing more of the systems architecture is always desirable. We made a start here, but this is no means the end. Modern systems software relies on much more of the architecture than just covered here, such as: the Arm generic interrupt controller, and virtualisation of interrupts; the variety of Arm features and extensions for virtual memory e.g. `FEAT_ETS2`, `FEAT_BBM`, `FEAT_nTLBPA`, access permissions, and cacheability, and shareability domains; device memory and DMA; and much more.

With the models themselves, they can always be improved to be executable more efficiently, and the tools easier to use. `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, covering overlapping sets of features: the usermode model, which we recalled in Chapter 2; for transactional memory, Chong et al. [131]; for persistent memory, Raad et al. [89]; and the official Arm model which contains an updated usermode, memory tagging, ifetch, and some overlapping parts of the VMSA, Alglave et al. [42, 97, 107]. Simply gluing these together into a single model would not be sound, as their interactions would need to be explored, and the architectural intent clarified first. However, it seems necessary for such work to be carried out to enable future verification efforts of complex systems.

Work on relaxed systems, either on virtual memory, instruction fetching, or exceptions has not ceased at the finalization of this work. We are continuing to improve all 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 work by researchers, academics, engineers, architects, and hardware designers, to better understand the environment as it is today. This hopefully will help them 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.

Appendix A

Pocket guide to the Arm ISA

The litmus tests, as found in this thesis, use a relatively small subset of the whole ISA.

Refer to the Arm Architecture Reference Manual, Section C6 ('A64 Base Instruction Descriptions') for a more complete explanation of all the instructions.

A.1 Architectural concepts

Some terminology:

- ▷ AArch64 is the 64-bit execution mode.
- ▷ A64 is the name of the 64-bit ISA which AArch64 executes.
- ▷ PE ('Processing Element') is generic Arm terminology for a hardware thread/core.
- ▷ GPR ('General-purpose register') is one of the 31 'general-purpose' registers.
- ▷ Immediate values are literal numeric values used in the instructions, as opposed to being read from registers.

Exception levels Arm execution is split into privilege levels (called *exception* levels in Arm), labelled from `EL0` (least privileged execution) to `EL3` (most privileged), see Fig A.1.

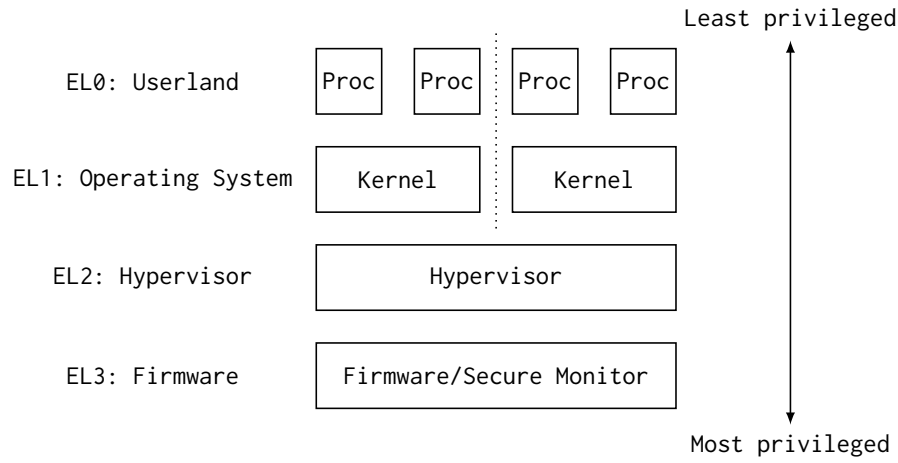


Figure A.1: Arm-A exception levels.

Registers A64 has:

- ▷ 31 general-purpose registers, named `R0`–`R30`.

  - `Rn` is an internal name, the register should be accessed via one of its aliases: `Xn` or `Wn` (see Fig A.2).
  - `X0`–`X30` are aliases for the whole 64-bit bitvector stored in `R0-R30`.
  - `W0`–`W30` are aliases for the least-significant 32-bit vector stored in `R0-R30`.

▷ a stack pointer, `SP`.

  - `WSP` is an alias for the least-significant 32-bit vector of `SP`.

▷ a program counter register, named `PC`, not directly accessible by software.

▷ a collection of 'special-purpose registers' which generally store some processor state, e.g.

  - `NZCV`, the flag register.
  - `DAIF`, interrupt mask register.
  - `CurrentEL`, the current exception level register.
  - ...

▷ a collection of 'system registers' which are generally configuration and identification registers, which control how the machine executes, e.g.

  - `SCTLR_EL1` the system configuration register, for EL1 and below.
  - `CTR_EL0` the cache-type identification register, accessible from EL0.
  - ...

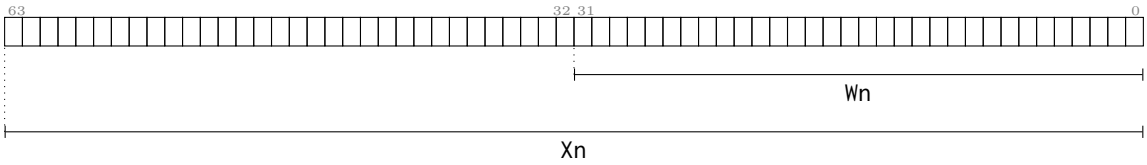


Figure A.2: Views of general-purpose register `Rn`.

A.2 Guide to Instructions

Chapter contents

A.2.1 Branches

Branches are those instructions which write to the PC register.

B

```
B <LABEL>
```

Jumps to a given label.

Example The following code branches to label L2, then writes 2 to general-purpose register R0:

```
1  b L2
2  L1:
3    MOV X0,#1
4    RET
5  L2:
6    MOV X0,#2
7    RET
```

Example Labels can be numeric, and branches to them can be suffixed with `f` ('forward') or `b` ('back'). e.g. this code has the control-flow-graph shown on the right:

```
1  0:
2    B 0f
3  1:
4    B 1f
5  0:
6    B 1b
7  0:
8    RET
9  1:
10   B 0b
```

B.cond

```
B.<COND> <LABEL>
```

Jumps to the given label, if the condition given is true.

Conditions The conditions are based on the current value of the condition register, `NZCV`, which are set by condition instructions (e.g. `CMP`), and then the `<COND>` is one of:

- `eq`: `Z==1`
- `ne`: `Z==0`
- `gt`: `N==V && Z==0`
- `lt`: `N!=V`

Example This program returns 0, as the values are unequal.

```
1  MOV X0,#13
2  MOV X1,#11
3  CMP X0,X1
4  // now NZCV=={0,0,1,0}
5  B.eq Lequal
6  Ldifferent:
7    MOV X0,#0
8    RET
9  Lequal:
10   MOV X0,#1
11   RET
```

BL and RET

```
BL <LABEL>
RET
```

Branch-and-link (aka 'call') and return. `BL` jumps to given label, saving the return location (current value of `PC` + 4), to register `X30`. `RET` then returns, by branching to the location stored in general-purpose register `X30`.

Example The following example returns 1,2,3 to registers `R0`, `R1` and `R2`, respectively. Note that the general-purpose register `X30` is overwritten by `BL`, so the following example explicitly saves and restores the value to some arbitrarily-picked general-purpose registers.

```
1  MOV X20,X30
2  BL f
3  MOV X30,X20
4  RET
```

```
5  f:
6    MOV X0,#1
7    MOV X21,X30
8    BL g
9    MOV X30,X21
10   RET
11 g:
12   MOV X1,#2
13   MOV X22,X30
14   BL h
15   MOV X30,X22
16   RET
17 h:
18   MOV X2,#3
19   RET
```

BR and BLR

```
BR <GPR>
```

Branches to an address in the given general-purpose register. The address is absolute (not PC-relative).

```
BLR <GPR>
```

Branch-and-link register, behaves as BL as before, but jumps to the address stored in the register rather than to a label.

Example The following code places the address of label L into the register R0 using the ADR instruction, then branches to the label using the stored address, and writes 1 to register R1:

```
1  ADR X0,L
2  BR X0
3  L:
4    MOV X1,#1
5    RET
```

CBZ and CBNZ

```
CBZ <GPR>, <LABEL>
CBNZ <GPR>, <LABEL>
```

Jumps to given label, if the value in the given general-purpose register is zero (or *not* zero if CBNZ).

Example The following code returns with 3 in R3, since X0 is zero (so the first CBNZ) is not taken, X1 is not zero (so the first CBZ) is not taken, but X2 is zero so the final CBZ is taken, and label L2 is branched to, and the fallthrough case is missed:

```
1  MOV X0,#0
2  MOV X1,#1
3  MOV X2,#0
4
5  CBNZ X0,L0
6  CBZ X1,L1
7  CBZ X2,L2
8
9  // (fallthrough case)
10 MOV X3,#0
11 RET
12
13 L0:
14   MOV X3,#1
15   RET
```

```
16  L1:
17    MOV X3,#2
18    RET
19  L2:
20    MOV X3,#3
21    RET
```

A.2.2 Comparisons

For use with the `B.cond` instruction. These instructions write to the `NZCV` flag register.

CMP

```
CMP <GPR0>, <GPR1>
CMP <GPR0>, #<IMM>
```

Subtracts the value stored in the second argument (either from a general purpose register or an immediate value) from the value stored in the first general purpose register, setting the flag register.

Example At the end of this program the flag registers are set such that `NZCV=={0,1,0,0}` i.e. the result is not-**n**egative, it is **z**ero, no **c**arry, and no o**v**erflow.

```
1  MOV X0,#100
2  CMP X0,X0
```

Example At the end of this program the flag registers are set such that `NZCV=={1,0,0,0}` i.e. the result is **n**egative, not **z**ero, no **c**arry, and no o**v**erflow.

```
1  MOV X0,#1
2  MOV X1,#2
3  CMP X0,X1
```

Example At the end of this program the flag registers are set such that `NZCV=={1,0,0,1}` i.e. the result is **n**egative, not **z**ero, no **c**arry, and it o**v**erflowed.

```
1  MOV X0,#0
2  NEG X0,X0
3  CMP X0,#1
```

A.2.3 Register moving and arithmetic

MOV

```
MOV <GPR0>, <GPR1>
MOV <GPR0>, #<IMM>
MOV <GPR0>, #<IMM>, LSL #<IMM>
```

Copies a value stored in the second argument (either in a general-purpose register or a 16-bit immediate value) into the first argument.

Optionally, the immediate value can be shifted left a multiple of 16.

Example At the end of this program `X0` contains the value 2, and `X1` contains the value 1.

```
1  MOV X0,#1
2  MOV X1,#2
3  MOV X2,#3
4  MOV X2,X0
5  MOV X0,X1
6  MOV X1,X2
```

MRS and MSR

```
MSR <SYSREG>, <GPR>
MRS <GPR>, <SYSREG>
```

Writes (`MSR`) or reads (`MRS`) a system (or special-purpose) register.

Example This program sets bits 31-28 to one in the flags register then reads the `CTR_EL0` identification register into general-purpose register `R1` (note the flags are not relevant here, this is just an example register):

```
1 MOV X0,#0xf000 LSL 16
2 MSR NZCV,X0
3 MRS X1,CTR_EL0
```

ADD

```
ADD <GPR0>, <GPR1>, <GPR2>
ADD <GPR0>, <GPR1>, #<IMM>
```

Adds the values stored in the second and third arguments together, and stores the result in the register given as the first argument.

Examples It is common to pass the same register as input and output to do an increment:

```
1 MOV X0,#1
2 ADD X0,X0,#1
3 // {R0==2}
```

Simple addition:

```
1 MOV X0,#1
2 MOV X1,#2
3 ADD X2,X0,X1
4 // {R2==3}
```

EOR

```
EOR <GPR0>, <GPR1>, <GPR2>
EOR <GPR0>, <GPR1>, #<IMM>
```

Exclusive-or. Does a bitwise exclusive or on the values stored in the second and third arguments, and writes the result to the register given as the first argument.

Examples `EOR` behaves as a bitwise XOR over integers:

```
1 MOV X0,#3
2 MOV X1,#5
3 EOR X2,X0,X1
4 // {X2==6}
```

Exclusive-or'ing a register with itself zeroes it:

```
1 MOV X0,#13
2 EOR X0,X0,X0
3 // {X0==0}
```

LSL and LSR

```
LSL <GPR0>, <GPR1>, <GPR2>
LSL <GPR0>, <GPR1>, #<IMM>

LSR <GPR0>, <GPR1>, <GPR2>
LSR <GPR0>, <GPR1>, #<IMM>
```

Logical shift left/right. Shifts the value in the second argument by the amount in the third argument, and stores the result in the general-purpose register named as the first argument.

Examples Left-shifts are multiplication by 2:

```
1 MOV X0,#1
2 LSL X1,X0,12
3 // {X1==4096}
```

Right shifts are floor division by 2:

```
1 MOV X0,#5
2 LSR X1,X0,1
3 // {X1==2}
```

A.2.4 Memory accesses

LDR

```
LDR <GPR0>, [<GPR1>]
LDR <GPR0>, [<GPR1>, #<IMM>]

LDRB <Wn>, [<GPR1>]
LDRB <Wn>, [<GPR1>, #<IMM>]
```

Reads the value at the memory address stored in the register `<GPR1>`, and stores the value in register `<GPR0>`.

Optionally, an offset to the address can be provided as an immediate value.

There are also address register writeback versions of these instructions, see the full manual.

NOTE: if the first argument is a 32-bit alias `Wn` then a 32-bit value is read from memory, if the first argument is a 64-bit alias `Xn` then a 64-bit value is read from memory. If the mnemonic is `LDRB` then it loads a single byte, and the register must be a 32-bit alias.

STR

```
STR <GPR0>, [<GPR1>]
STR <GPR0>, [<GPR1>, #<IMM>]

STRB <Wn>, [<GPR1>]
STRB <Wn>, [<GPR1>, #<IMM>]
```

Writes the value stored in register named by the `<GPR0>` argument, into the memory address stored in the register `<GPR1>`.

Optionally, an offset to that address can be provided as an immediate value.

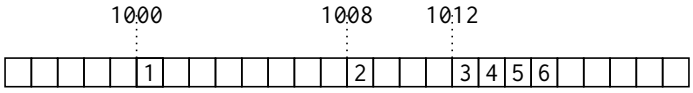
NOTE: if the first argument is a 32-bit alias `Wn` then a 32-bit value is written to memory, if the first argument is a 64-bit alias `Xn` then a 64-bit value is written to memory.

Example The following code writes the 1 as a 64-bit vector to address 1000, the value 2 as a 32-bit vector to address 1004, and the values 3,4,5 and 6 to addresses 1008,1009,1010, and 1011.

```
1 MOV X0,#1000
2 MOV X1,#1
3 MOV W2,#2
4 MOV W3,#3
5 MOV W4,#4
6 MOV W5,#5
7 MOV W6,#6
8 STR X1,[X0]
```

```
 9  STR W2,[X0,#8]
10  STRB W3,[X0,#12]
11  STRB W4,[X0,#13]
12  STRB W5,[X0,#14]
13  STRB W6,[X0,#15]
```

Resulting in memory like (noting Arm is little-endian by default):



LDP and STP

```
LDP <GPR0>, <GPR1>, [<GPR2>]
STP <GPR0>, <GPR1>, [<GPR2>]
```

Load and store pair variants of the load and store instructions. These read or write from two adjacent 32- or 64-bit locations, starting at the address stored in `<GPR2>`, using two separate general-purpose registers for the data.

A.2.5 Barriers

DMB

```
DMB.<KIND>
```

Data memory barrier.

Arm categorise the kinds into two partitions:

- ▷ The *types*: whether this orders reads or writes or both.
- ▷ The *domain*: whether the effect is visible to just this core, or all.

A sample of the kinds used in litmus tests are given below:

Kind	Types	Domain
SY	RW.RW	Full system
ISH	RW.RW	Full system
ST	W.W	Full system
LD	R.RW	Full system

See the full Arm architecture reference manual for the rest.

DSB

```
DSB.<KIND>
```

Data synchronisation barrier. Like a `DMB`, but affects (some) implicit memory effects, too.

Arm categorise the kinds two ways:

- ▷ The *types*: whether this orders reads or writes or both.
- ▷ The *domain*: whether the effect is visible to just this core, or all.

A sample of the kinds used in litmus tests are given below:

Kind	Types	Domain
SY	RW.RW	Full system
ISH	RW.RW	Full system
ST	W.W	Full system
LD	R.RW	Full system
NSH	RW.RW	This CPU only

See the full Arm architecture reference manual for the rest.

ISB

```
ISB
```

Instruction synchronisation barrier.

A.2.6 Cache maintenance

DC

```
DC <OP>, <GPR>
```

Data Cache maintenance by address. Performs the cache maintenance operation `OP` to the address stored in the given general-purpose register.

A sample of the kinds used in litmus tests are given below:

Kind	Clean/Invalidate	To
`CVAU`	Clean	Point of Unification
`CVAC`	Clean	Point of Coherency
`CIVAC`	Clean&Invalidate	Point of Coherency

See the full Arm architecture reference manual for the rest.

IC

```
IC <OP>, <GPR>
```

Instruction Cache maintenance by address. Performs the cache maintenance operation `OP` to the address stored in the given general-purpose register.

A sample of the kinds used in litmus tests are given below:

Kind	Clean/Invalidate	To
`IVAU`	Invalidate	Point of Unification
`IVAC`	Invalidate	Point of Coherency

See the full Arm architecture reference manual for the rest.

```
IC <ALLOP>
```

Instruction Cache maintenance, not by address.

`ALLOP` can be one of:

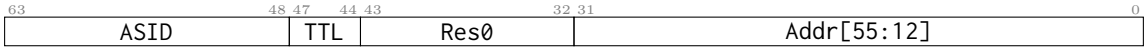
Op	Clean/Invalidate	To	Domain
`IALLU`	Invalidate	Point of Unification	This CPU only
`IALLUIS`	Invalidate	Point of Unification	All CPUs

A.2.7 TLB maintenance

TLBI-by-address

```
TLBI <OP>, <GPR>
```

TLB maintenance, by page number stored in the general-purpose register given as argument:



Encoding of `TLBI`-by-Address argument register.

A sample of `TLBI`-by-Address operations:

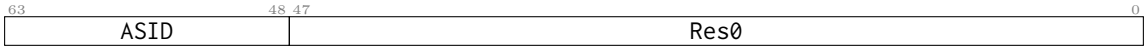
- ▷ `VAE1`: by virtual address and ASID, for the EL1&0 regime, for this PE.
- ▷ `VAE1IS`: by virtual address and ASID, for the EL1&0 regime, for all PEs.
- ▷ `VAAE1`: by virtual address, for all ASIDs, for the EL1&0 regime, for this PE.
- ▷ `VAAE1IS`: by virtual address, for all ASIDs, for the EL1&0 regime, for all PEs.

- ▷ VAE2: by virtual address and ASID, for the EL2 regime, for this PE.
- ▷ IPAS2E1: by intermediate physical address, for the current VMID, for the EL1&0 regime, for this PE, second stage only.
- ▷ IPAS2E1IS: by intermediate physical address, for the current VMID, for the EL1&0 regime, for all PEs, second stage only.

TLBI-by-ASID

```
TLBI ASIDE1, <GPR>
TLBI ASIDE2, <GPR>
```

TLB maintenance, for an ASID stored in the general-purpose register given as argument:



Encoding of TLBI-by-ASID argument register.

TLBI-ALL

```
TLBI <OP>
```

TLB maintenance, for an ASID stored in the general-purpose register given as argument:

A sample of TLBI-ALL operations:

- ▷ ALLE1: for any ASID, any VMID, stage 1 and stage2, for the EL1&0 regime, this PE only.
- ▷ ALLE1IS: for any ASID, any VMID, stage 1 and stage2, for the EL1&0 regime, for all PEs.
- ▷ ALLE2: for any ASID, for the EL2 regime, for this PE.
- ▷ VMALLE1IS: for any ASID, for the current VMID, for stage1, for the EL1&0 regime, for all PEs.
- ▷ VMALLS12E1IS: for any ASID, for the current VMID, for stage1 and stage2, for the EL1&0 regime, for all PEs.

A.2.8 Exceptions

SVC and ERET

```
SVC #<IMM>
```

Take an exception right here. Saves the current processor state (current exception level, flags, etc. but not register values) to the saved processor status register (SPSR_ELn) and then jumps to the address stored in the vector base address register (VBAR_ELn). Jumps to the address stored in the exception link register (ELR_ELn), and restores the processor status which was saved on taking the exception (in the SPSR_ELn).

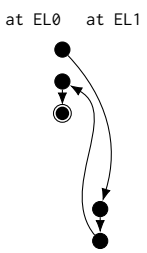
The immediate value is stored in the exception syndrome register, which software can read.

```
ERET
```

Return from exception. Jumps to the address stored in the exception link register (ELR_ELn), and restores the processor status which was saved on taking the exception (in the SPSR_ELn).

Example Execution jumps between process and the exception handler, as shown by the control-flow-graph on the right, with columns showing the current exception level.

```
1  MOV X0,#1
2  SVC #0
3  MOV X2,#3
4  RET
5
6  at_VBAR:
7    MOV X1,#2
8    ERET
```



Appendix B

The (i)Flat model

This Appendix is based on: ARMv8-A system semantics: instruction fetch in relaxed architectures [35] 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).

This appendix reproduces, in full, the Flat model with extensions for instruction fetching (see Chapter 4).

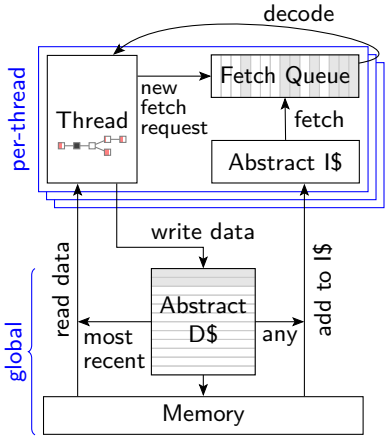
Since the instruction-fetching parts are additive, it can also serve as a reference for the original Flat model of [7], by simply ignoring the ifetch-specific parts.

To help reading this document we have colour-coded some text as follows:

▷ [ifetch] Instruction fetch and cache maintenance instructions

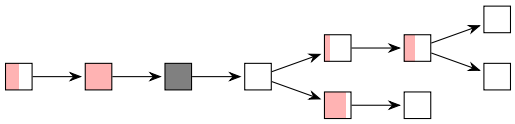
The operational model is expressed as a state machine, with states that are an abstract representation of hardware machine states. We first introduce the model states and transitions informally.

Model states A model state consists just of a shared memory and a tuple of thread model states:



The shared memory state effectively just records the most recent write to each location, together with some additional data for exclusives. To handle instruction fetching, the shared memory is extended with a data cache buffer of all the writes still visible to instruction fetches. Each thread is extended with an instruction cache that can be fetched from and fetch queue of buffered pre-fetched instructions.

Each thread model state consists principally of a list or tree of instruction instances, some of which have been finished, and some of which have not. Below we show an example for a thread that is executing 10 instruction instances. Some (grey) are finished; others (pink) have run some but perhaps not all of their instruction semantics; instructions are not necessarily atomic. Those with multiple children are branch instructions with multiple potential speculative successors being explored simultaneously.



Non-finished instruction instances can be subject to restart, e.g. if they depend on an out-of-order or speculative read that turns out to be unsound. The finished instances are not necessarily contiguous: in the example, the third is finished even though its predecessors are not, which can only happen if they are sufficiently independent. Instruction instances with multiple children are conditional branches for which the thread has fetched multiple possible successors. When a conditional branch is finished, any un-taken alternative paths are discarded, and instruction instances that follow (in program order) a non-finished conditional branch cannot be finished until that conditional branch is. One can choose whether or not to allow simultaneous exploration of multiple successors of a conditional branch (as shown above); this does not affect the set of allowed outcomes.

The intra-instruction behaviour of a single instruction can largely be treated as sequential (but not atomic) execution of its ASL/Sail pseudocode. Each instruction instance state includes a pseudocode execution state, which one can think of as a representation of the pseudocode control state, pseudocode call stack, and local variable values. An instruction instance state also includes information, detailed below, about the instruction instance's memory and register footprints, its register and memory reads and writes, whether it is finished, etc.

Model transitions For any state, the model defines the set of allowed transitions, each of which is a single atomic step to a new abstract machine state. Each transition arises from the next step of a single instruction instance; it will change the state of that instance, and it may depend on or change the rest of its thread state and/or the shared memory state. Instructions cannot be treated as atomic units: complete execution of a single instruction instance may involve many transitions, which can be interleaved with those of other instances in the same or other threads, and some of this is programmer-visible. The transitions are introduced below and defined in §B.0.4, with a precondition and a construction of the post-transition model state for each. The transitions labelled ○ can always be taken eagerly, as soon as they are enabled, without excluding other behaviour; the – cannot.

Transitions for all instructions:

- ○ [ifetch] Fetch request: This transition speculates the next address as a po-successor of a previously speculated instruction.
- ▷ Fetch instruction: Satisfy the fetch from instruction memory.
- ○ [ifetch] Decode instruction: Decode the instruction.
- ○ Register read: This is a read of a register value from the most recent program-order predecessor instruction instance that writes to that register.
- ○ Register write
- ○ Pseudocode internal step: this covers ASL/Sail internal computation, function calls, etc.
- ○ Finish instruction: At this point the instruction pseudocode is done, the instruction cannot be restarted or discarded, and all memory effects have taken place. For a conditional branch, any non-taken po-successor branches are discarded.

Load instructions:

- ○ Initiate memory reads of load instruction: At this point the memory footprint of the load is provisionally known and its individual reads can start being satisfied.
- ▷ Satisfy memory read by forwarding from writes: This partially or entirely satisfies a single read by forwarding from its po-previous writes.
- ▷ Satisfy memory read from memory: This entirely satisfies the outstanding slices of a single read, from memory.
- ○ Complete load instruction (when all its reads are entirely satisfied): At this point all the reads of the load have been entirely satisfied and the instruction pseudocode can continue execution. A load instruction can be subject to being restarted until the Finish instruction transition. In some cases it is possible to tell that a load instruction will not be restarted or discarded before that, e.g. when all the instructions po-before the load instruction are finished. The Restart condition over-approximates the set of instructions that might be restarted.

Store instructions:

- ○ Initiate memory writes of store instruction, with their footprints: At this point the memory footprint of the store is provisionally known.
- ○ Instantiate memory write values of store instruction: At this point the writes have their values and program-order-subsequent reads can be satisfied by forwarding from them.
- ○ Commit store instruction: At this point the store is guaranteed to happen (it cannot be restarted or discarded), and the writes can start being propagated to memory.
- ▷ Propagate memory write: This propagates a single write to memory.
- ○ Complete store instruction (when its writes are all propagated): At this point all writes have been propagated to memory, and the instruction pseudocode can continue execution.

Barrier instructions:

- ○ Commit barrier

Cache maintenance instructions:

- ▷ [ifetch] Begin IC: Initiate instruction cache maintenance.
- ○ [ifetch] Propagate IC to thread: Do instruction cache maintenance for a specific thread.
- ▷ [ifetch] Perform DC: Clean the abstract data cache for a specific cache line.

Instruction cache updates:

- ○ [ifetch] Add to instruction cache for thread: Update instruction cache for thread with write.

B.0.1 Intra-instruction Pseudocode Execution

To link the model transitions introduced above to the execution of the instructions an interface is needed between Sail and the rest of the concurrency model. For each instruction instance this intra-instruction semantics is expressed as a state machine, essentially running the instruction pseudocode, where each pseudocode execution state is a request of one of the following forms:

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

Each of these states is a suspended computation with a request for an action or input from the concurrency model and, except in the case of DONE, a continuation for the remaining execution.

Here memory values are lists of bytes, addresses are 64-bit numbers, read and write kinds identify whether they are regular, exclusive, and/or release/acquire operations, register names identify a register and slice thereof (start and end bit indices), and the continuations describe how the instruction instance will continue for any value that might be provided by the surrounding memory model. This largely follows [15, §2.2], except that memory writes are split into two steps, WRITE_EA and WRITE_MEMV. We ensure these are paired in the pseudocode, but there may be other steps between them: it is observable that the WRITE_EA can occur before the value to be written is determined, because the potential memory footprint of the instruction becomes provisionally known then.

We ensure that each instruction has at most one memory read, memory write, or barrier step, by rewriting the pseudocode to coalesce multiple reads or writes, which are then split apart into the architecturally atomic units by the thread semantics; this gives a single commit point for all memory writes of an instruction.

Each bit of a register read should be satisfied from a register write by the most recent (in program order) instruction instance that can write that bit, or from the thread's initial register state if there is no such.

That instance may not have executed its register write yet, in which case the register read should block. The semantics therefore has to know the register write footprint of each instruction instance, which it calculates when the instruction instance is created. We ensure in the pseudocode that each instruction does exactly one register write to each bit of its register footprint, and also that instructions do not do register reads from their own register writes. In some cases, but not in the fragment of ARM that we cover at present, register write footprints need to be dynamically recalculated, when the actual footprint only becomes known during pseudocode execution.

Data-flow dependencies in the model emerge from the fact that a register read has to wait for the appropriate register write to be executed (as described above). This has to be carefully handled in order not to create unintentional strength. First, for some instructions we need to ensure that the pseudocode is in the maximally liberal order, e.g. to allow early computed-address register writebacks before the corresponding memory write. Leaving load-pair aside (which we do not cover), and the treatment of the multiple reads or writes that can be associated with a single load or store instruction (which we do), we have not so far needed other intra-instruction concurrency. Second, the model has to be able to know when a register read value can no longer change (i.e. due to instruction restart). We approximate that by recording, for each register write, the set of register and memory reads the instruction instance has performed at the point of executing the write. This information is then used as follows to determine whether a register read value is final: if the instruction instance that performed the register write from which the register reads from is finished, the value is final; otherwise check that the recorded reads for the register write do not include memory reads, and continue recursively with the recorded register reads. For the instructions we cover this approximation is exact.

We express the pseudocode execution semantics in two ways: a definitional interpreter for Sail [15], with an exhaustive symbolic mode to (re)calculate an instruction's memory and register footprints, and as a shallow embedding, translating Sail into directly executable code, with separate hand-written definitions of the footprint functions. The two are essentially equivalent: the first lets one small-step through the pseudocode interactively, while the second is more efficient and should be more convenient for proof.

B.0.2 Instruction Instance States

Each instruction instance i has a state comprising:

- ▷ *program_loc*, the memory address from which the instruction was fetched;
- ▷ *instruction_kind*, identifying whether this is a load, store, or barrier instruction, each with the associated kind; or a conditional branch; or a 'simple' instruction.
- ▷ *regs_in*, the set of input *reg_name*s, as statically determined;
- ▷ *regs_out*, the output *reg_name*s, as statically determined;
- ▷ *pseudocode_state* (or sometimes just 'state' for short), one of
  - – PLAIN *next_state*, ready to make a pseudocode transition;
  - – PENDING_MEM_READS *k*, performing the read(s) from memory of a load; or
  - – PENDING_MEM_WRITES *k*, performing the write(s) to memory of a store;
- ▷ *reg_reads*, the accumulated register reads, including their sources and values, of this instance's execution so far;
- ▷ *reg_writes*, the accumulated register writes, including dependency information to identify the register reads and memory reads (by this instruction) that might have affected each;
- ▷ *mem_reads*, a set of memory read requests. Each request includes a memory footprint (an address and size) and, if the request has already been satisfied, the set of write slices (each consisting of a write and a set of its byte indices) that satisfied it.
- ▷ *mem_writes*, a set of memory write requests. Each request includes a memory footprint and, when available, the memory value to be written. In addition, each write has a flag that indicates whether the write has been propagated (passed to the memory) or not.
- ▷ information recording whether the instance is committed, finished, etc.

Read requests include their read kind and their memory footprint (their address and size), the as-yet-unsatisfied slices (the byte indices that have not been satisfied), and, for the satisfied slices, information about the write(s) that they were satisfied from. Write requests include their write kind, their memory footprint, and their value. When we refer to a write or read request without mentioning the kind of request we mean the request can be of any kind. A load instruction which has initiated (so its read request

list *mem_reads* is not empty) and for which all its read requests are satisfied (i.e. there are no unsatisfied slices) is said to be *entirely satisfied*.

B.0.3 Thread States

The model state of a single hardware thread includes:

- *thread_id*, a unique identifier of the thread;
- *register_data*, the name, bit width, and start bit index for each register;
- *initial_register_state*, the initial register value for each register;
- *initial_fetch_address*, the initial fetch address for this thread;
- *instruction_tree*, a tree or list of the instruction instances that have been fetched (and not discarded), in program order.

B.0.4 Model Transitions

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).

Action: add an unfetched entry for *loc* to the fetch queue for *i*'s thread.

Note that this allows speculation past conditional branches and calculated jumps.

Fetch instruction (ifetch) *In ifetch mode this transition replaces the original 'Fetch instruction' transition.*

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 instruction cache in a constrained-unpredictable way, if:

1. there exists a set of write-slices, each of which can be constructed in the same way as above;
2. that set contains write-slices corresponding to distinct opcodes, and at least one of those is 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.

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

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

Note that this transition is a proxy for the point the instructions will be decoded, but that it is the intra-instruction semantics that actually performs the decoding, with this transition merely starting the execution of the pseudocode.

Fetch instruction In ifetch mode this transition is replaced by Fetch instruction (ifetch).

A possible program-order successor of instruction instance i can be fetched from address *loc* if:

1. it has not already been fetched, i.e., none of the immediate successors of i in the thread's *instruction_tree* are from *loc*;
2. *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 an instruction that has performed a write to the program counter register (`_PC`), the value that was written;
   (c) for a conditional branch, either the successor address or the branch target address[1]; or
   (d) for a jump to an address which is not yet determined, any address (this is approximated in our tool implementation, necessarily); and
3. there is a decodable instruction in program memory at *loc*.

Note that this allows speculation past conditional branches and calculated jumps.

Action: construct a freshly initialized instruction instance i' for the instruction in the program memory at *loc*, including the static information available from the ISA model such as its *instruction_kind*, *regs_in*, and *regs_out*, and add i' to the thread's *instruction_tree* as a successor of i.

This involves only the thread, not the storage subsystem, as we assume a fixed program rather than modelling fetches with memory reads; we do not model self-modifying code.

Initiate memory reads of load instruction An instruction instance i with next state READ_MEM*(read_kind, address, size, k)* can initiate the corresponding memory reads. Action:

1. Construct the appropriate read requests *rrs*:
   ▷ if *address* is aligned to *size* then *rrs* is a single read request of *size* bytes from *address*;
   ▷ otherwise, *rrs* is a set of *size* read requests, each of one byte, from the addresses *address...address+size-1*.
2. set *i.mem_reads* to *rrs*; and
3. update the state of i to PENDING_MEM_READS k.

Complete load instruction (when all its reads are entirely satisfied) A load instruction instance i in state PENDING_MEM_READS k can be completed (not to be confused with finished) if all the read requests *i.mem_reads* are entirely satisfied (i.e., there are no unsatisfied slices).

Action: update the state of i to PLAIN *(read_cont (memory_value))*, where *memory_value* is assembled from all the write slices that satisfied *i.mem_reads*.

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

Initiate memory writes of store instruction, with their footprints An instruction instance i with next state WRITE_EA(*write_kind, address, size, next_state′*) can announce its pending write footprint. Action:

1. construct the appropriate write requests:
   - ▷ if *address* is aligned to *size* then *ws* is a single write request of *size* bytes to *address*;
   - ▷ otherwise *ws* is a set of *size* write requests, each of one byte size, to the addresses *address*...*address+size-1*.
2. set *i.mem_writes* to *ws*; and
3. update the state of i to PLAIN *next_state′*.

Note that at this point the write requests do not yet have their values. This state allows non-overlapping po-following writes to propagate.

Instantiate memory write values of store instruction An instruction instance i with next state WRITE_MEMV(*memory_value, k*) can initiate the corresponding memory writes. Action:

1. split *memory_value* between the write requests *i.mem_writes*; and
2. update the state of i to PENDING_MEM_WRITES *k*.

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; and
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;; and
6. [ifetch] if i is a `dsb sy` instruction, all po-previous memory access instructions, barriers, and cache maintenance instructions have finished.

Action:

1. Update the state of i to PLAIN(*next_state′*);
2. [ifetch] If i is an `isb` instruction, for any instruction instance in this thread's instruction tree, if that instruction instance is in the FETCHED state, set it to the UNFETCHED state.

Note that this corresponds to an `ISB` discarding any already-fetched entries from the fetch queue.

Satisfy memory read by forwarding from writes For a load instruction instance i in state PENDING_MEM_READS(*k*), 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; and
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*.
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*(k)*, and a read request r in *i.mem_reads*, that has unsatisfied slices, the read request can be satisfied from memory, if:

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

Action:

let *wss* be the write slices from memory or the abstract data cache, 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*(k)*, 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*(k)*, 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.
3. Add w as a complete slice to the abstract data cache.

Complete store instruction (when its writes are all propagated) A store instruction i in state PENDING_MEM_WRITES*(k)*, for which all the memory writes in *i.mem_writes* have been propagated, can be completed.

Action:

Update the state of i to PLAIN*(k(true))*.

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 instruction instance 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 be spontaneously updated with a write w from the storage subsystem, if this write (as a single slice) does not already exist in the instruction cache.

Action: Add this write (as a single slice) to thread *tid*'s instruction cache.

Register read An instruction instance i with next state *READ_REG(reg_name, k)* can do a register read if every instruction instance that it needs to read from has already performed the expected register write.

Let *read_sources* include, for each bit of *reg_name*, the write to that bit by the most recent (in program order) instruction instance that can write to that bit, if any. If there is no such instruction, the source is the initial register value from *initial_register_state*. Let *register_value* be the assembled value from *read_sources*. Action:

1. add *reg_name* to *i.reg_reads* with *read_sources* and *register_value*; and
2. update the state of i to *PLAIN (k(register_value))*.

Register write An instruction instance i with next state *WRITE_REG(reg_name, register_value, next_state′)* can do the register write. Action:

1. add *reg_name* to *i.reg_writes* with *write_deps* and *register_value*; and
2. update the state of i to *PLAIN next_state′*.

where *write_deps* is the set of all *read_sources* from *i.reg_reads* and a flag that is set to true if i is a load instruction that has already been entirely satisfied.

Pseudocode internal step An instruction instance i with next state *INTERNAL(next_state′)* can do that pseudocode-internal step. Action: update the state of i to *PLAIN next_state′*.

Finish instruction A non-finished instruction i with next state DONE can be finished if:

1. if i is a load instruction:
   (a) all po-previous `dmb sy` and `isb` instructions are finished;
   (b) it is guaranteed that the values read by the read requests of i will not cause coherence violations, i.e., for any po-previous instruction instance i', let *cfp* be the combined footprint of propagated writes from store instructions po-between i and i' and fixed writes that were forwarded to i from store instructions po-between i and i' including i', and let *cfp′* be the complement of *cfp* in the memory footprint of i. If *cfp′* is not empty:

i. i' has a fully determined memory footprint;
ii. i' has no unpropagated memory write that overlaps with cfp'; and
iii. If i' is a load with a memory footprint that overlaps with cfp', then all the read requests of i' that overlap with cfp' are satisfied and i' can not be restarted (see §B.0.5).

Here a memory write is called fixed if it is the write of a store instruction that has fully determined data.

2. i has fully determined data; and
3. all po-previous conditional branches are finished.

Action:

1. if i is a branch instruction, discard any untaken path of execution, i.e., remove any (non-finished) instructions that are not reachable by the branch taken in *instruction_tree*; and
2. record the instruction as finished, i.e., set *finished* to *true*.

B.0.5 Auxiliary Definitions

Fully determined An instruction is said to have fully determined footprint if the memory reads feeding into its footprint are finished: A register write w, of instruction i, with the associated *write_deps* from *i.reg_writes* is said to be *fully determined* if one of the following conditions hold:

1. i is finished; or
2. the load flag in *write_deps* is *false* and every register write in *write_deps* is fully determined.

An instruction i is said to have *fully determined data* if all the register writes of *read_sources* in *i.reg_reads* are fully determined. An instruction i is said to have a *fully determined memory footprint* if all the register writes of *read_sources* in *i.reg_reads* that are associated with registers that feed into i's memory access footprint are fully determined.

Restart condition To determine if instruction i might be restarted we use the following recursive condition: i is a non-finished instruction and at least one of the following holds,

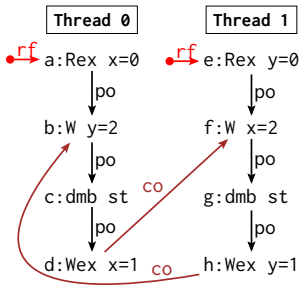
1. there exists an unpropagated write w such that applying the action of the Propagate memory write transition to s will result in the restart of i;
2. there exists a non-finished load instruction l such that applying the action of the Satisfy memory read from memory transition to l will result in the restart of i (even if l is already entirely satisfied); or
3. there exists a non-finished instruction i' that might be restarted and i is in its data-flow dependents.

Cache Line of Minimum Size Cache maintenance operations work over entire cache lines, not individual addresses. Each address is associated with at least one cache line for the data (and unified) caches, and one for the instruction caches. The cache line of minimum size is the smallest possible cache line for each of these. The `CTR_EL0.{DMinLine, IMinLine}` values 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.

B.0.6 Remarks about load/store exclusive instructions

The MCA ARMv8 architecture intends that the success bit of store exclusives does not introduce dependencies, to allow (e.g.) hardware optimisations that dynamically replace load/store exclusive pairs by atomic read-modify-write operations that can execute in the memory subsystem and therefore be guaranteed to succeed. The ARMv8-axiomatic definition assumes all address/data/control dependencies to be from reads, not writes. In the operational model, matching this weakness has proved to be difficult: it means the operational model must be able to promise the success or failure of a store-exclusive instruction even before any of its registers reads/writes have been done, so before the store-exclusive's address and data are available. The early success promises are the source of deadlocks in the operational model. To illustrate this consider, for example, the following litmus test and a state where both `a` and `e` are satisfied and finished, and where `b` and `f` are not propagated. Then `d` can promise its success, locking memory location `x`, and `h` can promise its success, locking location `y`. But now there is a deadlock:

▷ For `d` to propagate `c` has to be committed and hence `b` propagated.
But `b` cannot propagate since `y` is locked.



▷ For h to propagate g has to be committed and hence f propagated.
But f cannot propagate since x is locked.

Similar situations arise from cases where there are other barriers or release/acquire instructions in-between the load and the store exclusive, or if the store exclusive has additional dependencies that the load exclusive does not have. These are cases that are not really intended to be supported by the architecture.

The model can also currently deadlock if a load and a store-exclusive are paired successfully but later turn out to have different addresses: if the store-exclusive promises its success before its address is known it locks the matched load-exclusive's memory location; when they later turns out to be to a different addresses it never unlocks it. This issue can be fixed, but it is currently still being clarified what exactly the architecturally allowed behaviour should be.

Appendix C

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.
- MAIR_NORMAL_RA_WA (0xff): normal cacheable memory.

Finally, the harness allocator can be guided to place variables in locations with particular relationships between them (in the same page or cache line, or at the same offset into their respective regions):

- ▷ INIT_REGION_OWN(var, region): that var owns a region of memory larger than the default of a page, region can take values:
  - REGION_OWN_CACHE_LINE: this variable only takes up a single cache line.
  - REGION_OWN_PAGE: don't allocate other variables in the same page (the default).
  - REGION_OWN_PMD: don't allocate other variables in the same 2MiB region.
  - REGION_OWN_PUD: don't allocate other variables in the same 1GiB region.
- ▷ INIT_REGION_PIN(var1, var2, region): place var1 and var2 in the same region, where region is one of:
  - REGION_SAME_CACHE_LINE: place both in the same cache line.
  - REGION_SAME_PAGE: place both in same page.
  - REGION_SAME_PMD: place both same 2MiB region.
  - REGION_SAME_PUD: place both same 1GiB region.
- ▷ INIT_REGION_OFFSET(var1, var2, region): ensure that var1 and var2 have the same *offset* into the region (that is, the least significant bits overlap), where region can be one of:
  - REGION_SAME_CACHE_LINE_OFFSET: ensure both have same lower CACHE_LINE_SHIFT bits.
  - REGION_SAME_PAGE_OFFSET: ensure both have same offset into the page (bits 12-0).
  - REGION_SAME_PMD_OFFSET: ensure both have same offset into the 2MiB region (bits 20-12).
  - REGION_SAME_PUD_OFFSET: ensure both have same offset into the 1GiB region (bits 29-20).

Appendix D

Proof of virtual memory abstraction

This Appendix is based on: Relaxed virtual memory in Armv8-A [37] 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). In particular, much of the proof is the work of Jean Pichon-Pharabod.

We consider a simple case when the virtual address abstraction ought to hold: under some conditions, the model with translation and the original model without translations coincide. Here, we only consider the consistency of the pre-executions, but not how these pre-executions arise.

D.1 Abstraction

Definition 1 (VA abstraction subcondition)**.** `G` satisfies the *VA abstraction subcondition* when it has no page-table-affecting instructions: no TLBI, no context-changing operations (for example via writing to registers, for example via MSR TTBR), etc.

Definition 2 (VA abstraction condition)**.** `Gtr` satisfies the *VA abstraction condition* when it satisfies the VA abstraction subcondition, and has a static injective page table.

Theorem 1 (VA abstraction)**.** For all (`Gtr : concrete execution`)
if `Gtr` is consistent wrt. the model with translation
and respects the VA abstraction condition, then
let `Gabs` = erase `Gtr` in
`Gabs` is consistent wrt. the model without translation.

Proof. First, the builtin `addr` of the abstract model is assumed to coincide with the derived `addr` of the concrete model by the erasure. Showing that the two definitions of pre-executions do relate in this way is outside of our scope. Given that the definitions `addr` coincide, the definitions of all the other derived relations of the abstract model, including `ob` in the translation model, are syntactically supersets of their definition in the concrete model, so a cycle in `ob` in the abstract model is also a cycle in `ob` in the concrete model. □

D.2 Anti-abstraction

For this direction, we need to be able to put the translation table somewhere.

Step 1: Building the candidate execution in the translation model

Definition 3 (translation extension condition)**.** The *translation extension condition* is the data of
(`Gabs : execution`)
such that `Gabs` is consistent wrt. the model without translation
and has no TLBI, and no MSR TTBR
and

(va_space : va_address -> bool)
such that all the memory accesses of Gabs are in va_space
and
(pt_pa_space : pa_address -> bool)
(pt_initial_state : pa_address -> option (list byte)),
such that the domains of pt_pa_space and pt_initial_state coincide
and
(tr_ctxt : translation_context),
such that id_map_lifted va_space and pt_pa_space are disjoint address spaces
and
(translate : translation_function),
such that translating abstract_va_space translate-reads from within pt_pa_space and gives the injective map.

Definition 4 (translation extension)**.** Given the translation extension condition, the *translation extension* Gtr of Gabs is constructed by:

- ▷ adding all the initial writes for the page tables,
- ▷ adding all the translate reads obtained by running the translate function with the tr_ctxt,
- ▷ adding the translate reads in iio between the fetch and the explicit event,
- ▷ adding tdata to match addr,
- ▷ adding trf from the corresponding initial writes to the translates.

Definition 5 (VA anti abstraction condition)**.** Gtr satisfies the *VA anti-abstraction condition* when it is derived from a consistent execution which satisfies the VA abstraction subcondition by the translation extension.

Lemma 1 (VA abstraction condition for translation extension)**.** If Gtr satisfies the *VA anti-abstraction condition*, then Gtr satisfies the VA abstraction condition.

Proof. The translation extension does not add any extra instructions, and sets up static injective page tables. □

Lemma 2 (obtlbi-empty)**.** If Gtr satisfies the VA anti-abstraction condition, then obtlbi is empty.

Proof. obtlbi has

- ▷ obtlbi_translate which has
  - – tcache1
    which is [T & Stage1]; tfr; tseq1
    the latter is
    [W]; (maybe_TLB_barriered_by_va & ob); [TLBI VA]
    which requires a TLBI, so it is empty
  - – tcache2 & ...
    which requires a TLBI, so it is empty
  - – (tcache2; ...) & ...
    which requires a TLBI, so it is empty
- ▷ [M]; iio^-1; obtlbi_translate
  to which the same reasoning applies

□

Step 2: Consistency

Lemma 3. If Gtr satisfies the VA anti-abstraction condition, then translation-internal is acylic.

Proof. po-pa; [W]; trf is empty
because by the VA anti-abstraction condition there are no non-initial writes to page tables. □

So we only need to show `external` is acyclic.

Lemma 4 (ob-to-T)**.** If `G` satisfies the VA anti-abstraction condition, then, for all $n \geq 1$,

```
imm(ob)^n; [T] ==
  iio
  | imm(ob)^(n-1); trfe
  | imm(ob)^(n-1); [T]; iio; [T]
  | imm(ob)^(n-1); [CSE]; instruction-order
  | imm(ob)^(n-1); po; [ERET]; instruction-order; [T]
```

Proof. ▷ The `addr` clause
`| tdata; [T_f]`
is empty because there are no translation failures.

▷ `tob` does not contribute: there are no faults, and no non-initial writes to page table entries.

▷ The first clause of `ctxob` is empty because there are no `MSR TTBR`. The third and fourth are also empty, because they do not end in a `[T]`.

▷ Given a static injective mapping, the new `| (addr | data | ctrl); trfi` clause of `dob` is empty.

□

Lemma 5 (no-cycle-ob-to-init)**.** If `Gtr` is well-formed and consistent (in either model), then there is cycle in `ob` via the initial writes.

Proof. By well-formedness, `wco; [INIT] = [INIT]; wco; [INIT]`, and `wco` is acyclic.
By examination of the other edges. □

Lemma 6 (ob-from-T)**.** If `Gtr` satisfies the VA anti-abstraction condition, then

```
[T]; imm(ob) ==
  iio
  | [T]; iio; [M]; po; [W]
```

Proof. By examination of the edges. □

Lemma 7 (instruction-order-compress)**.**
`instruction-order; [T]; iio; [M]; po` $\subseteq$ `instruction-order`

Proof. If we unfold the definitions of `instruction-order` and `po`, we have
`iio^-1; fpo; iio; [T]; iio; [M]; [M|F|C]; iio^-1; fpo; iio; [M|F|C]`
which we can simplify into
`iio^-1; fpo; fpo; iio; [M|F|C]`
which means we have
`instruction-order`. □

Lemma 8 (instruction-order-compress-iio)**.** `instruction-order; iio; po` $\subseteq$ `instruction-order`

Proof. `iio` is transitive, and is the RHS of `instruction-order`. □

Lemma 9 (ob-acyclic-preserved)**.** If `G` satisfies the VA anti-abstraction condition, if there is a cycle in translate-`ob`, then there is a cycle in plain-`ob`.

Proof. Consider a minimal cycle in translate-`imm(ob)` (that is, the transitive closure of the `ob` of the model with translation). Let n be its length.
We show that there is a cycle in plain-`ob`.
Assume, for contradiction, that the cycle contains an edge that is not in plain-`ob` (that is, the `ob` of the model without translation):

▷ iio
by case split:

- `[T]; iio; [M]`: by Lemma ob-to-T, the `ob` edge to the left has to be either
  - `iio` in which case, by transitivity of `iio`, there is a shorter cycle, so we have a contradiction. Let us call this Case IIOtrans.
  - `trfe`, which is from an initial write by the VA abstraction condition, but by Lemma no-cycle-ob-to-init, the cycle cannot exist.
  - `imm(ob)^(n-2); [T]; iio; [T]; iio; [M]` then we have `imm(ob)^(n-2); [T]; iio; [M]`, which involves one fewer translate, so we have a contradiction.
  - `imm(ob)^(n-2); [CSE]; instruction-order` This is similar to IIOtrans.
  - `imm(ob)^(n-2); po; [ERET]; instruction-order; [T]` This is similar to IIOtrans.
- `[T]; iio; [T]`:
  So the whole cycle looks like `imm(ob)^(n-1); [T]; iio; [T]`

  By Lemma ob-to-T, we have either
  - `imm(ob)^(n-2); iio; [T]; iio; [T]` See Case IIOtrans.
  - `imm(ob)^(n-2); trfe` the `trfe` is from an initial write by the VA abstraction condition, and by Lemma no-cycle-ob-to-init, the cycle cannot exist.
  - `imm(ob)^(n-2); [T]; iio; [T]` but we already have `iio` to the second T, so we have a cycle involving one fewer translate, so we have a contradiction.
  - `imm(ob)^(n-2); [CSE]; instruction-order` This is similar to IIOtrans.
  - `imm(ob)^(n-2); po; [ERET]; instruction-order; [T]` This is similar to IIOtrans.

▷ `tob` has

- `[T_f]; tfr` which has a fault, so we have a contradiction.
- `([T_f]; tfri) & (po; [dsb.sy]; instruction-order)^-1` which has a fault, so we have a contradiction.
- `speculative; trfi` which is empty, because of the static page table.

▷ `obtlbi`, which is empty by Lemma obtlbi-empty.

▷ `ctxob` has

- `speculative; [MSR TTBR]` by the VA abstraction condition, there is no MSR TTBR
- `[CSE]; instruction-order`
  So the whole cycle looks like
  `[CSE]; instruction-order; imm(ob)^(n-1)`
  Because `instruction-order` is acyclic, $n \geq 1$, so we have
  `[CSE]; instruction-order; imm(ob); imm(ob)^(n-2)`
  By Lemma ob-from-T, we have either:

* `[CSE]; instruction-order; iio; imm(ob)^(n-2)`
  which means that by Lemma instruction-order-compress, we have
  `[CSE]; instruction-order; imm(ob)^(n-2)`
  so we have a cycle involving one edge fewer, so we have a contradiction.

* `[CSE]; instruction-order; [T]; iio; [M]; po; [W]; imm(ob)^(n-2)`
  which means that by Lemma instruction-order-compress, we have
  `[CSE]; instruction-order; imm(ob)^(n-2)`
  so we have a cycle involving one edge fewer, so we have a contradiction.

– `[ContextChange]; po; [CSE]`
  by the VA abstraction condition, there is no `ContextChange`.

– `speculative; [CSE]`
  The `CSE` has to be an `ISB`, because there are no exceptions, and the `speculative` is either in `dob` in the plain model, so we have a contradiction, or in `[T]; instruction-order`.
  So the whole cycle looks like `imm(ob)^(n-1); [T]; iio; [M]; po; [ISB]`

  Because `po | iio` is acyclic, $n-1$ has to be ≥ 1, so by Lemma ob-to-T, we have either

  * `imm(ob)^(n-2); iio; [T]; iio; [M]; po; [ISB]`
    See Case IIOtrans.

  * `trfe`, which is from an initial write by the VA abstraction condition,
    but by Lemma no-cycle-ob-to-init, the cycle cannot exist

  * `imm(ob)^(n-2); [T]; iio; [T]; iio; [M]; po; [ISB]`
    but we already have `iio` to the second T,
    so we have a cycle involving one fewer translate,
    so we have a contradiction.

  * `imm(ob)^(n-2); [CSE]; instruction-order; [T]; iio; [M]; po; [ISB]`
    which means that by Lemma instruction-order-compress, we have
    `imm(ob)^(n-2); [CSE]; instruction-order`
    so we have a cycle involving one edge fewer,
    so we have a contradiction.

  * `imm(ob)^(n-2); po; [ERET]; instruction-order; [T]; iio; [M]; po; [ISB]`
    is similar

– `po; [ERET]; instruction-order; [T]`
  So the whole cycle looks like
  `po; [ERET]; instruction-order; [T]; imm(ob)^(n-1)`
  Because instruction-order is acyclic, $n \geq 1$, so we have
  `po; [ERET]; instruction-order; [T]; imm(ob); imm(ob)^(n-2)`
  By Lemma ob-from-T, we have either:

  * `po; [ERET]; instruction-order; [T]; iio; imm(ob)^(n-2)`
    which means that by Lemma instruction-order-compress-iio, we have
    `po; [ERET]; instruction-order; imm(ob)^(n-2)`
    so we have a cycle involving one edge fewer, so we have a contradiction.

  * `po; [ERET]; instruction-order; [T]; ([T]; iio; [M]; po; [W]) ; imm(ob)^(n-2)`
    which means that by Lemma instruction-order-compress, we have
    `po; [ERET]; instruction-order; imm(ob)^(n-2)`
    so we have a cycle involving one edge fewer, so we have a contradiction.

▷ extended `dob`:

– involving `trfi` from non-initial writes, which contradicts our assumption about static translation.

– or `[T]; instruction-order; [W]`,
  so `[T]; iio; [M]; po; [W]`
  So the whole cycle looks like `imm(ob)^(n-1); [T]; iio; [M]; po; [W]`

  Because `po | iio` is acyclic, $n-1$ has to be ≥ 1, so by Lemma ob-to-T, we have either

    - ∗ `imm(ob)^(n-2); iio; [T]; iio; [M]; po; [W]`
      See Case IIOtrans.
    - ∗ `trfe`, which is from an initial write by the VA abstraction condition,
      but by Lemma no-cycle-ob-to-init, the cycle cannot exist
    - ∗ `imm(ob)^(n-2); [T]; iio; [T]; iio; [M]; po; [W]`
      but we already have `iio` to the second T,
      so we have a cycle involving one fewer translate,
      so we have a contradiction.
    - ∗ `imm(ob)^(n-2); [CSE]; instruction-order; [T]; iio; [M]; po; [W]`
      which means that by Lemma instruction-order-compress, we have
      `imm(ob)^(n-2); [CSE]; instruction-order`
      so we have a cycle involving one edge fewer,
      so we have a contradiction.
    - ∗ `imm(ob)^(n-2); po; [ERET]; instruction-order; [T]; iio; [M]; po; [W]`
      is similar

  - ▷ extended `bob`, but only involving TLBI, which contradicts our assumption of no TLBI.
  - ▷ extended `obs`, but only involving `trfe`, by the VA abstraction condition, the only writes to page tables are from initial writes, and by Lemma no-cycle-ob-to-init, there are no `ob` cycles via initial writes, so there is no cycle.
  - ▷ `obfault`, which involves a fault, which contradicts our assumptions.
  - ▷ `obets`, which involves a fault or a TLBI, which contradicts our assumptions.

All the other edges are in plain-`ob` by definition. □

Theorem 2 (VA anti-abstraction)**.** If the translation extension condition holds, then there exists a `Gtr` that satisfies the VA anti-abstraction condition such that `Gtr` is a stitching of `Gabs` with the `pt_initial_state` according to `translate` in `tr_ctxt` and `Gtr` is consistent wrt. the model with translation.

Proof. `Gtr` exists by the translation extension construction,
and it is consistent by Lemma ob-acyclic-preserved. □

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. doi:10.1145/1594834.1480929.

[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. doi:10.1007/978-3-642-03359-9_27.

[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). doi:10.1145/1785414.1785443.

[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. doi:10.1007/978-3-642-14052-5_18.

[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. doi:10.1145/3158107.

[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. doi:10.1145/1993498.1993520.

[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. URL: https://alastairreid.github.io/papers/fmcad2016-trustworthy.pdf, doi:10.1109/FMCAD.2016.7886675.

[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, Upper Saddle River, NJ, USA, 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. doi:10.1145/1481839.1481842.

[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. doi:10.1145/2830772.2830775.

[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. doi:10.1145/3009837.3009839.

[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), October 2018. doi:10.1145/3276507.

[18] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *ESOP*, 2007. doi:10.1007/978-3-540-71316-6_23.

[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. doi:10.1145/2103656.2103717.

[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. doi:10.1145/2254064.2254102.

[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. doi:10.1109/2.546611.

[24] William Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(6):445–455, 2000. doi:https://doi.org/10.1002/1096-9128(200005)12:6<445::AID-CPE484>3.0.CO;2-A.

[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, 2014. 2015 SIGPLAN John C. Reynolds Doctoral Dissertation award and 2015 CPHC/BCS Distinguished Dissertation Competition winner.

[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. JSR-395.

[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] 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. doi:10.1145/2627752.

[33] 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.

[34] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In *Proc. CAV*, 2010. doi:10.1007/978-3-642-14295-6_25.

[35] 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. doi:10.1007/978-3-030-44914-8_23.

[36] 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. doi:10.1007/978-3-030-81685-8_14.

[37] 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. doi:10.1007/978-3-030-99336-8_6.

[38] 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*, 63(1):110–133, October 2024. doi:10.1007/s10703-023-00409-y.

[39] Ben Simner, Alasdair Armstrong, Thomas Baueriss, Brian Campbell, Ohad Kammar, Jean Pichon-Pharabod, and Peter Sewell. Precise exceptions in relaxed architectures (pre-publication), December 2024. URL: https://arxiv.org/abs/2412.15140.

[40] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the ARM and POWER relaxed memory models, October 2012. Draft. URL: https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf.

[41] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C-28(9):690–691, 1979.

[42] 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), July 2021. doi:10.1145/3458926.

[43] 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. URL: http://dx.doi.org/10.1007/BF01784241, doi:10.1007/BF01784241.

[44] Arm Limited. Exploration Tools – Arm Developer. https://developer.arm.com/downloads/-/exploration-tools, 2022. Accessed May 2022.

[45] 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. doi:10.1145/3290384.

[46] 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.

[47] Shaked Flur. *Thesis (pre-publication)*. PhD thesis, University of Cambridge, January 2024.

[48] 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. doi:10.1145/2914770.2837615.

[49] Scott Owens, Peter Böhm, Francesco Zappa Nardelli, and Peter Sewell. Lem: A lightweight tool for heavyweight semantics. In *Proceedings of Interactive Theorem Proving – Second International Conference (previously TPHOLs) (Berg en Dal), LNCS 6898*, pages 363–369, 2011. (Rough Diamond). doi:10.1007/978-3-642-22863-6_27.

[50] 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. doi:10.1145/2628136.2628143.

[51] Shaked Flur, Jon French, Kathryn Gray, Christopher Pulte, Susmit Sarkar, and Peter Sewell. rmem. www.cl.cam.ac.uk/~pes20/rmem/, 2017.

[52] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 378391, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1040305.1040336.

[53] Will Deacon. The ARMv8 application level memory model. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat (accessed 2019-07-01), 2016.

[54] Arm. The Armv9 application level memory model. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat (accessed 2023-12-21), 2016.

[55] 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. doi:10.1145/2560537.

[56] 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. doi:10.1145/3192366.3192381.

[57] 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. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu.

[58] 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. doi:10.1145/3132747.3132782.

[59] 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. doi:10.3233/JCS-160558.

[60] 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. URL: http://www.easychair.org/publications/paper/wkpS.

[61] 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. doi:10.1017/S0956796818000229.

[62] 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. doi:10.1145/2535838.2535841.

[63] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009. doi:10.1007/s10817-009-9155-4.

[64] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. *SIGPLAN Not.*, 42(6):6677, June 2007. doi:10.1145/1273442.1250743.

[65] 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. doi:10.1145/1706299.1706313.

[66] 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.

[67] Richard Grisenthwaite. personal communication, 2020.

[68] Arm Limited. Arm architecture reference manual. https://developer.arm.com/documentation/ddi0487/ka/?lang=en, March 2024. K.a Armv9 EAC. ARM DDI 0487K.a (ID032224). 14777pp.

[69] 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. URL: http://dl.acm.org/citation.cfm?id=1987389.1987395, doi:10.1007/978-3-642-19835-9_5.

[70] Jade Alglave and Luc Maranget. The `diy7` tool. http://diy.inria.fr/, 2019. Accessed 2021-07-01.

[71] Hrutvik Kanabar, Anthony C. J. Fox, and Magnus O. Myreen. Taming an Authoritative Armv8 ISA Specification: L3 Validation and CakeML Compiler Verification. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving (ITP 2022)*, volume 237 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:22, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2022.20, doi:10.4230/LIPIcs.ITP.2022.20.

[72] 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.

[73] Arm Limited. Arm Cortex-A53 MPCore Processor Technical Reference Manual, 2022. ARM DDI 0500J.

[74] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017.

[75] 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.

[76] 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.

[77] 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.

[78] The Chromium source code. Jump table assembler - src/wasm/jump-table-assembler.cc - v8/v8.git.

[79] The Chromium source code. Commit ea82d: [arm64] Use BTI instructions for forward CFI.

[80] Jim Handy. *The cache memory book*. Academic Press Professional, Inc., 1993.

[81] 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.

[82] 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 6677, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1250734.1250743.

[83] 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. doi:10.1007/978-3-642-14107-2_24.

[84] 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. doi:10.1007/978-3-642-54108-7_12.

[85] 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. doi:10.1109/FMCAD.2014.6987600.

[86] 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 4760, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372885.3373811.

[87] 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. doi:10.1145/2954680.2872399.

[88] 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. doi:10.1109/SP40001.2021.00049.

[89] 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. doi:10.1145/3360561.

[90] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the Intel-x86 architecture. *PACMPL*, 4(POPL):11:1–11:31, 2020. doi:10.1145/3371079.

[91] Alasdair Armstrong, Brian Campbell, Ben Simner, Thibaut Pérami, and Peter Sewell. Isla Axiomatic Models. https://github.com/rems-project/system-semantics-arm-axiomatic-models.

[92] Arm Limited. Arm Cortex-A57 MPCore Processor Technical Reference Manual, 2016. ARM DDI 0488H.

[93] Arm Limited. Arm Cortex-A72 MPCore Processor Technical Reference Manual, 2016. ARM 100095 r0p3.

[94] Arm Limited. Arm Cortex-A76 Core Processor Technical Reference Manual, 2023. ARM 100798 r4p1.

[95] Arm Limited. Arm Cortex-A78 Core Processor Technical Reference Manual, 2023. ARM 101430 r1p2.

[96] Arm Limited. Arm Cortex-X1 Core Processor Technical Reference Manual, 2023. ARM 101433 r1p2.

[97] Arm Limited. Arm architecture reference manual. https://developer.arm.com/documentation/ddi0487/la/?lang=en, November 2024. L.a Armv9 EAC. ARM DDI 0487L.a. 14568pp.

[98] 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. https://arxiv.org/abs/1705.01225. doi:10.4204/EPTCS.249.1.

[99] 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.

[100] 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.

[101] 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. doi:10.29007/c2f1.

[102] 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. doi:10.1007/978-3-319-94821-8_32.

[103] 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. doi:10.1145/1736020.1736057.

[104] Bogdan Romanescu, Alvin Lebeck, and Daniel J. Sorin. Address translation aware memory consistency. *IEEE Micro*, 31(1):109–118, January 2011. doi:10.1109/MM.2010.99.

[105] Naorin Hossain, Caroline Trippel, and Margaret Martonosi. Transform: Formally specifying transistency models and synthesizing enhanced litmus tests. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*, pages 874–887. IEEE, 2020. doi:10.1109/ISCA45697.2020.00076.

[106] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. Formal verification of a multiprocessor hypervisor on arm relaxed memory hardware. In *SOSP 2021: Proceedings of the 28th ACM Symposium on Operating Systems Principles*, October 2021.

[107] Jade Alglave, Richard Grisenthwaite, Artem Khyzha, Luc Maranget, and Nikos Nikoleris. Puss in Boots: Formalizing Arm's Virtual Memory System Architecture. *IEEE Micro*, 44(6):83–91, 2024. doi:10.1109/MM.2024.3422668.

[108] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 405418, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3173162.3177156.

[109] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Amsterdam, 5 edition, 2012.

[110] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. The IBM System/360 Model 91: Machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 11(1):8–24, 1967. doi:10.1147/rd.111.0008.

[111] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Shaked Flur Jon French Kathryn E. Gray, Gabriel Kerneis, Neel Krishnaswami, Prashanth Mundkur, Robert Norton-Wright, Christopher Pulte, Alastair Reid, Peter Sewell, Ian Stark, and Mark Wassell. Sail. https://www.cl.cam.ac.uk/~pes20/sail/, 2019.

[112] Thomas Bauereiss, Brian Campbell, Alasdair Armstrong, Alastair Reid, Kathryn E. Gray, Anthony Fox, Peter Sewell, and Arm Limited. Sail Armv9.4-A instruction-set architecture (ISA) model, 2024. https://github.com/rems-project/sail-arm. Accessed 2024-05-11.

[113] Allon Adir, Hagit Attiya, and Gil Shurek. Information-flow models for shared memory with an application to the PowerPC architecture. *IEEE Trans. Parallel Distrib. Syst.*, 14(5):502–515, 2003. doi:10.1109/TPDS.2003.1199067.

[114] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 15–26, New York, NY, USA, 1990. ACM. doi:10.1145/325164.325102.

[115] Kourosh Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, 1995.

[116] Pradeep S. Sindhu, Jean-Marc Frailong, and Michel Cekleov. *Formal Specification of Memory Models*, pages 25–41. Springer US, Boston, MA, 1992. doi:10.1007/978-1-4615-3604-8_2.

[117] Intel. A formal specification of Intel Itanium processor family memory ordering, 2002. http://download.intel.com/design/Itanium/Downloads/25142901.pdf.

[118] Jade Alglave. *A Shared Memory Poetics*. PhD thesis, Université Paris 7 – Denis Diderot, November 2010.

[119] Luc Maranget. Personal communication, 2024.

[120] Peter Sewell, Christopher Pulte, Shaked Flur, Mark Batty, Luc Maranget, and Alasdair Armstrong. Multicore semantics: Making sense of relaxed memory (MPhil slides), October 2022. URL: https://www.cl.cam.ac.uk/~pes20/slides-acs-2022.pdf.

[121] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 618632, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3062341.3062352.

[122] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The problem of programming language concurrency semantics. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 283–307, 2015. doi:10.1007/978-3-662-46669-8_12.

[123] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158105.

[124] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 96110, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3314221.3314609.

[125] Michalis Kokologiannakis and Viktor Vafeiadis. GenMC: A model checker for weak memory models. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 427–440, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-81685-8_20.

[126] William W. Pugh. Fixing the Java memory model. In Geoffrey C. Fox, Klaus E. Schauser, and Marc Snir, editors, *Proceedings of the ACM 1999 Conference on Java Grande, JAVA '99, San Francisco, CA, USA, June 12-14, 1999*, pages 89–98. ACM, 1999. doi:10.1145/304065.304106.

[127] Jean Pichon-Pharabod and Peter Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 622–633. ACM, 2016. doi:10.1145/2837614.2837616.

[128] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 175189, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3009837.3009850.

[129] Radha Jagadeesan, Alan Jeffrey, and James Riely. Pomsets with preconditions: a simple model of relaxed memory. *Proc. ACM Program. Lang.*, 4(OOPSLA):194:1–194:30, 2020. doi:10.1145/3428262.

[130] Soham Chakraborty. *Correct Compilation of Relaxed Memory Concurrency*. PhD thesis, Kaiserslautern University of Technology, Germany, 2019. URL: https://kluedo.ub.rptu.de/frontdoor/index/index/docId/5697.

[131] Nathan Chong, Tyler Sorensen, and John Wickerson. The semantics of transactions and weak memory in x86, Power, ARM, and C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 211225, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3192366.3192373.