

C/C++ Atomics Application Binary Interface Standard for the Arm[®] 64-bit Architecture

2024Q1

Date of Issue: 5th April 2024

The logo for Arm, consisting of the lowercase letters 'arm' in a bold, blue, sans-serif font.

1 Preamble

1.1 Abstract

This document describes the C/C++ Atomics Application Binary Interface for the Arm 64-bit architecture. This document concerns the valid mappings from C/C++ Atomic Operations to sequences of A64 instructions. For matters concerning the memory model, please consult §B2 of the Arm Architecture Reference Manual [[ARMARM](#)]. We focus only on a subset of the C11 atomic operations at the time of writing.

1.2 Keywords

C++, C, Application Binary Interface, ABI, AArch64, C++ ABI, generic C++ ABI, Atomics, Concurrency

1.3 Latest release and defects report

Please check [Atomics Application Binary Interface for the Arm® Architecture](#) for the latest release of this document.

Please report defects in this specification to the [issue tracker page on GitHub](#).

1.4 Acknowledgement

This document came about in the process of Luke Geeson's PhD on testing the compilation of concurrent C/C++ with assistance from Wilco Dijkstra from Arm's Compiler Teams.

1.5 Licence

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/4.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Grant of Patent License. Subject to the terms and conditions of this license (both the Public License and this Patent License), each Licensor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Licensed Material, where such license applies only to those patent claims licensable by such Licensor that are necessarily infringed by their contribution(s) alone or by combination of their contribution(s) with the Licensed Material to which such contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Licensed Material or a contribution incorporated within the Licensed Material constitutes direct or contributory patent infringement, then any licenses granted to You under this license for that Licensed Material shall terminate as of the date such litigation is filed.

1.6 About the license

As identified more fully in the [Licence](#) section, this project is licensed under CC-BY-SA-4.0 along with an additional patent license. The language in the additional patent license is largely identical to that in Apache-2.0 (specifically, Section 3 of Apache-2.0 as reflected at <https://www.apache.org/licenses/LICENSE-2.0>) with two exceptions.

First, several changes were made related to the defined terms so as to reflect the fact that such defined terms need to align with the terminology in CC-BY-SA-4.0 rather than Apache-2.0 (e.g., changing "Work" to "Licensed Material").

Second, the defensive termination clause was changed such that the scope of defensive termination applies to "any licenses granted to You" (rather than "any patent licenses granted to You"). This change is intended to help maintain a healthy ecosystem by providing additional protection to the community against patent litigation claims.

1.7 Contributions

Contributions to this project are licensed under an inbound=outbound model such that any such contributions are licensed by the contributor under the same terms as those in the [Licence](#) section.

1.8 Trademark notice

The text of and illustrations in this document are licensed by Arm under a Creative Commons Attribution-Share Alike 4.0 International license ("CC-BY-SA-4.0"), with an additional clause on patents. The Arm trademarks featured here are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Please visit <https://www.arm.com/company/policies/trademarks> for more information about Arm's trademarks.

1.9 Copyright

Copyright (c) 2024, Arm Limited and its affiliates. All rights reserved.

Contents

1 Preamble	2
1.1 Abstract	2
1.2 Keywords	2
1.3 Latest release and defects report	2
1.4 Acknowledgement	3
1.5 Licence	3
1.6 About the license	3
1.7 Contributions	3
1.8 Trademark notice	3
1.9 Copyright	3
2 About this document	5
2.1 Change control	5
2.1.1 Current status and anticipated changes	5
2.2 Change History	5
2.3 References	5
2.4 Terms and Abbreviations	6
3 Overview	8
4 Mappings from Atomic Operations to Assembly Sequences	8
4.1 Notational Conventions	8
4.2 Mappings for 32-bit types	9
4.3 Mappings for 8-bit types	11
4.4 Mappings for 16-bit types	11
4.5 Mappings for 64-bit types	11
4.6 Mappings for 128-bit types	12
4.7 Special Cases	18
4.7.1 Re-Ordering of Read-Modify-Write Effects and Acquire Fence	18
4.7.2 Const-Qualified 128-bit Atomic Loads	20
5 Declarative statement of Mappings compatibility	20
5.1 The Mix Testing Process	20
5.2 Definition of ABI-Compatibility for Atomic Operations	21

2 About this document

2.1 Change control

2.1.1 Current status and anticipated changes

The following support level definitions are used by the Arm Atomics ABI specifications:

Release

Arm considers this specification to have enough implementations, which have received sufficient testing, to verify that it is correct. The details of these criteria are dependent on the scale and complexity of the change over previous versions: small, simple changes might only require one implementation, but more complex changes require multiple independent implementations, which have been rigorously tested for cross-compatibility. Arm anticipates that future changes to this specification will be limited to typographical corrections, clarifications and compatible extensions.

Beta

Arm considers this specification to be complete, but existing implementations do not meet the requirements for confidence in its release quality. Arm may need to make incompatible changes if issues emerge from its implementation.

Alpha

The content of this specification is a draft, and Arm considers the likelihood of future incompatible changes to be significant.

All content in this document is at the **Alpha** quality level.

2.2 Change History

If there is no entry in the change history table for a release, there are no changes to the content of the document for that release.

Issue	Date	Change
00alp 0	5 th April 2024.	Alpha release.

2.3 References

This document refers to, or is referred to by, the following documents.

Ref	External reference or URL	Title
ARMA RM	DDI 0487	Arm Architecture Reference Manual Armv8 for Armv8-A architecture profile
CSTD	ISO/IEC 9899:2018	International Standard ISO/IEC 9899:2018 – Programming languages C.

Note: At the time of writing C23 is not released, as such ISO C17 is considered the latest published document.

2.4 Terms and Abbreviations

The Atomics ABI for the Arm 64-bit Architecture uses the following terms and abbreviations.

A64

The instruction set available when in AArch64 state.

AArch64

The 64-bit general-purpose register width state of the Armv8 architecture.

ABI

Application Binary Interface:

1. The specifications to which an executable must conform in order to execute in a specific execution environment. For example, the *Linux ABI for the Arm Architecture*.
2. A particular aspect of the specifications to which independently produced relocatable files must conform in order to be statically linkable and executable. For example, the C++ ABI for the Arm 64-bit Architecture [[CPPABI64](#)], or ELF for the Arm Architecture [[AAELF64](#)].

Arm-based

... based on the Arm architecture ...

Concurrent Program

A C or C++ program that consists of one or more Threads of Execution. Each Thread of Execution must communicate with other threads in the Concurrent Program through Shared-Memory Locations, using Atomic Operations to be deemed *concurrent*.

Thread of Execution

A unit of computation that executes one or more Atomic Operations, Synchronization Operations or other C language statements. The Arm Architecture Reference Manual [[ARMARM](#)] calls these *Observers*. Typically a thread is defined as a function (e.g. a POSIX thread) although we do not limit threads to such implementations.

Atomic Operation

A C/C++ operation on a Shared-Memory Location. Typically either a load, store, exchange, compare, or arithmetic instruction (such as a fetch and add operation). Atomics are used to define higher level primitives including locks and concurrent queues. ISO C defines the range of supported atomic operations and the `atomic` type. Operations on atomic-qualified data are guaranteed not to be interrupted by another Thread of Execution.

Synchronization Operation

The order that atomic operations are executed by each Thread of Execution may not be the same as the order they are written in the program. Synchronization Operations are statements that constrain the order of accesses made to Shared-Memory Locations by each thread. Synchronization Operations include Thread Fences, and certain control flow structures.

Shared-Memory Location

A memory location that can be accessed by any Thread of Execution in the program.

Memory Order Parameters

Describes a constraint on an Atomic Operation or Synchronization Operation. Memory Order describes how memory accesses made by Atomic Operations may be ordered with respect to other Atomic Operations and Synchronization Operations. ISO C defines a `memory_order` enum type to capture the possible memory order parameters.

Thread Fence

A Thread Fence is a Synchronization Operation that constrains the order of Accesses made by Atomic Operations on a given Thread of Execution. Fences are equipped with a Memory Order Parameter that specifies which kinds of accesses may be reordered before or after the fence. ISO C defines the `atomic_thread_fence` to synchronize the order of accesses made by atomic operations on `_Atomic` qualified data.

Atomic Instruction

An A64 instruction that may have Memory Order semantics. For instance an A64 LDR instruction has no atomicity, but the LDAR instruction has *acquire* semantics. (see [[ARMARM](#)]).

Assembly Sequence

A sequence of Atomic Instructions.

Mapping

A pair of Atomic Operation and Assembly Sequence. A compiler generates the Assembly Sequence, given an Atomic Operation and Compiler Profile as input.

Compiler Profile

A combination of a compiler and command-line flags that implements a set of Mappings from Atomic Operations to A64 Assembly Sequences. When the compiler is provided with a Concurrent Program and Compiler Profile, it generates an Assembly Sequence.

More specific terminology is defined when it is first used.

3 Overview

The C/C++ Atomics ABI for the Arm 64-bit architecture (AABI64) comprises the following sub-components.

- The [Mappings from Atomic Operations to Assembly Sequences](#), which defines the mappings from C/C++ atomic operations to sequences of A64 assembly that are interoperable with respect to each other.
- A [Declarative statement of Mappings compatibility](#), as far as non-exhaustive testing can validate, that the aforementioned Mappings can be used together. That is, there is no tested combination of Mappings that induces unexpected program behaviour when a compiled program that uses atomics is executed on a multi-core Arm-based machine.

4 Mappings from Atomic Operations to Assembly Sequences

We now describe the compatible Mappings for C/C++ Atomic Operations and Assembly Sequences. Since there is a large number of ways these mappings may be combined, we break down the tables by the width of the access, and list compatible Assembly Sequences for each Atomic Operation.

This is an open ABI, we encourage improvements to this specification to be submitted to the [issue tracker page on GitHub](#).

These mappings are not exhaustive, but aim to cover the atomics we have tested. Please request more atomics using the issue tracker.

4.1 Notational Conventions

To reduce repetition, we use the following notational conventions

Memory Order Parameter	Notation
memory_order_relaxed	relaxed
memory_order_acquire	acq
memory_order_release	rel
memory_order_acq_rel	acq_rel
memory_order_seq_cst	sc

In what follows `loc` refers to the location, `val` refers to a value parameter.

Arbitrary registers may be used in the Assembly Sequences that may change in compiler implementations. Cases where arbitrary registers may *not* be used are covered in the Special Cases section.

Further, in what follows there may be multiple valid Mappings from Atomic Operation to Assembly Sequence, as made available by a given architecture extension. In this case we split the rows of the table to represent multiple options.

Atomic Operation		Assembly Sequence
atomic_store_explicit(<code>loc, val, relaxed</code>)	ARCH1	option A
	ARCH2	option B

Where ARCH is for example BASE (armv8), LSE, LSE2, LSE128, RCPC, or LRCPC3.

Lastly, all operations are in a shorthand form:

Atomic Operation	ShortHand Atomic Operation
<code>atomic_store_explicit(...)</code>	<code>store(...)</code>
<code>atomic_load_explicit(...)</code>	<code>load(...)</code>
<code>atomic_thread_fence(...)</code>	<code>fence(...)</code>
<code>atomic_exchange_explicit(...)</code>	<code>exchange(...)</code>
<code>atomic_fetch_add_explicit(...)</code>	<code>fetch_add(...)</code>
<code>atomic_fetch_sub_explicit(...)</code>	<code>fetch_sub(...)</code>
<code>atomic_fetch_or_explicit(...)</code>	<code>fetch_or(...)</code>
<code>atomic_fetch_xor_explicit(...)</code>	<code>fetch_xor(...)</code>
<code>atomic_fetch_and_explicit(...)</code>	<code>fetch_and(...)</code>

4.2 Mappings for 32-bit types

In what follows, register `x1` contains the location `loc` and `w2` contains `val`. The result is returned in `w0`.

Atomic Operation		Assembly Sequence
<code>store(loc, val, relaxed)</code>		<code>STR W2, [X1]</code>
<code>store(loc, val, rel)</code> <code>store(loc, val, sc)</code>		<code>STLR W2, [X1]</code>
<code>load(loc, relaxed)</code>		<code>LDR W2, [X1]</code>
<code>load(loc, acq)</code>	BASE	<code>LDAR W2, [X1]</code>
	RCPC	<code>LDAPR W2, [X1]</code>
<code>load(loc, sc)</code>		<code>LDAR W2, [X1]</code>
<code>fence(relaxed)</code>		<code>NOP</code>
<code>fence(acq)</code>		<code>DMB ISHLD</code>
<code>fence(rel)</code> <code>fence(acq_rel)</code> <code>fence(sc)</code>		<code>DMB ISH</code>
<code>exchange(loc, val, relaxed)</code>	BASE	<pre> loop: LDXR W0, [X1] STXR W3, W2, [X1] CBNZ W3, loop </pre>
	LSE	<code>SWP W2, W0, [X1]</code>

Atomic Operation		Assembly Sequence
exchange(loc, val, acq)	BASE	<pre> loop: LDAXR W0, [X1] STXR W3, W2, [X1] CBNZ W3, loop </pre>
	LSE	SWPA W2, W0, [X1]
exchange(loc, val, rel)	BASE	<pre> loop: LDXR W0, [X1] STLXR W3, W2, [X1] CBNZ W3, loop </pre>
	LSE	SWPL W2, W0, [X1]
exchange(loc, val, acq_rel) exchange(loc, val, sc)	BASE	<pre> loop: LDAXR W0, [X1] STLXR W3, W2, [X1] CBNZ W3, loop </pre>
	LSE	SWPAL W2, W0, [X1]
fetch_add(loc, val, relaxed)	BASE	<pre> loop: LDXR W0, [X1] ADD W2, W2, W0 STXR W3, W2, [X1] CBNZ W3, loop </pre>
	LSE	LDADD W2, W0, [X1]
fetch_add(loc, val, acq)	BASE	<pre> loop: LDAXR W0, [X1] ADD W2, W2, W0 STXR W3, W2, [X1] CBNZ W3, loop </pre>
	LSE	LDADDA W2, W0, [X1]
fetch_add(loc, val, rel)	BASE	<pre> loop: LDXR W0, [X1] ADD W2, W2, W0 STLXR W3, W2, [X1] CBNZ W3, loop </pre>
	LSE	LDADDL W2, W0, [X1]
fetch_add(loc, val, acq_rel) fetch_add(loc, val, sc)	BASE	<pre> loop: LDXAR W0, [X1] ADD W2, W2, W0 STLXR W3, W2, [X1] CBNZ W3, loop </pre>
	LSE	LDADDAL W2, W0, [X1]

Atomic Operation		Assembly Sequence
<pre>compare_exchange_strong(loc, &exp, val, relaxed, relaxed)</pre>	BASE	<pre>loop: LDXR W0, [X1] CMP W0, W4 B.NE fail STXR W3, W2, [X1] CBNZ W3, loop fail:</pre>
	LSE	CAS W0, W2, [X1]
<pre>compare_exchange_strong(loc, &exp, val, acq, acq)</pre>	BASE	<pre>loop: LDAXR W0, [X1] CMP W0, W4 B.NE fail STXR W3, W2, [X1] CBNZ W3, loop fail:</pre>
	LSE	CASA W0, W2, [X1]
<pre>compare_exchange_strong(loc, &exp, val, rel, rel)</pre>	BASE	<pre>loop: LDXR W0, [X1] CMP W0, W4 B.NE fail STLXR W3, W2, [X1] CBNZ W3, loop fail:</pre>
	LSE	CASL W0, W2, [X1]
<pre>compare_exchange_strong(loc, &exp, val, acq_rel, acq) compare_exchange_strong(loc, &exp, val, sc, sc)</pre>	BASE	<pre>loop: LDAXR W0, [X1] CMP W0, W4 B.NE fail STLXR W3, W2, [X1] CBNZ W3, loop fail:</pre>
	LSE	CASAL W0, W2, [X1]

4.3 Mappings for 8-bit types

The mappings for 8-bit types are the same as 32-bit types except they use the **B** variants of instructions.

4.4 Mappings for 16-bit types

The mappings for 16-bit types are the same as 32-bit types except they use the **H** variants of instructions.

4.5 Mappings for 64-bit types

The mappings for 64-bit types are the same as 32-bit types except the registers used are X-registers.

4.6 Mappings for 128-bit types

Since the access width of 128-bit types is double that of the 64-bit register width, the following Mappings use *pair* instructions, which require their own table.

In what follows, register `x4` contains the location `loc`, `x2` and `x3` contain the input value. The result is returned in `x0` and `x1`.

Atomic Operation		Assembly Sequence
store(loc, val, relaxed)	BASE	<pre>loop: LDXP XZR, X1, [X4] STXP W5, X2, X3, [X4] CBNZ W5, loop</pre>
	LSE	<pre>LDP X0, X1, [X4] loop: MOV X6, X0 MOV X7, X1 CASP X0, X1, X2, X3, [X4] CMP X0, X6 CCMP X1, X7, 0, EQ B.NE loop</pre>
	LSE2	<pre>STP x2, X3, [X4]</pre>
store(loc, val, rel)	BASE	<pre>loop: LDXP XZR, X1, [X4] STLXP W5, X2, X3, [X4] CBNZ W5, loop</pre>
	LSE	<pre>LDP X0, X1, [X4] loop: MOV X6, X0 MOV X7, X1 CASPL X0, X1, X2, X3, [X4] CMP X0, X6 CCMP X1, X7, 0, EQ B.NE loop</pre>
	LSE2	<pre>DMB ISH STP X2, X3, [X4]</pre>
	LRCPC3	<pre>STILP X2, X3, [X4]</pre>

Atomic Operation	Assembly Sequence	
store(loc, val, sc)	BASE	<pre> loop: LDXP XZR, X1, [X4] STLXP W5, X2, X3, [X4] CBNZ W5, loop </pre>
	LSE	<pre> LDP X0, X1, [X4] loop: MOV X6, X0 MOV X7, X1 CASPL X0, X1, X2, X3, [X4] CMP X0, X6 CCMP X1, X7, 0, EQ B.NE loop </pre>
	LSE2	<pre> DMB ISH STP X2, X3, [X4] DMB ISH </pre>
	LRCPC3	<pre> STILP X2, X3, [X4] </pre>
load(loc, relaxed)	BASE	<pre> loop: LDXP X0, X1, [X4] STXP W5, X0, X1, [X4] CBNZ W5, loop </pre>
	LSE	<pre> CASP X0, X1, X0, X1, [X4] </pre>
	LSE2	<pre> LDP X0, X1, [X4] </pre>
load(loc, acq)	BASE	<pre> loop: LDAXP X0, X1, [X4] STXP W5, X0, X1, [X4] CBNZ W5, loop </pre>
	LSE	<pre> CASPA X0, X1, X0, X1, [X4] </pre>
	LSE2	<pre> LDP X0, X1, [X4] DMB ISHLD </pre>
	LRCPC3	<pre> LDIAPP X0, X1, [X4] </pre>
load(loc, sc)	BASE	<pre> loop: LDAXP X0, X1, [X4] STXP W5, X0, X1, [X4] CBNZ W5, loop </pre>
	LSE	<pre> CASPA X0, X1, X0, X1, [X4] </pre>
	LSE2	<pre> LDAR X5, [X4] LDP X0, X1, [X4] DMB ISHLD </pre>
	LRCPC3	<pre> LDAR X5, [X4] LDIAPP X0, X1, [X4] </pre>

Atomic Operation		Assembly Sequence
exchange(loc, val, relaxed)	BASE	<pre> loop: LDXP X0, X1, [X4] STXP W5, X2, X3, [X4] CBNZ W5, loop </pre>
	LSE	<pre> LDP X0, X1, [X4] loop: MOV X6, X0 MOV X7, X1 CASP X0, X1, X2, X3, [X4] CMP X0, X6 CCMP X1, X7, 0, EQ B.NE loop </pre>
	LSE128	<pre> MOV X0, X2 MOV X1, X3 SWPP X0, X1, [X4] </pre>
exchange(loc, val, acq)	BASE	<pre> loop: LDAXP X0, X1, [X4] STXP W5, X2, X3, [X4] CBNZ W5, loop </pre>
	LSE	<pre> LDP X0, X1, [X4] loop: MOV X6, X0 MOV X7, X1 CASPA X0, X1, X2, X3, [X4] CMP X0, X6 CCMP X1, X7, 0, EQ B.NE loop </pre>
	LSE128	<pre> MOV X0, X2 MOV X1, X3 SWPPA X0, X1, [X4] </pre>
exchange(loc, val, rel)	BASE	<pre> loop: LDXP X0, X1, [X4] STLXP W5, X2, X3, [X4] CBNZ W5, loop </pre>
	LSE	<pre> LDP X0, X1, [X4] loop: MOV X6, X0 MOV X7, X1 CASPL X0, X1, X2, X3, [X4] CMP X0, X6 CCMP X1, X7, 0, EQ B.NE loop </pre>
	LSE128	<pre> MOV X0, X2 MOV X1, X3 SWPPL X0, X1, [X4] </pre>

Atomic Operation		Assembly Sequence
exchange(loc, val, acq_rel) exchange(loc, val, sc)	BASE	<pre> loop: LDAXP X0, X1, [X4] STLXP W5, X2, X3, [X4] CBNZ W5, loop </pre>
	LSE	<pre> LDP X0, X1, [X4] loop: MOV X6, X0 MOV X7, X1 CASPAL X0, X1, X2, X3, [X4] CMP X0, X6 CCMP X1, X7, 0, EQ B.NE loop </pre>
	LSE128	<pre> MOV X0, X2 MOV X1, X3 SWPPAL X0, X1, [X4] </pre>
fetch_add(loc, val, relaxed)	BASE	<pre> loop: LDXP X0, X1, [X4] ADDS X0, X0, X2 ADC X1, X1, X3 STXP W5, X2, X3, [X4] CBNZ W5, loop </pre>
	LSE	<pre> LDP X0, X1, [X4] loop: MOV X6, X0 MOV X7, X1 ADDS X8, X0, X2 ADC X9, X1, X3 CASP X0, X1, X8, X9, [X4] CMP X0, X6 CCMP X1, X7, 0, EQ B.NE loop </pre>
fetch_add(loc, val, acq)	BASE	<pre> loop: LDAXP X0, X1, [X4] ADDS X0, X0, X2 ADC X1, X1, X3 STXP W5, X2, X3, [X4] CBNZ W5, loop </pre>
	LSE	<pre> LDP X0, X1, [X4] loop: MOV X6, X0 MOV X7, X1 ADDS X8, X0, X2 ADC X9, X1, X3 CASPA X0, X1, X8, X9, [X4] CMP X0, X6 CCMP X1, X7, 0, EQ B.NE loop </pre>

Atomic Operation	Assembly Sequence	
fetch_add(loc, val, rel)	BASE	<pre> loop: LDXP X0, X1, [X4] ADDS X0, X0, X2 ADC X1, X1, X3 STLXP W5, X2, X3, [X4] CBNZ W5, loop </pre>
	LSE	<pre> LDP X0, X1, [X4] loop: MOV X6, X0 MOV X7, X1 ADDS X8, X0, X2 ADC X9, X1, X3 CASPL X0, X1, X8, X9, [X4] CMP X0, X6 CCMP X1, X7, 0, EQ B.NE loop </pre>
fetch_add(loc, val, acq_rel) fetch_add(loc, val, sc)	BASE	<pre> loop: LDAXP X0, X1, [X4] ADDS X0, X0, X2 ADC X1, X1, X3 STXLP W5, X2, X3, [X4] CBNZ W5, loop </pre>
	LSE	<pre> LDP X0, X1, [X4] loop: MOV X6, X0 MOV X7, X1 ADDS X8, X0, X2 ADC X9, X1, X3 CASPAL X0, X1, X8, X9, [X4] CMP X0, X6 CCMP X1, X7, 0, EQ B.NE loop </pre>
fetch_or(loc, val, relaxed)	LSE128	<pre> MOV X0, X2 MOV X1, X3 LDSETP X0, X1, [X4] </pre>
fetch_or(loc, val, acq)	LSE128	<pre> MOV X0, X2 MOV X1, X3 LDSETPA X0, X1, [X4] </pre>
fetch_or(loc, val, rel)	LSE128	<pre> MOV X0, X2 MOV X1, X3 LDSETPL X0, X1, [X4] </pre>
fetch_or(loc, val, acq_rel) fetch_or(loc, val, sc)	LSE128	<pre> MOV X0, X2 MOV X1, X3 LDSETPAL X0, X1, [X4] </pre>
fetch_and(loc, val, relaxed)	LSE128	<pre> MVN X0, X2 MVN X1, X3 LDCLRP X0, X1, [X4] </pre>

Atomic Operation		Assembly Sequence
fetch_and(loc, val, acq)	LSE128	MVN X0, X2 MVN X1, X3 LDCLRPA X0, X1, [X4]
fetch_and(loc, val, rel)	LSE128	MVN X0, X2 MVN X1, X3 LDCLRPL X0, X1, [X4]
fetch_and(loc, val, acq_rel) fetch_and(loc, val, sc)	LSE128	MVN X0, X2 MVN X1, X3 LDCLRPAL X0, X1, [X4]
compare_exchange_strong(loc, &exp, val, relaxed, relaxed)	BASE	loop: LDXP X6, x7, [X4] CMP X6, X0 CCMP X7, X1, 0, EQ CSEL X8, X2, X6, EQ CSEL X9, X3, X7, EQ STXP W5, X8, X9, [X4] CBNZ W5, loop MOV X0, X6 MOV X1, X7
	LSE	CASP X0, X1, X2, X3, [X4]
compare_exchange_strong(loc, &exp, val, acq, acq)	BASE	loop: LDAXP X6, x7, [X4] CMP X6, X0 CCMP X7, X1, 0, EQ CSEL X8, X2, X6, EQ CSEL X9, X3, X7, EQ STXP W5, X8, X9, [X4] CBNZ W5, loop MOV X0, X6 MOV X1, X7
	LSE	CASPA X0, X1, X2, X3, [X4]
compare_exchange_strong(loc, &exp, val, rel, rel)	BASE	loop: LDXP X6, x7, [X4] CMP X6, X0 CCMP X7, X1, 0, EQ CSEL X8, X2, X6, EQ CSEL X9, X3, X7, EQ STLXP W5, X8, X9, [X4] CBNZ W5, loop MOV X0, X6 MOV X1, X7
	LSE	CASPL X0, X1, X2, X3, [X4]

Atomic Operation		Assembly Sequence
<pre>compare_exchange_strong(loc, &exp, val, acq_rel, acq) compare_exchange_strong(loc, &exp, val, sc, sc)</pre>	BASE	<pre>loop: LDAXP X6, x7, [X4] CMP X6, X0 CCMP X7, X1, 0, EQ CSEL X8, X2, X6, EQ CSEL X9, X3, X7, EQ STLXP W5, X8, X9, [X4] CBNZ W5, loop MOV X0, X6 MOV X1, X7</pre>
	LSE	CASPAL X0, X1, X2, X3, [X4]

We do not list other variants of `fetch_<op>` since their mappings should be the same (modulo implementations of `<op>` that are not in scope of this document). Precisely implementations that use loops should use the instructions that load or store from memory with the relevant memory order, and the appropriate `<op>` Assembly Sequence inside the loop. Exceptions, where Assembly Sequences exist, are stated (for instance `fetch_or` can be implemented using `LDSETP` when the LSE128 extension is enabled).

4.7 Special Cases

There are special cases in the Mappings presented above, these must be handled in order to prevent unexpected outcomes of the compiled program.

4.7.1 Re-Ordering of Read-Modify-Write Effects and Acquire Fence

Consider the following Concurrent Program:

```
// Shared-Memory Locations
_Atomic int* x;
_Atomic int* y;

// Memory Order Parameter
#define relaxed memory_order_relaxed
#define release memory_order_release
#define acquire memory_order_acquire

// Threads of Execution
void thread_0 () {
    atomic_store_explicit(x, 1, relaxed);
    atomic_thread_fence(release);
    atomic_store_explicit(y, 1, relaxed);
}

void thread_1 () {
    atomic_exchange_explicit(y, 2, release);
    atomic_thread_fence(acquire);
    int r0 = atomic_load_explicit(x, relaxed);
}
```

Under ISO C, the above Concurrent Program finishes execution in one of three possible outcomes:

```
{ thread_1:r0=0; y=1; }
{ thread_1:r0=1; y=1; }
{ thread_1:r0=1; y=2; }
```

In this case the value read by the exchange on `thread_1` is not used, and a compiler is free to remove references to unused data. It is thus legal under ISO C for a compiler to translate the program into the following Assembly Sequences:

```
thread_0:
  MOV W9,#1
  STR W9,[X2]
  DMB ISH
  STR W3,[X4]

thread_1:
  MOV W9,#2
  SWP W9, WZR, [X2]
  DMB ISHLD
  LDR W3,[X4]
```

where `thread_0:X2` contains the address of `x`, `thread_0:X4` contains the address of `y`, and `thread_1:X2` contains the address of `y`, `thread_1:X4` contains the address of `x`.

Note: the `exchange` Atomic Operation is compiled to a `SWP` Assembly Instruction, where its destination register is the zero register `WZR`. The `acquire` fence on `thread_1` is compiled to the `DMB ISHLD` Assembly Instruction.

Executing the compiled program on an Arm-based machine from a fixed initial state (where `x` and `y` are 0) produces one of the following outcomes, according to the AArch64 Memory Model contained in §B2 of the Arm Architecture Reference Manual [ARMARM]:

```
{ thread_1:r0=0; [y]=1; }
{ thread_1:r0=0; [y]=2; } <-- Forbidden by source model, a bug!
{ thread_1:r0=1; [y]=1; }
{ thread_1:r0=1; [y]=2; }
```

By comparing `w3` and the local variable `r0` of the original Concurrent Program we see there is one additional Outcome of executing the compiled program that is not an outcome of executing the Concurrent Program. This is due to the fact that according to the Arm Architecture Reference Manual [ARMARM] instructions where the destination register is `WZR` or `XZR`, are not regarded as doing a read for the purpose of a `DMB LD` barrier.

ISO C permits a conforming implementation to delete unused data, but in this case it introduces another Outcome of Execution. To fix this issue, a compiler should not rewrite the destination register to be the zero register in this case:

```
thread_0:
  MOV W9,#1
  STR W9,[X2]
  DMB ISH
  STR W3,[X4]

thread_1:
  MOV W9,#2
  SWP W9, W10, [X2]
  DMB ISHLD
  LDR W3,[X4]
```

Executing the compiled program on an Arm-based machine from a fixed initial state (where `x` and `y` are 0) produces one of the following outcomes, according to the AArch64 Memory Model contained in §B2 of the Arm Architecture Reference Manual [ARMARM]:

```

{ thread_1:r0=0; [y]=1; }
{ thread_1:r0=1; [y]=1; }
{ thread_1:r0=1; [y]=2; }

```

As such the unexpected outcome has disappeared. There are multiple Mappings that exhibit this behaviour, those effected make use of `SWP` and `LD<OP>` Assembly instructions. These include but are not limited to:

Atomic Operation	Assembly Sequence
<code>exchange(loc, val, sc)</code>	<code>MOV W4, #val; SWP W4, W10, [X1]</code>
<code>fetch_add(loc, val, sc)</code>	<code>MOV W4, #val; LDADD W4, W10, [X1]</code>

Where `x1` contains the address of `loc`.

4.7.2 Const-Qualified 128-bit Atomic Loads

Const-qualified data containing 128-bit atomic types should not be placed in readonly memory (the `.rodata` section).

Before LSE2, the only way to implement a single-copy 128-bit atomic load is by using a Read-Modify-Write sequence. The write is not visible to software if the memory is writeable. Compilers and runtimes should use the LSE2/LRCPC3 sequence when available.

5 Declarative statement of Mappings compatibility

To ensure that the above Mappings are ABI-compatible we test the compilation of Concurrent Programs, where each Atomic Operation is compiled to one of the aforementioned Mappings. We test if there is a compiled program that exhibits an outcome of execution according to the AArch64 Memory Model contained in §B2 of the Arm Architecture Reference Manual [ARMARM] that is not an outcome of execution of the source program under the ISO C model. In this section we define the process by which we test compatibility.

5.1 The Mix Testing Process

We test for Compiler bugs, a compiler bug is defined as an Outcome of a compiled program execution (under the AArch64 model) that is not an Outcome of execution of the source Concurrent Program (under the ISO C model). Consider the hypothetical example where a source Concurrent Program finishes execution in one of three possible outcomes:

```

{ thread_0:r0=0, thread_1:r0=1 }
{ thread_0:r0=1, thread_1:r0=0 }
{ thread_0:r0=1, thread_1:r0=1 }

```

and one possible compiled program outcome has the following according to the AArch64 Memory Model contained in §B2 of the Arm Architecture Reference Manual [ARMARM]:

```

{ thread_0:X3=0, thread_1:X3=0 } <--- Forbidden by source model, compiler bug!
{ thread_0:X3=0, thread_1:X3=1 }
{ thread_0:X3=1, thread_1:X3=0 }
{ thread_0:X3=1, thread_1:X3=1 }

```

By comparing x_3 and the local variable x_0 of the original Concurrent Program in this example we see there is one additional outcome of executing the compiled program that is not an outcome of executing the source program (under the respective models). This suggests the Mappings under question are incompatible, and a compiler that implements them exhibits a compiler bug. To ensure compatibility we therefore test for the absence of such Outcomes of the compiled programs when mixing all combinations of the above Mappings. We define the *Mix Testing* process as follows:

1. Given a C/C++ Concurrent Program.
2. Split it into its representative Atomic Operations.
3. Compile each Atomic Operation separately using a Compiler Profile that generates Assembly Sequences under a given Mapping.
4. Combine the Assembly Sequences into *multiple* possible Compiled Programs.
5. Compute the outcomes of executing the Source Concurrent Program under the ISO C memory model. Get source program outcomes S .
6. Compute the outcomes of each compiled program under the AArch64 memory model [ARMARM]. Get a set of compiled program outcomes C .
7. If any c in C exhibits a compiler bug with respect to the outcomes S then the given mappings are not interoperable.

Using Mix Testing we now define ABI-Compatibility of Atomic Operations.

5.2 Definition of ABI-Compatibility for Atomic Operations

A compiler that implements the above set of Mappings is ABI-Compatible with respect to other compilers that implement the Mappings, if Mix Testing their code generation finds no compiler bugs.

We impose some constraints on this definition:

- This is not a correctness guarantee, but rather a statement backed up by bounded testing. Atomic ABI-compatibility is thus tested for the Mappings above by generating C/C++ Concurrent Programs that permute combinations of Atomic Operations on each Thread of Execution. We bound our test size between 2 and 5 Threads of Execution, where each Thread has at least 1 Atomic Operation or Synchronization Operation and at most 5 Atomic Operations or Synchronization Operations. We do not make any statement about the ABI-Compatibility of Concurrent Programs outside these bounds.
- We test Concurrent Programs with a fixed initial state, loop unroll factor (equal to 1 loop unroll), and function calls or recursion.
- The above Mappings are not exhaustive, We hope that Arm's partners will submit requests for other Mappings to the ABI team using the issue tracker page on GitHub.
- This document makes no statement about the ABI-Compatibility of optimised Concurrent Programs, nor does a statement concerning the performance of compiled programs under the above Mappings when executed on a given Arm-based machine.
- This document makes no statement about the ABI-Compatibility of compilers that implement Mappings other than what is stated in this document.