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

2024Q1

Date of Issue: 5th April 2024

arm

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 C/C++ Atomics Application Binary Interface Standard for the Arm 64-bit 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

1.6 About the license

1.7 Contributions

1.8 Trademark notice

1.9 Copyright

Contents

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
00alp0	5th 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
ARMARM	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.
AAELF64	ELF for the Arm 64-bit Architecture (AArch64)	ELF for the Arm 64-bit Architecture (AArch64)
PAPER	CGO paper	Compiler Testing with Relaxed Memory Models

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 C/C++ 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 ...

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.

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 both Atomic Operations and Non-Atomic Operations (Operations that lack the atomic qualifier) to be deemed *concurrent*. This document focuses on compiling such programs for Arm-based machines that run the A64 instruction set.

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.

Shared-Memory Location

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

Memory Order Parameter

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.

Assembly Sequence

A sequence of A64 instructions, optionally including Atomic Instructions.

Mapping

A Mapping takes an Atomic Operation and Compiler Profile as input, producing an Assembly Sequence as output.

Compiler Profile

A Compiler implementation and command-line flags or attributes that use Mappings.

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 sto one of more Assembly Sequences 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(loc,val,relaxed)`	ARCH1	`option A`
	ARCH2	`option B`

Where ARCH is for example BASE (armv8), LSE, LSE2, LSE128, RCPC, or LRCPC3. ARCH describes the required extension, with BASE meaning Armv8-A with no extensions and LSE is shorthand for FEAT_LSE (likewise for the other extensions).

Lastly, all operations are in a shorthand form:

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

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`.

Note
* Using `WZR` or `XZR` for the destination register is invalid (Section 4.7).

Atomic Operation		Assembly Sequence
`store(loc,val,relaxed)`		`STR  W2, [X1]`
`store(loc,val,rel)` `store(loc,val,sc)`		`STLR  W2, [X1]`
`load(loc,relaxed)`		`LDR  W2, [X1]`
`load(loc,acq)`	BASE	`LDAR  W2, [X1]`
	RCPC	`LDAPR W2, [X1]`
`load(loc,sc)`		`LDAR  W2, [X1]`
`fence(relaxed)`		`NOP`
`fence(acq)`		`DMB ISHLD`
`fence(rel)` `fence(acq_rel)` `fence(sc)`		`DMB ISH`

`exchange(loc,val,relaxed)`	BASE	`loop:`
		`  LDXR  W0, [X1]`
		`  STXR  W3, W2, [X1]`
		`  CBNZ  W3, loop`

	LSE	`SWP  W2, W0, [X1]` *

Atomic Operation		Assembly Sequence
exchange(loc,val,acq)	BASE	loop: LDAXR W0, [X1] STXR W3, W2, [X1] CBNZ W3, loop
	LSE	SWPA W2, W0, [X1] *
exchange(loc,val,rel)	BASE	loop: LDXR W0, [X1] STLXR W3, W2, [X1] CBNZ W3, loop
	LSE	SWPL W2, W0, [X1] *
exchange(loc,val,acq_rel) exchange(loc,val,sc)	BASE	loop: LDAXR W0, [X1] STLXR W3, W2, [X1] CBNZ W3, loop
	LSE	SWPAL W2, W0, [X1] *

fetch_add(loc,val,relaxed)	BASE	loop:
		LDXR W0, [X1]
		ADD W2, W2, W0
		STXR W3, W2, [X1]
		CBNZ W3, loop

	LSE	LDADD W2, W0, [X1] *
fetch_add(loc,val,acq)	BASE	loop: LDAXR W0, [X1] ADD W2, W2, W0 STXR W3, W2, [X1] CBNZ W3, loop
	LSE	LDADDA W2, W0, [X1] *

fetch_add(loc,val,rel)	BASE	loop:
		LDXR W0, [X1]
		ADD W2, W2, W0
		STLXR W3, W2, [X1]
		CBNZ W3, loop

	LSE	LDADDL W2, W0, [X1] *
fetch_add(loc,val,acq_rel) fetch_add(loc,val,sc)	BASE	loop: LDXAR W0, [X1] ADD W2, W2, W0 STLXR W3, W2, [X1] CBNZ W3, loop
	LSE	LDADDAL W2, W0, [X1] *

Atomic Operation		Assembly Sequence

`compare_exchange_strong(`
`  loc,&exp,val,relaxed,`
`  relaxed)`
	BASE	`loop:`
		`  LDXR  W0, [X1]`
		`  CMP  W0, W4`
		`  B.NE  fail`
		`  STXR  W3, W2, [X1]`
		`  CBNZ  W3, loop`
		`fail:`

	LSE	`CAS  W0, W2, [X1]` *

`compare_exchange_strong(`
`  loc,&exp,val,acq,acq)`
	BASE	`loop:`
		`  LDAXR  W0, [X1]`
		`  CMP  W0, W4`
		`  B.NE  fail`
		`  STXR  W3, W2, [X1]`
		`  CBNZ  W3, loop`
		`fail:`

	LSE	`CASA  W0, W2, [X1]` *

`compare_exchange_strong(`
`  loc,&exp,val,rel,rel)`
	BASE	`loop:`
		`  LDXR  W0, [X1]`
		`  CMP  W0, W4`
		`  B.NE  fail`
		`  STLXR  W3, W2, [X1]`
		`  CBNZ  W3, loop`
		`fail:`

	LSE	`CASL  W0, W2, [X1]` *

`compare_exchange_strong(`
`  loc,&exp,val,acq_rel,acq)`
`compare_exchange_strong(`
`  loc,&exp,val,sc,sc)`
	BASE	`loop:`
		`  LDAXR  W0, [X1]`
		`  CMP  W0, W4`
		`  B.NE  fail`
		`  STLXR  W3, W2, [X1]`
		`  CBNZ  W3, loop`
		`fail:`

	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 Msappings 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	`loop:`
		`  LDXP  XZR, X1, [X4]`
		`  STXP  W5, X2, X3, [X4]`
		`  CBNZ  W5, loop`

	LSE	`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`

	LSE2	`STP  x2, X3, [X4]`

`store(loc,val,rel)`	BASE	`loop:`
		`  LDXP  XZR, X1, [X4]`
		`  STLXP  W5, X2, X3, [X4]`
		`  CBNZ  W5, loop`

	LSE	`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`

	LSE2	`DMB  ISH` `STP  X2, X3, [X4]`
	LRCPC3	`STILP  X2, X3, [X4]`

Atomic Operation		Assembly Sequence
store(loc,val,sc)	BASE	loop: LDXP XZR, X1, [X4] STLXP W5, X2, X3, [X4] CBNZ W5, loop

	LSE	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

	LSE2	DMB ISH STP X2, X3, [X4] DMB ISH
	LRCPC3	STILP X2, X3, [X4]
load(loc,relaxed)	BASE	loop: LDXP X0, X1, [X4] STXP W5, X0, X1, [X4] CBNZ W5, loop
	LSE	CASP X0, X1, X0, X1, [X4]
	LSE2	LDP X0, X1, [X4]
load(loc,acq)	BASE	loop: LDAXP X0, X1, [X4] STXP W5, X0, X1, [X4] CBNZ W5, loop
	LSE	CASPA X0, X1, X0, X1, [X4]
	LSE2	LDP X0, X1, [X4] DMB ISHLD
	LRCPC3	LDIAPP X0, X1, [X4]
load(loc,sc)	BASE	loop: LDAXP X0, X1, [X4] STXP W5, X0, X1, [X4] CBNZ W5, loop
	LSE	CASPA X0, X1, X0, X1, [X4]

	LSE2	LDAR X5, [X4]
		LDP X0, X1, [X4]
		DMB ISHLD

	LRCPC3	LDAR X5, [X4] LDIAPP X0, X1, [X4]

Atomic Operation		Assembly Sequence
exchange(loc,val,relaxed)	BASE	loop: LDXP X0, X1, [X4] STXP W5, X2, X3, [X4] CBNZ W5, loop

	LSE	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

	LSE128	MOV X0, X2
		MOV X1, X3
		SWPP X0, X1, [X4]

exchange(loc,val,acq)	BASE	loop: LDAXP X0, X1, [X4] STXP W5, X2, X3, [X4] CBNZ W5, loop

	LSE	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

	LSE128	MOV X0, X2
		MOV X1, X3
		SWPPA X0, X1, [X4]

exchange(loc,val,rel)	BASE	loop: LDXP X0, X1, [X4] STLXP W5, X2, X3, [X4] CBNZ W5, loop

	LSE	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

	LSE128	MOV X0, X2
		MOV X1, X3
		SWPPL X0, X1, [X4]

Atomic Operation		Assembly Sequence
exchange(loc,val,acq_rel) exchange(loc,val,sc)	BASE	loop: LDAXP X0, X1, [X4] STLXP W5, X2, X3, [X4] CBNZ W5, loop

	LSE	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

	LSE128	MOV X0, X2
		MOV X1, X3
		SWPPAL X0, X1, [X4]

fetch_add(loc,val,relaxed)	BASE	loop:
		LDXP X0, X1, [X4]
		ADDS X0, X0, X2
		ADC X1, X1, X3
		STXP W5, X2, X3, [X4]
		CBNZ W5, loop

	LSE	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

fetch_add(loc,val,acq)	BASE	loop: LDAXP X0, X1, [X4] ADDS X0, X0, X2 ADC X1, X1, X3 STXP W5, X2, X3, [X4] CBNZ W5, loop

	LSE	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

Atomic Operation		Assembly Sequence

`fetch_add(loc,val,rel)`	BASE	`loop:`
		`  LDXP  X0, X1, [X4]`
		`  ADDS  X0, X0, X2`
		`  ADC  X1, X1, X3`
		`  STLXP  W5, X2, X3, [X4]`
		`  CBNZ  W5, loop`

	LSE	`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`

`fetch_add(loc,val,acq_rel)`
`fetch_add(loc,val,sc)`
	BASE	`loop:`
		`  LDAXP  X0, X1, [X4]`
		`  ADDS  X0, X0, X2`
		`  ADC  X1, X1, X3`
		`  STXLP  W5, X2, X3, [X4]`
		`  CBNZ  W5, loop`

	LSE	`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`

`fetch_or(loc,val,relaxed)`	LSE128	`MOV  X0, X2`
		`MOV  X1, X3`
		`LDSETP  X0, X1, [X4]`

`fetch_or(loc,val,acq)`	LSE128	`MOV  X0, X2`
		`MOV  X1, X3`
		`LDSETPA  X0, X1, [X4]`

`fetch_or(loc,val,rel)`	LSE128	`MOV  X0, X2`
		`MOV  X1, X3`
		`LDSETPL  X0, X1, [X4]`

`fetch_or(loc,val,acq_rel)`
`fetch_or(loc,val,sc)`
	LSE128	`MOV  X0, X2`
		`MOV  X1, X3`
		`LDSETPAL X0, X1, [X4]`

`fetch_and(loc,val,relaxed)`	LSE128	`MVN  X0, X2`
		`MVN  X1, X3`
		`LDCLRP  X0, X1, [X4]`

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

`compare_exchange_strong(`
`loc,&exp,val,acq_rel,acq)`
`compare_exchange_strong(`
`loc,&exp,val,sc,sc)`
	BASE	`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`

	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. The special cases are identified below.

- Re-Ordering of Read-Modify-Write Effects and Acquire Fence
- Const-Qualified 128-bit Atomic Loads

4.7.1 Destination Register Should Not Be Zero Register for Read-Modify-Writes

A compiler is not permitted to rewrite the destination register to be the zero register for atomic operations that make use of `SWP` and `LD<OP>` Assembly instructions. These include but are not limited to:

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

Where `X1` contains the address of `loc`.

We annotate Mappings affected with * in section 4.2.

Please refer to Appendix: Read-Modify-Write Destination Register Semantics for information on why this example must be documented.

4.7.2 Const-Qualified 128-bit Atomic Loads Should Be Marked Mutable

Const-qualified data containing 128-bit atomic types should not be placed in read-only memory (such as 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 tested 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. Please refer to Appendix: Mix Testing for information on how ABI-compatibility is tested.

5.1 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. C/C++ Atomics 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 recommend that Arm's partners 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.

6 Appendix: Mix Testing

The status of this appendix is informative.

6.1 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 Memory Model contained in §B2 of the Arm Architecture Reference Manual [ARMARM]) that is not an outcome of execution of the source Concurrent Program (under the ISO C memory model). Consider the hypothetical example where a source Concurrent Program finishes execution in one of three possible outcomes (a reference for this notation is found here [PAPER]):

```
{ 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 `X3` and the local variable `r0` 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. Take an arbitrary Concurrent Program, when executed on the C/C++ memory model will produce outcomes *S*.
2. Split out the individual Atomic Operations from the initial concurrent program into individual source files.
3. Compile each individual source file containing an Atomic Operation using each Compiler Profile under test that generates Assembly Sequences under a given Mapping.
4. Combine the Assembly Sequences from above into *multiple* possible Compiled Programs.
5. Compute the outcomes of each compiled program under the AArch64 Memory Model contained in §B2 of the Arm Architecture Reference Manual [ARMARM]. Get a *set* of compiled program outcomes *C*.
6. If any compiled program set of outcomes *c* in *C* exhibits a Compiler Bug (Check that *c* is a subset of *S*) with then the given Mappings are not interoperable.

7 Appendix: Read-Modify-Write Destination Register Semantics

We elaborate on why in the following example.

Consider the following Concurrent Program:

code-block:

```
// 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 (a reference for this notation is found here [PAPER]):

```
{ 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 not legal according to this ABI for a compliant implementation piler 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`.

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.*

In this case the compiler introduces another outcome of Execution. To fix this issue, a compiler is not permitted to 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 affected make use of `SWP` and `LD<OP>` Assembly instructions.