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

2024Q1

Date of Issue: 19th August 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 lists the valid mappings from C/C++ Atomic Operations to sequences of AArch64 instructions. For further information on the memory model, refer to §B2 of the Arm Architecture Reference Manual [ARMARM].

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 ABI was written as part of Luke Geeson's PhD on testing the compilation of concurrent C/C++ with assistance from Wilco Dijkstra from Arm's Compiler Teams.

It is an offshoot from a paper that will be presented at OOPSLA 2024 [OOPSLA]: *Mix Testing: Specifying and Testing ABI Compatibility Of C/C++ Atomics Implementations* by Luke Geeson, James Brotherston, Wilco Dijkstra, Alastair Donaldson, Lee Smith, Tyler Sorensen, and John Wickerson.

1.5 Licence

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

1.9 Copyright

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 7

3 Overview **8**

4 AArch64 atomic mappings **9**

4.1 Synchronization Fences 9

4.2 32-bit types 9

4.3 8-bit types 12

4.4 16-bit types 12

4.5 64-bit types 12

4.6 128-bit types 12

5 Special Cases **19**

5.1 Unused result in Read-Modify-Write atomics 19

5.2 Const-Qualified 128-bit Atomic Loads 20

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	19th August 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)
CPPABI64	C++ ABI for the Arm 64-bit Architecture (AArch64)	C++ ABI for the Arm 64-bit Architecture (AArch64)
RATIONALE	Rationale Document for C11 Atomics ABI	Rationale Document for C11 Atomics ABI

Ref	External reference or URL	Title
PAPER	CGO paper	Compiler Testing with Relaxed Memory Models

2.4 Terms and Abbreviations

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

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

A unit of computation (e.g. a POSIX thread) of a process, managed by the OS.

Atomic Operation

An indivisble operation on a memory location. This can be a load, store, exchange, compare, or arithmetic operation. Atomics may be used to define higher level primitives including locks and concurrent queues. ISO C/C++ defines a range of supported atomic types and operations.

Concurrent Program

A C or C++ program that consists of one or more threads. Threads may communicate with each other through memory locations, using both Atomic Operations and standard memory accesses.

Memory Order Parameter

The order of memory accesses as executed by each thread may not be the same as the order they are written in the program. The Memory Order describes how memory accesses are ordered with respect to other memory accesses or Atomic Operations. ISO C/C++ defines a `memory_order` enum type for the set of memory orders.

Mapping

A mapping from an Atomic Operation to a sequence of AArch64 instructions.

3 Overview

AArch64 atomic mappings defines the mappings from C/C++ atomic operations to AArch64 that are interoperable.

Arbitrary registers may be used in the mappings. Instructions marked with * in the tables cannot use `WZR` or `XZR` as a destination register. This is further detailed in Special Cases.

Only some variants of `fetch_<op>` are listed since the mappings are identical except for a different `<op>`.

Atomic operations and Memory Order are abbreviated as follows:

Atomic Operation	Short form
`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(...)`

Memory Order Parameter	Short form
`memory_order_relaxed`	`relaxed`
`memory_order_acquire`	`acquire`
`memory_order_release`	`release`
`memory_order_acq_rel`	`acq_rel`
`memory_order_seq_cst`	`seq_cst`

If there are multiple mappings for an Atomic Operation, the rows of the table show the options:

Atomic Operation		AArch64
`store(loc,val,relaxed)`	ARCH1	`option A`
	ARCH2	`option B`

Where ARCH is either the base architecture (Armv8-A) or an extension like FEAT_LSE.

Suggestions and improvements to this specification may be submitted to the: issue tracker page on GitHub.

4 AArch64 atomic mappings

4.1 Synchronization Fences

Fence	AArch64
`atomic_thread_fence(relaxed)`	`NOP`
`atomic_thread_fence(acquire)`	`DMB ISHLD`

`atomic_thread_fence(release)`
`atomic_thread_fence(acq_rel)`
`atomic_thread_fence(seq_cst)`
	`DMB ISH`

4.2 32-bit types

In what follows, register `X1` contains the location `loc` and `W2` contains `val`. `W0` contains input `exp` in compare-exchange. The result is returned in `W0`.

Atomic Operation		AArch64
`store(loc,val,relaxed)`		`STR  W2, [X1]`
`store(loc,val,release)` `store(loc,val,seq_cst)`		`STLR  W2, [X1]`
`load(loc,relaxed)`		`LDR  W2, [X1]`
`load(loc,acquire)`	`Armv8-A`	`LDAR  W2, [X1]`
	`FEAT_RCPC`	`LDAPR  W2, [X1]`
`load(loc,seq_cst)`		`LDAR  W2, [X1]`

`exchange(loc,val,relaxed)`	`Armv8-A`	`loop:`
		`  LDXR  W0, [X1]`
		`  STXR  W3, W2, [X1]`
		`  CBNZ  W3, loop`

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

Atomic Operation		AArch64

exchange(loc,val,acquire)	Armv8-A	loop:
		  LDAXR W0, [X1]
		  STXR W3, W2, [X1]
		  CBNZ W3, loop

	FEAT_LSE	SWPA W2, W0, [X1] *

exchange(loc,val,release)	Armv8-A	loop:
		  LDXR W0, [X1]
		  STLXR W3, W2, [X1]
		  CBNZ W3, loop

	FEAT_LSE	SWPL W2, W0, [X1] *

exchange(loc,val,acq_rel)
exchange(loc,val,seq_cst)
	Armv8-A	loop:
		  LDAXR W0, [X1]
		  STLXR W3, W2, [X1]
		  CBNZ W3, loop

	FEAT_LSE	SWAL W2, W0, [X1] *

fetch_add(loc,val,relaxed)	Armv8-A	loop:
		  LDXR W0, [X1]
		  ADD W2, W2, W0
		  STXR W3, W2, [X1]
		  CBNZ W3, loop

	FEAT_LSE	LDADD W0, W2, [X1] *

fetch_add(loc,val,acquire)	Armv8-A	loop:
		  LDAXR W0, [X1]
		  ADD W2, W2, W0
		  STXR W3, W2, [X1]
		  CBNZ W3, loop

	FEAT_LSE	LDADDA W0, W2, [X1] *

fetch_add(loc,val,release)	Armv8-A	loop:
		  LDXR W0, [X1]
		  ADD W2, W2, W0
		  STLXR W3, W2, [X1]
		  CBNZ W3, loop

	FEAT_LSE	LDADDL W0, W2, [X1] *

Atomic Operation		AArch64

`fetch_add(loc,val,acq_rel)`
`fetch_add(loc,val,seq_cst)`
	Armv8-A	`loop:`
		`  LDAXR  W0, [X1]`
		`  ADD    W2, W2, W0`
		`  STLXR  W3, W2, [X1]`
		`  CBNZ   W3, loop`

	FEAT_LSE	`LDADDAL W0, W2, [X1] *`

`compare_exchange_strong(`
`  loc,exp,val,relaxed,relaxed)`
	Armv8-A	`  MOV    W4, W0`
		`loop:`
		`  LDXR   W0, [X1]`
		`  CMP    W0, W4`
		`  B.NE   fail`
		`  STXR   W3, W2, [X1]`
		`  CBNZ   W3, loop`
		`fail:`

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

`compare_exchange_strong(`
`  loc,exp,val,acquire,acquire)`
	Armv8-A	`  MOV    W4, W0`
		`loop:`
		`  LDAXR  W0, [X1]`
		`  CMP    W0, W4`
		`  B.NE   fail`
		`  STXR   W3, W2, [X1]`
		`  CBNZ   W3, loop`
		`fail:`

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

`compare_exchange_strong(`
`  loc,exp,val,release,release)`
	Armv8-A	`  MOV    W4, W0`
		`loop:`
		`  LDXR   W0, [X1]`
		`  CMP    W0, W4`
		`  B.NE   fail`
		`  STLXR  W3, W2, [X1]`
		`  CBNZ   W3, loop`
		`fail:`

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

Atomic Operation		AArch64

`compare_exchange_strong(`
`  loc,exp,val,acq_rel,acquire)`
`compare_exchange_strong(`
`  loc,exp,val,seq_cst,seq_cst)`
	Armv8-A	`  MOV    W4, W0`
		`loop:`
		`  LDAXR  W0, [X1]`
		`  CMP    W0, W4`
		`  B.NE   fail`
		`  STLXR  W3, W2, [X1]`
		`  CBNZ   W3, loop`
		`fail:`

	FEAT_LSE	`CASAL  W0, W2, [X1] *`

4.3 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 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 64-bit types

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

4.6 128-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 `val`. `X0` and `X1` contain input `exp` in compare-exchange. The result is returned in `X0` and `X1`.

Atomic Operation		AArch64

`store(loc,val,relaxed)`	Armv8-A	`loop:`
		`  LDXP   XZR, X1, [X4]`
		`  STXP   W5, X2, X3, [X4]`
		`  CBNZ   W5, loop`

	FEAT_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`

	FEAT_LSE2	`STP   X2, X3, [X4]`

Atomic Operation		AArch64

store(loc,val,release)	Armv8-A	loop:
		  LDXP XZR, X1, [X4]
		  STLXP W5, X2, X3, [X4]
		  CBNZ W5, loop

	FEAT_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

	FEAT_LSE2	DMB ISH STP X2, X3, [X4]
	FEAT_LRCPC3	STILP X2, X3, [X4]

store(loc,val,seq_cst)	Armv8-A	loop:
		  LDAXP XZR, X1, [X4]
		  STLXP W5, X2, X3, [X4]
		  CBNZ W5, loop

	FEAT_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

	FEAT_LSE2	DMB ISH STP X2, X3, [X4] DMB ISH
	FEAT_LRCPC3	STILP x2, X3, [X4]

load(loc,relaxed)	Armv8-A	loop:
		  LDXP X0, X1, [X4]
		  STXP W5, X0, X1, [X4]
		  CBNZ W5, loop

	FEAT_LSE	CASP X0, X1, X0, X1, [X4]
	FEAT_LSE2	LDP X0, X1, [X4]

Atomic Operation		AArch64

load(loc,acquire)	Armv8-A	loop:
		  LDAXP X0, X1, [X4]
		  STXP W5, X0, X1, [X4]
		  CBNZ W5, loop

	FEAT_LSE	CASPA X0, X1, X0, X1, [X4]
	FEAT_LSE2	LDP X0, X1, [X4] DMB ISHLD
	FEAT_LRCPC3	LDIAPP X0, X1, [X4]

load(loc,seq_cst)	Armv8-A	loop:
		  LDAXP X0, X1, [X4]
		  STXP W5, X0, X1, [X4]
		  CBNZ W5, loop

	FEAT_LSE	CASPA X0, X1, X0, X1, [X4]

	FEAT_LSE2	LDAR X5, [X4]
		LDP X0, X1, [X4]
		DMB ISHLD

	FEAT_LRCPC3	LDAR X5, [X4] LDIAPP X0, X1, [X4]

exchange(loc,val,relaxed)	Armv8-A	loop:
		  LDXP X0, X1, [X4]
		  STXP W5, X2, X3, [X4]
		  CBNZ W5, loop

	FEAT_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

	FEAT_LSE128	MOV X0, X2
		MOV X1, X3
		SWPP X0, X1, [X4]

Atomic Operation		AArch64

exchange(loc,val,acquire)	Armv8-A	loop:
		  LDAXP X0, X1, [X4]
		  STXP W5, X2, X3, [X4]
		  CBNZ W5, loop

	FEAT_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

	FEAT_LSE128	MOV X0, X2
		MOV X1, X3
		SWPPA X0, X1, [X4]

exchange(loc,val,release)	Armv8-A	loop:
		  LDXP X0, X1, [X4]
		  STLXP W5, X2, X3, [X4]
		  CBNZ W5, loop

	FEAT_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

	FEAT_LSE128	MOV X0, X2
		MOV X1, X3
		SWPPL X0, X1, [X4]

Atomic Operation		AArch64

exchange(loc,val,acq_rel)
exchange(loc,val,seq_cst)
	Armv8-A	loop:
		  LDAXP X0, X1, [X4]
		  STLXP W5, X2, X3, [X4]
		  CBNZ W5, loop

	FEAT_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

	FEAT_LSE128	MOV X0, X2
		MOV X1, X3
		SWPPAL X0, X1, [X4]

fetch_add(loc,val,relaxed)	Armv8-A	loop:
		  LDXP X0, X1, [X4]
		  ADDS X0, X0, X2
		  ADC X1, X1, X3
		  STXP W5, X0, X1, [X4]
		  CBNZ W5, loop

	FEAT_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

Atomic Operation		AArch64

`fetch_add(loc,val,acquire)`	Armv8-A	`loop:`
		`  LDAXP  X0, X1, [X4]`
		`  ADDS   X0, X0, X2`
		`  ADC    X1, X1, X3`
		`  STXP   W5, X0, X1, [X4]`
		`  CBNZ   W5, loop`

	FEAT_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`

`fetch_add(loc,val,release)`	Armv8-A	`loop:`
		`  LDXP   X0, X1, [X4]`
		`  ADDS   X0, X0, X2`
		`  ADC    X1, X1, X3`
		`  STLXP  W5, X0, X1, [X4]`
		`  CBNZ   W5, loop`

	FEAT_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,seq_cst)`
	Armv8-A	`loop:`
		`  LDAXP  X0, X1, [X4]`
		`  ADDS   X0, X0, X2`
		`  ADC    X1, X1, X3`
		`  STLXP  W5, X0, X1, [X4]`
		`  CBNZ   W5, loop`

	FEAT_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`

Atomic Operation		AArch64

`fetch_or(loc,val,relaxed)`	FEAT_LSE128	`MOV    X0, X2`
		`MOV    X1, X3`
		`LDSETP X0, X1, [X4]`

`fetch_or(loc,val,acquire)`	FEAT_LSE128	`MOV     X0, X2`
		`MOV     X1, X3`
		`LDSETPA X0, X1, [X4]`

`fetch_or(loc,val,release)`	FEAT_LSE128	`MOV     X0, X2`
		`MOV     X1, X3`
		`LDSETPL X0, X1, [X4]`

`fetch_or(loc,val,acq_rel)`
`fetch_or(loc,val,seq_cst)`
	FEAT_LSE128	`MOV      X0, X2`
		`MOV      X1, X3`
		`LDSETPAL X0, X1, [X4]`

`fetch_and(loc,val,relaxed)`	FEAT_LSE128	`MVN    X0, X2`
		`MVN    X1, X3`
		`LDCLRP X0, X1, [X4]`

`fetch_and(loc,val,acquire)`	FEAT_LSE128	`MVN     X0, X2`
		`MNV     X1, X3`
		`LDCLRPA X0, X1, [X4]`

`fetch_and(loc,val,release)`	FEAT_LSE128	`MVN     X0, X2`
		`MVN     X1, X3`
		`LDCLRPL X0, X1, [X4]`

`fetch_and(loc,val,acq_rel)`
`fetch_and(loc,val,seq_cst)`
	FEAT_LSE128	`MVN      X0, X2`
		`MVN      X1, X3`
		`LDCLRPAL X0, X1, [X4]`

`compare_exchange_strong(`
`loc,exp,val,relaxed,relaxed)`
	Armv8-A	`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`

	FEAT_LSE	`CASP    X0, X1, X2, X3, [X4]`

Atomic Operation		AArch64

`compare_exchange_strong(`
`  loc,exp,val,acquire,acquire)`
`compare_exchange_strong(`
`  loc,exp,val,acquire,relaxed)`
	Armv8-A	`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`

	FEAT_LSE	`CASPA  X0, X1, X2, X3, [X4]`

`compare_exchange_strong(`
`  loc,exp,val,release,relaxed)`
	Armv8-A	`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`

	FEAT_LSE	`CASPL  X0, X1, X2, X3, [X4]`

`compare_exchange_strong(`
`  loc,exp,val,acq_rel,acquire)`
`compare_exchange_strong(`
`  loc,exp,val,seq_cst,acquire)`
	Armv8-A	`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`

	FEAT_LSE	`CASPAL  X0, X1, X2, X3, [X4]`

5 Special Cases

5.1 Unused result in Read-Modify-Write atomics

`CAS`, `SWP` and `LD<OP>` instructions must not use the zero register if the result is not used since it allows reordering of the read past a `DMB ISHLD` barrier. Affected instructions are marked with `*`.

5.2 Const-Qualified 128-bit Atomic Loads

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

Before FEAT_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 prefer the FEAT_LSE2/FEAT_LRCPC3 sequence when available.