ARM® C Language Extensions Release 1.1

Document number: IHI 0053B
Date of Issue: 13/11/13

Abstract

This document specifies the ARM C Language Extensions to enable C/C++ programmers to exploit the ARM architecture with minimal restrictions on source code portability.

Keywords

ACLE, AARCH, ABI, C, C++, compiler, armcc, gcc, intrinsic, macro, attribute, NEON, SIMD, vector, atomic

How to find the latest release of this specification or report a defect in it

Please check the *ARM Information Center* (http://infocenter.arm.com/) for a later release if your copy is more than one year old. This document may be found under "Developer Guides and Articles", "Software Development".

Please report defects in this specification to arm dot acle at arm dot com.

Licence

Confidentiality status

This document is Open Access. This document has no restriction on distribution.

Proprietary notice

Contents

1 ABOUT THIS DOCUMENT

1.1 Change control

1.1.1 Current status and anticipated changes

This document is release 1.1 of the ARM C Language Extensions (ACLE).

Anticipated changes to this document include:

- Typographical corrections.
- Clarifications.
- Compatible extensions.

1.1.2 Change history

Issue	Date	By	Change
A	11/11/11	AG	First release
B	13/11/13	AG	Version 1.1. Editorial changes. Corrections and completions to intrinsics as detailed in 3.3. Updated for C11/C++11.

1.2 References

This document refers to the following documents.

Ref	Doc No	Author(s)	Title
ARMARM	ARM DDI 0406C	ARM	ARM Architecture Reference Manual (7-A / 7-R)
ARMv7M	ARM DDI 0403C	ARM	ARM Architecture Reference Manual (7-M)
AAPCS	ARM IHI 0042D	ARM	Procedure Call Standard
BA	ARM IHI 0045C	ARM	EABI Addenda and Errata – Build Attributes
C++11	ISO/IEC 14882:2011	ISO	Standard C++ (based on draft N3337)
C11	ISO/IEC 9899:2011	ISO	Standard C (based on draft N1570)
C99	ISO 9899:1999	ISO	Standard C (“C99”)
cxxabi	http://mentorembedded.github.com/cxx-abi/abi.html	Code-Sourcery	Itanium C++ ABI (rev. 1.86)
G.191	T-REC-G.191-200508-I	ITU-T	Software Tool Library 2005 User’s Manual
GNUC	http://gcc.gnu.org/onlinedocs	GNU/FSF	GNU C Compiler Collection
IA-64	245370-003	Intel	Intel Itanium Processor-Specific ABI
IEEE-FP	IEEE 754-2008	IEEE	IEEE floating-point
POSIX	IEEE 1003.1	IEEE / TOG	The Open Group base specifications
Warren	ISBN 0-201-91465-4	H. Warren	“Hacker’s Delight”, pub. Addison-Wesley 2003

1.3 Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
AAPCS	ARM Procedure Call Standard, part of the ABI, defined in [AAPCS]
ABI	ARM Application Binary Interface
ACLE	ARM C Language Extensions, as defined in this document
Advanced SIMD	a 64-bit/128-bit SIMD instruction set defined as part of the ARM architecture
build attributes	object build attributes indicating configuration, as defined in [BA]
ILP32	a 32-bit address mode where '`long`' is a 32-bit type
LLP64	a 64-bit address mode where '`long`' is a 32-bit type
LP64	a 64-bit address mode where '`long`' is a 64-bit type
NEON™	an implementation of the ARM Advanced SIMD extensions
quadword	a 128-bit quantity, not necessarily 128-bit aligned
SIMD	any instruction set that operates simultaneously on multiple elements of a vector data type
Thumb®	the Thumb instruction set extension to ARM
VFP	the original ARM non-SIMD floating-point instruction set
word	a 32-bit quantity, in memory or a register

1.4 Your licence to use this specification

IMPORTANT: THIS IS A LEGAL AGREEMENT ("LICENCE") BETWEEN YOU (AN INDIVIDUAL OR SINGLE ENTITY WHO IS RECEIVING THIS DOCUMENT DIRECTLY FROM ARM LIMITED) ("LICENSEE") AND ARM LIMITED ("ARM") FOR THE SPECIFICATION DEFINED IMMEDIATELY BELOW. BY DOWNLOADING OR OTHERWISE USING IT, YOU AGREE TO BE BOUND BY ALL OF THE TERMS OF THIS LICENCE. IF YOU DO NOT AGREE TO THIS, DO NOT DOWNLOAD OR USE THE SPECIFICATION.

"Affiliate" means any entity controlling, controlled by or under common control with a party (in fact or in law, via voting securities, management control or otherwise) and "affiliated" shall be construed accordingly.

"Assert" means to allege infringement in legal or administrative proceedings, or proceedings before any other competent trade, arbitral or international authority.

"Necessary" means with respect to any claims of any patent, those claims which, without the appropriate permission of the patent owner, will be infringed when implementing the Specification because no alternative, commercially reasonable, non-infringing way of implementing the Specification is known.

"Specification" means, and is limited to, the version of the specification for the ARM C Language Extensions for the ARM architecture comprised in this document. Notwithstanding the foregoing, "Specification" shall not include (i) the implementation of other published specifications referenced in this Specification; (ii) any enabling technologies that may be necessary to make or use any product or portion thereof that complies with this Specification, but are not themselves expressly set forth in this Specification (e.g. compiler front ends, code generators, back ends, libraries or other compiler, assembler or linker technologies; validation or debug software or hardware; applications, operating system or driver software; RISC architecture; processor microarchitecture); (iii) maskworks and physical layouts of integrated circuit designs; or (iv) RTL or other high level representations of integrated circuit designs.

"Specification Intellectual Property" means ARM copyrights in the Specification and the claims of patents (if any) owned by ARM that are Necessary to develop, have developed, sell, supply or otherwise distribute software products which comply with the Specification, but specifically excludes any trademark, service mark, or trade name of ARM or its Affiliates.

1. The Specification is owned by ARM or its licensors and is protected by copyright laws and international copyright treaties as well as other intellectual property laws and treaties. Use of any trademarks is also not covered by this Licence. The Specification is licensed not sold.
2. ARM hereby grants to LICENSEE, subject to the terms and conditions of this Licence, under the Specification Intellectual Property a perpetual, non-exclusive, non-transferable, fully paid, worldwide limited licence (without the right to sublicense) to use and copy this Specification solely for the purpose of developing, having developed, offering to sell, selling, supplying or otherwise distributing software products which comply with the Specification. No right is granted under this Licence to implement the Specification in any hardware.
3. THE SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES EXPRESS, IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF SATISFACTORY QUALITY, MERCHANTABILITY, NONINFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE. THE SPECIFICATION MAY INCLUDE ERRORS. ARM RESERVES THE RIGHT TO INCORPORATE MODIFICATIONS TO THE SPECIFICATION IN LATER REVISIONS OF IT, AND TO MAKE IMPROVEMENTS OR CHANGES IN THE SPECIFICATION OR THE PRODUCTS OR TECHNOLOGIES DESCRIBED THEREIN AT ANY TIME.
4. This Licence shall immediately terminate and shall be unavailable to LICENSEE if LICENSEE or any party affiliated to LICENSEE Asserts any patents against ARM, ARM Affiliates, third parties who have a valid licence from ARM for the Specification, or any customers or distributors of any of them based upon a claim that a LICENSEE (or LICENSEE affiliate) patent is Necessary to implement the Specification.
5. THE MAXIMUM LIABILITY OF ARM TO LICENSEE IN AGGREGATE FOR ALL CLAIMS MADE AGAINST ARM IN CONTRACT, TORT OR OTHERWISE UNDER OR IN CONNECTION WITH THE SUBJECT MATTER OF THIS LICENCE SHALL NOT EXCEED THE GREATER OF (I) THE TOTAL OF SUMS PAID BY YOU TO ARM (IF ANY) FOR THIS LICENCE AND (II) US$10.00. THE LIMITATIONS, EXCLUSIONS AND DISCLAIMERS IN THIS LICENCE SHALL APPLY TO THE MAXIMUM EXTENT ALLOWED BY APPLICABLE LAW.
6. This Licence is governed by English Law and the jurisdiction of the English courts shall apply to all aspects of this Licence, its interpretation and enforcement. Except where ARM agrees otherwise in a written contract signed by you and ARM, this is the only agreement between you and ARM relating to the Specification and it may only be modified by written agreement between you and ARM. Except as expressly agreed in writing, this Licence may not be modified by purchase orders, advertising or other representation by any person. If any clause or sentence in this Licence is held by a court of law to be illegal or unenforceable the remaining provisions of this Licence shall not be affected thereby. The failure by ARM to enforce any of the provisions of this Licence, unless waived in writing, shall not constitute a waiver of ARM's rights to enforce such provision or any other provision of this Licence in the future.
7. US Government Restrictions: Use, duplication, reproduction, release, modification, disclosure or transfer of this commercial documentation is restricted in accordance with the terms of this Licence.

ARM Contract reference LEC-PRE-00530 AP/AG (February 2011)

2 SCOPE

The ARM C Language Extensions (ACLE) specification specifies source language extensions and implementation choices that C/C++ compilers can implement in order to allow programmers to better exploit the ARM architecture.

The extensions include:

- Predefined macros that provide information about the functionality of the target architecture (for example, whether it has hardware floating-point)
- Intrinsic functions
- Attributes that can be applied to functions, data and other entities

This specification does not standardize command-line options, diagnostics or other external behavior of compilers.

The intended users of this specification are:

- Application programmers wishing to adapt or hand-optimize applications and libraries for ARM targets
- System programmers needing low-level access to ARM targets beyond what C/C++ provides for
- Compiler implementors, who will implement this specification
- Implementors of IDEs, static analysis tools etc. who wish to deal with the C/C++ source language extensions when encountered in source code

Some of the material – specifically, the architecture/CPU namings, and the feature test macros – may also be applicable to assemblers and other tools.

ACLE is not a hardware abstraction layer (HAL), and does not specify a library component – but it may make it easier to write a HAL or other low-level library in C rather than assembler.

3 INTRODUCTION

Modern computer architectures (such as ARM) include architectural features that go beyond the set of operations available in C/C++. These features may include SIMD and saturating instructions. Exploiting these features to improve program efficiency has in the past caused "lock-in" to compilers, or to individual CPUs.

The intention of the ARM C Language Extensions (ACLE) is to allow the writing of applications and middleware code that is portable across compilers, and across ARM architecture variants, while exploiting the unique features of the ARM architecture family.

The design principles for ACLE can be summarized as:

- be implementable in (or as an addition to) current C/C++ implementations
- build on and standardize existing practice where possible

Notably, ACLE standardizes the NEON (Advanced SIMD) intrinsics.

ACLE incorporates some language extensions introduced in the GCC C compiler. Current GCC documentation [GCC] can be found at http://gcc.gnu.org/onlinedocs/gcc. Formally it should be assumed that ACLE refers to the documentation for GCC 4.5.1: http://gcc.gnu.org/onlinedocs/gcc-4.5.1/gcc/.

Some of the ACLE extensions are not specific to the ARM architecture but have proven to be of particular benefit in low-level and systems programming; examples include features for controlling the alignment and packing of data, and some common operations such as word rotation and reversal. As and when features become available in international standards (and implementations), it is recommended to use these in preference to ACLE. When implementations are widely available, any ACLE-specific features can be expected to be deprecated.

3.1 ACLE and 64-bit architectures

This revision of ACLE is intended to be efficiently implementable on 64-bit architectures, and to aid in writing code that runs efficiently on both 32-bit and 64-bit architectures.

Full support for features introduced in 64-bit architectures will be added in future revisions of ACLE.

3.2 Change history

The following sections highlight changes which implementors should be aware of. For tracking purposes the internal defect references (e.g. "[ACLE-123]") are given.

3.3 Changes between ACLE 1.0 and ACLE 1.1

Most changes in ACLE 1.1 are corrections to bring ACLE in line with current practice, or remove inconsistencies.

3.3.1 General changes

- updated specification for C11/C++11 [ACLE-52]
- clarified behavior of header files in C++ (section 4.4) [ACLE-73]
- added some 64-bit intrinsics (section 9.2) [ACLE-49]
- added `__sevl()` and `__pldx()` intrinsics [ACLE-43]
- added `__cls()` intrinsic (section 9.2) [ACLE-76]
- added `__ARM_32BIT_STATE` (section 6.4.1) [ACLE-67]
- added `__ARM_FEATURE_IDIV` (section 6.4.10) [ACLE-95]

3.3.2 NEON intrinsics

- removed `vrhsub()` intrinsic [ACLE-37]
- added `vld1Q_ST_xN` [ACLE-51]
- added `vst1Q_ST_xN` [ACLE-94]
- added quadword form of `vldN_lane` [ACLE-39]
- `vtst` intrinsics work on `poly16` types [ACLE-77]
- added section 12.2.3 on additional scalar types [ACLE-53, ACLE-54]
- noted restrictions on some other NEON intrinsics
- clarified constant range for `vextQ_ST` [ACLE-96]

4 C LANGUAGE EXTENSIONS

4.1 Fundamental data types

This section overlaps with the specification of the ARM Procedure Call Standard, particularly [AAPCS 4.1]. ACLE extends C by providing some types not present in Standard C and defining how they are dealt with by the AAPCS. It also extends some of the guarantees of C, allowing assumptions to be made in source code beyond those permitted by Standard C.

Plain ‘`char`’ is unsigned, as specified in the ABI [AAPCS 7.1.1].

When pointers are 32 bits, the ‘`long`’ type is 32 bits (ILP32 model).

When pointers are 64 bits, the ‘`long`’ type may be either 64 bits (LP64 model) or 32 bits (LLP64 model).

This release of ACLE does not deal with platforms with 64-bit pointers.

4.1.1 Implementation-defined type properties

ACLE and the ARM ABI allow implementations some freedom in order to conform to long-standing conventions in various environments. It is suggested that implementations set suitable defaults for their environment (and document what these are) but allow the default to be overridden.

The signedness of a plain ‘`int`’ bit-field is implementation-defined [C99 6.7.2#4].

Whether the underlying type of an enumeration is minimal or at least 32-bit, is implementation-defined. The predefined macro `__ARM_SIZEOF_MINIMAL_ENUM` should be defined as 1 or 4 according to the size of a minimal enumeration type such as `enum { X=0 }`.

`wchar_t` may be 2 or 4 bytes. The predefined macro `__ARM_SIZEOF_WCHAR_T` should be defined as the same number.

Where an implementation conforms to the ARM ABI, an object that is sensitive to implementation-defined settings must indicate those settings in object build attributes, e.g. `Tag_ABI_enum_size` and `Tag_ABI_PCS_wchar_t`, as specified in [BA]. See also section 6.8.

4.1.2 Half-precision floating-point

The `__fp16` type denotes half-precision (16-bit) floating-point. It is not required to be provided when not implemented in hardware. The recommended way to test for this hardware support is to test bit 1 in `__ARM_FP`.

Implementations which support 16-bit floating-point support two formats: the “binary16” format defined in [IEEE-FP], and an alternative format, defined by ARM, which extends the range by removing support for infinities and NaNs. Both formats are described in [ARM ARM A2.7.4]. Toolchains are not required to support the alternative format. The format in use can be selected at runtime but ACLE assumes it is fixed for the life of a program. If 16-bit floating-point is available, one of `__ARM_FP16_FORMAT_IEEE` and `__ARM_FP16_FORMAT_ALTERNATIVE` will be defined to indicate the format in use. An implementation conforming to the ARM ABI will set the `Tag_ABI_FP_16bit_format` build attribute.

16-bit floating point is a storage and interchange format only. Values of `__fp16` type promote to (at least) `float` when used in arithmetic operations, in the same way that values of `char` or `short` types promote to `int`. There is no arithmetic directly on 16-bit values.

Conversion from 64-bit to 16-bit, i.e. from `double` to `__fp16`, must round only once. (With round-to-nearest, converting first to 32-bit and then to 16-bit could give an incorrectly rounded result.) Because in current ARM hardware floating-point architectures this is not a primitive operation, it may be faster to convert first to single-precision and then to half-precision:

```
double xd;
__fp16 xs = (float)xd;
```

rather than:

```
double xd;
__fp16 xs = xd;
```

__fp16 cannot be used as an argument or result type, though it can be used as a field in a structure passed as an argument or result, or passed via a pointer. C++ name mangling is "Dh" as defined in [cxxabi], and is the same for both the IEEE and alternative formats.

In this example, the floating-point addition is done in single (32-bit) precision:

```
void add(__fp16 *z, __fp16 const *x, __fp16 const *y, int n) {
  int i;
  for (i = 0; i < n; ++i) z[i] = x[i] + y[i];
}
```

4.2 Predefined macros

Several predefined macros are defined. Generally these define features of the architecture being targeted, or how the C/C++ implementation uses the architecture. These macros are detailed in section 6 and there is a summary table in 6.9. All ACLE predefined macros start with the prefix __ARM.

4.3 Intrinsics

ACLE standardizes intrinsics to access the NEON (Advanced SIMD) extension. These intrinsics are intended to be compatible with existing implementations. Before using the NEON intrinsics or data types, the <arm_neon.h> header must be included. The NEON intrinsics are defined in section 12. Note that the NEON intrinsics and data types are in the user namespace.

ACLE also standardizes other intrinsics to access ARM instructions which do not map directly to C operators – generally either for optimal implementation of algorithms, or for accessing specialist system-level features. Intrinsics are defined further in various following sections.

Before using the non-NEON intrinsics, the <arm_acle.h> header should be included.

Whether intrinsics are macros, functions or built-in operators is unspecified. For example:

- it is unspecified whether applying #undef to an intrinsic removes the name from visibility
- it is unspecified whether it is possible to take the address of an intrinsic

However, each argument must be evaluated at most once. So this definition is acceptable:

```
#define __rev(x) __builtin_bswap32(x)
```

but this is not:

```
#define __rev(x) ((((x) & 0xff) << 24) | (((x) & 0xff00) << 8) | \
                 (((x) & 0xff0000) >> 8) | ((x) >> 24))
```


4.3.1 Constant arguments to intrinsics

Some intrinsics may require arguments that are constant at compile-time, to supply data that is encoded into the immediate fields of an instruction. Typically, these intrinsics require an integral-constant-expression in a specified range, or sometimes a string literal. An implementation should produce a diagnostic if the argument does not meet the requirements.

4.4 ACLE headers

`<arm_acle.h>` is provided to make the non-NEON intrinsics available. These intrinsics are in the C implementation namespace and begin with double underscores. It is unspecified whether they are available without the header being included. The `__ARM_ACLE` macro should be tested before including the header:

```
#ifdef __ARM_ACLE
#include <arm_acle.h>
#endif /* __ARM_ACLE */
```

`<arm_neon.h>` is provided to define the NEON intrinsics. As these intrinsics are in the user namespace, an implementation would not normally define them until the header is included. The `__ARM_NEON` macro should be tested before including the header:

```
#ifdef __ARM_NEON
#include <arm_neon.h>
#endif /* __ARM_NEON */
```

These headers behave as standard library headers; repeated inclusion has no effect beyond the first include.

4.4.1 Interaction with standard headers

It is unspecified whether the ACLE headers include the standard headers `<assert.h>`, `<stdint.h>` or `<inttypes.h>`. However, the ACLE headers will not define the standard type names (`uint32_t` etc.) except by inclusion of the standard headers. Programmers wanting to use the standard types in their own code are recommended to use the standard headers explicitly rather than relying on their inclusion by ACLE headers.

4.4.2 Use of ACLE headers in C++

In C++, the following source code fragments are expected to work correctly:

```
#include <stdint.h>
// UINT64_C not defined here since we did not set __STDC_FORMAT_MACROS
...
#include <arm_neon.h>
```

and

```
#include <arm_neon.h>
...
#define __STDC_FORMAT_MACROS
#include <stdint.h>
// ... UINT64_C is now defined
```

Names defined by ACLE are in the global namespace – there is no separate ACLE namespace.

ACLE headers must not have the side-effect of injecting the standard namespace.

4.5 Attributes

GCC-style attributes are provided to annotate types, objects and functions with extra information, such as alignment. These attributes are defined in section 7.

4.6 Implementation strategies

An implementation may choose to define all the ACLE non-NEON intrinsics as true compiler intrinsics, i.e. built-in functions. The <arm_acle.h> header would then have no effect.

Alternatively, <arm_acle.h> could define the ACLE intrinsics in terms of already supported features of the implementation, e.g. compiler intrinsics with other names, or inline functions using inline assembler.

5 ARCHITECTURE AND CPU NAMES

5.1 Introduction

The intention of this section is to standardize architecture names, e.g. for use in compiler command lines. Toolchains should accept these names case-insensitively where possible, or use all lowercase where not possible. Tools may apply local conventions such as using hyphens instead of underscores.

(Note: processor names, including from the ARM Cortex™ family, are used as illustrative examples. This specification is applicable to any processors implementing the ARM architecture.)

5.2 Architecture names

5.2.1 CPU architecture

The recommended CPU architecture names are as specified under `Tag_CPU_arch` in [BA]. For details of how to use predefined macros to test architecture in source code, see 6.4.1.

The following table lists the architectures and the ARM and Thumb® instruction set versions.

Name	Features	ARM	Thumb	Example processor
`ARMv4`	ARM v4	4		DEC/Intel StrongARM
`ARMv4T`	ARM v4 with Thumb instruction set	4	2	ARM7TDMI
`ARMv5T`	ARM v5 with Thumb instruction set	5	2	ARM10TDMI
`ARMv5TE`	ARM v5T with DSP extensions	5	2	ARM9E, Intel XScale
`ARMv5TEJ`	ARM v5TE with Jazelle® extensions	5	2	ARM926EJ
`ARMv6`	ARM v6 (includes TEJ)	6	2	ARM1136J r0
`ARMv6K`	ARM v6 with kernel extensions	6	2	ARM1136J r1
`ARMv6T2`	ARM v6 with Thumb-2 architecture	6	3	ARM1156T2
`ARMv6Z`	ARM v6K with TrustZone® extensions (includes K)	6	2	ARM1176JZ-S
`ARMv6-M`	Thumb-1 only (M-profile)		2	Cortex-M0, Cortex-M1
`ARMv7-A`	ARM v7 application profile	7	4	Cortex-A8, Cortex-A9
`ARMv7-R`	ARM v7 realtime profile	7	4	Cortex-R4
`ARMv7-M`	ARM v7 microcontroller profile: Thumb-2 instructions only		4	Cortex-M3
`ARMv7E-M`	ARM v7-M with DSP extensions		4	Cortex-M4

Note that there is some architectural variation that is not visible through ACLE; either because it is only relevant at the system level (e.g. the large physical address extension) or because it would be handled by the compiler (e.g. hardware integer divide might or might not be present in the ARM v7-A architecture).

5.2.2 FPU architecture

For details of how to test FPU features in source code, see 6.5. In particular, for testing which precisions are supported in hardware, see 6.5.1.

Name	Features	Example processor
`VFPv2`	VFPv2	ARM1136JF-S
`VFPv3`	VFPv3	Cortex-A8
`VFPv3_FP16`	VFPv3 with FP16	Cortex-A9 (with NEON)
`VFPv3_D16`	VFPv3 with 16 D-registers	Cortex-R4F
`VFPv3_D16_FP16`	VFPv3 with 16 D-registers and FP16	Cortex-A9 (without NEON), Cortex-R7
`VFPv3_SP_D16`	VFPv3 with 16 D-registers, single-precision only	Cortex-R5 with SP-only
`VFPv4`	VFPv4 (including FMA and FP16)	Cortex-A15
`VFPv4_D16`	VFPv4 (including FMA and FP16) with 16 D-registers	Cortex-A5 (VFP option)
`FPv4_SP`	FPv4 with single-precision only	Cortex-M4.fp

5.3 CPU names

ACLE does not standardize CPU names for use in command-line options and similar contexts. Standard vendor product names should be used.

Object producers should place the CPU name in the `Tag_CPU_name` build attribute.

6 FEATURE TEST MACROS

6.1 Introduction

The feature test macros allow programmers to determine the availability of ACLE or subsets of it, or of target architectural features. This may indicate the availability of some source language extensions (e.g. intrinsics) or the likely level of performance of some standard C features, such as integer division and floating-point.

Several macros are defined as numeric values to indicate the level of support for particular features. These macros are undefined if the feature is not present. (Aside: in Standard C/C++, references to undefined macros expand to 0 in preprocessor expressions, so a comparison such as

```
#if __ARM_ARCH >= 7
```

will have the expected effect of evaluating to false if the macro is not defined.)

All ACLE macros begin with the prefix `__ARM_`. All ACLE macros expand to integral constant expressions suitable for use in an `#if` directive, unless otherwise specified. Syntactically, they must be primary-expressions – generally this means an implementation should enclose them in parentheses if they are not simple constants.

6.2 Testing for ARM C Language Extensions

`__ARM_ACLE` is defined to the version of this specification implemented, as 100*major version + minor_version. An implementation implementing version 1.1 (this version) of the ACLE specification will define `__ARM_ACLE` as `101`.

6.3 Endianness

`__ARM_BIG_ENDIAN` is defined as 1 if data is stored by default in big-endian format. If the macro is not set, data is stored in little-endian format. (Aside: the “mixed-endian” format for double-precision numbers, used on some very old ARM FPU implementations, is not supported by ACLE or the ARM ABI.)

6.4 ARM and Thumb instruction set architecture and features

References to “the target architecture” refer to the target as configured in the tools, for example by appropriate command-line options. This may be a subset or intersection of actual targets, in order to produce a binary that runs on more than one real architecture. For example, use of specific features may be disabled.

In some cases, hardware features may be accessible from only one or other of ARM or Thumb instruction state. For example, in the v5TE and v6 architectures, “DSP” instructions and (where available) VFP instructions, are only accessible in ARM state, while in the v7-R architecture, hardware integer divide is only accessible from Thumb state. Where both states are available, the implementation should set feature test macros indicating that the hardware feature is accessible. To provide access to the hardware feature, an implementation might override the programmer’s preference for target instruction set, or generate an interworking call to a helper function. This mechanism is outside the scope of ACLE. In cases where the implementation is given a hard requirement to use only one state (e.g. to support validation, or post-processing) then it should set feature test macros only for the hardware features available in that state – as if compiling for a core where the other instruction set was not present.

An implementation that allows a user to indicate which functions go into which state (either as a hard requirement or a preference) is not required to change the settings of architectural feature test macros.

6.4.1 ARM/Thumb instruction set architecture

`__ARM_ARCH` is defined as an integer value indicating the current ARM instruction set architecture (e.g. 7 for the ARM v7-A architecture implemented by Cortex-A8 or the ARM v7-M architecture implemented by Cortex-M3). Since ACLE only supports the ARM architecture, this macro would always be defined in an ACLE implementation.

Note that the `__ARM_ARCH` macro is defined even for cores which only support the Thumb instruction set.

`__ARM_ARCH_ISA_ARM` is defined to 1 if the core supports the ARM instruction set. It is not defined for M-profile cores.

`__ARM_ARCH_ISA_THUMB` is defined to 1 if the core supports the original Thumb instruction set (including the v6-M architecture) and 2 if it supports the Thumb-2 instruction set as found in the v6T2 architecture and all v7 architectures.

`__ARM_32BIT_STATE` is defined to 1 if code is being generated for a 32-bit instruction set such as ARM or Thumb. This macro was introduced in ACLE 1.1.

6.4.2 Architectural profile (A, R, M or pre-Cortex)

`__ARM_ARCH_PROFILE` is defined as 'A', 'R', 'M' or 'S', or unset, according to the architectural profile of the target. 'S' indicates the common subset of 'A' and 'R'.

This macro corresponds to the `Tag_CPU_arch_profile` object build attribute. It may be useful to writers of system code. It is expected in most cases programmers will use more feature-specific tests.

The macro is undefined for architectural targets which predate the use of architectural profiles.

6.4.3 Unaligned access supported in hardware

`__ARM_FEATURE_UNALIGNED` is defined if the target supports unaligned access in hardware, at least to the extent of being able to load or store an integer word at any alignment with a single instruction. (There may be restrictions on load-multiple and floating-point accesses.) Note that whether a code generation target permits unaligned access will in general depend on the settings of system register bits, so an implementation should define this macro to match the user's expectations and intentions. For example, a command-line option might be provided to disable the use of unaligned access, in which case this macro would not be defined.

6.4.4 LDREX/STREX

`__ARM_FEATURE_LDREX` is defined if the load/store-exclusive instructions (LDREX/STREX) are supported. Its value is a set of bits indicating available widths of the access, as powers of 2. The following bits are used:

Bit	Value	Access width	Instruction
0	0x01	byte	LDREXB/STREXB
1	0x02	halfword	LDREXH/STREXH
2	0x04	word	LDREX/STREX
3	0x08	doubleword	LDREXD/STREXD

Other bits are reserved.

The following values of `__ARM_FEATURE_LDREX` may occur:

Macro value	Access widths	Example architecture
(undefined)	none	ARM v5, ARM v6-M

0x04	word	ARM v6
0x07	word, halfword, byte	ARM v7-M
0x0F	doubleword, word, halfword, byte	ARM v6K, ARM v7-A/R

Other values are reserved.

The LDREX/STREX instructions are introduced in recent versions of the ARM architecture and supersede the SWP instruction. Where both are available, ARM strongly recommends programmers to use LDREX/STREX rather than SWP. Note that platforms may choose to make SWP unavailable in user mode and emulate it through a trap to a platform routine, or fault it.

6.4.5 CLZ

`__ARM_FEATURE_CLZ` is defined to 1 if the CLZ (count leading zeroes) instruction is supported in hardware. Note that ACLE provides the `__clz()` family of intrinsics (see 9.2) even when `__ARM_FEATURE_CLZ` is not defined.

6.4.6 Q (saturation) flag

`__ARM_FEATURE_QBIT` is defined to 1 if the Q (saturation) global flag exists and the intrinsics defined in 9.1.1 are available. This flag is used with the DSP saturating-arithmetic instructions (such as QADD) and the width-specified saturating instructions (SSAT and USAT). Note that either of these classes of instructions may exist without the other: for example, v5E has only QADD while v7-M has only SSAT.

6.4.7 DSP instructions

`__ARM_FEATURE_DSP` is defined to 1 if the DSP (v5E) instructions are supported and the intrinsics defined in 9.4 are available. These instructions include QADD, SMULBB etc. This feature also implies support for the Q flag.

6.4.8 Saturation instructions

`__ARM_FEATURE_SAT` is defined to 1 if the SSAT and USAT instructions are supported and the intrinsics defined in 9.4.1 are available. This feature also implies support for the Q flag.

6.4.9 32-bit SIMD instructions

`__ARM_FEATURE_SIMD32` is defined to 1 if the 32-bit SIMD instructions are supported and the intrinsics defined in 9.5 are available. This also implies support for the GE global flags which indicate byte-by-byte comparison results.

6.4.10 Hardware integer divide

`__ARM_FEATURE_IDIV` is defined to 1 if the target has hardware support for 32-bit integer division in all available instruction sets. Signed and unsigned versions are both assumed to be available. The intention is to allow programmers to choose alternative algorithm implementations depending on the likely speed of integer division.

Some older R-profile targets have hardware divide available in the Thumb instruction set only. This can be tested for using the following test:

```
#if __ARM_FEATURE_IDIV || (__ARM_ARCH_PROFILE == 'R')
```


6.5 Floating-point and Advanced SIMD (NEON) hardware

6.5.1 Hardware floating point

`__ARM_FP` is set if hardware floating-point is available. The value is a set of bits indicating the floating-point precisions supported. The following bits are used:

Bit	Value	Precision
1	0x02	half (16-bit) – data type only
2	0x04	single (32-bit)
3	0x08	double (64-bit)

Bits 0 and 4..31 are reserved.

Currently, the following values of `__ARM_FP` may occur (assuming the processor configuration option for hardware floating-point support is selected where available):

Value	Precisions	Example processor
(undefined)	none	any processor without hardware floating-point support
0x04	single	Cortex-R5 when configured with SP only
0x06	single, half	Cortex-M4.fp
0x0C	double, single	ARM9, ARM11, Cortex-A8, Cortex-R4
0x0E	double, single, half	Cortex-A9, Cortex-A15, Cortex-R7

Other values are reserved.

Standard C implementations support single and double precision floating-point irrespective of whether floating-point hardware is available. However, an implementation might choose to offer a mode to diagnose or fault use of floating-point arithmetic at a precision not supported in hardware.

Support for 16-bit floating-point language extensions (see 6.5.2) is only required to be available if supported in hardware. Hardware support for 16-bit floating-point is limited to conversions. Values are promoted to 32-bit (single-precision) type for arithmetic.

6.5.2 Half-precision (16-bit) floating-point format

`__ARM_FP16_FORMAT_IEEE` is defined to 1 if the IEEE 754-2008 [IEEE-FP] 16-bit floating-point format is used.

`__ARM_FP16_FORMAT_ALTERNATIVE` is defined to 1 if the ARM alternative [ARMARM] 16-bit floating-point format is used. This format removes support for infinities and NaNs in order to provide an extra exponent bit.

At most one of these macros will be defined. See 4.1.2 for details of half-precision floating-point types.

6.5.3 Fused multiply-accumulate (FMA)

`__ARM_FEATURE_FMA` is defined to 1 if the hardware floating-point architecture supports fused floating-point multiply-accumulate, i.e. without intermediate rounding. Note that C implementations are encouraged [C99 7.12] to ensure that `<math.h>` defines `FP_FAST_FMAF` or `FP_FAST_FMA`, which can be tested by portable C code. A C implementation on ARM might define these macros by testing `__ARM_FEATURE_FMA` and `__ARM_FP`.

6.5.4 Advanced SIMD architecture extension (NEON)

__ARM_NEON is defined to a value indicating the Advanced SIMD (NEON) architecture supported. The only current value is 1. Section 12 of this document describes a comprehensive set of vector types and intrinsics for use with NEON.

In principle, the NEON architecture can exist in an integer-only version. To test for the presence of NEON floating-point vector instructions, test __ARM_NEON_FP. When NEON does occur in an integer-only version, the VFP scalar instruction set is also not present. See [ARMARM table A2-4] for architecturally permitted combinations.

6.5.5 NEON floating-point

__ARM_NEON_FP is defined as a bitmap to indicate floating-point support in the NEON architecture. The meaning of the values is the same as for __ARM_FP. This macro is undefined when the NEON extension is not present or does not support floating-point.

Current NEON implementations do not support double-precision floating-point even when it is present in VFP. 16-bit floating-point format is supported in NEON if and only if it is supported in VFP. Consequently, the definition of __ARM_NEON_FP is the same as __ARM_FP except that the bit to indicate double-precision is not set.

If __ARM_FEATURE_FMA and __ARM_NEON_FP are both defined, fused-multiply instructions are available in NEON also.

6.5.6 Wireless MMX

If Wireless MMX operations are available on the target, __ARM_WMMX is defined to a value that indicates the level of support, corresponding to the Tag_WMMX_arch build attribute.

This specification does not further define source-language features to support Wireless MMX.

6.6 Floating-point model

These macros test the floating-point model implemented by the compiler and libraries. The model determines the guarantees on arithmetic and exceptions.

__ARM_FP_FAST is defined to 1 if floating-point optimizations may occur such that the computed results are different from those prescribed by the order of operations according to the C standard. Examples of such optimizations would be reassociation of expressions to reduce depth, and replacement of a division by constant with multiplication by its reciprocal.

__ARM_FP_FENV_ROUNDING is defined to 1 if the implementation allows the rounding to be configured at runtime using the standard C fesetround() function and will apply this rounding to future floating-point operations. The rounding mode applies to both scalar floating-point and NEON.

The floating-point implementation might or might not support denormal values. If denormal values are not supported then they are flushed to zero. NEON does not support denormals.

Implementations may also define the following macros in appropriate floating-point modes:

__STDC_IEC_559__ is defined if the implementation conforms to IEC 559. This implies support for floating-point exception status flags, including the inexact exception. This macro is specified by [C99 6.10.8].

__SUPPORT_SNAN__ is defined if the implementation supports signalling NaNs. This macro is specified by the C standards proposal WG14 N965 “Optional support for Signaling NaNs”. (Note: this was not adopted into C11.)

6.7 Procedure call standard

`__ARM_PCS` is defined to 1 if the default procedure calling standard for the translation unit conforms to the "base PCS" defined in [AAPCS].

`__ARM_PCS_VFP` is defined to 1 if the default is to pass floating-point parameters in hardware floating-point registers using the "VFP variant PCS" defined in [AAPCS].

Note that this should reflect the implementation default for the translation unit. Implementations which allow the PCS to be set for a function, class or namespace are not expected to redefine the macro within that scope.

6.8 Mapping of object build attributes to predefines

This section is provided for guidance. Details of build attributes can be found in [BA].

Tag no.	Tag	Predefined macro
6	`Tag_CPU_arch`	`__ARM_ARCH`, `__ARM_FEATURE_DSP`
7	`Tag_CPU_arch_profile`	`__ARM_ARCH_PROFILE`
8	`Tag_ARM_ISA_use`	`__ARM_ARCH_ISA_ARM`
9	`Tag_THUMB_ISA_use`	`__ARM_ARCH_ISA_THUMB`
11	`Tag_WMMX_arch`	`__ARM_WMMX`
18	`Tag_ABI_PCS_wchar_t`	`__ARM_SIZEOF_WCHAR_T`
20	`Tag_ABI_FP_denormal`	
21	`Tag_ABI_FP_exceptions`	
22	`Tag_ABI_FP_user_exceptions`	
23	`Tag_ABI_FP_number_model`	
26	`Tag_ABI_enum_size`	`__ARM_SIZEOF_MINIMAL_ENUM`
34	`Tag_CPU_unaligned_access`	`__ARM_FEATURE_UNALIGNED`
36	`Tag_FP_HP_extension`	`__ARM_FP16_FORMAT_IEEE`, `__ARM_FP16_FORMAT_ALTERNATIVE`
38	`Tag_ABI_FP_16bit_format`	`__ARM_FP16_FORMAT_IEEE`, `__ARM_FP16_FORMAT_ALTERNATIVE`

6.9 Summary of predefined macros

Macro name	Meaning	Example	See section
`__ARM_32BIT_STATE`	code is for 32-bit state (new in 1.1)	1	6.4.1
`__ARM_ACLE`	indicates ACLE implemented	101	6.2
`__ARM_ALIGN_MAX_PWR`	log of maximum alignment of static object	20	7.5.2
`__ARM_ALIGN_MAX_STACK_PWR`	log of maximum alignment of stack object	3	7.5.3
`__ARM_ARCH`	ARM architecture level	7	6.4.1
`__ARM_ARCH_ISA_ARM`	ARM instruction set present	1	6.4.1
`__ARM_ARCH_ISA_THUMB`	Thumb instruction set present	2	6.4.1
`__ARM_ARCH_PROFILE`	architecture profile	'A'	6.4.2
`__ARM_BIG_ENDIAN`	memory is big-endian	1	6.3
`__ARM_FEATURE_CLZ`	CLZ instruction	1	6.4.5, 9.2
`__ARM_FEATURE_DSP`	DSP instructions (ARM v5E)	1	6.4.6, 9.4
`__ARM_FEATURE_FMA`	floating-point fused multiply-accumulate	1	6.5.3, 9.6
`__ARM_FEATURE_IDIV`	integer divide in hardware (new in 1.1)	1	6.4.10
`__ARM_FEATURE_LDREX`	load/store exclusive instructions	0x0F	6.4.4, 8
`__ARM_FEATURE_QBIT`	Q (saturation) flag	1	6.4.6, 9.1.1
`__ARM_FEATURE_SAT`	width-specified saturation instructions	1	6.4.8, 9.4.1
`__ARM_FEATURE_SIMD32`	32-bit SIMD instructions (ARM v6)	1	6.4.8, 9.5
`__ARM_FEATURE_UNALIGNED`	hardware support for unaligned access	1	6.4.3
`__ARM_FP`	hardware floating-point	0x0C	6.5.1
`__ARM_FP_FAST`	accuracy-losing optimizations	1	6.6
`__ARM_FP_FENV_ROUNDING`	rounding is configurable at runtiume	1	6.6
`__ARM_FP16_FORMAT_ALTERNATIVE`	16-bit floating-point, alternative format	1	6.5.2
`__ARM_FP16_FORMAT_IEEE`	16-bit floating-point, IEEE format	1	6.5.2
`__ARM_NEON`	Advanced SIMD (NEON) extension	1	6.5.4
`__ARM_NEON_FP`	Advanced SIMD (NEON) floating-point	0x04	6.5.5
`__ARM_PCS`	ARM procedure call standard	1	6.7
`__ARM_PCS_VFP`	ARM PCS hardware FP variant in use	1	6.7
`__ARM_SIZEOF_MINIMAL_ENUM`	size of minimal enumeration type: 1 or 4	1	4.1.1
`__ARM_SIZEOF_WCHAR_T`	size of `wchar_t`: 2 or 4	2	4.1.1
`__ARM_WMMX`	Wireless MMX extension	1	6.5.6

7 ATTRIBUTES AND PRAGMAS

7.1 Attribute syntax

The general rules for attribute syntax are described in the GCC documentation at http://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html. Briefly, for this declaration:

```
A int B x C, D y E;
```

attribute A applies to both x and y; B and C apply to x only, and D and E apply to y only. Programmers are recommended to keep declarations simple if attributes are used.

Unless otherwise stated, all attribute arguments must be compile-time constants.

7.2 Hardware/software floating-point calling convention

On targets with hardware FP the AAPCS provides for procedure calls to use either integer or floating-point argument and result registers. ACLE allows this to be selectable per function.

```
__attribute__((pcs("aapcs")))
```

applied to a function, selects software (integer) FP calling convention.

```
__attribute__((pcs("aapcs-vfp")))
```

applied to a function, selects hardware FP calling convention.

The `pcs` attribute applies to functions and function types. Implementations are allowed to treat the procedure call specification as part of the type, i.e. as a “language linkage” in the sense of [C++ 7.5#1].

7.3 Target selection

The following target selection attributes are supported:

```
__attribute__((target("arm")))
```

when applied to a function, forces ARM state code generation.

```
__attribute__((target("thumb")))
```

when applied to a function, forces Thumb state code generation.

The implementation must generate code in the required state unless it is impossible to do so. For example, on an ARM v5 or v6 target with VFP (and without the Thumb2 instruction set), if a function is forced to Thumb state, any floating-point operations or intrinsics that are only available in ARM state must be generated as calls to library functions or compiler-generated functions.

7.4 Weak linkage

`__attribute__((weak))` can be attached to declarations and definitions to indicate that they have weak static linkage (STB_WEAK in ELF objects). As definitions, they can be overridden by other definitions of the same symbol. As references, they do not need to be satisfied and will be resolved to zero if a definition is not present.

7.4.1 Patchable constants

In addition, this specification requires that weakly defined initialized constants are not used for constant propagation, allowing the value to be safely changed by patching after the object is produced.

7.5 Alignment

The new standards for C [C11 6.7.5] and C++ [C++11 7.6.2] add syntax for aligning objects and types. ACLE provides an alternative syntax described in this section.

7.5.1 Alignment attribute

`__attribute__((aligned(N)))` can be associated with data, functions, types and fields. N must be an integral constant expression and must be a power of 2, e.g. 1, 2, 4, 8. The maximum alignment depends on the storage class of the object being aligned. The size of a data type is always a multiple of its alignment. This is a consequence of the rule in C that the spacing between array elements is equal to the element size.

The `aligned` attribute does not act as a type qualifier. For example, given

```
char x __attribute__((aligned(8)));
int y __attribute__((aligned(1)));
```

the type of `&x` is "`char *`" and the type of `&y` is "`int *`". The following declarations are equivalent:

```
struct S x __attribute__((aligned(16)));    /* ACLE */

struct S _Alignas(16) x;      /* C11 */

#include <stdalign.h>         /* C11 (alternative) */
struct S alignas(16) x;

struct S alignas(16) x;       /* C++11 */
```

7.5.2 Alignment of static objects

The macro `__ARM_ALIGN_MAX_PWR` indicates (as the exponent of a power of 2) the maximum available alignment of static data – for example 4 for 16-byte alignment. So the following is always valid:

```
int x __attribute__((aligned(1 << __ARM_ALIGN_MAX_PWR)));
```

or, using the C11/C++11 syntax:

```
alignas(1 << __ARM_ALIGN_MAX_PWR) int x;
```

Since an alignment request on an object does not change its type or size, `x` in this example would have type `int` and size 4.

There is in principle no limit on the alignment of static objects, within the constraints of available memory. In the ARM ABI an object with a requested alignment would go into an ELF section with at least as strict an alignment requirement. However, an implementation supporting position-independent dynamic objects or overlays may need to place restrictions on their alignment demands.

7.5.3 Alignment of stack objects

It must be possible to align any local object up to the stack alignment as specified in the AAPCS (i.e. 8 bytes), this being also the maximal alignment of any native type.

An implementation may, but is not required to, permit the allocation of local objects with greater alignment, e.g. 16 or 32 bytes. (This would involve some runtime adjustment such that the object address was not a fixed offset from the stack pointer on entry.)

If a program requests alignment greater than the implementation supports, it is recommended that the compiler warn but not fault this. Programmers should expect over-alignment of local objects to be treated as a hint.

The macro `__ARM_ALIGN_MAX_STACK_PWR` indicates (as the exponent of a power of 2) the maximum available stack alignment. For example, a value of 3 indicates 8-byte alignment.

7.5.4 Procedure calls

For procedure calls, where a parameter has aligned type, data should be passed as if it was a basic type of the given type and alignment. For example, given the aligned type

```
struct S { int a[2]; } __attribute__((aligned(8)));
```

the second argument of

```
f(int, struct S);
```

should be passed as if it were

```
f(int, long long);
```

with the second parameter in R2/R3 rather than R1/R2.

7.5.5 Alignment of C heap storage

The standard C allocation functions [C99 7.20.3], such as `malloc()`, return storage aligned to the normal maximal alignment, i.e. the largest alignment of any (standard) type.

Implementations may, but are not required to, provide a function to return heap storage of greater alignment. Suitable functions are

```
int posix_memalign(void **memptr, size_t alignment, size_t size);
```

as defined in [POSIX], or

```
void *aligned_alloc(size_t alignment, size_t size);
```

as defined in [C11 7.22.3.1].

7.5.6 Alignment of C++ heap allocation

In C++, an allocation (with '`new`') knows the object's type. If the type is aligned, the allocation should also be aligned. There are two cases to consider depending on whether the user has provided an allocation function.

If the user has provided an allocation function for an object or array of over-aligned type, it is that function's responsibility to return suitably aligned storage. The size requested by the runtime library will be a multiple of the alignment (trivially so, for the non-array case).

(The ARM C++ ABI does not explicitly deal with the runtime behavior when dealing with arrays of alignment greater than 8. In this situation, any 'cookie' will be 8 bytes as usual, immediately preceding the array; this means that the cookie is not necessarily at the address seen by the allocation and deallocation functions. Implementations will need to make some adjustments before and after calls to the ABI-defined C++ runtime, or may provide additional non-standard runtime helper functions.) Example:

```
struct float4 {
  void *operator new[](size_t s) {
    void *p;
    posix_memalign(&p, 16, s);
    return p;
  }
```



```
  float data[4];
} __attribute__((aligned(16)));
```

If the user has not provided their own allocation function, the behavior is implementation-defined.

7.6 Other attributes

The following attributes should be supported and their definitions follow [GCC]. These attributes are not specific to ARM or the ARM ABI.

`alias`, `common`, `nocommon`, `noinline`, `packed`, `section`, `visibility`, `weak`

Some specific requirements on the `weak` attribute are detailed in 7.4.

8 SYNCHRONIZATION, BARRIER AND HINT INTRINSICS

8.1 Introduction

This section provides intrinsics for managing data that may be accessed concurrently between processors, or between a processor and a device. Some intrinsics atomically update data, while others place barriers around accesses to data to ensure that accesses are visible in the correct order.

Memory prefetch intrinsics are also described in this section.

8.2 Atomic update primitives

8.2.1 C/C++ standard atomic primitives

The new C and C++ standards [C11 7.17, C++11 clause 29] provide a comprehensive library of atomic operations and barriers, including operations to read and write data with particular ordering requirements. Programmers are recommended to use this where available.

8.2.2 IA-64/GCC atomic update primitives

The `__sync` family of intrinsics (introduced in [IA-64 section 7.4], and as documented in the GCC documentation) may be provided, especially if the C/C++ atomics are not available, and are recommended as being portable and widely understood. These may be expanded inline, or call library functions. Note that, unusually, these intrinsics are polymorphic – they will specialize to instructions suitable for the size of their arguments.

8.3 Memory barriers

Memory barriers ensure specific ordering properties between memory accesses. For more details on memory barriers, see ARM ARM [v7 section A3.8.3]. The intrinsics in this section are available for all targets. They may be no-ops (i.e. generate no code, but possibly act as a code motion barrier in compilers) on targets where the relevant instructions do not exist, but only if the property they guarantee would have held anyway. On targets where the relevant instructions exist but are implemented as no-ops, these intrinsics generate the instructions.

The memory barrier intrinsics take a numeric argument indicating the scope and access type of the barrier, as shown in the following table. (The assembler mnemonics for these numbers, as shown in the table, are not available in the intrinsics.) The argument should be an integral constant expression within the required range – see section 4.3.1.

Argument	Mnemonic	Domain	Ordered Accesses (before-after)
15	SY	Full system	Any-Any
14	ST	Full system	Store-Store
13	LD	Full system	Load-Load, Load-Store
11	ISH	Inner shareable	Any-Any
10	ISHST	Inner shareable	Store-Store
9	ISHLD	Inner shareable	Load-Load, Load-Store
7	NSH or UN	Non-shareable	Any-Any

6	NSHST	Non-shareable	Store-Store
5	NSHLD	Non-shareable	Load-Load, Load-Store
3	OSH	Outer shareable	Any-Any
2	OSHST	Outer shareable	Store-Store
1	OSHLD	Outer shareable	Load-Load, Load-Store

The following memory barrier intrinsics are available:

```
void __dmb(/*constant*/ unsigned int);
```

Generates a DMB (data memory barrier) instruction or equivalent CP15 instruction. DMB ensures the observed ordering of memory accesses. Memory accesses of the specified type issued before the DMB are guaranteed to be observed (in the specified scope) before memory accesses issued after the DMB. For example, DMB should be used between storing data, and updating a flag variable that makes that data available to another core.

The `__dmb()` intrinsic also acts as a compiler memory barrier of the appropriate type.

```
void __dsb(/*constant*/ unsigned int);
```

Generates a DSB (data synchronization barrier) instruction or equivalent CP15 instruction. DSB ensures the completion of memory accesses. A DSB behaves as the equivalent DMB and has additional properties. After a DSB instruction completes, all memory accesses of the specified type issued before the DSB are guaranteed to have completed.

The `__dsb()` intrinsic also acts as a compiler memory barrier of the appropriate type.

```
void __isb(/*constant*/ unsigned int);
```

Generates an ISB (instruction synchronization barrier) instruction or equivalent CP15 instruction. This instruction flushes the processor pipeline fetch buffers, so that following instructions are fetched from cache or memory. An ISB is needed after some system maintenance operations.

An ISB is also needed before transferring control to code that has been loaded or modified in memory, for example by an overlay mechanism or just-in-time code generator. (Note that if instruction and data caches are separate, privileged cache maintenance operations would be needed in order to unify the caches.)

The only supported argument for the `__isb()` intrinsic is 15, corresponding to the SY (full system) scope of the ISB instruction.

8.3.1 Examples

In this example, process P1 makes some data available to process P2 and sets a flag to indicate this.

P1:

```
value = x;
/* issue full-system memory barrier for previous store:
   setting of flag is guaranteed not to be observed before
   write to value */
__dmb(14);
flag = true;
```

P2:


```
/* busy-wait until the data is available */
while (!flag) {}
/* issue full-system memory barrier: read of value is guaranteed
   not to be observed by memory system before read of flag */
__dmb(15);
use value;
```

In this example, process P1 makes data available to P2 by putting it on a queue.

P1:

```
work = new WorkItem;
work->payload = x;
/* issue full-system memory barrier for previous store:
   consumer cannot observe work item on queue before write to
   work item's payload */
__dmb(14);
queue_head = work;
```

P2:

```
/* busy-wait until work item appears */
while (!(work = queue_head)) {}
/* no barrier needed: load of payload is data-dependent */
use work->payload;
```

8.4 Hints

The intrinsics in this section are available for all targets. They may be no-ops (i.e. generate no code, but possibly act as a code motion barrier in compilers) on targets where the relevant instructions do not exist. On targets where the relevant instructions exist but are implemented as no-ops, these intrinsics generate the instructions.

```
void __wfi(void);
```

Generates a WFI (wait for interrupt) hint instruction, or nothing. The WFI instruction allows (but does not require) the processor to enter a low-power state until one of a number of asynchronous events occurs.

```
void __wfe(void);
```

Generates a WFE (wait for event) hint instruction, or nothing. The WFE instruction allows (but does not require) the processor to enter a low-power state until some event occurs such as a SEV being issued by another processor.

```
void __sev(void);
```

Generates a SEV (send a global event) hint instruction. This causes an event to be signaled to all processors in a multiprocessor system. It is a NOP on a uniprocessor system.

```
void __sevl(void);
```

Generates a “send a local event” hint instruction. This causes an event to be signaled to only the processor executing this instruction. In a multiprocessor system, it is not required to affect the other processors. New in ACLE 1.1.

```
void __yield(void);
```

Generates a YIELD hint instruction. This enables multithreading software to indicate to the hardware that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance.

```
void __dbg(/*constant*/ unsigned int);
```

Generates a DBG instruction. This provides a hint to debugging and related systems. The argument must be a constant integer from 0 to 15 inclusive. See implementation documentation for the effect (if any) of this instruction and the meaning of the argument.

8.5 Swap

`__swp` is available for all targets. This intrinsic expands to a sequence equivalent to the deprecated (and possibly unavailable) SWP instruction.

```
uint32_t __swp(uint32_t, volatile void *);
```

unconditionally stores a new value at the given address, and returns the old value.

As with the IA-64/GCC primitives described in 8.2.2, the `__swp` intrinsic is polymorphic. The second argument must provide the address of a byte-sized object or an aligned word-sized object and it must be possible to determine the size of this object from the argument expression.

This intrinsic is implemented by LDREX/STREX (or LDREXB/STREXB) where available, as if by

```
uint32_t __swp(uint32_t x, volatile uint32_t *p) {
  uint32_t v;
  /* use LDREX/STREX intrinsics not specified by ACLE */
  do v = __ldrex(p); while (__strex(x, p));
  return v;
}
```

or alternatively,

```
uint32_t __swp(uint32_t x, uint32_t *p) {
  uint32_t v;
  /* use IA-64/GCC atomic builtins */
  do v = *p; while (!__sync_bool_compare_and_swap(p, v, x));
  return v;
}
```

It is recommended that compilers should produce a downgradeable/upgradeable warning on encountering the `__swp` intrinsic.

Only if load-store exclusive instructions are not available will the intrinsic use the SWP/SWPB instructions.

It is strongly recommended to use standard and flexible atomic primitives such as those available in the C++ `<atomic>` header. `__swp` is provided solely to allow straightforward (and possibly automated) replacement of explicit use of SWP in inline assembler. SWP is obsolete in the ARM architecture, and in recent versions of the architecture, may be configured to be unavailable in user-mode. (Aside: unconditional atomic swap is also less powerful as a synchronization primitive than load-exclusive/store-conditional.)

8.6 Memory prefetch intrinsics

Intrinsics are provided to prefetch data or instructions. The size of the data or function is ignored. Note that the intrinsics may be implemented as no-ops (i.e. not generate a prefetch instruction, if none is available). Also, even where the architecture does provide a prefetch instruction, a particular implementation may implement the instruction as a no-op (i.e. the instruction has no effect).

8.6.1 Data prefetch

```
void __pld(void const volatile *addr);
```

Generates a data prefetch instruction, if available. The argument should be any expression that may designate a data address. The data is prefetched to the innermost level of cache, for reading.

```
void __pldx(/*constant*/ unsigned int /*access_kind*/,
            /*constant*/ unsigned int /*cache_level*/,
            /*constant*/ unsigned int /*retention_policy*/,
            void const volatile *addr);
```

Generates a data prefetch instruction. This intrinsic allows the specification of the expected access kind (read or write), the cache level to load the data, the data retention policy (temporal or streaming), The relevant arguments can only be one of the following values. New in ACLE 1.1.

Access Kind	**Value**	**Summary**
PLD	0	Fetch the addressed location for reading
PST	1	Load the addressed location for writing

Cache Level	Value	Summary
L1	0	Load the addressed location to L1 cache
L2	1	Load the addressed location to L2 cache
L3	2	Load the addressed location to L3 cache

Retention Policy	**Value**	**Summary**
KEEP	0	Temporal fetch of the addressed location (i.e. allocate in cache normally)
STRM	1	Streaming fetch of the addressed location (i.e. memory used only once)

8.6.2 Instruction prefetch

```
void __pli(T addr);
```

Generates a code prefetch instruction, if available. If a specific code prefetch instruction is not available, this intrinsic may generate a data-prefetch instruction to fetch the addressed code to the innermost level of unified cache. It will not fetch code to data-cache in a split cache level.

8.7 NOP

```
void __nop(void);
```

generates an unspecified no-op instruction. Note that not all architectures provide a distinguished NOP instruction. On those that do, it is unspecified whether this intrinsic generates it or another instruction. It is not guaranteed that inserting this instruction will increase execution time.

9 DATA-PROCESSING INTRINSICS

The intrinsics in this section are provided for algorithm optimization.

The `<arm_acle.h>` header should be included before using these intrinsics.

Implementations are not required to introduce precisely the instructions whose names match the intrinsics. However, implementations should aim to ensure that a computation expressed compactly with intrinsics will generate a similarly compact sequence of machine code. In general, C's "as-if rule" [C99 5.1.2.3] applies, meaning that the compiled code must behave *as if* the instruction had been generated.

In general, these intrinsics are aimed at DSP algorithm optimization on M-profile and R-profile. Use on A-profile is discouraged. However, the miscellaneous intrinsics described in 9.2 are suitable for all profiles.

9.1 Programmer's model of global state

9.1.1 The Q (saturation) flag

The Q flag is a cumulative ('sticky') saturation bit in the APSR (Application Program Status Register) indicating that an operation saturated, or in some cases, overflowed. It is set on saturation by most intrinsics in the DSP and SIMD intrinsic sets, though some SIMD intrinsics feature saturating operations which do not set the Q flag.

[AAPCS 5.1.1] states:

> The N, Z, C, V and Q flags (bits 27-31) and the GE[3:0] bits (bits 16-19) are undefined on entry to or return from a public interface.

Note that this does not state that these bits (in particular the Q flag) are undefined across any C/C++ function call boundary – only across a "public interface". The Q and GE bits could be manipulated in well-defined ways by local functions, for example when constructing functions to be used in DSP algorithms.

Implementations must avoid introducing instructions (such as SSAT/USAT, or SMLABB) which affect the Q flag, if the programmer is testing whether the Q flag was set by explicit use of intrinsics and if the implementation's introduction of an instruction may affect the value seen. The implementation might choose to model the definition and use (liveness) of the Q flag in the way that it models the liveness of any visible variable, or it might suppress introduction of Q-affecting instructions in any routine in which the Q flag is tested.

ACLE does not define how or whether the Q flag is preserved across function call boundaries. (This is seen as an area for future specification.)

In general, the Q flag should appear to C/C++ code in a similar way to the standard floating-point cumulative exception flags, as global (or thread-local) state that can be tested, set or reset through an API.

The following intrinsics are available when `__ARM_FEATURE_QBIT` is defined:

```
int __saturation_occurred(void);
```

Returns 1 if the Q flag is set, 0 if not.

```
void __set_saturation_occurred(int);
```

Sets or resets the Q flag according to the LSB of the value. `__set_saturation_occurred(0)` might be used before performing a sequence of operations after which the Q flag is tested. (In general, the Q flag cannot be assumed to be unset at the start of a function.)

```
void __ignore_saturation(void);
```


This intrinsic is a hint and may be ignored. It indicates to the compiler that the value of the Q flag is not live (needed) at or subsequent to the program point at which the intrinsic occurs. It may allow the compiler to remove preceding instructions, or to change the instruction sequence in such a way as to result in a different value of the Q flag. (A specific example is that it may recognize clipping idioms in C code and implement them with an instruction such as SSAT that may set the Q flag.)

9.1.2 The GE flags

The GE (Greater than or Equal to) flags are four bits in the APSR. They are used with the 32-bit SIMD intrinsics described in section 9.5.

There are four GE flags, one for each 8-bit lane of a 32-bit SIMD operation. Certain non-saturating 32-bit SIMD intrinsics set the GE bits to indicate overflow of addition or subtraction. For 4x8-bit operations the GE bits are set one for each byte. For 2x16-bit operations the GE bits are paired together, one for the high halfword and the other pair for the low halfword. The only supported way to read or use the GE bits (in this specification) is by using the `__sel` intrinsic.

9.1.3 Floating-point environment

An implementation should implement the features of `<fenv.h>` for accessing the floating-point runtime environment. Programmers should use this rather than accessing the VFP FPSCR directly. For example, on a target supporting VFP the cumulative exception flags (IXC, OFC etc.) can be read from the FPSCR by using the `fetestexcept()` function, and the rounding mode (RMode) bits can be read using the `fegetround()` function.

ACLE does not support changing the DN, FZ or AHP bits at runtime.

VFP “short vector” mode (enabled by setting the Stride and Len bits) is deprecated, and is unavailable on later VFP implementations. ACLE provides no support for this mode.

9.2 Miscellaneous data-processing intrinsics

The following intrinsics perform general data-processing operations. They have no effect on global state.

[Note: documentation of the `__nop` intrinsic has moved to 8.7.]

The 64-bit versions of these intrinsics (‘`ll`’ suffix) are new in ACLE 1.1. For completeness and to aid portability between LP64 and LLP64 models, ACLE 1.1 also defines intrinsics with ‘`l`’ suffix.

```
uint32_t __ror(uint32_t x, uint32_t y);
unsigned long __rorl(unsigned long x, uint32_t y);
uint64_t __rorll(uint64_t x, uint32_t y);
```

rotates the argument `x` right by `y` bits. `y` can take any value. These intrinsics are available on all targets.

```
unsigned int __clz(uint32_t x);
unsigned int __clzl(unsigned long x);
unsigned int __clzll(uint64_t x);
```

returns the number of leading zero bits in x. When x is zero it returns the argument width, i.e. 32 or 64. These intrinsics are available on all targets. On targets without the `CLZ` instruction it should be implemented as an instruction sequence or a call to such a sequence. A suitable sequence can be found in [Warren] (fig. 5-7). Hardware support for these intrinsics is indicated by `__ARM_FEATURE_CLZ`.

```
unsigned int __cls(uint32_t x);
unsigned int __clsl(unsigned long x);
unsigned int __clsll(uint64_t x);
```

returns the number of leading sign bits in x. When x is zero it returns the argument width, i.e. 32 or 64. These intrinsics are available on all targets. On targets without the `CLZ` instruction it should be implemented as an instruction sequence or a call to such a sequence. Fast hardware implementation (using a CLS instruction or a short code sequence involving the CLZ instruction) is indicated by `__ARM_FEATURE_CLZ`. New in ACLE 1.1.

```
uint32_t __rev(uint32_t);
unsigned long __revl(unsigned long);
uint64_t __revll(uint64_t);
```

reverses the byte order within a word or doubleword. These intrinsics are available on all targets and should be expanded to an efficient straight-line code sequence on targets without byte reversal instructions.

```
uint32_t __rev16(uint32_t);
unsigned long __rev16l(unsigned long);
uint64_t __rev16ll(uint64_t);
```

reverses the byte order within each halfword of a word. For example, 0x12345678 becomes 0x34127856. These intrinsics are available on all targets and should be expanded to an efficient straight-line code sequence on targets without byte reversal instructions.

```
int16_t __revsh(int16_t);
```

reverses the byte order in a 16-bit value and returns the (sign-extended) result. For example, 0x00000080 becomes 0xFFFF8000. This intrinsic is available on all targets and should be expanded to an efficient straight-line code sequence on targets without byte reversal instructions.

```
uint32_t __rbit(uint32_t x);
unsigned long __rbitl(unsigned long x);
uint64_t __rbitll(uint64_t x);
```

reverses the bits in x. These intrinsics are only available on targets with the `RBIT` instruction.

9.2.1 Examples

```
#ifdef __ARM_BIG_ENDIAN
#define htonl(x) (uint32_t)(x)
#define htons(x) (uint16_t)(x)
#else /* little-endian */
#define htonl(x) __rev(x)
#define htons(x) (uint16_t)__revsh(x)
#endif /* endianness */
#define ntohl(x) htonl(x)
#define ntohs(x) htons(x)

/* Count leading sign bits */
inline unsigned int count_sign(int32_t x) { return __clz(x ^ (x << 1)); }

/* Count trailing zeroes */
inline unsigned int count_trail(uint32_t x) {
#if (__ARM_ARCH >= 6 && __ARM_ARCH_ISA_THUMB >= 2) || __ARM_ARCH >= 7
  /* RBIT is available */
  return __clz(__rbit(x));
#else
  unsigned int n = __clz(x & -x);   /* get the position of the last bit */
  return n == 32 ? n : (31-n);
#endif
}
```


9.3 16-bit multiplications

The intrinsics in this section provide direct access to the 16x16 and 16x32 bit multiplies introduced in ARM v5E. Compilers are also encouraged to exploit these instructions from C code. These intrinsics are available when `__ARM_FEATURE_DSP` is defined, and are not available on non-5E targets. These multiplies cannot overflow.

```
int32_t __smulbb(int32_t, int32_t);
```

Multiplies two 16-bit signed integers, i.e. the low halfwords of the operands.

```
int32_t __smulbt(int32_t, int32_t);
```

Multiplies the low halfword of the first operand and the high halfword of the second operand.

```
int32_t __smultb(int32_t, int32_t);
```

Multiplies the high halfword of the first operand and the low halfword of the second operand.

```
int32_t __smultt(int32_t, int32_t);
```

Multiplies the high halfwords of the operands.

```
int32_t __smulwb(int32_t, int32_t);
```

Multiplies the 32-bit signed first operand with the low halfword (as a 16-bit signed integer) of the second operand. Return the top 32 bits of the 48-bit product.

```
int32_t __smulwt(int32_t, int32_t);
```

Multiplies the 32-bit signed first operand with the high halfword (as a 16-bit signed integer) of the second operand. Return the top 32 bits of the 48-bit product.

9.4 Saturating intrinsics

9.4.1 Width-specified saturation intrinsics

These intrinsics are available when `__ARM_FEATURE_SAT` is defined. They saturate a 32-bit value at a given bit position. The saturation width must be an integral constant expression – see section 4.3.1.

```
int32_t __ssat(int32_t, /*constant*/ unsigned int);
```

Saturates a signed integer to the given bit width in the range 1 to 32. For example, the result of saturation to 8-bit width will be in the range -128 to 127. The Q flag is set if the operation saturates.

```
uint32_t __usat(int32_t, /*constant*/ unsigned int);
```

Saturates a signed integer to an unsigned (non-negative) integer of a bit width in the range 0 to 31. For example, the result of saturation to 8-bit width is in the range 0 to 255, with all negative inputs going to zero. The Q flag is set if the operation saturates.

9.4.2 Saturating addition and subtraction intrinsics

These intrinsics are available when `__ARM_FEATURE_DSP` is defined.

The saturating intrinsics operate on 32-bit signed integer data. There are no special 'saturated' or 'fixed point' types.

```
int32_t __qadd(int32_t, int32_t);
```

Adds two 32-bit signed integers, with saturation. Sets the Q flag if the addition saturates.

```
int32_t __qsub(int32_t, int32_t);
```

Subtracts two 32-bit signed integers, with saturation. Sets the Q flag if the subtraction saturates.

```
int32_t __qdbl(int32_t);
```

Doubles a signed 32-bit number, with saturation. `__qdbl(x)` is equal to `__qadd(x,x)` except that the argument `x` is evaluated only once. Sets the Q flag if the addition saturates.

9.4.3 Accumulating multiplications

These intrinsics are available when `__ARM_FEATURE_DSP` is defined.

```
int32_t __smlabb(int32_t, int32_t, int32_t);
```

Multiplies two 16-bit signed integers, the low halfwords of the first two operands, and adds to the third operand. Sets the Q flag if the addition overflows. (Note that the addition is the usual 32-bit modulo addition which wraps on overflow, not a saturating addition. The multiplication cannot overflow.)

```
int32_t __smlabt(int32_t, int32_t, int32_t);
```

Multiplies the low halfword of the first operand and the high halfword of the second operand, and adds to the third operand, as for `__smlabb`.

```
int32_t __smlatb(int32_t, int32_t, int32_t);
```

Multiplies the high halfword of the first operand and the low halfword of the second operand, and adds to the third operand, as for `__smlabb`.

```
int32_t __smlatt(int32_t, int32_t, int32_t);
```

Multiplies the high halfwords of the first two operands and adds to the third operand, as for `__smlabb`..

```
int32_t __smlawb(int32_t, int32_t, int32_t);
```

Multiplies the 32-bit signed first operand with the low halfword (as a 16-bit signed integer) of the second operand. Adds the top 32 bits of the 48-bit product to the third operand. Sets the Q flag if the addition overflows. (See note for `__smlabb`.)

```
int32_t __smlawt(int32_t, int32_t, int32_t);
```

Multiplies the 32-bit signed first operand with the high halfword (as a 16-bit signed integer) of the second operand and adds the top 32 bits of the 48-bit result to the third operand as for `__smlawb`.

9.4.4 Examples

The ACLE DSP intrinsics can be used to define ETSI/ITU-T basic operations [G.191]:

```
#include <arm_acle.h>
inline int32_t L_add(int32_t x, int32_t y) { return __qadd(x, y); }
inline int32_t L_negate(int32_t x) { return __qsub(0, x); }
inline int32_t L_mult(int16_t x, int16_t y) { return __qdbl(x*y); }
inline int16_t add(int16_t x, int16_t y) { return (int16_t)(__qadd(x<<16, y<<16) >> 16); }
inline int16_t norm_l(int32_t x) { return __clz(x ^ (x<<1)) & 31; }
...
```

This example assumes the implementation preserves the Q flag on return from an inline function.

9.5 32-bit SIMD intrinsics

9.5.1 Availability

ARM v6 introduced instructions to perform 32-bit SIMD operations (i.e. two 16-bit operations or four 8-bit operations) on the ARM general-purpose registers. These instructions are not related to the much more versatile Advanced SIMD (NEON) extension, whose support is described in section 12.

The 32-bit SIMD intrinsics are available on targets featuring ARM v6 and upwards, including the A and R profiles. In the M profile they are available in the v7E-M architecture. Availability of the 32-bit SIMD intrinsics implies availability of the saturating intrinsics.

Availability of the SIMD intrinsics is indicated by the `__ARM_FEATURE_SIMD32` predefine.

To access the intrinsics, the `<arm_acle.h>` header should be included.

9.5.2 Data types for 32-bit SIMD intrinsics

The header `<arm_acle.h>` should be included before using these intrinsics.

The SIMD intrinsics generally operate on and return 32-bit words consisting of two 16-bit or four 8-bit values. These are represented as `int16x2_t` and `int8x4_t` below for illustration. Some intrinsics also feature scalar accumulator operands and/or results.

When defining the intrinsics, implementations can define SIMD operands using a 32-bit integral type (such as '`unsigned int`').

The header `<arm_acle.h>` defines typedefs `int16x2_t`, `uint16x2_t`, `int8x4_t` and `uint8x4_t`. These should be defined as 32-bit integral types of the appropriate sign. There are no intrinsics provided to pack or unpack values of these types. This can be done with shifting and masking operations.

9.5.3 Use of the Q flag by 32-bit SIMD intrinsics

Some 32-bit SIMD instructions may set the Q flag described in section 9.1.1. The behavior of the intrinsics matches that of the instructions.

Generally, instructions that perform lane-by-lane saturating operations do not set the Q flag. For example, `__qadd16` does not set the Q flag, even if saturation occurs in one or more lanes.

The explicit saturation operations `__ssat` and `__usat` set the Q flag if saturation occurs. Similarly, `__ssat16` and `__usat16` set the Q flag if saturation occurs in either lane.

Some instructions, such as `__smlad`, set the Q flag if overflow occurs on an accumulation, even though the accumulation is not a saturating operation (i.e. does not clip its result to the limits of the type).

In the following descriptions of intrinsics, if the description does not mention whether the intrinsic affects the Q flag, the intrinsic does not affect it.

9.5.4 Parallel 16-bit saturation

These intrinsics are available when `__ARM_FEATURE_SIMD32` is defined. They saturate two 16-bit values to a given bit width as for the `__ssat` and `__usat` intrinsics defined in 9.4.1.

```
int16x2_t __ssat16(int16x2_t, /*constant*/ unsigned int);
```

Saturates two 16-bit signed values to a width in the range 1 to 16. The Q flag is set if either operation saturates.

```
int16x2_t __usat16(int16x2_t, /*constant */ unsigned int);
```

Saturates two 16-bit signed values to a bit width in the range 0 to 15. The input values are signed and the output values are non-negative, with all negative inputs going to zero. The Q flag is set if either operation saturates.

9.5.5 Packing and unpacking

These intrinsics are available when `__ARM_FEATURE_SIMD32` is defined.

```
int16x2_t __sxtab16(int16x2_t, int8x4_t);
```

Two values (at bit positions 0..7 and 16..23) are extracted from the second operand, sign-extended to 16 bits, and added to the first operand.

```
int16x2_t __sxtb16(int8x4_t);
```

Two values (at bit positions 0..7 and 16..23) are extracted from the first operand, sign-extended to 16 bits, and returned as the result.

```
uint16x2_t __uxtab16(uint16x2_t, uint8x4_t);
```

Two values (at bit positions 0..7 and 16..23) are extracted from the second operand, zero-extended to 16 bits, and added to the first operand.

```
uint16x2_t __uxtb16(uint8x4_t);
```

Two values (at bit positions 0..7 and 16..23) are extracted from the first operand, zero-extended to 16 bits, and returned as the result.

9.5.6 Parallel selection

This intrinsic is available when `__ARM_FEATURE_SIMD32` is defined.

```
uint8x4_t __sel(uint8x4_t, uint8x4_t);
```

Selects each byte of the result from either the first operand or the second operand, according to the values of the GE bits. For each result byte, if the corresponding GE bit is set then the byte from the first operand is used, otherwise the byte from the second operand is used. Because of the way that `int16x2_t` operations set two (duplicate) GE bits per value, the `__sel` intrinsic works equally well on (`u`)`int16x2_t` and (`u`)`int8x4_t` data.

9.5.7 Parallel 8-bit addition and subtraction

These intrinsics are available when `__ARM_FEATURE_SIMD32` is defined. Each intrinsic performs 8-bit parallel addition or subtraction. In some cases the result may be halved or saturated.

```
int8x4_t __qadd8(int8x4_t, int8x4_t);
```

4x8-bit addition, saturated to the range -2**7 to 2**7-1.

```
int8x4_t __qsub8(int8x4_t, int8x4_t);
```

4x8-bit subtraction, with saturation.

```
int8x4_t __sadd8(int8x4_t, int8x4_t);
```

4x8-bit signed addition. The GE bits are set according to the results.

```
int8x4_t __shadd8(int8x4_t, int8x4_t);
```

4x8-bit signed addition, halving the results.

```
int8x4_t __shsub8(int8x4_t, int8x4_t);
```

4x8-bit signed subtraction, halving the results.

```
int8x4_t __ssub8(int8x4_t, int8x4_t);
```

4x8-bit signed subtraction. The GE bits are set according to the results.

```
uint8x4_t __uadd8(uint8x4_t, uint8x4_t);
```

4x8-bit unsigned addition. The GE bits are set according to the results.

```
uint8x4_t __uhadd8(uint8x4_t, uint8x4_t);
```

4x8-bit unsigned addition, halving the results.

```
uint8x4_t __uhsub8(uint8x4_t, uint8x4_t);
```

4x8-bit unsigned subtraction, halving the results.

```
uint8x4_t __uqadd8(uint8x4_t, uint8x4_t);
```

4x8-bit unsigned addition, saturating to the range 0 to 2**8-1.

```
uint8x4_t __uqsub8(uint8x4_t, uint8x4_t);
```

4x8-bit unsigned subtraction, saturating to the range 0 to 2**8-1.

```
uint8x4_t __usub8(uint8x4_t, uint8x4_t);
```

4x8-bit unsigned subtraction. The GE bits are set according to the results.

9.5.8 Sum of 8-bit absolute differences

These intrinsics are available when `__ARM_FEATURE_SIMD32` is defined. They perform an 8-bit sum-of-absolute differences operation, typically used in motion estimation.

```
uint32_t __usad8(uint8x4_t, uint8x4_t);
```

Performs 4x8-bit unsigned subtraction, and adds the absolute values of the differences together, returning the result as a single unsigned integer.

```
uint32_t __usada8(uint8x4_t, uint8x4_t, uint32_t);
```

Performs 4x8-bit unsigned subtraction, adds the absolute values of the differences together, and adds the result to the third operand.

9.5.9 Parallel 16-bit addition and subtraction

These intrinsics are available when `__ARM_FEATURE_SIMD32` is defined. Each intrinsic performs 16-bit parallel addition and/or subtraction. In some cases the result may be halved or saturated.

```
int16x2_t __qadd16(int16x2_t, int16x2_t);
```

2x16-bit addition, saturated to the range -2**15 to 2**15-1.

```
int16x2_t __qasx(int16x2_t, int16x2_t);
```

Exchanges halfwords of second operand, adds high halfwords and subtracts low halfwords, saturating in each case.

```
int16x2_t __qsax(int16x2_t, int16x2_t);
```

Exchanges halfwords of second operand, subtracts high halfwords and adds low halfwords, saturating in each case.

```
int16x2_t __qsub16(int16x2_t, int16x2_t);
```

2x16-bit subtraction, with saturation.

```
int16x2_t __sadd16(int16x2_t, int16x2_t);
```

2x16-bit signed addition. The GE bits are set according to the results.

```
int16x2_t __sasx(int16x2_t, int16x2_t);
```

Exchanges halfwords of the second operand, adds high halfwords and subtracts low halfwords. The GE bits are set according to the results.

```
int16x2_t __shadd16(int16x2_t, int16x2_t);
```

2x16-bit signed addition, halving the results.

```
int16x2_t __shasx(int16x2_t, int16x2_t);
```

Exchanges halfwords of the second operand, adds high halfwords and subtract low halfwords, halving the results.

```
int16x2_t __shsax(int16x2_t, int16x2_t);
```

Exchanges halfwords of the second operand, subtracts high halfwords and add low halfwords, halving the results.

```
int16x2_t __shsub16(int16x2_t, int16x2_t);
```

2x16-bit signed subtraction, halving the results.

```
int16x2_t __ssax(int16x2_t, int16x2_t);
```

Exchanges halfwords of the second operand, subtracts high halfwords and adds low halfwords. The GE bits are set according to the results.

```
int16x2_t __ssub16(int16x2_t, int16x2_t);
```

2x16-bit signed subtraction. The GE bits are set according to the results.

```
uint16x2_t __uadd16(uint16x2_t, uint16x2_t);
```

2x16-bit unsigned addition. The GE bits are set according to the results.

```
uint16x2_t __uasx(uint16x2_t, uint16x2_t);
```

Exchanges halfwords of the second operand, adds high halfwords and subtracts low halfwords. The GE bits are set according to the results of unsigned addition.

```
uint16x2_t __uhadd16(uint16x2_t, uint16x2_t);
```

2x16-bit unsigned addition, halving the results.

```
uint16x2_t __uhasx(uint16x2_t, uint16x2_t);
```

Exchanges halfwords of the second operand, adds high halfwords and subtracts low halfwords, halving the results.

```
uint16x2_t __uhsax(uint16x2_t, uint16x2_t);
```

Exchanges halfwords of the second operand, subtracts high halfwords and adds low halfwords, halving the results.

```
uint16x2_t __uhsub16(uint16x2_t, uint16x2_t);
```

2x16-bit unsigned subtraction, halving the results.

```
uint16x2_t __uqadd16(uint16x2_t, uint16x2_t);
```

2x16-bit unsigned addition, saturating to the range 0 to 2**16-1.


```
uint16x2_t __uqasx(uint16x2_t, uint16x2_t);
```

Exchanges halfwords of the second operand, and performs saturating unsigned addition on the high halfwords and saturating unsigned subtraction on the low halfwords.

```
uint16x2_t __uqsax(uint16x2_t, uint16x2_t);
```

Exchanges halfwords of the second operand, and performs saturating unsigned subtraction on the high halfwords and saturating unsigned addition on the low halfwords.

```
uint16x2_t __uqsub16(uint16x2_t, uint16x2_t);
```

2x16-bit unsigned subtraction, saturating to the range 0 to 2**16-1.

```
uint16x2_t __usax(uint16x2_t, uint16x2_t);
```

Exchanges the halfwords of the second operand, subtracts the high halfwords and adds the low halfwords. Sets the GE bits according to the results of unsigned addition.

```
uint16x2_t __usub16(uint16x2_t, uint16x2_t);
```

2x16-bit unsigned subtraction. The GE bits are set according to the results.

9.5.10 Parallel 16-bit multiplication

These intrinsics are available when `__ARM_FEATURE_SIMD32` is defined. Each intrinsic performs two 16-bit multiplications.

```
int32_t __smlad(int16x2_t, int16x2_t, int32_t);
```

Performs 2x16-bit multiplication and adds both results to the third operand. Sets the Q flag if the addition overflows. (Overflow cannot occur during the multiplications.)

```
int32_t __smladx(int16x2_t, int16x2_t, int32_t);
```

Exchanges the halfwords of the second operand, performs 2x16-bit multiplication, and adds both results to the third operand. Sets the Q flag if the addition overflows. (Overflow cannot occur during the multiplications.)

```
int64_t __smlald(int16x2_t, int16x2_t, int64_t);
```

Performs 2x16-bit multiplication and adds both results to the 64-bit third operand. Overflow in the addition is not detected.

```
int64_t __smlaldx(int16x2_t, int16x2_t, int64_t);
```

Exchanges the halfwords of the second operand, performs 2x16-bit multiplication and adds both results to the 64-bit third operand. Overflow in the addition is not detected.

```
int32_t __smlsd(int16x2_t, int16x2_t, int32_t);
```

Performs two 16-bit signed multiplications. Takes the difference of the products, subtracting the high-halfword product from the low-halfword product, and adds the difference to the third operand. Sets the Q flag if the addition overflows. (Overflow cannot occur during the multiplications or the subtraction.)

```
int32_t __smlsdx(int16x2_t, int16x2_t, int32_t);
```

Performs two 16-bit signed multiplications. The product of the high halfword of the first operand and the low halfword of the second operand is subtracted from the product of the low halfword of the first operand and the high halfword of the second operand, and the difference is added to the third operand. Sets the Q flag if the addition overflows. (Overflow cannot occur during the multiplications or the subtraction.)

```
int64_t __smlsld(int16x2_t, int16x2_t, int64_t);
```

Perform two 16-bit signed multiplications. Take the difference of the products, subtracting the high-halfword product from the low-halfword product, and add the difference to the third operand. Overflow in the 64-bit addition is not detected. (Overflow cannot occur during the multiplications or the subtraction.)

```
int64_t __smlsldx(int16x2_t, int16x2_t, int64_t);
```

Perform two 16-bit signed multiplications. The product of the high halfword of the first operand and the low halfword of the second operand is subtracted from the product of the low halfword of the first operand and the high halfword of the second operand, and the difference is added to the third operand. Overflow in the 64-bit addition is not detected. (Overflow cannot occur during the multiplications or the subtraction.)

```
int32_t __smuad(int16x2_t, int16x2_t);
```

Perform 2x16-bit signed multiplications, adding the products together. Set the Q flag if the addition overflows.

```
int32_t __smuadx(int16x2_t, int16x2_t);
```

Exchange the halfwords of the second operand (or equivalently, the first operand), perform 2x16-bit signed multiplications, and add the products together. Set the Q flag if the addition overflows.

```
int32_t __smusd(int16x2_t, int16x2_t);
```

Perform two 16-bit signed multiplications. Take the difference of the products, subtracting the high-halfword product from the low-halfword product.

```
int32_t __smusdx(int16x2_t, int16x2_t);
```

Perform two 16-bit signed multiplications. The product of the high halfword of the first operand and the low halfword of the second operand is subtracted from the product of the low halfword of the first operand and the high halfword of the second operand.

9.5.11 Examples

Taking the elementwise maximum of two SIMD values each of which consists of four 8-bit signed numbers:

```
int8x4_t max8x4(int8x4_t x, int8x4_t y) { __ssub8(x, y); return __sel(x, y); }
```

As described in section 9.5.6, where SIMD values consist of two 16-bit unsigned numbers:

```
int16x2_t max16x2(int16x2_t x, int16x2_t y) { __usub16(x, y); return __sel(x, y); }
```

Note that even though the result of the subtraction is not used, the compiler must still generate the instruction, because of its side-effect on the GE bits which are tested by the `__sel()` intrinsic.

9.6 Floating-point data-processing intrinsics

The intrinsics in this section provide direct access to selected floating-point instructions. They are defined only if the appropriate precision is available in hardware, as indicated by `__ARM_FP` (6.5.1).

```
double __sqrt(double x);
float __sqrtf(float x);
```

The `__sqrt` intrinsics compute the square root of their operand. They have no effect on `errno`. Negative values will produce a default NaN result and possible floating-point exception as described in [ARM ARM A2.7.7].

```
double __fma(double x, double y, double z);
float __fmaf(float x, float y, float z);
```

The `__fma` intrinsics compute `(x*y)+z`, without intermediate rounding. These intrinsics are available only if `__ARM_FEATURE_FMA` is defined. On a Standard C implementation it should not normally be necessary to use these intrinsics, as the `fma` functions defined in [C99 7.12.13] should expand directly to the instructions if available.

10 SYSTEM REGISTER ACCESS

10.1 Special register intrinsics

Intrinsics are provided to read and write system and coprocessor registers, collectively referred to as "special registers".

```
uint32_t __arm_rsr(const char *special_register);
```

reads a 32-bit system register.

```
uint64_t __arm_rsr64(const char *special_register);
```

reads a 64-bit system register.

```
void* __arm_rsrp(const char *special_register);
```

reads a system register containing an address.

```
void __arm_wsr(const char *special_register, uint32_t value);
```

writes a 32-bit system register.

```
void __arm_wsr64(const char *special_register, uint64_t value);
```

writes a 64-bit system register.

```
void __arm_wsrp(const char *special_register, const void *value);
```

writes a system register containing an address.

10.2 Special register designations

The special_register parameter must be a compile time string literal. This means that the implementation can determine the register being accessed at compile-time and produce the correct instruction without having to resort to self-modifying code. All register specifiers are case-insensitive (so "apsr" is equivalent to "APSR"). The string literal should have one of the forms described below.

10.2.1 32-bit coprocessor register

When specifying a 32-bit coprocessor register to __arm_rsr, __arm_rsrp, __arm_wsr, or __arm_wsrp:

```
cp<coprocessor>:<opc1>:c<CRn>:c<CRm>:<opc2>
```

or (equivalently)

```
p<coprocessor>:<opc1>:c<CRn>:c<CRm>:<opc2>
```

where:

<coprocessor> is a decimal integer in the range [0, 15]
<opc1>, <opc2> are decimal integers in the range [0, 7]
<CRn>, <CRm> are decimal integers in the range [0, 15].

The values of the register specifiers will be as described in [ARM ARM] or the Technical Reference Manual (TRM) for the specific processor.

So to read MIDR:

```
unsigned int midr = __arm_rsr("cp15:0:c0:c0:0");
```

ACLE does not specify predefined strings for the system coprocessor register names documented in the ARM ARM (e.g. "MIDR").

10.2.2 32-bit system register

When specifying a 32-bit system register to __arm_rsr, __arm_rsrp, __arm_wsr, or __arm_wsrp, one of:

- the values accepted in the spec_reg field of the MRS instruction [ARMARM-AR B6.1.5], e.g. “CPSR”
- the values accepted in the spec_reg field of the MSR (immediate) instruction [ARMARM B6.1.6]
- the values accepted in the spec_reg field of the VMRS instruction [ARMARM B6.1.14], e.g. “FPSID”
- the values accepted in the spec_reg field of the VMSR instruction [ARMARM B6.1.15], e.g. “FPSCR”
- the values accepted in the spec_reg field of the MSR and MRS instructions with virtualization extensions [ARM ARM B1.7], e.g. “ELR_Hyp”
- the values specified in ‘Special register encodings used in ARMv7-M system instructions.’ [ARMv7M B5.1.1], e.g. “PRIMASK”

10.2.3 64-bit coprocessor register

When specifying a 64-bit coprocessor register to `__arm_rsr64` or `__arm_wsr64`:

```
cp<coprocessor>:<opc1>:c<CRm>
```

or (equivalently)

```
p<coprocessor>:<opc1>:c<Rm>
```

where:

<coprocessor> is a decimal integer in the range [0, 15]
<opc1> is a decimal integer in the range [0, 7]
<CRm> is a decimal integer in the range [0, 15]

10.3 Unspecified behavior

ACLE does not specify how the implementation should behave in the following cases:

- when merging multiple reads/writes of the same register
- when writing to a read-only register, or a register that is undefined on the architecture being compiled for
- when reading or writing to a register which the implementation models by some other means (this covers – but is not limited to – reading/writing cp10 and cp11 registers when VFP is enabled, and reading/writing the CPSR)
- when reading or writing a register using one of these intrinsics with an inappropriate type for the value being read or written to.

11 INSTRUCTION GENERATION

11.1 Instruction generation, arranged by instruction

The following table indicates how instructions may be generated by intrinsics, and/or C code. The table includes integer data processing and certain system instructions.

Compilers are encouraged to use opportunities to combine instructions, or to use shifted/rotated operands where available. In general, intrinsics are not provided for accumulating variants of instructions in cases where the accumulation is a simple addition (or subtraction) following the instruction.

The table indicates which architectures the instruction is supported on, as follows:

- architecture '7' means ARM v7-A and ARM v7-R
- in the sequence of ARM architectures { 5, 5TE, 6, 6T2, 7 } each architecture includes its predecessor instruction set
- in the sequence of Thumb-only architectures { 6-M, 7-M, 7E-M } each architecture includes its predecessor instruction set

Instruction	Flgs	Arch.	Intrinsic or C code
BKPT		5	none
BFC		6T2, 7M	C
BFI		6T2, 7M	C
CLZ		5	__clz, __builtin_clz
DBG		7, 7M	__dbg
DMB		7, 6M	__dmb
DSB		7, 6M	__dsb
ISB		7, 6M	__isb
LDREX		6, 7M	__sync_xxx
LDRT		all	none
MCR/MRC		all	see 10
MSR/MRS		6M	see 10
PKHBT		6	C
PKHTB		6	C
PLD		5TE, 7M	__pld
PLDW		7MP	__pldx
PLI		7	__pli
QADD	Q	5E, 7EM	__qadd
QADD16		6, 7EM	__qadd16
QADD8		6, 7EM	__qadd8
QASX		6, 7EM	__qasx
QDADD	Q	5E, 7EM	__qadd(__qdbl)
QDSUB	Q	5E, 7EM	__qsub(__qdbl)
QSAX		6, 7EM	__qsax
QSUB	Q	5E, 7EM	__qsub
QSUB16		6, 7EM	__qsub16
QSUB8		6, 7EM	__qsub8
RBIT		6T2, 7M	__rbit, __builtin_rbit
REV		6, 6	__rev, __builtin_bswap32
REV16		6, 6M	__rev16
REVSH		6, 6M	__revsh
ROR		all	__ror
SADD16	GE	6, 7EM	__sadd16
SADD8	GE	6, 7EM	__sadd8
SASX	GE	6, 7EM	__sasx
SBFX		6T2, 7M	C
SDIV		7M+	C
SEL	(GE)	6, 7EM	__sel
SETEND		6	n/a
SEV		6K	__sev
SHADD16		6, 7EM	__shadd16
SHADD8		6, 7EM	__shadd8
SHASX		6, 7EM	__shasx
SHSAX		6, 7EM	__shsax
SHSUB16		6, 7EM	__shsub16
SHSUB8		6, 7EM	__shsub8
SMC		6Z, T2	none
SMI		6Z, T2	none
SMLABB	Q	5E, 7EM	__smlabb
SMLABT	Q	5E, 7EM	__smlabt
SMLAD	Q	6, 7EM	__smlad
SMLADX	Q	6, 7EM	__smladx
SMLAL		all, 7M	C
SMLALBB		5E, 7EM	__smulbb and C
SMLALBT		5E, 7EM	__smulbt and C
SMLALTB		5E, 7EM	__smultb and C
SMLALTT		5E, 7EM	__smultt and C
SMLALD		6, 7EM	__smlald
SMLALDX		6, 7EM	__smlaldx
SMLATB	Q	5E, 7EM	__smlatb
SMLATT	Q	5E, 7EM	__smlatt
SMLAWB	Q	5E, 7EM	__smlawb
SMLAWT	Q	5E, 7EM	__smlawt
SMLSD	Q	6, 7EM	__smlsd
SMLSDX	Q	6, 7EM	__smlsdx
SMLSLD		6, 7EM	__smlsld
SMLSLDX		6, 7EM	__smlsldx
SMMLA		6, 7EM	C
SMMLAR		6, 7EM	C
SMMLS		6, 7EM	C
SMMLSR		6, 7EM	C
SMMUL		6, 7EM	C
SMMULR		6, 7EM	C

SMUAD	Q	6, 7EM	__smuad
SMUADX	Q	6, 7EM	__smuadx
SMULBB		5E, 7EM	__smulbb; C
SMULBT		5E, 7EM	__smulbt ; C
SMULTB		5E, 7EM	__smultb; C
SMULTT		5E, 7EM	__smultt; C
SMULL		all, 7M	C
SMULWB		5E, 7EM	__smulwb; C
SMULWT		5E, 7EM	__smulwt; C
SMUSD		6, 7EM	__smusd
SMUSDX		6, 7EM	__smusd
SSAT	Q	6, 7M	__ssat
SSAT16	Q	6, 7EM	__ssat16
SSAX	GE	6, 7EM	__ssax
SSUB16	GE	6, 7EM	__ssub16
SSUB8	GE	6, 7EM	__ssub8
STREX		6, 7M	__sync_xxx
STRT		all	none
SVC		all	none
SWP		ARM only	__swp [deprecated; see 8.5]
SXTAB		6, 7EM	(int8_t)x + a
SXTAB16		6, 7EM	__sxtab16
SXTAH		6, 7EM	(int16_t)x + a
SXTB		6, 6M	(int8_t)x
SXTB16		6, 7EM	__sxtb16
SXTH		6, 6M	(int16_t)x
UADD16	GE	6, 7EM	__uadd16
UADD8	GE	6, 7EM	__uadd8
UASX	GE	6, 7EM	__uasx
UBFX		6T2, 7M	C
UDIV		7M+	C
UHADD16		6, 7EM	__uhadd16
UHADD8		6, 7EM	__uhadd8
UHASX		6, 7EM	__uhasx
UHSAX		6, 7EM	__uhsax
UHSUB16		6, 7EM	__uhsub16
UHSUB8		6, 7EM	__uhsub8
UMAAL		6, 7EM	C
UMLAL		all, 7M	acc += (uint64_t)x * y
UMULL		all, 7M	C
UQADD16		6, 7EM	__uqadd16
UQADD8		6, 7EM	__uqadd8
UQASX		6, 7EM	__uqasx
UQSAX		6, 7EM	__uqsax
UQSUB16		6, 7EM	__uqsub16
UQSUB8		6, 7EM	__uqsub8
USAD8		6, 7EM	__usad8
USADA8		6, 7EM	__usad8 + acc
USAT	Q	6, 7M	__usat
USAT16	Q	6, 7EM	__usat16
USAX		6, 7EM	__usax
USUB16		6, 7EM	__usub16
USUB8		6, 7EM	__usub8
UXTAB		6, 7EM	(uint8_t)x + i
UXTAB16		6, 7EM	__uxtab16
UXTAH		6, 7EM	(uint16_t)x + i
UXTB16		6, 7EM	__uxtb16
UXTH		6, 6M	(uint16_t)x
VFMA		VFPv4	fma, __fma
VSQRT		VFP	sqrt, __sqrt
WFE		6K, 6M	__wfe
WFI		6K, 6M	__wfi
YIELD		6K, 6M	__yield

12 NEON INTRINSICS

12.1 Availability of NEON intrinsics

The NEON intrinsics correspond to operations on the ARM NEON extension. This architectural extension provides for arithmetic, logical and saturated arithmetic operations on 8-bit, 16-bit and 32-bit integers (and sometimes on 64-bit integers) and on 32-bit floating-point data, arranged in 64-bit and 128-bit vectors.

NEON intrinsics are available if the __ARM_NEON macro is predefined (see 6.5.4), but in order to access NEON intrinsics it is necessary to include the <arm_neon.h> header.

12.1.1 16-bit floating-point availability

When the 16-bit floating-point data type is available in the scalar VFP instruction set, it is also available in NEON. This will be indicated by the setting of bit 1 in __ARM_NEON_FP (see 6.5.5).

12.1.2 Fused multiply-accumulate availability

Fused multiply-accumulate is available in the NEON extension when available in the scalar instruction set, as indicated by __ARM_FEATURE_FMA. When fused multiply-accumulate is available, extra NEON intrinsics are defined to access it.

12.2 NEON data types

12.2.1 Vector data types

Vector data types are named as a lane type and a multiple. Lane type names are based on the scalar types defined in <stdint.h> with a few additional types as detailed in 12.2.3. For example,. int16x4_t is a vector of four int16_t values. The base type prefixes are int8, uint8, int16, uint16, int32, uint32, int64, uint16, float16, float32, poly8 and poly16. The multiples are such that the resulting vector types are 64-bit and 128-bit.

Not all types can be used in all operations. Generally, the operations available on a type correspond to the operations available on the corresponding scalar type.

A few NEON intrinsics operate on a single 64-bit quantity as a degenerate case of vector instructions. ACLE does not define whether int64x1_t is the same type as int64_t, or whether uint64x1_t is the same type as uint64_t, e.g. for C++ overloading purposes.

float16 vector types are only available when the __fp16 type is defined, i.e. when supported by the hardware. As with scalar (VFP) operations, 16-bit floating-point types cannot be used in arithmetic operations. They can be used in conversions to and from 32-bit floating-point types, in loads and stores, and in reinterpret operations.

12.2.2 Vector array data types

Array types are defined for multiples of 2, 3 or 4 of all the vector types, for use in load and store operations, in table-lookup operations, and as the result type of operations that return a pair of vectors. For a vector type <type>_t the corresponding array type is <type>x<length>_t. Concretely, an array type is a structure containing a single array element called val.

For example an array of two int16x4_t types is int16x4x2_t, and is represented as

```
struct int16x4x2_t { int16x4_t val[2]; };
```


Note that this array type of two 64-bit vectors is distinct from the 128-bit vector type `int16x8_t`.

12.2.3 Scalar data types

For consistency, `<arm_neon.h>` defines some additional scalar data types to match the vector types.

`float32_t` is defined as an alias for `float`.

If the `__fp16` type is defined, `float16_t` is defined as an alias for it.

`poly8_t` and `poly16_t` are defined as unsigned integer types. It is unspecified whether these are the same type as `uint8_t` and `uint16_t` for overloading and mangling purposes.

12.2.4 Operations on data types

ACLE does not define implicit conversion between different data types. E.g. the following is not portable:

```
int32x4_t x;
uint32x4_t y = x;    // No representation change
float32x4_t z = x;   // Conversion of integer to floating type
```

Use the `vreinterpret` intrinsics to convert from one vector type to another without changing representation, and use the `vcvt` intrinsics to convert between integer and floating types; for example:

```
int32x4_t x;
uint32x4_t y = vreinterpretq_u32_s32(x);
float32x4_t z = vcvt_f32_s32(x);
```

ACLE does not define static construction of vector types. E.g. the following is not portable:

```
int32x4_t x = { 1, 2, 3, 4 };    // not portable
```

Use the `vcreate` or `vdup` intrinsics to construct values from scalars.

In C++, ACLE does not define whether NEON data types are POD types or whether they can be inherited from.

12.3 Specification of NEON intrinsics

12.3.1 Introduction

The intrinsics are presented in the following order:

- construction and deconstruction of vectors
- loading and storing vectors
- lane-by-lane (SIMD) operations
- reductions
- rearrangements and table lookups

12.3.2 Explanation of NEON intrinsics templates

In order to present the NEON intrinsics in a compact form, they are specified here in a generic way, as templates. In this specification, all capital letters in intrinsic names and types are template parameters. Both names and types are specified with placeholders to be filled in with a vector type parameter T or some type or string derived from T. T is a vector type such as `int16x4_t` for a vector of four lanes of signed 16-bit integers. Other types are derived from T as follows:

Key	Meaning	Restriction on T
B	number of bits in a lane, i.e. number of bits in type ET	any
DT	128-bit vector type with lanes twice as wide as T (where this exists). This is used in widening operations	64-bit vector
ET	element (lane) type of vector type T	any
FT	vector type with lanes of 32-bit floating type	lane type is 32-bit integral type
HT	64-bit vector type with lanes half as wide as T (where this exists). This is used in narrowing operations. There are no types with 4-bit lanes.	128-bit vector
N	number of lanes in a 64-bit vector of the same lane type; this is used as a limit for a lane selection intrinsic	any
Q	the string "`q`" if T is a 128-bit vector type; this is used in forming the names of intrinsics	any
ST	the short-form name of the lane type, such as `s16` for a signed 16-bit integer	any
T2	the 128-bit vector type with the same lane type	64-bit vector
T64	the 64-bit vector type with the same lane type	any
TxN for N from 1 to 4	array of T; so where T is an `int8x8_t`, Tx3 is `int8x8x3_t`; this is used in intrinsics which return multiple results, or where input operands consist of multiple vectors. Where N is 1, the array type is simply T	any
UHT	the 64-bit vector type with lanes half as wide as T and of unsigned type. This is used as the result of signed-to-unsigned narrowing saturation operations	128-bit vector
UT	the vector type of the same size and lane size as T but whose lane type is an unsigned integer; used as the result of comparison operations and signed-to-unsigned saturation operations, and as an operand in selection operations	any

12.3.2.1 Examples of template type parameters

T	Q	lanes	ET	ST	DT	HT	UT	T64	T2
`int8x8_t`		0..7	`int8_t`	`s8`	`int16x8_t`	n/a	`uint8x8_t`	`int8x8_t`	`int8x16_t`
`uint16x8_t`	`q`	0..7	`uint16_t`	`u16`	n/a	`uint8x8_t`	`uint16x8_t`	`uint16x4_t`	
`float32x4_t`	`q`	0..3	`float32_t`	`f32`	n/a	n/a	`uint32x4_t`	`float32x2_t`	

12.3.3 Intrinsics with scalar operands

Some NEON vector operations use a scalar (non-vector) value. Depending on the intrinsic, scalar values may be obtained in one of two ways:

- directly supplied as a scalar operand. These intrinsics are identified with the string "`_n`" in their name. Depending on the intrinsic, this operand may be a compile-time integral constant (e.g. a shift count), or it may be a general expression (usually of the same type as the vector lanes).
- from one lane of an input vector. These intrinsics are identified with the string "`_lane`" in their name. The lane number is the last argument and must be a compile-time constant and within range. The input vector from which the scalar operand is taken is the preceding operand and is always a 64-bit vector.

12.3.4 Summary of intrinsic naming conventions

All capital letters in intrinsic descriptions in this specification are template parameters. Intrinsic names are modelled after the NEON instruction set and generally follow the following scheme:

`v`[`q`][`r`]*name*[`u`][`n`][`q`][`_lane`][`_n`][_*result*]_*type*

where

- `q` indicates a saturating operation
- `r` indicates a rounding operation
- *name* is the descriptive name of the basic operation
- `u` indicates signed-to-unsigned saturation
- `n` indicates a narrowing operation
- `q` postfixing the name indicates an operation on 128-bit vectors
- `_n` indicates a scalar operand supplied as an argument
- `_lane` indicates a scalar operand taken from the lane of a vector
- *result* is the result type in short form
- *type* is the primary operand type in short form

12.3.5 Lane type classes

Type classes indicate permissible sets of types for data-processing intrinsics. For example, whether an intrinsic can operate on integer, floating or polynomial types, and the size of operands, depends on the data operation.

class	count	types
int	6	int8, int16, int32, uint8, uint16, uint32
int/64	8	int8, int16, int32, int64, uint8, uint16, uint32, uint64
sint	3	int8, int16, int32
sint/64	4	int8, int16, int32, int64
sint/float	4	int8, int16, int32, float32
int16/32	4	int16, uint16, int32, uint32
sint16/32	2	int16, int32
int32	2	int32, uint32

8-bit	3	int8, uint8, poly8
int/poly8	7	int8, int16, int32, uint8, uint16, uint32, poly8
int/poly	8	int8, int16, int32, uint8, uint16, uint32, poly8, poly16
int/64/poly	10	int8, int16, int32, int64, uint8, uint16, uint32, uint64, poly8, poly16
arith	7	int8, int16, int32, uint8, uint16, uint32, float32
arith16/32	5	int16, int32, uint16, uint32, float32
arith/64	9	int8, int16, int32, int64, uint8, uint16, uint32, uint64, float32
arith/poly8	8	int8, int16, int32, uint8, uint16, uint32, poly8, float32
floating	1	float32
any	12*	int8, int16, int32, int64, uint8, uint16, uint32, uint64, poly8, poly16, float32, (float16)

* Note: float16 is only available if supported in target hardware.

12.3.6 Constructing and deconstructing NEON vectors

The intrinsics in this section construct and deconstruct NEON vectors. In some cases, they may be “free” operations, in the sense that a compiler can achieve the effect (e.g. of combining two 64-bit vectors into a 128-bit vector) by register allocation. Also, in many cases the most natural way to construct a NEON vector is to load it from an array.

```
T vcreate_ST(uint64_t a);
```

creates a vector directly from a 64-bit value. T can be any 64-bit vector type. There are 12 intrinsics.

```
T vdupQ_n_ST(ET value);
```

creates a vector by duplicating a scalar value across all lanes. T can be any vector type for which ET exists. There are 22 intrinsics.

```
T vdupQ_lane_ST(T64 vec, const int lane);
```

creates a vector by duplicating one lane of a source vector. T can be any vector type. T64 is the 64-bit vector type corresponding to T. The scalar value is obtained from a designated lane of the input vector. There are 22 intrinsics.

```
T2 vcombine_ST(T low, T high);
```

creates a 128-bit vector by combining two 64-bit vectors. T can be any 64-bit vector type. There are 12 intrinsics.

```
T vget_high_ST(T2 a);
T vget_low_ST(T2 a);
```

gets the high, or low, half of a 128-bit vector. There are 24 intrinsics.

```
T vsetQ_lane_ST(ET value, T vec, const int lane);
```

sets the specified lane of an input vector to be a new value. There are 24 intrinsics.

```
ET vgetQ_lane_ST(T vec, const int lane);
```

gets the value from the specified lane of an input vector. There are 24 intrinsics.

```
T’ vreinterpretQ_ST’_ST(T a);
```


reinterprets a vector of one type T as a vector of another type T', without any operation taking place. The lane sizes may be the same or different. For example, `vreinterpretq_s8_f32()` reinterprets a vector of four 32-bit floating-point lanes as a vector of sixteen 8-bit signed integer lanes.

12.3.6.1 Examples

The following "no-op" expressions demonstrate some relationships between these intrinsics:

```
vcombine_ST(vget_low_ST(a), vget_high_ST(a))
```

```
vset_lane_ST(vget_lane_ST(a, N), a, N)
```

```
vreinterpret_ST_u8(vreinterpret_u8_ST(a))
```

12.3.7 NEON loads and stores

There are separate load and store intrinsics for each lane type, but they are implemented as common instructions determined by lane size and vector size.

```
T vld1Q_ST(ET const *ptr);
```

loads a vector elementwise from an array.

```
T vld1Q_lane_ST(ET const *ptr, T vec, int lane);
```

loads one lane of a vector.

```
T vld1Q_dup_ST(ET const *ptr);
```

duplicates a loaded value into all lanes of a vector.

```
void vst1Q_ST(ET *ptr, T val);
```

stores a vector elementwise into an array.

```
void vst1Q_lane_ST(ET *ptr, T val, const int lane);
```

stores one lane of a vector.

```
TxN vldNQ_ST(ET const *ptr);
```

for N from 2 to 4, loads N vectors from an array, with de-interleaving. The array consists of a sequence of sets of N values. The first element of the array is placed in the first lane of the first vector, the second element in the first lane of the second vector, and so on. For example, `vld3_s32` will load the six 32-bit elements { A, B, C, D, E, F } into the three 64-bit vectors { DA, EB, FC }. Not available for 64-bit lanes when T is a quadword type.

```
TxN vldNQ_dup_ST(ET const *ptr);
```

for N from 2 to 4, loads a single N-element structure to all lanes of N vectors. N values are loaded, then duplicated across all lanes. For example, `vld3_dup_s16` will load the three consecutive 16-bit elements { A, B, C } and produce the three 64-bit vectors { AAAA, BBBB, CCCC }, while the quadword form `vld3q_dup_s16` will produce the three 128-bit vectors { AAAAAAAA, BBBBBBBB, CCCCCCCC }.

```
void vstNQ_ST(ET *ptr, TxN val);
```

for N from 2 to 4, stores N vectors to an array, with interleaving. Every element of each vector is stored. Not available for 64-bit lanes when T is a quadword type.

```
TxN vldNQ_lane_ST(ET const *ptr, TxN src, const int lane);
```

for N from 2 to 4, loads a single N-element structure to the designated lane of N vectors. Not available for 64-bit lanes; not available for 8-bit lanes and quadword vectors.


```
void vstNQ_lane_ST(ET *ptr, TxN val, const int lane);
```

for N from 2 to 4, stores a single N-element structure from the designated lane of N vectors. Not available for 64-bit lanes; not available for 8-bit lanes and quadword vectors.

```
TxN vld1Q_ST_xN(ET const *ptr);
```

for N from 2 to 4, loads N vectors from an array without de-interleaving. The first element (at the lowest address) of the array is placed in the first lane of the first vector, the second element in the second lane of the first vector and so on. For example, `vld1_s32_x4` will load the eight 32-bit array elements {A, B, C, D, E, F, G, H} into the four 64-bit vectors {BA, DC, FE, HG}. This intrinsic is new in ACLE 1.1.

```
void vst1Q_ST_xN(ET *ptr, TxN vec);
```

for N from 2 to 4, stores N vectors from a register to an array without de-interleaving. The first element (at LSB) of the register is placed in the lowest address of the array, the second lane of the first vector in the second element of the array and so on. For example, vst1_s32_x4 will store four 64-bit vectors {BA, DC, FE, HG} into the eight 32-bit array elements {A, B, C, D, E, F, G, H}. This intrinsic is new in ACLE 1.1.

12.3.7.1 Examples

This is an example of iterating through an array, with fixup code for any elements left over:

```
void scale_values(float *a, int n, float scale) {
  int i;
  for (i = 0; i < (n & ~3); ++i) {
    vst1q_f32(&a[i], vmulq_n_f32(vld1q_f32(&a[i]), scale));
  }
  if (i & 2) {
    vst1_f32(&a[i], vmul_n_f32(vld1_f32(&a[i]), scale));
    i += 2;
  }
  if (i & 1) {
    a[i] *= scale;
  }
}
```

If the array is known to contain an integral number of whole vectors, fixup code is not necessary.

12.3.7.2 Alignment assertions

The NEON load and store instructions provide for alignment assertions, which may speed up access to aligned data (and will fault access to unaligned data). The NEON intrinsics as defined in this document do not directly provide a means for asserting alignment. Implementations may be able to introduce these assertions by analyzing the alignment of types or data. In the C++ example below, the type alignment of the `Point` type would allow the compiler to assert alignment on the NEON loads and stores:

```
#if !(__cplusplus >= 201103L)
#define alignas(X) __attribute__((aligned(X)))
#endif
struct Point alignas(16) { float32x4_t point; };

void scale_points(Point *a, int n, float scale) {
  for (int i = 0; i < n; ++i) {
    a[i].point = vmulq_n_f32(a[i].point, scale);
  }
}
```


12.3.8 NEON lane-by-lane (SIMD) operations

The operations in this table perform lane-by-lane (SIMD) operations, where operations proceed independently in parallel. The result is a vector and at least one input operand is a vector. In some cases an input operand may be a scalar, supplied as a parameter or extracted from a lane of a vector, as described in 12.2.3.

Comparison operations result in a lane of all 1s where the condition is true, all 0s otherwise. The resulting bit vector is typically used with the `vbsl` intrinsic.

Saturation clips the result of an operation to the output range when it is narrowed to a smaller result type or shifted left. Signed-to-unsigned saturation clips a signed result to an unsigned range, so that negative results go to zero.

Rounding is used when a value is shifted right, or when the high part of a result is taken. It effectively adds a value equivalent to 0.5 bits to the value before truncating it, i.e. applying the “round half up” rule.

Variable shift operations are bidirectional, i.e. a shift count is encoded as a signed integer. A shift operation may be both saturating (when the value is shifted left, or narrowed) and rounding (when the value is shifted right).

template	count	types	operation	instruction
T vaddQ_ST(T a, T b)	18	arith/64	add	VADD
DT vaddl_ST(T a, T b)	6	int	long add	VADDL
DT vaddw_ST(DT a, T b)	6	int	wide add	VADDW
T vhaddQ_ST(T a, T b)	12	int	halving add	VHADD
T vrhaddQ_ST(T a, T b)	12	int	rounding halving add	VRHADD
T vqaddQ_ST(T a, T b)	16	int/64	saturating add	VQADD
HT vaddhn_ST(T a, T b)	6	int/64	add high half	VADDHN
HT vraddhn_ST(T a, T b)	6	int/64	rounding add high half	VRADDHN
T vmulQ_ST(T a, T b)	16	arith/poly8	multiply	VMUL
T vmlaQ_ST(T a, T b, T c)	14	arith	multiply accumulate (a + b*c)	VMLA
DT vmlal_ST(DT a, T b, T c)	6	int	multiply accumulate long	VMLAL
T vmlsQ_ST(T a, T b, T c)	14	arith	multiply subtract (a – b*c)	VMLS
DT vmlsl_ST(DT a, T b, T c)	6	int	multiply subtract long	VMLSL
T vfmaQ_ST(T a, T b, T c)	2	floating	fused multiply-accumulate (where available)	VFMA
T vfmsQ_ST(T a, T b, T c)	2	floating	fused multiply-subtract (where available)	VFMS
T vqdmulhQ_ST(T a, T b)	4	sint16/32	saturating doubling multiply high	VQDMULH
T vqrdmulhQ_ST(T a, T b)	4	sint16/32	saturating rounding doubling multiply high	VQRDMULH
DT vqdmlal_ST(DT a, T b, T c)	2	sint16/32	saturating doubling multiply accumulate long	VQDMLAL
DT vqdmlsl_ST(DT a, T b, T c)	2	sint16/32	saturating doubling multiply subtract long	VQDMLSL
DT vmull_ST(T a, T b)	7	int/poly8	long multiply	VMULL

DT vqdmull_ST(T a, T b)	2	sint16/32	saturating doubling long multiply	VQDMULL
T vsubQ_ST(T a, T b)	18	arith/64	subtract	VSUB
DT vsubl_ST(T a, T b)	6	int	long subtract	VSUBL
DT vsubw_ST(DT a, T b)	6	int	wide subtract	VSUBW
T vhsubQ_ST(T a, T b)	12	int	halving subtract	VHSUB
T vqsubQ_ST(T a, T b)	16	int/64	saturating subtract	VQSUB
HT vsubhn_ST(T a, T b)	6	int/64	subtract high half	VSUBHN
HT vrsubhn_ST(T a, T b)	6	int/64	rounding subtract high half	VRSUBHN
UT vceqQ_ST(T a, T b)	16	arith/poly8	compare equal	VCEQ
UT vcgeQ_ST(T a, T b)	14	arith	compare greater-than or equal	VCGE
UT vcleQ_ST(T a, T b)	14	arith	compare less-than or equal	VCGE
UT vcgtQ_ST(T a, T b)	14	arith	compare greater-than	VCGT
UT vcltQ_ST(T a, T b)	14	arith	compare less-than	VCGT
UT vcageQ_ST(T a, T b)	2	floating	compare absolute greater-than or equal	VACGE
UT vcaleQ_ST(T a, T b)	2	floating	compare absolute less-than or equal	VACGE
UT vcagtQ_ST(T a, T b)	2	floating	compare absolute greater-than	VACGT
UT vcaltQ_ST(T a, T b)	2	floating	compare absolute less-than	VACGT
UT vtstQ_ST(T a, T b)	16	int/poly	test bits	VTST
T vabdQ_ST(T a, T b)	14	arith	absolute difference	VABD
DT vabdl_ST(T a, T b)	6	int	absolute difference - long	VABDL
T vabaQ_ST(T a, T b, T c)	12	int	absolute difference and accumulate	VABA
DT vabal_ST(DT a, T b, T c)	6	int	absolute difference and accumulate - long	VABAL
T vmaxQ_ST(T a, T b)	14	arith	maximum	VMAX
T vminQ_ST(T a, T b)	14	arith	minimum	VMIN
T vshlQ_ST(T a, T b)	16	int/64	shift left by signed variable	VSHL
T vqshlQ_ST(T a, T b)	16	int/64	saturating shift left by signed variable	VQSHL
T vrshlQ_ST(T a, T b)	16	int/64	rounding shift left by signed variable	VRSHL
T vqrshlQ_ST(T a, T b)	16	int/64	saturating rounding shift left by signed variable	VQRSHL
T vshrQ_n_ST(T a, 1..B)	16	int/64	shift right by constant	VSHR
T vshlQ_n_ST(T a, 0..B-1)	16	int/64	shift left by constant	VSHL

T vrshrQ_n_ST(T a, 1..B)	16	int/64	rounding shift right by constant	VRSHR
T vsraQ_n_ST(T a, T b, 1..B)	16	int/64	shift right by constant and accumulate	VSRA
T vrsraQ_n_ST(T a, T b, 1..B)	16	int/64	rounding shift right by constant and accumulate	VRSRA
T vqshlQ_n_ST(T a, 0..B-1)	16	int/64	saturating shift left by constant	VQSHL
T vqshluQ_n_ST(T a, 0..B-1)	8	sint/64	signed-to-unsigned saturating shift left by constant	VQSHLU
HT vshrn_n_ST(T a, 1..B/2)	6	int/64	narrowing shift right by constant	VSHRN
UHT vqshrun_n_ST(T a, 1..B/2)	3	sint/64	signed-to-unsigned narrowing saturating shift right by constant	VQSHRUN
UHT vqrshrun_n_ST(T a, 1..B/2)	3	sint/64	signed-to-unsigned narrowing saturating rounding shift right by constant	VQRSHRUN
HT vqshrn_n_ST(T a, 1..B/2)	6	int/64	narrowing saturating shift right by constant	VQSHRN
HT vrshrn_n_ST(T a, 1..B/2)	6	int/64	rounding narrowing shift right by constant	VRSHRN
HT vqrshrn_n_ST(T a, 1..B/2)	6	int/64	rounding narrowing saturating shift right by constant	VQRSHRN
DT vshll_n_ST(T a, 0..B-1)	6	int	widening shift left by constant	VSHLL
T vsriQ_n_ST(T a, T b, 1..B)	20	int/64/poly	shift right and insert	VSRI
T vsliQ_n_ST(T a, T b, 0..B-1)	20	int/64/poly	shift left and insert	VSLI
T vcvtQ_ST_f32(FT a)	4	int32	convert float to integer	VCVT
T vcvtQ_n_ST_f32(FT a, 1..32)	4	int32	convert float to fixed-point	VCVT
FT vcvtQ_f32_ST(T a)	4	int32	convert integer to float	VCVT
FT vcvtQ_n_f32_ST(T a, 1..32)	4	int32	convert fixed-point to float	VCVT
HT vcvt_f16_f32(T a)	1	floating	convert from 32-bit float to 16-bit float	VCVT
T vcvt_f32_f16(HT a)	1	floating	convert from 16-bit float to 32-bit float	VCVT
HT vmovn_ST(T a)	6	int/64	narrow integer	VMOVN
DT vmovl_ST(T a)	6	int	long move	VMOVL
HT vqmovn_ST(T a)	6	int/64	saturating narrow integer	VQMOVN
UHT vqmovun_ST(T a)	3	signed/64	saturating narrow integer signed to unsigned	VQMOVUN
T vmlaQ_lane_ST(T a, T b, T64 v, 0..N-1)	10	arith16/32	multiply-acccumulate with scalar	VMLA
DT vmlal_lane_ST(DT a, T b, T64 v, 0..N-1)	4	int16/32	widening multiply-accumulate with scalar	VMLAL
DT vqdmlal_lane_ST(DT a, T b, T64 v, 0..N-1)	2	sint16/32	widening saturating doubling multiply-accumulate with scalar	VQDMLAL

T vmlsQ_lane_ST(T a, T b, T64 v, 0..N-1)	10	arith16/32	multiply-subtract with scalar	VMLS
DT vmlsl_lane_ST(DT a, T b, T64 v, 0..N-1)	4	int16/32	widening multiply-subtract with scalar	VMLSL
DT vqdmlsl_lane_ST(DT a, T b, T64 v, 0..N-1)	2	sint16/32	widening saturating doubling multiply-subtract with scalar	VQDMLSL
T vmulQ_n_ST(T a, ET b)	10	arith16/32	multiply by scalar	VMUL
DT vmull_n_ST(T a, ET b)	4	int16/32	long multiply by scalar	VMULL
DT vmull_lane_ST(T a, T64 b, 0..N-1)	4	int16/32	long multiply by scalar lane	VMULL
DT vqdmull_n_ST(T a, ET b)	2	sint16/32	saturating doubling long multiply by scalar	VQDMULL
DT vqdmull_lane_ST(T a, T64 b, 0..N-1)	2	sint16/32	saturating doubling long multiply by scalar lane	VQDMULL
T vqdmulhQ_n_ST(T a, ET b)	4	sint16/32	saturating doubling multiply high by scalar	VQDMULH
T vqdmulhQ_lane_ST(T a, T64 b, 0..N-1)	4	sint16/32	saturating doubling multiply high by scalar lane	VQDMULH
T vqrdmulhQ_n_ST(T a, ET b)	4	sint16/32	saturating rounding doubling multiply high by scalar	VQRDMULH
T vqrdmulhQ_lane_ST(T a, T64 b, 0..N-1)	4	sint16/32	saturating rounding doubling multiply high by scalar lane	VQRDMULH
T vmlaQ_lane_ST(T a, T b, T64 v, 0..N-1)	10	arith16/32	multiply accumulate with scalar	VMLA
DT vmlal_n_ST(DT a, T b, ET c)	4	int16/32	widening multiply accumulate with scalar	VMLAL
DT vqdmlal_n_ST(DT a, T b, ET c)	2	sint16/32	widening saturating doubling multiply accumulate with scalar	VQDMLAL
T vmlsQ_lane_ST(T a, T b, T64 v, 0..N-1)	10	arith16/32	multiply subtract with scalar	VMLS
DT vmlsl_n_ST(DT a, T b, ET c)	4	int16/32	widening multiply subtract with scalar	VMLSL
DT vqdmlsl_n_ST(DT a, T b, ET c)	2	sint16/32	widening saturating doubling multiply subtract with scalar	VQDMLSL
T vabsQ_ST(T a)	8	sint/float	absolute	VABS
T vqabsQ_ST(T a)	6	sint	saturating absolute	VQABS
T vnegQ_ST(T a)	8	sint/float	negate	VNEG
T vqnegQ_ST(T a)	6	sint	saturating negate	VQNEG
T vclsQ_ST(T a)	6	sint	count leading sign bits	VCLS
T vclzQ_ST(T a)	12	int	count leading zeros	VCLZ
T vcntQ_ST(T a)	6	8-bit	count number of set bits	VCNT
T vrecpeQ_ST(T a)	4	f32,u32	reciprocal estimate	VRECPE
T vrecpsQ_ST(T a, T b)	2	f32	reciprocal step	VRECPS

T vrsqrteQ_ST(T a)	4	f32,u32	reciprocal square root estimate	VRSQRTE
T vrsqrtsQ_ST(T a, T b)	2	f32	reciprocal square root step	VRSQRTS
T vmvnQ_ST(T a)	14	int/poly8	bitwise not	VMVN
T vandQ_ST(T a, T b)	16	int/64	bitwise and	VAND
T vorrQ_ST(T a, T b)	16	int/64	bitwise or	VORR
T veorQ_ST(T a, T b)	16	int/64	bitwise exclusive-or	VEOR
T vbicQ_ST(T a, T b)	16	int/64	bit clear	VBIC
T vornQ_ST(T a, T b)	16	int/64	bitwise or complement	VORN
T vbslQ_ST(UT a, T b, T c)	24	any	bitwise select	VBSL

12.3.9 NEON vector reductions

```
T vpadd_ST(T a, T b);
```

performs a pairwise add operation. For example, given the two input vectors ABCD and EFGH, the result is {A+B,C+D,E+F,G+H}. This is a 64-bit vector operation only. The lane type of T can be any 8-bit, 16-bit or 32-bit integer type or `float32_t`.

```
RT vpaddlQ_ST(T a);
RT vpadalQ_ST(RT a, T b);
```

adds elements pairwise in the input vector, with a long result. The input elements can be 8-bit, 16-bit or 32-bit integers. The result vector type RT is the same size as the input vector, with half as many lanes, each of twice the size. For example, given an `int16x4_t` input vector {A,B,C,D}, the output vector is the `int32x2_t` vector {A+B,C+D}. The `vpadal()` form accumulates the result with another vector.

```
T vpmax_ST(T a, T b);
T vpmin_ST(T a, T b);
```

performs pairwise maximum or minimum on a pair of 64-bit input vectors. The input elements can be 8-bit, 16-bit or 32-bit integers, or `float32_t`. Given inputs {A,B,C,D} and {E,F,G,H}, the output vector (for `vpmax`) is {max(A,B),max(C,D),max(E,F),max(G,H)}.

12.3.10 NEON vector rearrangements

Like loads and stores, the intrinsics which rearrange vectors are defined for all relevant lane data types, but are implemented by the same generic instructions.

```
T vextQ_ST(T a, T b, const int c); where 0 <= c <= (N - 1)
```

extracts one vector from a pair of concatenated input vectors, starting at a given lane position. Given inputs ABCD and EFGH, and a lane position of 3, the concatenation EFGHABCD is formed and a vector is extracted at lane 3 to produce a result of FGHA.

```
T vrevBQ_ST(T vec);
```

reverses the order of lanes within B-bit sets. For example, `vrev32_s8` reverses the order of 8-bit lanes within 32-bit groups of four lanes in an `int8x8_t` vector, so that the input ABCDEFGH would result in DCBAHGFE. (At the machine level, this can also be understood as a SIMD operation on 32-bit elements, reversing the byte order in each, but to use the `vrev` intrinsic with `int32x2_t` vectors it would be necessary to reinterpret the input and output vector types.) B must be greater than the lane size: i.e. for 8-bit lanes B must be 16, 32 or 64; for 16-bit lanes B must be 32 or 64; and for 32-bit lanes B must be 64.

```
Tx2 vzipQ_ST(T a, T b);
```

interleaves elements pairwise from two vectors, returning a pair (i.e. a 2-element array) of vectors. The inputs ABCD and EFGH result in AEBF and CGDH. Not available for 64-bit lanes.

```
Tx2 vuzpQ_ST(T a, T b);
```

de-interleaves elements from two vectors. The inputs ABCD and EFGH result in ACEG and BDFH. Not available for 64-bit lanes.

```
Tx2 vtrnQ_ST(T a, T b);
```

transposes elements from two vectors, treating them as 2x2 matrices. Not available for 64-bit lanes.

12.3.11 NEON vector table lookup

```
T vtblN_ST(TxN a, UT b);
```

performs 8-bit table lookup. T must be a 64-bit vector type with 8-bit lanes, i.e. `int8x8_t`, `uint8x8_t` or `poly8x8_t`. The table is supplied in `a` as an array of 1 to 4 vectors, treated as one large vector consisting of (respectively) 8 to 32 table entries. The output is formed by using the vector b as a vector of 8 indexes into the table, and mapping each index by its table entry, or zero if the index is out of range. This operation can be thought of as either

- a lane-by-lane table-lookup operation on `b`, where the index value in each lane of `b` is replaced by the corresponding value from the table in `a`
- or as a general permutation/selection operation on data in `a`, where the data is rearranged, selected or duplicated according to the steering information in the array `b`.

```
T vtbxN_ST(T a, TxN b, UT c);
```

performs an extended table lookup operation. In contrast to `vtbl`, for `vtbx`, if the index is out of range, the resulting lane value is taken from the corresponding lane in the vector `a`, rather than zero.

13 FUTURE DIRECTIONS

13.1 Extensions under consideration

13.1.1 Procedure calls and the Q / GE bits

The ARM procedure call standard [AAPCS] says that the Q and GE bits are undefined across public interfaces, but in practice it is desirable to return saturation status from functions. There are at least two common use cases:

- to define small (inline) functions defined in terms of expressions involving intrinsics, which provide abstractions or emulate other intrinsic families; it is desirable for such functions to have the same well-defined effects on the Q/GE bits as the corresponding intrinsics
- DSP library functions

Options being considered are to define an extension to the “pcs” attribute to indicate that Q is meaningful on the return, and possibly also to infer this in the case of functions marked as inline.

13.1.2 Returning a value in registers

As a type attribute this would allow things like

```
struct __attribute__((value_in_regs)) Point { int x[2]; };
```

This would indicate that the result registers should be used as if the type had been passed as the first argument. The implementation should not complain if the attribute is applied inappropriately (i.e. where insufficient registers are available) – it might be a template instance.

13.1.3 Custom calling conventions

Some interfaces may use calling conventions that depart from the AAPCS. Examples include:

- using additional argument registers, e.g. passing an argument in R5, R7, R12 etc.
- using additional result registers, e.g. R0 and R1 for a combined divide-and-remainder routine (note that some implementations may be able to support this by means of a “value in registers” structure return)
- returning results in the condition flags
- preserving and possibly setting the Q (saturation) bit

13.1.4 Traps: system calls, breakpoints etc.

This release of ACLE does not define how to invoke a SVC (supervisor call), BKPT (breakpoint) etc.

One option would be to mark a function prototype with an attribute, e.g.

```
int __attribute__((svc(0xAB))) system_call(int code, void const *params);
```

When calling the function, arguments and results would be marshalled according to the AAPCS, the only difference being that the call would be invoked as a trap instruction rather than a branch-and-link.

One issue is that some calls may have non-standard calling conventions. (For example, ARM Linux system calls expect the code number to be passed in R7.)

Another issue is that the code may vary between ARM and Thumb state. This issue could be addressed by allowing two numeric parameters in the attribute.

13.1.5 Mixed-endian data

Extensions for accessing data in different endianness have been considered. However, this is not an issue specific to the ARM architecture, and it seems better to wait for a lead from language standards.

13.2 Features not considered for support

13.2.1 Load/store exclusive

In principle it would be possible to provide the following intrinsics to directly access the load/store exclusive instructions. These intrinsics would allow implementation of general atomic operations. However, this is problematic because of the need for a compiler not to interpolate other memory accesses (e.g. spills) in between the load and the store. Therefore it is not currently proposed to standardize these intrinsics.

```
uint32_t __ldrex(T const *)
```

returns the contents of the given address, and acquires exclusive access to the address.

```
int __strex(uint32_t, T *)
```

stores a new value at the given address, subject to holding exclusive access to the address. It returns 0 if the store succeeds and 1 if the store fails.

```
int64_t __ldrexd(int64_t const *)
```

returns the doubleword contents of the given address, and acquires exclusive access.

```
int __strexd(uint64_t, T *)
```

stores a doubleword value at the given address, subject to holding exclusive access.

```
void __clrex(void)
```

clears the exclusive access (monitor) bit.

Example use:

```
void atomic_inc(int *p) { while (__strex(__ldrex(p) + 1), p); }
```

Memory operations occurring between a load-exclusive and a store-exclusive may cause the monitor bit to be reset, leading to livelock. It is recommended that compilers and other tools avoid and/or diagnose this situation.

13.2.2 VFP vector mode

The “short vector” mode of the original VFP architecture is now deprecated, and unsupported in recent implementations of the ARM floating-point instructions set. There is no plan to support it through C extensions.

13.2.3 Bit-banded memory access

The bit-banded memory feature of certain Cortex-M cores is now regarded as being outside the architecture, and there is no plan to standardize its support.