ACLE Documentation

Release ACLE Q2 2017

ARM Ltd.

Jan 25, 2023

CONTENTS

Contents:

CHAPTER

ONE

PREFACE

1.1 ARM C Language Extensions

Document number: 101028

Date of Issue: 02/06/2017

1.2 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.

1.3 Keywords

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

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

Please check the ARM Information Center or ARM Developer Website for a later release if your copy is more than one year old. This document may be found under Developer Guides and ArticlesSoftware Development.

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

1.5 Confidentiality status

This document is Non-Confidential.

1.5.1 Proprietary Notice

1.6 About this document

1.6.1 Change control

1.6.1.1 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.
C	09/05/14	TB	Version 2.0. Updated for ARMv8 AArch32 and AArch64.
D	24/03/16	TB	Version 2.1. Updated for ARMv8.1 AArch32 and AArch64.
E	02/06/17	ARM	Version ACLE Q2 2017. Updated for ARMv8.2-A and ARMv8.3-A.

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

1.6.1.2 Changes between ACLE 2.1 and ACLE Q2 2017

Most changes in ACLE Q2 2017 are updates to support features introduced in ARMv8.2-A [ARMARMv82] and ARMv8.3-A [ARMARMv83].

1.6.1.3 General changes

[ACLE-134] - Add missing float16 intrinsics: vdup_lane, vmov_n, vzip, vtrn, etc.

[ACLE-135] - ACLE operator semantics for __fp16 wrt. ARMv8.2-A half-precision instructions.

[ACLE-138] - Support fp16 Intrinsics for all the new fp16 instructions in v8.2.

[ACLE-145] - Remove the terse Neon Intrinsics table.

[ACLE-146] - Add intrinsic support for co-processor instructions to ACLE.

[ACLE-149] - Correct statement for __ARM_ARCH_PROFILE for __ARM_ARCH > 7.

[ACLE-150] - __revsh intrinsic description does not match signature.

[ACLE-151] - Intrinsic for ARMv8.3 floating-point conversion instruction.

1.6.2 References

This document refers to the following documents.

1.6.3 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.
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.

1.7 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
- 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

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.

CHAPTER

TWO

INTRODUCTION

The ARM architecture includes features that go beyond the set of operations available to C/C++ programmers. 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 advanced features of the ARM architecture.

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.

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.

2.1 Portable Binary Objects

In AArch32, the *ABI for the ARM Architecture* defines a set of build attributes [BA]. These attributes are intended to facilitate generating cross-platform portable binary object files by providing a mechanism to determine the compatibility of object files. In AArch64, the ABI does not define a standard set of build attributes and takes the approach that binaries are, in general, not portable across platforms. References to build attributes in this document should be interpreted as applying only to AArch32.

CHAPTER

THREE

C LANGUAGE EXTENSIONS

3.1 Data types

This section overlaps with the specification of the ARM Procedure Call Standard, particularly [AAPCS] (4.1). ACLE 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] and [AAPCS64] (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).

ACLE extends C by providing some types not present in Standard C and defining how they are dealt with by the AAPCS.

- Vector types for use with the Advanced SIMD intrinsics (see *Vector data types*).
- The `__fp16` type for 16-bit floating-point values (see *Half-precision floating-point*).

3.1.1 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 but allow the default to be overridden.

The signedness of a plain int bit-field is implementation-defined.

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 }`. An implementation that conforms to the ARM ABI must reflect its choice in the `Tag_ABI_enum_size build` attribute.

`wchar_t` may be 2 or 4 bytes. The predefined macro `__ARM_SIZEOF_WCHAR_T` should be defined as the same number. An implementation that conforms to the ARM ABI must reflect its choice in the `Tag_ABI_PCS_wchar_t` build attribute.

3.2 Predefined macros

Several predefined macros are defined. Generally these define features of the ARM architecture being targeted, or how the C/C++ implementation uses the architecture. These macros are detailed in *Feature test macros*. All ACLE predefined macros start with the prefix `__ARM_`.

3.3 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 *Advanced SIMD (NEON) intrinsics*. 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))
```

3.3.1 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.

3.4 Header files

<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 */
```

<arm_fp16.h> is provided to define the scalar 16-bit floating point arithmetic intrinsics. As these intrinsics are in the user namespace, an implementation would not normally define them until the header is included. The __ARM_FEATURE_FP16_SCALAR_ARITHMETIC feature macro should be tested before including the header:

```
#ifdef __ARM_FEATURE_FP16_SCALAR_ARITHMETIC
#include <arm_fp16.h>
#endif /* __ARM_FEATURE_FP16_SCALAR_ARITHMETIC */
```

Including <arm_neon.h> will also cause <arm_fp16.h> to be included if appropriate.

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

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 are recommended to include the standard headers explicitly if the associated types and macros are needed.

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
```

3.5 Attributes

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

3.6 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.

3.6.1 Half-precision floating-point

ACLE defines the __fp16 type, which can be used for half-precision (16-bit) floating-point in one of two formats. The binary16 format defined in [IEEE-FP], and referred to as *IEEE* format, and an alternative format, defined by ARM, which extends the range by removing support for infinities and NaNs, referred to as *alternative* format. Both formats are described in [ARMARM] (A2.7.4), [ARMARMv8] (A1.4.2).

Toolchains are not required to support the alternative format, and use of the alternative format precludes use of the ISO/IEC TS 18661:3 [CFP15] _Float16 type and the ARMv8.2-A 16-bit floating-point extensions. For these reasons, ARM deprecates the use of the alternative format for half precision in ACLE.

The format in use can be selected at runtime but ACLE assumes it is fixed for the life of a program. If the __fp16 type 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.

The __fp16 type can be used in two ways; using the intrinsics ACLE defines when the ARMv8.2-A 16-bit floating point extensions are available, and using the standard C operators. When using standard C operators, 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 support for arithmetic directly on __fp16 values using standard C operators.

```
void add(__fp16 a, __fp16 b) {
  a + b; /* a and b are promoted to (at least) float.
            Operation takes place with (at least) 32-bit precision.  */
  vaddh_f16 (a, b); /* a and b are not promoted.
                       Operation takes place with 16-bit precision.  */
}
```

ARMv8 introduces floating point instructions to convert 64-bit to 16-bit i.e. from double to __fp16. They are not available in earlier architectures, therefore have to rely on emulation libraries or a sequence of instructions to achieve the conversion.

Providing emulation libraries for half-precision floating point conversions when not implemented in hardware is implementation-defined.

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

rather than:

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

In some older implementations, __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. The predefined macro __ARM_FP16_ARGS should be defined if __fp16 can be used as an argument and result. 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];
 }
```

3.6.2 Relationship between __fp16 and ISO/IEC TS 18661

ISO/IEC TS 18661-3 [CFP15] is a published extension to [C11] which describes a language binding for the [IEEE-FP] standard for floating point arithmetic. This language binding introduces a mapping to an unlimited number of interchange and extended floating-point types, on which binary arithmetic is well defined. These types are of the form _FloatN, where N gives size in bits of the type.

One instantiation of the interchange types introduced by [CFP15] is the _Float16 type. ACLE defines the __fp16 type as a storage and interchange only format, on which arithmetic operations are defined to first promote to a type with at least the range and precision of float.

This has implications for the result of operations which would result in rounding had the operation taken place in a native 16-bit type. As software may rely on this behaviour for correctness, arithmetic operations on __fp16 are defined to promote even when the ARMv8.2-A fp16 extension is available.

ARM recommends that portable software is written to use the _Float16 type defined in [CFP15].

Type conversion between a value of type __fp16 and a value of type _Float16 leaves the object representation of the converted value unchanged.

When __ARM_FP16_FORMAT_IEEE == 1, this has no effect on the value of the object. However, as the representation of certain values has a different meaning when using the ARM alternative format for 16-bit floating point values [ARMARM] (A2.7.4) [ARMARMv8] (A1.4.2), when __ARM_FP16_FORMAT_ALTERNATIVE == 1 the type conversion may introduce or remove infinity or NaN representations.

ARM recommends that software implementations warn on type conversions between __fp16 and _Float16 if __ARM_FP16_FORMAT_ALTERNATIVE == 1.

In an arithmetic operation where one operand is of __fp16 type and the other is of _Float16 type, the _Float16 type is first converted to __fp16 type following the rules above, and then the operation is completed as if both operands were of __fp16 type.

[CFP15] and [C11] do not define vector types, however many C implementations do provide these extensions. Where they exist, type conversion between a value of type vector of __fp16 and a value of type vector of _Float16 leaves the object representation of the converted value unchanged.

ACLE does not define vector of _Float16 types.

CHAPTER

FOUR

ARCHITECTURE AND CPU NAMES

4.1 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.)

4.2 Architecture names

4.2.1 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 *ARM/Thumb instruction set architecture*.

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

Name	Features	ARM	Thumb	Example processor
ARMv4	ARMv4	4		DEC/Intel StrongARM
ARMv4T	ARMv4 with Thumb instruction set	4	2	ARM7TDMI
ARMv5T	ARMv5 with Thumb instruction set	5	2	ARM10TDMI
ARMv5TE	ARMv5T with DSP extensions	5	2	ARM9E, Intel XScale
ARMv5TEJ	ARMv5TE with Jazelle® extensions	5	2	ARM926EJ
ARMv6	ARMv6 (includes TEJ)	6	2	ARM1136J r0
ARMv6K	ARMv6 with kernel extensions	6	2	ARM1136J r1
ARMv6T2	ARMv6 with Thumb-2 architecture	6	3	ARM1156T2
ARMv6Z	ARMv6K with Security Extensions (includes K)	6	2	ARM1176JZ-S
ARMv6-M	Thumb-1 only (M-profile)		2	Cortex-M0, Cortex-M1
ARMv7-A	ARMv7 application profile	7	4	Cortex-A8, Cortex-A9
ARMv7-R	ARMv7 realtime profile	7	4	Cortex-R4
ARMv7-M	ARMv7 microcontroller profile: Thumb-2 instructions only		4	Cortex-M3
ARMv7E-M	ARMv7-M with DSP extensions		4	Cortex-M4
ARMv8-A AArch32	ARMv8 application profile	8	4	Cortex-A57, Cortex-A53
ARMv8-A AArch64	ARMv8 application profile	8		Cortex-A57, Cortex-A53

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 divide might or might not be present in the ARMv7-A architecture).

4.2.2 FPU architecture

For details of how to test FPU features in source code, see *Floating-point and Advanced SIMD (NEON) hardware*. In particular, for testing which precisions are supported in hardware, see *_ssec-HWFP*.

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

4.3 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.

CHAPTER

FIVE

FEATURE TEST MACROS

5.1 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.

5.2 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 2.1 of the ACLE specification will define `__ARM_ACLE` as 201.

5.3 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.)

5.4 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 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.

5.4.1 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 ARMv7-M architecture implemented by Cortex-M3 or 8 for the ARMv8-A architecture implemented by Cortex-A57). ARMv8.1 [ARMARMv81] onwards, the value of `__ARM_ARCH` is scaled up to include minor versions. The formula to calculate the value of `__ARM_ARCH` from ARMv8.1 [ARMARMv81] onwards is given by the following formula:

```
For an ARM architecture ARMvX.Y, __ARM_ARCH = X * 100 + Y. E.g.
for ARMv8.1 __ARM_ARCH = 801.
```

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_ARCH_ISA_A64` is defined to 1 if the core supports AArch64's A64 instruction set.

`__ARM_32BIT_STATE` is defined to 1 if code is being generated for AArch32.

`__ARM_64BIT_STATE` is defined to 1 if code is being generated for AArch64.

5.4.2 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. The common subset of A, R and M is indicated by:

```
__ARM_ARCH == 7 && !defined (__ARM_ARCH_PROFILE)
```

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.

5.4.3 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.

5.4.4 LDREX/STREX

This feature was deprecated in ACLE 2.0. It is strongly recommended that C11/C++11 atomics be used instead.

__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	ARMv5, ARMv6-M
0x04	word	ARMv6
0x07	word, halfword, byte	ARMv7-M
0x0F	doubleword, word, halfword, byte	ARMv6K, ARMv7-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.

5.4.5 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 *Miscellaneous data-processing intrinsics*) even when __ARM_FEATURE_CLZ is not defined.

5.4.6 Q (saturation) flag

__ARM_FEATURE_QBIT is defined to 1 if the Q (saturation) global flag exists and the intrinsics defined in *The Q (saturation) flag* 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.

Intrinsics associated with the Q-bit and their feature macro __ARM_FEATURE_QBIT are deprecated in ACLE 2.0 for A-profile. They are fully supported for M-profile and R-profile. This macro is defined for AArch32 only.

5.4.7 DSP instructions

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

__ARM_FEATURE_DSP and its associated intrinsics are deprecated in ACLE 2.0 for A-profile. They are fully supported for M and R-profiles. This macro is defined for AArch32 only.

5.4.8 Saturation instructions

__ARM_FEATURE_SAT is defined to 1 if the SSAT and USAT instructions are supported and the intrinsics defined in *Width-specified saturation intrinsics* are available. This feature also implies support for the Q flag.

__ARM_FEATURE_SAT and its associated intrinsics are deprecated in ACLE 2.0 for A-profile. They are fully supported for M and R-profiles. This macro is defined for AArch32 only.

5.4.9 32-bit SIMD instructions

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

__ARM_FEATURE_SIMD32 is deprecated in ACLE 2.0 for A-profile. Users are encouraged to use NEON Intrinscs as an equivalent for the 32-bit SIMD intrinsics functionality. However they are fully supported for M and R-profiles. This is defined for AArch32 only.

5.4.10 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)
```

5.5 Floating-point and Advanced SIMD (NEON) hardware

5.5.1 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 *Half-precision (16-bit) floating-point format*) is only required to be available if supported in hardware.

5.5.2 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 *Half-precision floating-point* for details of half-precision floating-point types.

5.5.3 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.

5.5.4 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.

In principle, for AArch32, 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.

__ARM_NEON is always set to 1 for AArch64.

5.5.5 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 AArch32 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 for AArch32. Double-precision is always set for AArch64.

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

5.5.6 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.

5.5.7 Crypto Extension

__ARM_FEATURE_CRYPTO is defined to 1 if the Crypto instructions are supported and intrinsics targeting them are available. These instructions include AES{E, D}, SHA1{C, P, M} etc. This is only available when __ARM_ARCH >= 8.

5.5.8 CRC32 Extension

__ARM_FEATURE_CRC32 is defined to 1 if the CRC32 instructions are supported and the intrinsics defined in *CRC32 intrinsics* are available. These instructions include CRC32B, CRC32H etc. This is only available when __ARM_ARCH >= 8.

5.5.9 Directed Rounding

__ARM_FEATURE_DIRECTED_ROUNDING is defined to 1 if the directed rounding and conversion vector instructions are supported and rounding and conversion intrinsics are available. This is only available when __ARM_ARCH >= 8.

5.5.10 Numeric Maximum and Minimum

__ARM_FEATURE_NUMERIC_MAXMIN is defined to 1 if the IEEE 754-2008 compliant floating point maximum and minimum vector instructions are supported and intrinsics targeting these instructions are available. This is only available when __ARM_ARCH >= 8.

5.5.11 Half-precision argument and result

__ARM_FP16_ARGS is defined to 1 if __fp16 can be used as an argument and result.

5.5.12 Rounding Doubling Multiplies

__ARM_FEATURE_QRDMX is defined to 1 if SQRDMLAH and SQRDMLSH instructions and their associated intrinsics are available.

5.5.13 16-bit floating-point data processing operations

__ARM_FEATURE_FP16_SCALAR_ARITHMETIC is defined to 1 if the 16-bit floating-point arithmetic instructions are supported in hardware and the associated scalar intrinsics defined by ACLE are available. Note that this implies:

- __ARM_FP16_FORMAT_IEEE == 1
- __ARM_FP16_FORMAT_ALTERNATIVE == 0
- __ARM_FP & 0x02 == 1

__ARM_FEATURE_FP16_VECTOR_ARITHMETIC is defined to 1 if the 16-bit floating-point arithmetic instructions are supported in hardware and the associated vector intrinsics defined by ACLE are available. Note that this implies:

- __ARM_FP16_FORMAT_IEEE == 1
- __ARM_FP16_FORMAT_ALTERNATIVE == 0
- __ARM_FP & 0x02 == 1
- __ARM_NEON_FP & 0x02 == 1

5.5.14 Javascript floating-point conversion

__ARM_FEATURE_JCVT is defined to 1 if the FJCVTZS (AArch64) or VJCVT (AArch32) instruction and the associated intrinsic is available.

5.6 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.

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

__STDC_IEC_559__ is defined if the implementation conforms to IEC 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.)

5.7 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]. This is supported on AArch32 only.

__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]. This is supported on AArch32 only.

__ARM_PCS_AAPCS64 is defined to 1 if the default procedure calling standard for the translation unit conforms to the [AAPCS64].

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.

5.8 Position-independent code

__ARM_ROPI is defined to 1 if the translation unit is being compiled in read-only position independent mode. In this mode, all read-only data and functions are at a link-time constant offset from the program counter.

__ARM_RWPI is defined to 1 if the translation unit is being compiled in read-write position independent mode. In this mode, all writable data is at a link-time constant offset from the static base register defined in [AAPCS].

The ROPI and RWPI position independence modes are compatible with each other, so the __ARM_ROPI and __ARM_RWPI macros may be defined at the same time.

5.9 Coprocessor Intrinsics

__ARM_FEATURE_COPROC is defined as a bitmap to indicate the presence of coprocessor intrinsics for the target architecture. If __ARM_FEATURE_COPROC is undefined or zero, that means there is no support for coprocessor intrinsics on the target architecture. The following bits are used:

Bit	Value	Intrinsics Available
0	0x1	__arm_cdp __arm_ldc, __arm_ldcl, __arm_stc, __arm_stcl, __arm_mcr and __arm_mrc
1	0x2	__arm_cdp2, __arm_ldc2, __arm_stc2, __arm_ldc2l, __arm_stc2l, __arm_mcr2 and __arm_mrc2
2	0x4	__arm_mcrr and __arm_mrrc
3	0x8	__arm_mcrr2 and __arm_mrrc2

5.10 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_PROFILE
8	Tag_ARM_ISA_use	__ARM_ISA_ARM
9	Tag_THUMB_ISA_use	__ARM_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_for	__ARM_FP16_FORMAT_IEEE __ARM_FP16_FORMAT_ALTERNATIVE

5.11 Summary of predefined macros

Macro name	Meaning	Example	See section
__ARM_32BIT_STATE	Code is for AArch32 state	1	*ARM/Thumb*
__ARM_64BIT_STATE	Code is for AArch64 state	1	*ARM/Thumb*
__ARM_ACLE	Indicates ACLE implemented	101	*Testing for A*
__ARM_ALIGN_MAX_PWR	Log of maximum alignment of static object	20	*Alignment o*
__ARM_ALIGN_MAX_STACK_PWR	Log of maximum alignment of stack object	3	*Alignment o*
__ARM_ARCH	ARM architecture level	7	*ARM/Thumb*
__ARM_ARCH_ISA_A64	AArch64 ISA present	1	*ARM/Thumb*

Table 1 – continued from previous page

__ARM_ARCH_ISA_ARM	ARM instruction set present	1	*ARM/Thumb*
__ARM_ARCH_ISA_THUMB	Thumb instruction set present	2	*ARM/Thumb*
__ARM_ARCH_PROFILE	Architecture profile	A	*Architectura*
__ARM_BIG_ENDIAN	Memory is big-endian	1	*Endianness*
__ARM_FEATURE_CLZ	CLZ instruction	1	*CLZ*, *Misce*
__ARM_FEATURE_CRC32	CRC32 extension	1	*CRC32 Exte*
__ARM_FEATURE_CRYPTO	Crypto extension	1	*Crypto Exte*
__ARM_FEATURE_DIRECTED_ROUNDING	Directed Rounding	1	*Directed Ro*
__ARM_FEATURE_DSP	DSP instructions (ARM v5E) (32-bit-only)	1	*DSP instruc*
__ARM_FEATURE_FMA	Floating-point fused multiply-accumulate	1	*Fused multi*
__ARM_FEATURE_IDIV	Hardware Integer Divide	1	*Hardware In*
__ARM_FEATURE_JCVT	Javascript conversion (ARMv8.3)	1	*Javascript fl*
__ARM_FEATURE_LDREX *(Deprecated)*	Load/store exclusive instructions	0x0F	*LDREX/STR*
__ARM_FEATURE_NUMERIC_MAXMIN	Numeric Maximum and Minimum	1	*Numeric Ma*
__ARM_FEATURE_QBIT	Q (saturation) flag (32-bit-only)	1	*Q (saturatio*
__ARM_FEATURE_QRDMX	SQRDMLxH instructions and associated intrinsics availability	1	*Rounding D*
__ARM_FEATURE_SAT	Width-specified saturation instructions (32-bit-only)	1	*Saturation i*
__ARM_FEATURE_SIMD32	32-bit SIMD instructions (ARMv6) (32-bit-only)	1	*Saturation i*
__ARM_FEATURE_UNALIGNED	Hardware support for unaligned access	1	*Unaligned a*
__ARM_FP	Hardware floating-point	0x0C	*Hardware fl*
__ARM_FP16_ARGS	`__fp16` argument and result	1	*Half-precisi*
__ARM_FP16_FORMAT_ALTERNATIVE	16-bit floating-point, alternative format	1	*Half-precisi*
__ARM_FP16_FORMAT_IEEE	16-bit floating-point, IEEE format	1	*Half-precisi*
__ARM_FP_FAST	Accuracy-losing optimizations	1	*Floating-po*
__ARM_FP_FENV_ROUNDING	Rounding is configurable at runtime	1	*Floating-po*
__ARM_NEON	Advanced SIMD (NEON) extension	1	*NEON floati*
__ARM_NEON_FP	Advanced SIMD (NEON) floating-point	0x04	*Wireless MM*
__ARM_FEATURE_COPROC	Coprocessor Intrinsics	0x01	*Coprocesso*
__ARM_PCS	ARM procedure call standard (32-bit-only)	1	*Procedure c*
__ARM_PCS_AAPCS64	ARM PCS for AArch64.	1	*Procedure c*
__ARM_PCS_VFP	ARM PCS hardware FP variant in use (32-bit-only)	1	*Procedure c*
__ARM_ROPI	Read-only PIC in use	1	*Position-ind*
__ARM_RWPI	Read-write PIC in use	1	*Position-ind*
__ARM_SIZEOF_MINIMAL_ENUM	Size of minimal enumeration type: 1 or 4	1	*Implementa*
__ARM_SIZEOF_WCHAR_T	Size of `wchar_t`: 2 or 4	2	*Implementa*
__ARM_WMMX	Wireless MMX extension (32-bit-only)	1	*Wireless MM*

CHAPTER

SIX

ATTRIBUTES AND PRAGMAS

6.1 Attribute syntax

The general rules for attribute syntax are described in the GCC documentation <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.

6.2 Hardware/software floating-point calling convention

The AArch32 PCS defines a base standard, as well as several variants.

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 AArch64 PCS standard variants do not change how parameters are passed, so no PCS attributes are supported.

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++ #1].

6.3 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 ARMv5 or ARMv6 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.

This attribute does not apply to AArch64.

6.4 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.

6.4.1 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.

6.5 Alignment

The new standards for C [C11] (6.7.5) and C++ [CPP11] (7.6.2) add syntax for aligning objects and types. ACLE provides an alternative syntax described in this section.

6.5.1 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 */
```

6.5.2 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.

6.5.3 Alignment of stack objects

It must be possible to align any local object up to the stack alignment as specified in the AAPCS for AArch32 (i.e. 8 bytes) or as specified in AAPCS64 for AArch64 (i.e. 16 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 for AArch32. (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.

6.5.4 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);
```

which means that in AArch32 AAPCS the second parameter is in `R2/R3` rather than `R1/R2`.

6.5.5 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).

6.5.6 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 AArch32 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;
  }
```

(continues on next page)

(continued from previous page)

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

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

The generic itanium C++ ABI, which we use in AArch64, already handles arrays with arbitrarily aligned elements

6.6 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 *Weak linkage*.

CHAPTER

SEVEN

SYNCHRONIZATION, BARRIER, AND HINT INTRINSICS

7.1 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.

7.2 Atomic update primitives

7.2.1 C/C++ standard atomic primitives

The new C and C++ standards [C11] (7.17), [CPP11] (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.

7.2.2 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.

7.3 Memory barriers

Memory barriers ensure specific ordering properties between memory accesses. For more details on memory barriers, see [ARMARM] (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 *Constant arguments to intrinsics*.

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.

7.3.1 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 */
```

7.4 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.

```
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. This is available only when compliling for AArch32.

7.5 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 0, 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 */
```

(continues on next page)

(continued from previous page)

```
  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.)

7.6 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).

7.6.1 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.

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

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

7.6.2 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.

```
void __plix(/*constant*/ unsigned int /*cache_level*/,
            /*constant*/ unsigned int /*retention_policy*/,
            T addr);
```

Generates a code prefetch instruction. This intrinsic allows the specification of the cache level to load the code, the retention policy (temporal or streaming). The relevant arguments can have the same values as in `__pldx.`

`__pldx` and `__plix` arguments cache level and retention policy are ignored on unsupported targets.

7.7 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.

CHAPTER

EIGHT

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 deprecated. However, the miscellaneous intrinsics and CRC32 intrinsics described in *Miscellaneous data-processing intrinsics* and *CRC32 intrinsics* respectively are suitable for all profiles.

8.1 Programmer's model of global state

8.1.1 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 interfac. 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.)

8.1.2 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 *32-bit SIMD intrinsics*.

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, see *Parallel selection*.

8.1.3 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.

8.2 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 *NOP*]

The 64-bit versions of these intrinsics (ll suffix) are new in ACLE 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 - 1, i.e. 31 or 63. 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 signed 16-bit result. For example, 0x0080 becomes 0x8000. 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.

8.2.1 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_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
}
```

8.3 16-bit multiplications

The intrinsics in this section provide direct access to the 16x16 and 16x32 bit multiplies introduced in ARMv5E. 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.

8.4 Saturating intrinsics

8.4.1 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 *Constant arguments to intrinsics*.

```
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.

8.4.2 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.

8.4.3 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.

8.4.4 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.

8.5 32-bit SIMD intrinsics

8.5.1 Availability

ARMv6 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 *Advanced SIMD (NEON) intrinsics*.

The 32-bit SIMD intrinsics are available on targets featuring ARMv6 and upwards, including the A and R profiles. In the M profile they are available in the ARMv7E-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.

8.5.2 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.

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

Some 32-bit SIMD instructions may set the Q flag described in *The Q (saturation) flag*. 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.

8.5.4 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 *Width-specified saturation intrinsics*.

```
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.

8.5.5 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.

8.5.6 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.

8.5.7 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.

8.5.8 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.

8.5.9 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.

8.5.10 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.

8.5.11 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 :ref:sec-Parallel-selection, 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.

8.6 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` (see *Hardware floating point*).

```
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 [ARMARM] (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.

```
float __rintnf (float);
double __rintn (double);
```

The __rintn intrinsics perform a floating point round to integral, to nearest with ties to even. The __rintn intrinsic is available when __ARM_FEATURE_DIRECTED_ROUNDING is defined to 1. For other rounding modes like 'to nearest with ties to away' it is strongly recommended that C99 standard functions be used. To achieve a floating point convert to integer, rounding to 'nearest with ties to even' operation, use these rounding functions with a type-cast to integral values, eg.

```
(int) __rintnf (a);
```

Will map to a floating point convert to signed integer, rounding to nearest with ties to even operation.

```
int32_t __jcvt (double);
```

Converts a double-precision floating-point number to a 32-bit signed integer following the Javascript Convert instruction semantics [ARMARMv83]. The __jcvt intrinsic is available if __ARM_FEATURE_JCVT is defined.

8.7 CRC32 intrinsics

CRC32 intrinsics provide direct access to CRC32 instructions CRC32{C}{B, H, W, X} in both ARMv8 AArch32 and AArch64 execution states. These intrinsics are available when __ARM_FEATURE_CRC32 is defined.

```
uint32_t __crc32b (uint32_t a, uint8_t b);
```

Performs CRC-32 checksum from bytes.

```
uint32_t __crc32h (uint32_t a, uint16_t b);
```

Performs CRC-32 checksum from half-words.

```
uint32_t __crc32w (uint32_t a, uint32_t b);
```

Performs CRC-32 checksum from words.

```
uint32_t __crc32d (uint32_t a, uint64_t b);
```

Performs CRC-32 checksum from double words.

```
uint32_t __crc32cb (uint32_t a, uint8_t b);
```

Performs CRC-32C checksum from bytes.

```
uint32_t __crc32ch (uint32_t a, uint16_t b);
```

Performs CRC-32C checksum from half-words.

```
uint32_t __crc32cw (uint32_t a, uint32_t b);
```

Performs CRC-32C checksum from words.

```
uint32_t __crc32cd (uint32_t a, uint64_t b);
```

Performs CRC-32C checksum from double words.

To access these intrinsics, <arm_acle.h> should be included.

CHAPTER

NINE

SYSTEM REGISTER ACCESS

9.1 Special register intrinsics

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

```
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.

9.2 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.

9.2.1 AArch32 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 [ARMARM] 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").

9.2.2 AArch32 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] (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 [ARMARM] (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.

9.2.3 AArch32 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]

9.2.4 AArch64 system register

When specifying a system register to __arm_rsr, __arm_rsr64, __arm_rsrp, __arm_wsr, __arm_wsr64 or __arm_wsrp:

```
"o0:op1:CRn:CRm:op2"
```

Where:

- <o0> is a decimal integer in the range [0, 1]
- <op1>, <op2> are decimal integers in the range [0, 7]
- <CRm>, <CRn> are decimal integers in the range [0, 15]

9.2.5 AArch64 processor state field

When specifying a processor state field to __arm_rsr, __arm_rsp, __arm_wsr, or __arm_wsrp, one of the values accepted in the pstatefield of the MSR (immediate) instruction [ARMARMv8] (C5.6.130).

9.3 Coprocessor Intrinsics

9.3.1 AArch32 coprocessor intrinsics

In the intrinsics below coproc, opc1, opc2, CRn and CRd are all compile time integer constants with appropriate values as defined by the coprocessor for the intended architecture.

The argument order for all intrinsics is the same as the operand order for the instruction as described in the ARMARM, with the exception of MRC/ MRC2/ MRRC/MRRC2 which omit the ARM register arguments and instead returns a value and MCRR/MCRR2 which accepts a single 64 bit unsigned integer instead of two 32-bit unsigned integers.

9.3.2 AArch32 Data-processing coprocessor intrinsics

Intrinsics are provided to create coprocessor data-processing instructions as follows:

Intrinsics	Equivalent Instruction
void __arm_cdp(coproc, opc1, CRd, CRn, CRm, opc2)	CDP coproc, opc1, CRd, CRn, CRm, opc2
void __arm_cdp2(coproc, opc1, CRd, CRn, CRm, opc2)	CDP2 coproc, opc1, CRd, CRn, CRm, opc2

9.3.2.1 AArch32 Memory coprocessor transfer intrinsics

Intrinsics are provided to create coprocessor memory transfer instructions as follows:

Intrinsics	Equivalent Instruction
void __arm_ldc(coproc, CRd, const void* p)	LDC coproc, CRd, [...]
void __arm_ldcl(coproc, CRd, const void* p)	LDCL coproc, CRd, [...]
void __arm_ldc2(coproc, CRd, const void* p)	LDC2 coproc, CRd, [...]
void __arm_ldc2l(coproc, CRd, const void* p)	LDC2L coproc, CRd, [...]
void __arm_stc(coproc, CRd, void* p)	STC coproc, CRd, [...]
void __arm_stcl(coproc, CRd, void* p)	STCL coproc, CRd, [...]
void __arm_stc2(coproc, CRd, void* p)	STC2 coproc, CRd, [...]
void __arm_stc2l(coproc, CRd, void* p)	STC2L coproc, CRd, [...]

9.3.3 AArch32 Integer to coprocessor transfer intrinsics

Intrinsics are provided to map to coprocessor to core register transfer instructions as follows:

Intrinsics	Equivalent Instruction
void __arm_mcr(coproc, opc1, uint32_t value, CRn, CRm, opc2)	MCR coproc, opc1, Rt, CRn, CRm, opc2
void __arm_mcr2(coproc, opc1, uint32_t value, CRn, CRm, opc2)	MCR2 coproc, opc1, Rt, CRn, CRm, opc2
uint32_t __arm_mrc(coproc, opc1, CRn, CRm, opc2)	MRC coproc, opc1, Rt, CRn, CRm, opc2
uint32_t __arm_mrc2(coproc, opc1, CRn, CRm, opc2)	MRC2 coproc, opc1, Rt, CRn, CRm, opc2
void __arm_mcrr(coproc, opc1, uint64_t value, CRm)	MCRR coproc, opc1, Rt, Rt2, CRm
void __arm_mcrr2(coproc, opc1, uint64_t value, CRm)	MCRR2 coproc, opc1, Rt, Rt2, CRm
uint64_t __arm_mrrc(coproc, opc1, CRm)	MRRC coproc, opc1, Rt, Rt2, CRm
uint64_t __arm_mrrc2(coproc, opc1, CRm)	MRRC2 coproc, opc1, Rt, Rt2, CRm

The intrinsics `__arm_mcrr`/`__arm_mcrr2` accept a single unsigned 64-bit integer value instead of two 32-bit integers. The low half of the value goes in register `Rt` and the high half goes in `Rt2`. Likewise for `__arm_mrrc`/`__arm_mrrc2` which return an unsigned 64-bit integer.

9.4 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.
- When writing to a coprocessor register that carries out a "System operation".
- When using a register specifier which doesn't apply to the targetted architecture.

CHAPTER

TEN

INSTRUCTION GENERATION

10.1 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 8 means ARMv8-A AArch32 and AArch64, 8-32 means ARMv8-AArch32 only.

Architecture 7 means ARMv7-A and ARMv7-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.

7MP are the ARMv7 architectures that implement the Multiprocessing Extensions.

Instruction	Flgs	Arch.	Intrinsic or C code
BKPT		5	none
BFC		6T2, 7-M	C
BFI		6T2, 7-M	C
CLZ		5	`__clz, __builtin_clz`
DBG		7, 7-M	`__dbg`
DMB		8,7, 6-M	`__dmb`
DSB		8, 7, 6-M	`__dsb`
ISB		8, 7, 6-M	`__isb`
LDREX		6, 7-M	`__sync_xxx`
LDRT		all	none
MCR/MRC		all	see *System register access*
MSR/MRS		6-M	see *System register access*
PKHBT		6	C
PKHTB		6	C
PLD		8-32,5TE, 7-M	`__pld`
PLDW		7-MP	`__pldx`
PLI		8-32,7	`__pli`
QADD	Q	5E, 7E-M	`__qadd`
QADD16		6, 7E-M	`__qadd16`

continues on next page

Table 1 – continued from previous page

QADD8		6, 7E-M	`__qadd8`
QASX		6, 7E-M	`__qasx`
QDADD	Q	5E, 7E-M	`__qadd(__qdbl)`
QDSUB	Q	5E, 7E-M	`__qsub(__qdbl)`
QSAX		6, 7E-M	`__qsax`
QSUB	Q	5E, 7E-M	`__qsub`
QSUB16		6, 7E-M	`__qsub16`
QSUB8		6, 7E-M	`__qsub8`
RBIT		8,6T2, 7-M	`__rbit`, `__builtin_rbit`
REV		8,6, 6-M	`__rev`, `__builtin_bswap32`
REV16		8,6, 6-M	`__rev16`
REVSH		6, 6-M	`__revsh`
ROR		all	`__ror`
SADD16	GE	6, 7E-M	`__sadd16`
SADD8	GE	6, 7E-M	`__sadd8`
SASX	GE	6, 7E-M	`__sasx`
SBFX		8,6T2, 7-M	C
SDIV		7-M+	C
SEL	(GE)	6, 7E-M	`__sel`
SETEND		6	n/a
SEV		8,6K,6-M,7-M	`__sev`
SHADD16		6, 7E-M	`__shadd16`
SHADD8		6, 7E-M	`__shadd8`
SHASX		6, 7E-M	`__shasx`
SHSAX		6, 7E-M	`__shsax`
SHSUB16		6, 7E-M	`__shsub16`
SHSUB8		6, 7E-M	`__shsub8`
SMC		8,6Z, T2	none
SMI		6Z, T2	none
SMLABB	Q	5E, 7E-M	`__smlabb`
SMLABT	Q	5E, 7E-M	`__smlabt`
SMLAD	Q	6, 7E-M	`__smlad`
SMLADX	Q	6, 7E-M	`__smladx`
SMLAL		all, 7-M	C
SMLALBB		5E, 7E-M	`__smulbb` and C
SMLALBT		5E, 7E-M	`__smulbt` and C
SMLALTB		5E, 7E-M	`__smultb` and C
SMLALTT		5E, 7E-M	`__smultt` and C
SMLALD		6, 7E-M	`__smlald`
SMLALDX		6, 7E-M	`__smlaldx`
SMLATB	Q	5E, 7E-M	`__smlatb`
SMLATT	Q	5E, 7E-M	`__smlatt`
SMLAWB	Q	5E, 7E-M	`__smlawb`
SMLAWT	Q	5E, 7E-M	`__smlawt`
SMLSD	Q	6, 7E-M	`__smlsd`
SMLSDX	Q	6, 7E-M	`__smlsdx`
SMLSLD		6, 7E-M	`__smlsld`
SMLSLDX		6, 7E-M	`__smlsldx`
SMMLA		6, 7E-M	C
SMMLAR		6, 7E-M	C

continues on next page

Table 1 – continued from previous page

SMMLS		6, 7E-M	C
SMMLSR		6, 7E-M	C
SMMUL		6, 7E-M	C
SMMULR		6, 7E-M	C
SMUAD	Q	6, 7E-M	`__smuad`
SMUADX	Q	6, 7E-M	`__smuadx`
SMULBB		5E, 7E-M	`__smulbb`; C
SMULBT		5E, 7E-M	`__smulbt` ; C
SMULTB		5E, 7E-M	`__smultb`; C
SMULTT		5E, 7E-M	`__smultt`; C
SMULL		all, 7-M	C
SMULWB		5E, 7E-M	`__smulwb`; C
SMULWT		5E, 7E-M	`__smulwt`; C
SMUSD		6, 7E-M	`__smusd`
SMUSDX		6, 7E-M	`__smusd`
SSAT	Q	6, 7-M	`__ssat`
SSAT16	Q	6, 7E-M	`__ssat16`
SSAX	GE	6, 7E-M	`__ssax`
SSUB16	GE	6, 7E-M	`__ssub16`
SSUB8	GE	6, 7E-M	`__ssub8`
STREX		6, 7-M	`__sync_xxx`
STRT		all	none
SVC		all	none
SWP		ARM only	`__swp` [deprecated; see *Swap*]
SXTAB		6, 7E-M	`(int8_t)x` + a
SXTAB16		6, 7E-M	`__sxtab16`
SXTAH		6, 7E-M	`(int16_t)x` + a
SXTB		8,6, 6-M	`(int8_t)x`
SXTB16		6, 7E-M	`__sxtb16`
SXTH		8,6, 6-M	`(int16_t)x`
UADD16	GE	6, 7E-M	`__uadd16`
UADD8	GE	6, 7E-M	`__uadd8`
UASX	GE	6, 7E-M	`__uasx`
UBFX		8,6T2, 7-M	C
UDIV		7-M+	C
UHADD16		6, 7E-M	`__uhadd16`
UHADD8		6, 7E-M	`__uhadd8`
UHASX		6, 7E-M	`__uhasx`
UHSAX		6, 7E-M	`__uhsax`
UHSUB16		6, 7E-M	`__uhsub16`
UHSUB8		6, 7E-M	`__uhsub8`
UMAAL		6, 7E-M	C
UMLAL		all, 7-M	`acc += (uint64_t)x * y`
UMULL		all, 7-M	C
UQADD16		6, 7E-M	`__uqadd16`
UQADD8		6, 7E-M	`__uqadd8`
UQASX		6, 7E-M	`__uqasx`
UQSAX		6, 7E-M	`__uqsax`
UQSUB16		6, 7E-M	`__uqsub16`
UQSUB8		6, 7E-M	`__uqsub8`

continues on next page

Table 1 – continued from previous page

USAD8		6, 7E-M	`__usad8`
USADA8		6, 7E-M	`__usad8 + acc`
USAT	Q	6, 7-M	`__usat`
USAT16	Q	6, 7E-M	`__usat16`
USAX		6, 7E-M	`__usax`
USUB16		6, 7E-M	`__usub16`
USUB8		6, 7E-M	`__usub8`
UXTAB		6, 7E-M	`(uint8_t)x + i`
UXTAB16		6, 7E-M	`__uxtab16`
UXTAH		6, 7E-M	`(uint16_t)x + i`
UXTB16		6, 7E-M	`__uxtb16`
UXTH		8,6, 6-M	`(uint16_t)x`
VFMA		VFPv4	`fma`, `__fma`
VSQRT		VFP	`sqrt`, `__sqrt`
WFE		8,6K, 6-M	`__wfe`
WFI		8,6K, 6-M	`__wfi`
YIELD		8,6K, 6-M	`__yield`

CHAPTER

ELEVEN

ADVANCED SIMD (NEON) INTRINSICS

11.1 Introduction

The Advanced SIMD instructions provide packed Single Instruction Multiple Data (SIMD) and single-element scalar operations on a range of integer and floating-point types.

NEON is an implementation of the Advanced SIMD instructions which is provided as an extension for some Cortex A-Series processors. Where this document refers to NEON instructions, such instructions refer to the Advanced SIMD instructions as described by the ARM Architecture Reference Manual [ARMARMv8].

The Advanced SIMD 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 and 64-bit floating-point data, arranged in 64-bit and 128-bit vectors.

The intrinsics in this section provide C and C++ programmers with a simple programming model allowing easy access to code-generation of the Advanced SIMD instructions for both AArch64 and AArch32 execution states.

11.1.1 Concepts

The Advanced SIMD instructions are designed to improve the performance of multimedia and signal processing algorithms by operating on 64-bit or 128-bit *vectors* of *elements* of the same *scalar* data type.

For example, `uint16x4_t` is a 64-bit vector type consisting of four elements of the scalar `uint16_t` data type. Likewise, `uint16x8_t` is a 128-bit vector type consisting of eight `uint16_t` elements.

In a vector programming model, operations are performed in parallel across the elements of the vector. For example, `vmul_u16(a, b)` is a vector intrinsic which takes two `uint16x4_t` vector arguments `a` and `b`, and returns the result of multiplying corresponding elements from each vector together.

The Advanced SIMD extension also provides support for *vector-by-lane* and *vector-by-scalar* operations. In these operations, a scalar value is extracted from one element of a vector input, or provided directly, duplicated to create a new vector with the same number of elements as an input vector, and an operation is performed in parallel between this new vector and other input vectors.

For example, `vmul_lane_u16(a, b, 1)`, is a vector-by-lane intrinsic which takes two `uint16x4_t` vector elements. From `b`, element `1` is extracted, a new vector is formed which consists of four copies of `b`, and this new vector is multiplied by `a`.

Reduction, *cross-lane*, and *pairwise* vector operations work on pairs of elements within a vector, or across the whole of a single vector performing the same operation between elements of that vector. For example, `vaddv_u16(a)` is a reduction intrinsic which takes a `uint16x4_t` vector, adds each of the four `uint16_t` elements together, and returns a `uint16_t` result containing the sum.

11.1.2 Vector data types

Vector data types are named as a lane type and a multiple. Lane type names are based on the types defined in <stdint.h>. For example,. int16x4_t is a vector of four int16_t values. The base types are int8_t, uint8_t, int16_t, uint16_t, int32_t, uint32_t, int64_t, uint64_t, float16_t, float32_t, poly8_t, poly16_t. The multiples are such that the resulting vector types are 64-bit and 128-bit. In AArch64, float64_t, poly64_t and poly128_t are also base types.

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.

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, or whether poly64x1_t is the same as poly64_t e.g. for C++ overloading purposes.

float16 types are only available when the __fp16 type is defined, i.e. when supported by the hardware.

11.1.3 Advanced SIMD Scalar data types

AArch64 supports Advanced SIMD scalar operations that work on standard scalar data types viz. int8_t, uint8_t, int16_t, uint16_t, int32_t, uint32_t, int64_t, uint64_t, float32_t, float64_t.

11.1.4 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 of two 64-bit vector types is distinct from the 128-bit vector type int16x8_t.

11.1.5 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.

11.1.6 16-bit floating-point arithmetic scalar intrinsics

The architecture extensions introduced by ARMv8.2 [ARMARMv82] provide a set of data processing instructions which operate on 16-bit floating-point quantities. These instructions are available in both AArch64 and AArch32 execution states, for both Advanced SIMD and scalar floating-point values.

ACLE defines two sets of intrinsics which correspond to these data processing instructions; a set of scalar intrinsics, and a set of vector intrinsics.

The intrinsics introduced in this section use the data types defined by ACLE. In particular, scalar intrinsics use the `float16_t` type defined by ACLE as an alias for the `__fp16` type, and vector intrinsics use the `float16x4_t` and `float16x8_t` vector types.

Where the scalar 16-bit floating point intrinsics are available, an implementation is required to ensure that including <arm_neon.h> has the effect of also including <arm_fp16.h>.

To only enable support for the scalar 16-bit floating-point intrinsics, the header <arm_fp16.h> may be included directly.

11.1.7 Operations on data types

ACLE does not define implicit conversion between different data types. E.g.

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

Is not portable. 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.

```
int32x4_t x = { 1, 2, 3, 4 };
```

Is not portable. Use the `vcreate` or `vdup` intrinsics to construct values from scalars.

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

11.1.8 Compatibility with other vector programming models

ACLE does not specify how the Advanced SIMD Intrinsics interoperate with alternative vector programming models. Consequently, programmers should take particular care when combining the Advanced SIMD Intrinsics programming model with such programming models.

For example, the GCC vector extensions permit initialising a variable using array syntax, as so

```
#include "arm_neon.h"
...
uint32x2_t x = {0, 1}; // GCC extension.
uint32_t y = vget_lane_s32 (x, 0); // ACLE NEON Intrinsic.
```

But the definition of the GCC vector extensions is such that the value stored in y will depend on both the target architecture (AArch32 or AArch64) and whether the program is running in big- or little-endian mode.

It is recommended that Advanced SIMD Intrinsics be used consistently:

```
#include "arm_neon.h"
...
const int temp[2] = {0, 1};
uint32x2_t x = vld1_s32 (temp);
uint32_t y = vget_lane_s32 (x, 0);
```

11.1.9 Availability of Advanced SIMD intrinsics and Extensions

11.1.10 Availability of Advanced SIMD intrinsics

Advanced SIMD support is available if the `__ARM_NEON` macro is predefined (see *Advanced SIMD architecture extension (NEON)*). In order to access the Advanced SIMD intrinsics, it is necessary to include the `<arm_neon.h>` header.

```
#if __ARM_NEON
#include <arm_neon.h>
  /* Advanced SIMD intrinsics are now available to use.  */
#endif
```

Some intrinsics are only available when compiling for the AArch64 execution state. This can be determined using the `__ARM_64BIT_STATE` predefined macro (see *ARM/Thumb instruction set architecture*.

11.1.11 Availability of 16-bit floating-point vector interchange types

When the 16-bit floating-point data type `__fp16` is available as an interchange type for scalar values, it is also available in the vector interchange types `float16x4_t` and `float16x8_t`. When the vector interchange types are available, conversion intrinsics between vector of `__fp16` and vector of `float` types are provided.

This is indicated by the setting of bit 1 in `__ARM_NEON_FP` (see *NEON floating-point*).

```
#if __ARM_NEON_FP & 0x1
  /* 16-bit floating point vector types are available.  */
  float16x8_t storage;
#endif
```

11.1.12 Availability of fused multiply-accumulate intrinsics

Whenever fused multiply-accumulate is available for scalar operations, it is also available as a vector operation in the Advanced SIMD extension. When a vector fused multiply-accumulate is available, intrinsics are defined to access it.

This is indicated by `__ARM_FEATURE_FMA` (see *Fused multiply-accumulate (FMA)*).

```
#if __ARM_FEATURE_FMA
  /* Fused multiply-accumulate intrinsics are available.  */
  float32x4_t a, b, c;
  vfma_f32 (a, b, c);
#endif
```

11.1.13 Availability of ARMv8.1-A Advanced SIMD intrinsics

Two new instructions have been added to ARMv8.1 [ARMARMv81] SQRDMLAH and SQRDMLSH. ACLE specifies vector and vector-by-lane intrinsics to access these instructions where they are available in hardware.

This is indicated by `__ARM_FEATURE_QRDMX` (see *Rounding Doubling Multiplies*).

```
#if __ARM_FEATURE_QRDMX
  /* ARMv8.1-A RDMA extensions are available.  */
  int16x4_t a, b, c;
  vqrdmlah_s16 (a, b, c);
#endif
```

11.1.14 Availability of 16-bit floating-point arithmetic intrinsics

ARMv8.2-A [ARMARMv82] introduces new data processing instructions which operate on 16-bit floating point data in the IEEE754-2008 [IEEE-FP] format. ACLE specifies intrinsics which map to the vector forms of these instructions where they are available in hardware.

This is indicated by `__ARM_FEATURE_FP16_VECTOR_ARITHMETIC` (see *16-bit floating-point data processing operations*).

```
#if __ARM_FEATURE_FP16_VECTOR_ARITHMETIC
  float16x8_t a, b;
  vaddq_f16 (a, b);
#endif
```

ACLE also specifies intrinsics which map to the scalar forms of these instructions, see *16-bit floating-point arithmetic scalar intrinsics*. Availability of the scalar intrinsics is indicated by `__ARM_FEATURE_FP16_SCALAR_ARITHMETIC`.

```
#if __ARM_FEATURE_FP16_SCALAR_ARITHMETIC
  float16_t a, b;
  vaddh_f16 (a, b);
#endif
```

11.2 Specification of Advanced SIMD intrinsics

The Advanced SIMD intrinsics are specified in the ARM Neon Intrinsics Reference Architecture Specification [NEON].

The behavior of an intrinsic is specified to be equivalent to the AArch64 instruction it is mapped to in [NEON]. Intrinsics are specified as a mapping between their name, arguments and return values and the AArch64 instruction and assembler operands which they are equivalent to.

A compiler may make use of the as-if rule from C [C99] (5.1.2.3) to perform optimizations which preserve the instruction semantics.

11.3 Undefined behavior

Care should be taken by compiler implementers not to introduce the concept of undefined behavior to the semantics of an intrinsic. For example, the `vabsd_s64` intrinsic has well defined behaviour for all input values, while the C99 `llabs` has undefined behaviour if the result would not be representable in a `long long` type. It would thus be incorrect to implement `vabsd_s64` as a wrapper function or macro around `llabs`.

11.4 Alignment assertions

The AArch32 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 Advanced SIMD intrinsics do not directly provide a means for asserting alignment.

CHAPTER

TWELVE

FUTURE DIRECTIONS

12.1 Extensions under consideration

12.1.1 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.

12.1.2 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.

12.1.3 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.

12.1.4 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.

12.1.5 Traps: system calls, breakpoints, ...

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.

12.1.6 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.

12.1.7 Memory access with non-temporal hints

Supporting memory access with cacheability hints through language extensions is being investigated. Eg.

```
int *__attribute__((nontemporal)) p;
```

As a type attribute, will allow indirection of p with non-temporal cacheability hint.

12.2 Features not considered for support

12.2.1 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.

12.2.2 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.

BIBLIOGRAPHY

[ARMARM] ARM, ARM Architecture Reference Manual (7-A / 7-R), ARM DDI 0406C

[ARMARMv8] ARM, ARMv8-A Reference Manual (Issue A.b), ARM DDI0487A.B

[ARMARMv81] ARM, ARMv8.1 Extension, http://community.arm.com/groups/processors/blog/2014/12/02/the-armv8-a-architecture-and-its-ongoing-development

[ARMARMv82] ARM, ARMv8.2 Extension, https://community.arm.com/groups/processors/blog/2016/01/05/armv8-a-architecture-evolution

[ARMARMv83] ARM, ARMv8.3 Extension, https://community.arm.com/processors/b/blog/posts/armv8-a-architecture-2016-additions

[ARMv7M] ARM, ARM Architecture Reference Manual (7-M), ARM DDI 0403C

[AAPCS] ARM, Procedure Call Standard, ARM IHI 0042D

[AAPCS64] ARM, Procedure Call Standard (AArch64), ARM IHI0055C-BETA

[BA] ARM, EABI Addenda and Errata Build Attributes, ARM IHI 0045C

[CPP11] ISO, Standard C++ (based on draft N3337), ISO/IEC 14882:2011

[C11] ISO, Standard C (based on draft N1570), ISO/IEC 9899:2011

[C99] ISO, Standard C (C99), ISO 9899:1999

[cxxabi] Code-Sourcery, Itanium C++ ABI (rev. 1.86), http://mentorembedded.github.com/cxx-abi/abi.html

[G.191] ITU-T, Software Tool Library 2005 User's Manual, T-REC-G.191-200508-I

[GCC] GNU/FSF, GNU C Compiler Collection, http://gcc.gnu.org/onlinedocs

[IA-64] Intel, Intel Itanium Processor-Specific ABI, 245370-003

[IEEE-FP] IEEE, IEEE Floating Point, IEEE 754-2008

[CFP15] ISO/IEC, Floating point extensions for C, ISO/IEC TS 18661-3

[NEON] ARM, ARM NEON Intrinsics Reference, ARM IHI0073B

[POSIX] IEEE / TOG, The Open Group Base Specifications, IEEE 1003.1

[Warren] H. Warren, Hacker's Delight, pub. Addison-Wesley 2003