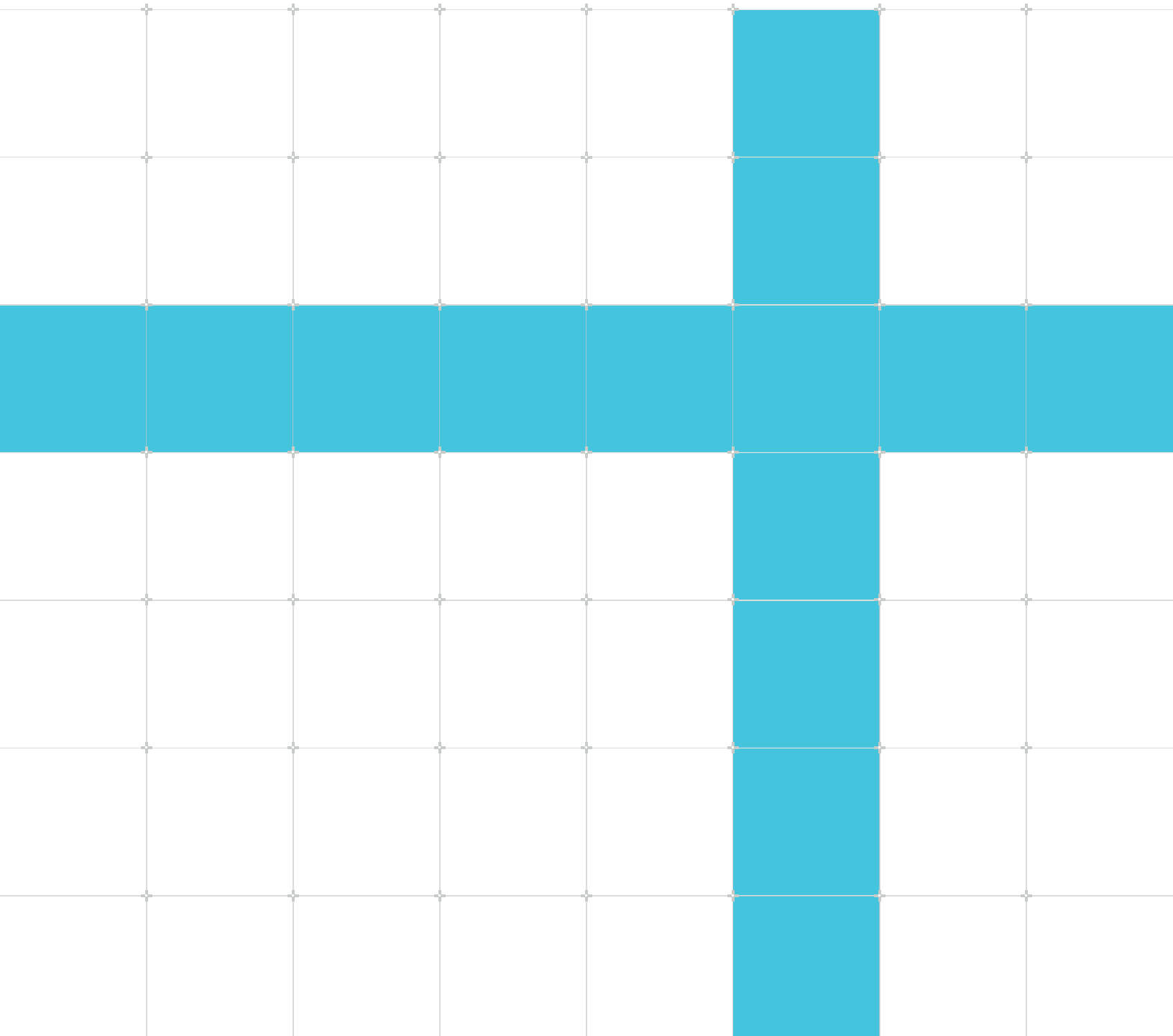
Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile

Known issues in Issue F.c

Issue 04
102105_F.c_04_en



Release information

Issue	Date	Confidentiality	Change
F.c-00	27 August 2020	Non-Confidential	Known Issues in Arm® Architecture Reference Manual, Issue F.c, as of 21 August 2020
F.c-01	30 September 2020	Non-Confidential	Known Issues in Arm® Architecture Reference Manual, Issue F.c, as of 25 September 2020
F.c-02	30 October 2020	Non-Confidential	Known Issues in Arm® Architecture Reference Manual, Issue F.c, as of 23 October 2020
F.c-03	30 November 2020	Non-Confidential	Known Issues in Arm® Architecture Reference Manual, Issue F.c, as of 20 November 2020
F.c-04	18 December 2020	Non-Confidential	Known Issues in Arm® Architecture Reference Manual, Issue F.c, as of 18 December 2020

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349 version 21.0

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Web address

developer.arm.com

Progressive terminology commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used terms that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive terms. If you find offensive terms in this document, please contact terms@arm.com.

Contents

1 Introduction

1.1 Conventions

The following subsections describe conventions used in Arm documents.

1.1.1 Glossary

The Arm Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm® Glossary for more information: developer.arm.com/glossary.

1.1.2 Typographic conventions

Convention	Use
italic	Introduces special terminology, denotes cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.
`monospace underline`	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<and>`	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.

Convention	Use
	This represents a recommendation which, if not followed, might lead to system failure or damage.
	This represents a requirement for the system that, if not followed, might result in system failure or damage.
	This represents a requirement for the system that, if not followed, will result in system failure or damage.
	This represents an important piece of information that needs your attention.
	This represents a useful tip that might make it easier, better or faster to perform a task.
	This is a reminder of something important that relates to the information you are reading.

1.2 Additional reading

This document contains information that is specific to this product. See the following documents for other relevant information:

Table 1: Arm publications

Document Name	Document ID	Licensee only
Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile, Issue F.c	DDI 0487F.c	No

1.3 Feedback

Arm welcomes feedback on this product and its documentation.

1.3.1 Feedback on this product

Information about how to give feedback on the product.

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

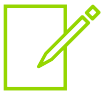
1.3.2 Feedback on content

Information about how to give feedback on the content.

If you have comments on content then send an e-mail to errata@arm.com. Give:

- The title Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile Known issues in Issue F.c.
- The number 102105_F.c_04_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.


Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

1.4 Other information

See the Arm website for other relevant information.

- Arm® Developer.
- Arm® Documentation.
- Technical Support
- Arm® Glossary.

2 Known issues

This document records known issues in the Arm Architecture Reference Manual, Armv8, for Armv8-A architecture profile (DD10487), Issue F.c.

Key

- C = Clarification.
- D = Defect.
- R = Relaxation.
- E = Enhancement.

2.1 D12791

In section J1.1 (Pseudocode for AArch64 operation), the Pseudocode function AArch64.TakePhysicalSErrorException() does not perform any checks before clearing the pending physical SError.

The following function is added:

```
// IsSErrorEdgeTriggered()
// =======================
// Returns TRUE if the physical SError interrupt is edge-triggered
// and FALSE otherwise.

boolean IsSErrorEdgeTriggered(bits(24) syndrome)
    if HaveRASExt() then
        if HaveDoubleFaultExt() then
            return TRUE;
        if UsingAArch32() && syndrome<11:10> != '00' then
            // AArch32 and not Uncontainable.
            return TRUE;
        if !UsingAArch32() && syndrome<23> == '0' && syndrome<5:0> != '000000' then
            // AArch64 and neither IMPLEMENTATION DEFINED syndrome nor
 Uncategorized.
            return TRUE;
    return boolean IMPLEMENTATION_DEFINED \"Edge-triggered SError\";
```

The code that reads:

```
AArch64.TakePhysicalSErrorException()
    ...
    exception = ExceptionSyndrome(Exception_SError);
    exception.syndrome<24> = if impdef_syndrome then '1' else '0';
    exception.syndrome<23:0> = syndrome;

    ClearPendingPhysicalSError();

    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return,
 vect_offset);
```

Is updated to read:

```
AArch64.TakePhysicalSErrorException()
    ...
    exception = ExceptionSyndrome(Exception_SError);
    exception.syndrome<24> = if impdef_syndrome then '1' else '0';
    exception.syndrome<23:0> = syndrome;

    if IsSErrorEdgeTriggered(syndrome) then
        ClearPendingPhysicalSError();

    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return,
 vect_offset);
```

A similar change is made to the Pseudocode function AArch32.TakePhysicalSErrorException() in section J1.2 (Pseudocode for AArch32 operation).

The code that reads:

```
AArch32.TakePhysicalSErrorException(boolean parity, bit extflag, bits(2) errortype,
 boolean impdef_syndrome, bits(24) full_syndrome)
    ClearPendingPhysicalSError();
    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    ...
```

Is updated to read:

```
AArch32.TakePhysicalSErrorException(boolean parity, bit extflag, bits(2) errortype,
 boolean impdef_syndrome, bits(24) full_syndrome)
    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    ...
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;

    if IsSErrorEdgeTriggered(full_syndrome) then
        ClearPendingPhysicalSError();

    fault = AArch32.AsynchExternalAbort(parity, errortype, extflag);
```

2.2 C14537

In section D7.11.3 (Common event numbers) in the Performance Monitors Extension chapter, the following paragraph is added:

```
It is IMPLEMENTATION DEFINED which events, including Common events, are generated
 by IMPLEMENTATION DEFINED extensions to the architecture, including accesses
 to IMPLEMENTATION DEFINED System registers and IMPLEMENTATION DEFINED System
 instructions. However, the functionality of the IMPLEMENTATION DEFINED extension
 must be appropriate for the generated events.
```

2.3 D15346

In section D1.7.1 (Accessing PSTATE fields), the following text:

```
PSTATE.{N, Z, C, V, TCO} can be accessed at EL0. Access to PSTATE.{D, A, I, F} at
 EL0 using AArch64 depends on SCTLR_EL1.UMA, see Traps to EL1 of EL0 accesses to
 the PSTATE.{D, A, I, F} interrupt masks on page D1-2371. All other PSTATE access
 instructions can be executed at EL1 or higher and are UNDEFINED at EL0.
```

is corrected to read:

```
PSTATE.{N, Z, C, V, SSBS, DIT,  TCO} can be accessed at EL0. Access to PSTATE.{D,
 A, I, F} at EL0 using AArch64 depends on SCTLR_EL1.UMA, see Traps to EL1 of EL0
 accesses to the PSTATE.{D, A, I, F} interrupt masks on page D1-2371. All other
 PSTATE access instructions can be executed at EL1 or higher and are UNDEFINED at
 EL0.
```

This text appears in two locations, for the register access and immediate access, and the correction is made in both.

2.4 C15549

In section D7.11.3 (Common event numbers), in the sub-section 'Common architectural events', the definition of the LD_RETIRED event is extended with the following text:

```
If Armv8 Memory Tagging is implemented, the counter increments for every executed
 Allocation tag load instruction.
```

Similarly, the definition of the ST_RETIRED event is extended with the following text:

```
If Armv8 Memory Tagging is implemented, the counter increments for every executed
 Allocation tag store instruction.
```

2.5 D15558

In section J1.1 (Pseudocode for AArch64 operation), during a translation table walk, the fault.write value for an External abort can be incorrectly reported resulting in the WnR field in the ISS encoding for an exception from a Data Abort not being set.

_Mem[] (non-assignment form) is updated to take a boolean parameter, read_for_write, signifying if a read is taking place as part of a write operation.

The function prototype that reads:

```
bits(8*size) _Mem[AddressDescriptor desc, integer size, AccessDescriptor accdesc]
```

Is updated to read:

```
bits(8*size) _Mem[AddressDescriptor desc, integer size, AccessDescriptor accdesc,
 boolean read_for_write]
```

2.6 D15648

In D13.2.117 (SCXTNUM_EL1, EL1 Read/Write Software Context Number), the HCR_EL2.<NV2,NV1,NV> == '011' trap is added at EL1. For example, the MRS accessor at EL1:

```
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED
 \"EL3 trap priority when SDD == '1'\" && SCR_EL3.EnSCXT == '0' then
       UNDEFINED;
   elsif EL2Enabled() && HCR_EL2.EnSCXT == '0' then
       AArch64.SystemAccessTrap(EL2, 0x18);
   elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
 HFGWTR_EL2.SCXTNUM_EL1 == '1' then
       AArch64.SystemAccessTrap(EL2, 0x18);
   elsif HaveEL(EL3) && SCR_EL3.EnSCXT == '0' then
       if Halted() && EDSCR.SDD == '1' then
           UNDEFINED;
       else
           AArch64.SystemAccessTrap(EL3, 0x18);
   elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
       NVMem[0x188] = X[t];
   else
       SCXTNUM_EL1 = X[t];
```

is updated to:

```
elsif PSTATE.EL == EL1 then
   if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED
 \"EL3 trap priority when SDD == '1'\" && SCR_EL3.EnSCXT == '0' then
       UNDEFINED;
   elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '011' then
       AArch64.SystemAccessTrap(EL2, 0x18);
   elsif EL2Enabled() && HCR_EL2.EnSCXT == '0' then
       AArch64.SystemAccessTrap(EL2, 0x18);
   elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
 HFGWTR_EL2.SCXTNUM_EL1 == '1' then
       AArch64.SystemAccessTrap(EL2, 0x18);
   elsif HaveEL(EL3) && SCR_EL3.EnSCXT == '0' then
       if Halted() && EDSCR.SDD == '1' then
           UNDEFINED;
       else
           AArch64.SystemAccessTrap(EL3, 0x18);
   elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
       NVMem[0x188] = X[t];
   else
       SCXTNUM_EL1 = X[t];
```

The equivalent change is made for the MSR accessor for SCXTNUM_EL1 at EL1.

In D13.2.47 (HCR_EL2, Hypervisor Configuration Register), in the NV1 field under the conditions, 'When FEAT_NV2 is implemented' and 'When FEAT_NV is implemented', in both descriptions for value `0b1`, the text that reads:

```
If HCR_EL2.NV2 is 0, EL1 accesses to VBAR_EL1, ELR_EL1, and SPSR_EL1, are trapped to
 EL2,
```

is updated to:

```
If HCR_EL2.NV2 is 0, EL1 accesses to VBAR_EL1, ELR_EL1, SPSR_EL1, and SCXTNUM_EL1
 are trapped to EL2,
```

In D5.7.1 (Armv8.3 nested virtualization functionality), in the subsection, 'Additional behaviors when HCR_EL2.NV == 1 and HCR_EL2.NV1 == 1', the line that reads:

```
Accesses to VBAR_EL1, ELR_EL1, and SPSR_EL1 from EL1 are trapped to EL2. In this
 case the exception is reported in ESR_EL2 using the EC code 0x18.
```

is enhanced to read:

```
Accesses to VBAR_EL1, ELR_EL1, SPSR_EL1, and, if implemented, the SCXTNUM_EL1, from
 EL1 are trapped to EL2. In this case the exception is reported in ESR_EL2 using the
 EC code 0x18.
```

2.7 D15740

In section J1.3 (Shared pseudocode), the function SetPSTATEFromPSR() does not correctly set PSTATE bits on an illegal exception return.

The code that reads:

```
SetPSTATEFromPSR(bits(32) spsr)
    PSTATE.SS = DebugExceptionReturnSS(spsr);
    if IllegalExceptionReturn(spsr) then
        PSTATE.IL = '1';
        if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr<20>;
        if spsr<4> == '1' then                    // AArch32 state
            AArch32.WriteMode(spsr<4:0>);         // Sets PSTATE.EL correctly
            if HaveSSBSExt() then PSTATE.SSBS = spsr<23>;
        else                                      // AArch64 state
            PSTATE.nRW = '0';
            PSTATE.EL  = spsr<3:2>;
            PSTATE.SP  = spsr<0>;
            if HaveSSBSExt() then PSTATE.SSBS = spsr<12>;
```

is updated to read:

```
SetPSTATEFromPSR(bits(32) spsr)
    PSTATE.SS = DebugExceptionReturnSS(spsr);
    if IllegalExceptionReturn(spsr) then
        PSTATE.IL = '1';
        if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
        if HaveBTIExt() then PSTATE.BTYPE = bits(2) UNKNOWN;
        if HaveUAOExt() then PSTATE.UAO = bit UNKNOWN;
        if HaveDITExt() then PSTATE.DIT = bit UNKNOWN;
        if HaveMTEExt() then PSTATE.TCO = bit UNKNOWN;
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr<20>;
        if spsr<4> == '1' then                    // AArch32 state
            AArch32.WriteMode(spsr<4:0>);         // Sets PSTATE.EL correctly
            if HaveSSBSExt() then PSTATE.SSBS = spsr<23>;
        else                                      // AArch64 state
            PSTATE.nRW = '0';
            PSTATE.EL  = spsr<3:2>;
            PSTATE.SP  = spsr<0>;
            if HaveBTIExt() then PSTATE.BTYPE = spsr<11:10>;
            if HaveSSBSExt() then PSTATE.SSBS = spsr<12>;
            if HaveUAOExt() then PSTATE.UAO = spsr<23>;
            if HaveDITExt() then PSTATE.DIT = spsr<24>;
            if HaveMTEExt() then PSTATE.TCO = spsr<25>;
```

2.8 D15876

In section G4.4.7 ('AArch32 cache and branch predictor maintenance instructions') in the subsection ('Effects of virtualization and security on the AArch32 cache maintenance instructions'), in the Table G4-6 ('Effects of virtualization and security on the AArch32 cache maintenance instructions') in the entry for ICIALLU/ICIALLUS, the effects of Secure EL2 on Secure EL1 AArch32 have not been added.

It should take the same form as the AArch64 equivalent instructions documented in Table D4-7 ('Effects of virtualization and security on the maintenance instructions').

2.9 D15893

In section D13.3.24 (OSECCR_EL1, OS Lock Exception Catch Control Register), each occurrence of the following text:

```
    else
        return OSECCR_EL1;
```

is replaced by:

```
   elsif OSLSR_EL1.OSLK == '0' then
        return bits(64) UNKNOWN;
```

```
    else
        return OSECCR_EL1;
```

and each occurrence of:

```
   else
        OSECCR_EL1 = X[t];
```

is replaced with:

```
elsif OSLSR_EL1.OSLK == '0' then
        //no operation
    else
        OSECCR_EL1 = X[t];
```

Similar changes are made in G8.3.21 (DBGOSECCR, Debug OS Lock Exception Catch Control Register).

2.10 C15932

In the Glossary, a new term is added:

```
Conventional memory
Memory locations from which generic OSs and application run-times will expect to
 create allocations for general software use.
```

In section D6.1 (Introduction) in the Memory Tagging Extension chapter, the following Note is removed:

```
Implementations are expected to provide one Allocation Tag for each 16 byte granule
 of bulk data memory.
```

and is replaced with the following text:

```
The extension defines two levels of support for Memory tagging:
Instruction only - Supports the Memory tagging instructions accessible in EL0,
 but does not support system level instructions or System registers defined by the
 extension, or Allocation Tags in memory.
Full - Supports all instructions and System registers defined by the extension,
 Allocation Tags in memory, and Tag Checking of accesses to tagged memory.
If Full Memory tagging is implemented, Allocation Tags are provided for each 16-byte
 granule of Conventional memory.
```

In section D6.2 (Allocation Tags), the following Note is removed:

```
Arm recommends that implementations provide storage for Allocation Tags at each tag
 PA where general-purpose memory exists at the same physical address in the data PA
 space.
```

and replaced with the following text:

```
If Full Memory tagging is implemented, storage is provided for Allocation Tags at
 each tag PA where Conventional memory exists at the same physical address in the
 data PA space.
```

In D13.2.65 (ID_AA64PFR1_EL1, AArch64 Processor Feature Register 1), the MTE field description that reads:

```
* 0b0001 - Memory Tagging Extension instructions accessible at EL0 are implemented.
 Instructions and System Registers defined by the extension not configurably
 accessible at EL0 are Unallocated and other System Register fields defined by the
 extension are RES0.
* 0b0010 - Memory Tagging Extension is implemented.
```

is replaced with

```
* 0b0001 - Instruction only Memory tagging is implemented.
* 0b0010 - Full Memory tagging is implemented.
```

2.11 C16013

In section B2.3.1 (Basic Definitions of the Arm Memory Model), the definition that reads:

```
Location
A Location refers to a single byte in memory.
```

is clarified to read:

```
Location
A Location is a byte that is associated with an address in the physical address
 space.

Note: It is expected that an operating system will present the illusion to the
 application programmer that is consistent with a location also being considered as
 a byte that is associated with an address in the virtual address space.
```

2.12 D16095

In section J1.1 (Pseudocode for AArch64 operation), a comment in the Pseudocode function _ChooseRandomNonExcludedTag does not reflect the relaxation of the architecture allowing

GCR_EL1.RRND to be treated as **RES0** other than for the purpose of reading and writing the register field.

The comment that reads:

```
// This function is expected to generate a non-deterministic selection from the set
 of non-excluded Allocation Tags.
// A reasonable implementation is described by the Pseudocode used when
// GCR_EL1.RRND is 0, but with a non-deterministic implementation of
 NextRandomTagBit().
bits(4) _ChooseRandomNonExcludedTag(bits(16) exclude);
```

is updated to read:

```
// This function is permitted to generate a non-deterministic selection from the set
 of non-excluded Allocation Tags.
// A reasonable implementation is described by the Pseudocode used when
// GCR_EL1.RRND is 0, but with a non-deterministic implementation of
 NextRandomTagBit().
// Implementations may choose to behave the same as GCR_EL1.RRND=0.
bits(4) ChooseRandomNonExcludedTag(bits(16) exclude);
```

In section C6.2.96 (IRG), the Pseudocode which describes the behavior of the IRG instruction does not clearly define for GCR_EL1.RRND=1, when the exclude bits are all ones.

The Pseudocode that reads:

```
if AArch64.AllocationTagAccessIsEnabled(AccType_NORMAL) then
    if GCR_EL1.RRND == '1' then
        RGSR_EL1 = bits(32) UNKNOWN;
        rtag = _ChooseRandomNonExcludedTag(exclude);
    else
```

is updated to read:

```
if AArch64.AllocationTagAccessIsEnabled(AccType_NORMAL) then
    if GCR_EL1.RRND == '1' then
        RGSR_EL1 = bits(32) UNKNOWN;
        if IsOnes(exclude) then
            rtag = '0000';
        else
            rtag = ChooseRandomNonExcludedTag(exclude);
    else
```

2.13 D16111

In D13.7.1 (DISR_EL1, Deferred Interrupt Status Register), the accessibility pseudocode is updated to include the **RAZ/WI** access when SCR_EL3.EA or SCR.EA is 1. The MRS accessibility pseudocode:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
```

```
    if EL2Enabled() && HCR_EL2.AMO == '1' then
        return VDISR_EL2;
   else
        return DISR_EL1;
elsif PSTATE.EL == EL2 then
    return DISR_EL1;
elsif PSTATE.EL == EL3 then
    return DISR_EL1;
```

is updated to :

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.AMO == '1' then
        return VDISR_EL2;
    elsif HaveEL(EL3) && !Halted() && SCR_EL3.EA == '1' then
        return Zeros();
    else
        return DISR_EL1;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !Halted() && SCR_EL3.EA == '1' then
        return Zeros();
    else
        return DISR_EL1;
elsif PSTATE.EL == EL3 then
    return DISR_EL1;
```

and the MSR accessibility is updated to:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && HCR_EL2.AMO == '1' then
        VDISR_EL2 = X[t];
    elsif HaveEL(EL3) && !Halted() && SCR_EL3.EA == '1' then
        //no operation
    else
        DISR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && !Halted() && SCR_EL3.EA == '1' then
        //no operation
    else
        DISR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    DISR_EL1 = X[t];
```

In D13.7.15 (VDISR_EL2, Virtual Deferred Interrupt Status Register), the MRS access to DISR_EL1 is similarly updated.

In G8.6.1 (DISR, Deferred Interrupt Status Register), the MRC accessibility code:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.AMO == '1' then
        return VDISR_EL2;
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.AMO == '1' then
```

```
        return VDISR;
    else
        return DISR;
elsif PSTATE.EL == EL2 then
        return DISR;
elsif PSTATE.EL == EL3 then
    return DISR;
```

is updated to:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.AMO == '1' then
        return VDISR_EL2;
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.AMO == '1' then
        return VDISR;
    elsif Have(EL3) && ELUsingAArch32(EL3) && !Halted() && SCR.EA == '1' then
        return Zeros();
    else
        return DISR;
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) && !Halted() && SCR.EA == '1' then
        return Zeros();
    else
        return DISR;
elsif PSTATE.EL == EL3 then
    return DISR;
```

and the MCR access updated to:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T12 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T12 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.AMO == '1' then
        VDISR_EL2 = R[t];
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HCR.AMO == '1' then
        VDISR = R[t];
    elsif Have(EL3) && ELUsingAArch32(EL3) && !Halted() && SCR.EA == '1' then
        //no operation
    else
        DISR = R[t];
elsif PSTATE.EL == EL2 then
    if HaveEL(EL3) && ELUsingAArch32(EL3) && !Halted() && SCR.EA == '1' then
        //no operation
    else
        DISR = R[t];
elsif PSTATE.EL == EL3 then
    DISR = R[t];
```

In G8.6.20 (VDISR, Virtual Deferred Interrupt Status Register) the MRC access to DISR is similarly updated.

2.14 D16140

In section D9.6 ('The profiling data'), the Note that reads:

```
This behavior describes when CNTEN.EN is cleared to 0. This behavior does not apply
 when the Generic Timer system counter is enabled but not accessible at the current
 Exception level.
```

is corrected to:

```
This behavior describes when CNTCR.EN is 0, the Generic Timer system counter is
 disabled. This behavior does not apply when the Generic Timer system counter is
 enabled but not accessible at the current Exception level.
```

2.15 D16243

In D5.4.5 (Data access permission controls), subsection 'Preventing EL0 access to halves of the address map', the text that reads:

```
If access is prevented, the fault is reported as a level 0 fault, and should take
 the same time to generate, whether the address is present in the TLB or not, to
 mitigate attacks that use fault timing.
```

is updated to read:

```
If access is prevented, the fault is reported as a level 0 translation fault. The
 fault should take the same time to generate, whether the address is present in the
 TLB or not, to mitigate attacks that use fault timing. This type of fault is not
 counted as a TLB miss for performance monitoring features.
```

In D13.2.120 (TCR_EL1, Translation Control Register (EL1)), D13.2.121 (TCR_EL2, Translation Control Register (EL2)), and D13.2.122 (TCR_EL3, Translation Control Register (EL3)), the following text is added to all of the E0PDn fields.

```
Level 0 translation faults generated as a result of this field are not counted
 as TLB misses for performance monitoring. The fault should take the same time to
 generate, whether the address is present in the TLB or not, to mitigate attacks
 that use fault timing.
```

2.16 D16329

In section J1.3 (Shared pseudocode), in the Pseudocode function DebugExceptionReturnSS() the check for ELUsingAArch32() is incorrect for the case where destination is EL0.

The code that reads:

```
if ELUsingAArch32(dest) then
    enabled_at_dest = AArch32.GenerateDebugExceptionsFrom(dest, secure);
else
    mask = spsr<9>;
    enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest, secure, mask);
```

Is updated to read:

```
dest_using_32 = (if dest == EL0 then spsr<4> == '1' else ELUsingAArch32(dest));
if dest_using_32 then
    enabled_at_dest = AArch32.GenerateDebugExceptionsFrom(dest, secure);
else
    mask = spsr<9>;
    enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest, secure, mask);
```

2.17 D16332

In C6.2 (Alphabetical list of A64 base instructions), the Pseudocode for the LDGM, STGM, and STZGM instructions does not check the value of ID_AA64PFR1_EL1.MTE.

The following code is added into the decode block of each instruction:

```
if !HaveMTE2Ext() then UNDEFINED;
```

2.18 D16367

In section J1.2 (Pseudocode for AArch32 operation) AArch32.CheckWatchpoint() is updated to ignore excluded access types from the watchpoint check.

The code that reads:

```
    assert ELUsingAArch32(S1TranslationRegime());

    match = FALSE;
```

Is updated to read:

```
    assert ELUsingAArch32(S1TranslationRegime());
```

```
    if acctype IN {AccType_PTW, AccType_IC, AccType_AT} then
        return AArch32.NoFault();
    if acctype == AccType_DC then
        if !iswrite then
            return AArch32.NoFault();
        elsif !(boolean IMPLEMENTATION_DEFINED \"DCIMVAC generates watchpoint\")
then
            return AArch32.NoFault();

    match = FALSE;
```

In section J1.1 (Pseudocode for AArch64 operation) AArch64.CheckWatchpoint() is also updated.

The code that reads:

```
     assert ELUsingAArch32(S1TranslationRegime());

    match = FALSE;
```

Is updated to read:

```
    assert ELUsingAArch32(S1TranslationRegime());
    if acctype IN {AccType_PTW, AccType_IC, AccType_AT} then
        return AArch32.NoFault();
    if acctype == AccType_DC then
        if !iswrite then
            return AArch32.NoFault();

    match = FALSE;
```

In section J1.2 (Pseudocode for AArch32 operation) AArch32.TranslateAddress() the check for the excluded access types is moved to AArch32.CheckWatchpoint(). The code that reads:

```
    result = AArch32.FullTranslate(vaddress, acctype, iswrite, wasaligned, size);
   if !(acctype IN {AccType_PTW, AccType_IC, AccType_AT}) && !IsFault(result) then
        result.fault = AArch32.CheckDebug(vaddress, acctype, iswrite, size);
```

Is updated to read:

```
    result = AArch32.FullTranslate(vaddress, acctype, iswrite, wasaligned, size);
    if !IsFault(result) then
        result.fault = AArch32.CheckDebug(vaddress, acctype, iswrite, size);
```

In section J1.1 (Pseudocode for AArch64 operation) AArch64.TranslateAddress() the check for the excluded access types is moved to AArch64.CheckWatchpoint().

The code that reads:

```
    result = AArch64.FullTranslate(vaddress, acctype, iswrite, wasaligned, size);
   if !(acctype IN {AccType_PTW, AccType_IC, AccType_AT}) && !IsFault(result) then
        result.fault = AArch64.CheckDebug(vaddress, acctype, iswrite, size);
```

Is updated to read:

```
    result = AArch64.FullTranslate(vaddress, acctype, iswrite, wasaligned, size);
    if !IsFault(result) then
        result.fault = AArch64.CheckDebug(vaddress, acctype, iswrite, size);
```

2.19 R16399

In section D9.7 (The Profiling Buffer), a new sub-section D9.7.4 is inserted after section D9.7.3 (Memory access types and coherency).

The section is titled 'Memory access and crossing page boundaries'.

The content of the new section is as follows:

```
A memory access from SPE that crosses a page boundary to a memory location that
 has a different memory type or Shareability attribute results in CONSTRAINED
 UNPREDICTABLE behavior. In this case, the implementation performs one of the
 following behaviors:

- Each memory access generated by the SPE uses the memory type and Shareability
 attribute associated with its own address.
- The access generates an Alignment fault caused by the memory type:
-- If only the stage 1 translation generated the mismatch, or there is only one
 stage of translation in the owning translation regime, the resulting Buffer
 Management event is a stage 1 Data Abort.
-- If only the stage 2 translation generated the mismatch, the resulting Buffer
 Management event is a stage 2 Data Abort.
-- If both stages of translation generate the mismatch, the resulting Buffer
 Management event is either a stage 1 Data Abort or a stage 2 Data Abort.
- Some or all of the data is discarded. The write pointer is either updated by
 the amount of data written not including the discarded data or the amount of data
 written including the discarded data.

A memory access from SPE to Device memory that crosses a boundary corresponding
 to the smallest translation granule size of the implementation causes CONSTRAINED
 UNPREDICTABLE behavior. In this case, the implementation performs one of the
 following behaviors:

- All memory accesses generated by SPE are performed as if the boundary has no
 effect on the memory accesses.
- All memory accesses generated by SPE are performed as if the boundary has no
 effect on the memory accesses except that there is no guarantee of ordering between
 memory accesses.
- The access generates an Alignment fault caused by the memory type:
-- If only the stage 1 translation causes the boundary to be crossed, or there
 is only one stage of translation in the owning translation regime, the resulting
 Buffer Management event is a stage 1 Data Abort.
-- If only the stage 2 translation causes the boundary to be crossed, the resulting
 Buffer Management event is a stage 2 Data Abort.
-- If both stages of translation cause the boundary to be crossed, the resulting
 Buffer Management event is either a stage 1 Data Abort or a stage 2 Data Abort.
- Some or all of the data is discarded. The write pointer is either updated by
 the amount of data written not including the discarded data or the amount of data
 written including the discarded data.

Note:
The boundary referred to is between two Device memory regions that are both:
- Of the size of the smallest implemented translation granule.
- Aligned to the size of the smallest implemented translation granule.
```

```
If PMSIDR_EL1.MaxSize indicates the same value as PMBIDR_EL1.Align, then records are
 a fixed size and never cross a page boundary.
```

2.20 D16409

In section J1.1 (Pseudocode for AArch64 operation) AArch64.CheckPermission() is corrected for checks in case of cache maintenance operations. IMPDEF conditions previously unaccounted for are also introduced.

The code that reads:

```
    if acctype == AccType_IFETCH then
        fail = xn;
        failedread = TRUE;
    elsif acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW,
 AccType_ORDEREDATOMICRW } then
        fail = !r || !w;
        failedread = !r;
    elsif iswrite then
        fail = !w;
        failedread = FALSE;
    elsif acctype == AccType_DC && PSTATE.EL != EL0 then
    // DC maintenance instructions operating by VA, cannot fault from stage 1
 translation,
    // other than DC IVAC, which requires write permission, and operations executed
 at EL0,
    // which require read permission.
        fail = FALSE;
    else
        fail = !r;
        failedread = TRUE;

    if fail then
        secondstage = FALSE;
        s2fs1walk = FALSE;
        ipaddress = bits(52) UNKNOWN;
        return AArch64.PermissionFault(ipaddress,boolean UNKNOWN, level, acctype,!
failedread, secondstage, s2fs1walk);
```

Is updated to read:

```
    if acctype == AccType_IFETCH then
        fail = xn;
    elsif acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW,
 AccType_ORDEREDATOMICRW } then
        fail = !r || !w;
        if fail then iswrite = r;    // Report as a read failure if a read of the
 location would fail.
    elsif acctype IN {AccType_IC, AccType_DC} then
        if UsingAArch32() then
            fail = FALSE;
        elsif iswrite && acctype == AccType_DC then
            fail = !w;
        elsif PSTATE.EL == EL0 && !iswrite then
            fail = !r && !(acctype == AccType_IC && !(boolean IMPLEMENTATION_DEFINED
 \"Permission fault on EL0 IC_IVAU execution\"));
        else
```

```
            fail = FALSE;

    elsif iswrite then
        fail = !w;
    else
        fail = !r;

    if fail then
        secondstage = FALSE;
        s2fs1walk = FALSE;
        ipaddress = bits(52) UNKNOWN;
        return AArch64.PermissionFault(ipaddress,boolean UNKNOWN,  level, acctype,
 iswrite, secondstage, s2fs1walk);
```

In section J1.2 (Pseudocode for AArch32 operation) AArch32.CheckPermission() is corrected for checks in case of cache maintenance operations.

The code that reads:

```
    if acctype == AccType_IFETCH then
        fail = xn;
        failedread = TRUE;
    elsif acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW,
 AccType_ORDEREDATOMICRW } then
        fail = !r || !w;
        failedread = !r;
    elsif acctype == AccType_DC then
        // DC maintenance instructions operating by VA, cannot fault from stage 1
 translation.
        fail = FALSE;
        elsif iswrite then
        fail = !w;
        failedread = FALSE;
    else
        fail = !r;
        failedread = TRUE;
    if fail then
        secondstage = FALSE;
        s2fs1walk = FALSE;
        ipaddress = bits(40) UNKNOWN;
        return AArch32.PermissionFault(ipaddress, domain, level, acctype, !
failedread, secondstage, s2fs1walk);
```

Is updated to read:

```
    if acctype == AccType_IFETCH then
        fail = xn;
    elsif acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW,
 AccType_ORDEREDATOMICRW } then
        fail = !r || !w;
        if fail then iswrite = r;    // Report as a read failure if a read of the
 location would fail.
    elsif acctype IN {AccType_IC, AccType_DC} then
        // AArch32 IC/DC maintenance instructions operating by VA cannot fault.
        fail = FALSE;
    elsif iswrite then
        fail = !w;
    else
        fail = !r;

    if fail then
        secondstage = FALSE;
        s2fs1walk = FALSE;
        ipaddress = bits(40) UNKNOWN;
```

```
        return AArch32.PermissionFault(ipaddress,  domain, level, acctype, iswrite,
secondstage, s2fs1walk);
```

In section J1.1 (Pseudocode for AArch64 operation) AArch64.CheckS2Permission() is corrected for checks in case of cache maintenance operations. IMPDEF conditions previously unaccounted for are also introduced.

The code that reads:

```
   if acctype == AccType_IFETCH && !s2fs1walk then
       fail = xn;
       failedread = TRUE;
   elsif (acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW,
AccType_ORDEREDATOMICRW }) && !s2fs1walk then
       fail = !r || !w;\t\t
       failedread = !r;
   elsif iswrite && !s2fs1walk then
       fail = !w;
       failedread = FALSE;
   elsif acctype == AccType_DC && PSTATE.EL != EL0 && !s2fs1walk then
       // DC maintenance instructions operating by VA, with the exception of DC
IVAC, do
       // not generate Permission faults from stage 2 translation, other than when
       // performing a stage 1 translation table walk.
       fail = FALSE;
   elsif hwupdatewalk then
       fail = !w;
       failedread = !iswrite;
   else
       fail = !r;
       failedread = !iswrite;
   if fail then
       domain = bits(4) UNKNOWN;
       secondstage = TRUE;
       return AArch64.PermissionFault(ipaddress,NS, level, acctype,!failedread,
secondstage, s2fs1walk);
```

Is updated to read:

```
   if acctype == AccType_IFETCH && !s2fs1walk then
       fail = xn;
   elsif (acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW,
AccType_ORDEREDATOMICRW }) && !s2fs1walk then
       fail = !r || !w;
       if fail then iswrite = r;    // Report as a read failure if a read of the
location would fail.
   elsif acctype IN {AccType_IC, AccType_DC} && !s2fs1walk then
       if UsingAArch32() then
           fail = FALSE;
       elsif iswrite && acctype == AccType_DC then
           fail = !w;
       elsif PSTATE.EL == EL0 && !iswrite then
           fail = !r && !(acctype == AccType_IC && !(boolean IMPLEMENTATION_DEFINED
\"Permission fault on EL0 IC_IVAU execution\"));
       else
           fail = FALSE;

   elsif iswrite && !s2fs1walk then
       fail = !w;

   elsif hwupdatewalk then
       fail = !w;
   else
       fail = !r;
```

```
    if fail then
        domain = bits(4) UNKNOWN;
        secondstage = TRUE;
        return AArch64.PermissionFault(ipaddress,NS,  level, acctype,
                                       iswrite, secondstage, s2fs1walk);
```

In section J1.2 (Pseudocode for AArch32 operation) AArch32.CheckS2Permission() is corrected for checks in case of cache maintenance operations.

The code that reads:

```
    if acctype == AccType_IFETCH && !s2fs1walk then
        fail = xn;
        failedread = TRUE;
    elsif (acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW,
 AccType_ORDEREDATOMICRW }) && !s2fs1walk then
        fail = !r || !w;
        failedread = !r;
    elsif acctype == AccType_DC && !s2fs1walk then
        // DC maintenance instructions operating by VA, do not generate Permission
 faults
        // from stage 2 translation, other than from stage 1 translation table walk.
        fail = FALSE;
    elsif iswrite && !s2fs1walk then
        fail = !w;
        failedread = FALSE;
    else
        fail = !r;
        failedread = !iswrite;
    if fail then
        domain = bits(4) UNKNOWN;
        secondstage = TRUE;
        return AArch32.PermissionFault(ipaddress, domain, level, acctype,!
failedread, secondstage, s2fs1walk);
```

Is updated to read:

```
    if acctype == AccType_IFETCH && !s2fs1walk then
        fail = xn;
    elsif (acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW,
 AccType_ORDEREDATOMICRW }) && !s2fs1walk then
        fail = !r || !w;
        if fail then iswrite = r;    // Report as a read failure if a read of the
 location would fail.
    elsif acctype IN {AccType_IC, AccType_DC} && !s2fs1walk then
        // AArch32 IC/DC maintenance instructions operating by VA cannot fault.
        fail = FALSE;
    elsif iswrite && !s2fs1walk then
        fail = !w;
    else
        fail = !r;

    if fail then
        domain = bits(4) UNKNOWN;
        secondstage = TRUE;
        return AArch32.PermissionFault(ipaddress,  domain, level, acctype,
                                       iswrite, secondstage, s2fs1walk);
```

In section J1.1 (Pseudocode for AArch64 operation) AArch64.FirstStageTranslate() is corrected to always check for permissions.

The code that reads:

```
    TLBRecord S1;
    S1.addrdesc.fault = AArch32.NoFault();
    ipaddress = bits(40) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;
```

Is updated to read:

```
    TLBRecord S1;
    S1.addrdesc.fault = AArch32.NoFault();
    ipaddress = bits(40) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;
    permissioncheck = TRUE;
```

In section J1.2 (Pseudocode for AArch32 operation) AArch32.FirstStageTranslate() is corrected to always check for permissions.

The code that reads:

```
    TLBRecord S1;
    S1.addrdesc.fault = AArch32.NoFault();
    ipaddress = bits(40) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;
```

Is updated to read:

```
    TLBRecord S1;
    S1.addrdesc.fault = AArch32.NoFault();
    ipaddress = bits(40) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;
    permissioncheck = TRUE;
```

In section J1.1 (Pseudocode for AArch64 operation) AArch64.SecondStageTranslate() is corrected to always check for permissions.

The code that reads:

```
    if s2_enabled then                          // Second stage enabled
        ipaddress = S1.paddress.address<51:0>;
        NS = S1.paddress.NS == '1';
        S2 = AArch64.TranslationTableWalk(ipaddress, NS, vaddress, acctype, iswrite,
 secondstage,
                                          s2fs1walk, size);
\t.
\t.
\t.
\t        if !IsFault(S2.addrdesc) then
            S2.addrdesc.fault = AArch64.CheckS2Permission(S2.perms, vaddress,
 ipaddress, S2.level,
                                                          acctype, iswrite,
 NS,s2fs1walk, hwupdatewalk);
```

Is updated to read:

```
    if s2_enabled then                          // Second stage enabled
        permissioncheck = TRUE;
        ipaddress = S1.paddress.address<51:0>;
        NS = S1.paddress.NS == '1';
        S2 = AArch64.TranslationTableWalk(ipaddress, NS, vaddress, acctype, iswrite,
 secondstage,
                                            s2fs1walk, size);
\t.
\t.
\t.
\t        if !IsFault(S2.addrdesc) && permissioncheck then
            S2.addrdesc.fault = AArch64.CheckS2Permission(S2.perms, vaddress,
 ipaddress, S2.level,
```

In section J1.2 (Pseudocode for AArch32 operation) AArch32.SecondStageTranslate() is corrected to always check for permissions.

The code that reads:

```
    if s2_enabled then                          // Second stage enabled
        ipaddress = S1.paddress.address<51:0>;
        NS = S1.paddress.NS == '1';
        S2 = AArch64.TranslationTableWalk(ipaddress, NS, vaddress, acctype, iswrite,
 secondstage,
                      .                      s2fs1walk, size);
\t.
\t.
\t.
\t        if !IsFault(S2.addrdesc) then
            S2.addrdesc.fault = AArch64.CheckS2Permission(S2.perms, vaddress,
 ipaddress, S2.level,
                                                          acctype, iswrite,
 NS,s2fs1walk, hwupdatewalk);
```

Is updated to read:

```
    if s2_enabled then                          // Second stage enabled
        permissioncheck = TRUE;
        ipaddress = S1.paddress.address<51:0>;
        NS = S1.paddress.NS == '1';
        S2 = AArch64.TranslationTableWalk(ipaddress, NS, vaddress, acctype, iswrite,
 secondstage,
                                            s2fs1walk, size);
\t.
\t.
\t.
\t        if !IsFault(S2.addrdesc) && permissioncheck then
            S2.addrdesc.fault = AArch64.CheckS2Permission(S2.perms, vaddress,
 ipaddress, S2.level,
                                                          acctype, iswrite,
 NS,s2fs1walk, hwupdatewalk);
```

2.21 D16451

In section C7.2 (Alphabetical list of A64 Advanced SIMD and floating-point instructions), the Pseudocode for instructions FADDP (scalar), FMAXNMP (scalar), FMAXP (scalar), FMINNMP (scalar), FMINP (scalar) did not appear in full as it was improperly trimmed due to a tooling issue. This will be corrected in a future release.

The code in FADDP (scalar), FMAXNMP (scalar), FMAXP (scalar), FMINNMP (scalar), and FMINP (scalar) that reads:

```
integer esize = 32;
integer datasize = 64;
```

is corrected to read:

```
integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
```

2.22 D16454

In the following AArch64 TLBI System instructions: C5.5.1-3 TLBI_ASIDE1 (,IS,OS), C5.5.25-27 TLBI_RVAAE1(,IS,OS), C5.5.28-30 TLBI_RVAALE1(,IS, OS), C5.5.31-33 TLBI_RVAE1(,IS, OS), C5.5.40-42 TLBI_RVALE1(,IS,OS), C5.5.49-51 TLBI_VAAE1(,IS, OS), C5.5.52-54 TLBI_VAALE1(,IS,OS), C5.5.55-57 TLBI_VAE1(,IS, OS), C5.5.64 TLBI_VALE1(,IS,OS), C5.5.73-75 TLBI_VMALLE1(,IS,OS), the text that reads:

```
When EL2 is implemented and enabled in the Security state described by the current
 value of SCR_EL3.NS:
- If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry would be used with the current VMID
 and would be required to translate the specified VA using the EL1&0 translation
 regime.
- If HCR_EL2.{E2H, TGE} is {1, 1}, the entry would be required to translate the
 specified VA using the EL2&0 translation regime.

When EL2 is not implemented or is disabled in the current Security state, the entry
 would be required to translate the specified VA using the EL1&0 translation regime.
```

is replaced with:

```
When EL2 is implemented and enabled in the Security state described by the current
 value of SCR_EL3.NS:
- If HCR_EL2.{E2H, TGE} is not {1, 1}, the entry that would be used with the
 current VMID and would be required to translate the specified VA using the EL1&0
 translation regime for the Security state indicated by the value of the SCR_EL3.NS
 bit.
- If HCR_EL2.{E2H, TGE} is {1, 1}, the entry that would be required to translate the
 specified VA using the EL2&0 translation regime for the Security state indicated by
 the value of the SCR_EL3.NS bit.
```

```
When EL2 is not implemented or is disabled in the current Security state, the entry
 that would be required to translate the specified VA using the EL1&0 translation
 regime for the Security state indicated by the value of the SCR_EL3.NS bit.
```

Similarly, in C5.5.34-36 TLBI_RVAE2(,IS, OS), C5.5.43-45 TLBI_RVALE2(,IS, OS), C5.5.58-60 TLBI_VAE2(,IS, OS), C5.5.67-69 TLBI_VALE2(,IS,OS), the text that reads:

```
using the EL2 or EL2&0 translation regime.
```

is replaced with:

```
using the EL2 or EL2&0 translation regime for the Security state indicated by the
 value of the SCR_EL3.NS bit.
```

2.23 D16498

In section D9.6.3 (Additional information for each profiled memory access operation) the text that reads:

```
- An optional, IMPLEMENTATION DEFINED, record of whether the sampled operation
 accessed Last Level data cache and the result.
- An optional, IMPLEMENTATION DEFINED, record of whether the sampled operation
 accessed another socket in a multi-socket system.
- An optional, IMPLEMENTATION DEFINED, indicator of the data source for a load.
```

is corrected to:

```
- An optional record of whether the sampled operation accessed Last Level data cache
 and the result.
- An optional record of whether the sampled operation accessed another socket in a
 multi-socket system.
- An optional, IMPLEMENTATION DEFINED, indicator of the data source for a load. If
 the sampled operation makes multiple accesses, it is IMPLEMENTATION DEFINED whether
 this indicator combines information for all parts of the load or applies only for a
 chosen part of the load.
```

The following paragraphs:

```
If architecture instructions are sampled, for a sampled load or store operation that
 is not single-copy atomic, the data
addresses are the lowest address that is accessed by the sampled operation
 regardless of whether architecture
instructions are sampled or not.
Otherwise the information is for the micro-op that is sampled.
```

are replaced with:

```
If the sampled load, store, or atomic operation performs multiple accesses, it is
 IMPLEMENTATION DEFINED whether the implementation chooses to profile all of the
```

```
 access or a chosen part of that access. If the implementation chooses to profile a
 chosen part of the access:
- It is IMPLEMENTATION DEFINED how the PE chooses the part of the access. The choice
 does not introduce any systematic bias.
- If the accesses are architecturally contiguous, it is further IMPLEMENTATION
 DEFINED whether the recorded data addresses is the lowest address that is accessed
 by the sampled operation or apply to the chosen part of the access.
- If the accesses are not architecturally contiguous, the recorded data addresses
 apply for the chosen part of the access.
- It is further IMPLEMENTATION DEFINED whether the events and total operation
 latency apply to the whole operation or the chosen part of the operation. See the
 example below.
- The translation latency applies to the chosen part of the operation, and is the
 count of cycles for which the chosen part of the operation is waiting for the MMU
 to complete an address translation.
- The same chosen access is used in each case. The recorded virtual and physical
 data addresses apply to the same access.

Arm recommends that if the implementation chooses to profile a chosen part of the
 access that the recorded addresses, events, and total operation latency apply to
 the chosen access. That is, the PE behaves as if the chosen part of the access is
 the sampled operation.

If the sampled load, store, or atomic operation performs a single access, or the
 implementation chooses to profile all parts of a multiple access:
- If the accesses are architecturally contiguous, the recorded data addresses is the
 lowest address that is accessed by the sampled operation. See the example below.
- If the accesses are not architecturally contiguous, the recorded data addresses
 apply for the chosen part of the access.
- The events and total operation latency apply to the whole operation. For example,
 when recording whether the sampled operation accessed the Level 1 data cache, the
 PE records whether any part of the access accessed the Level 1 data cache, and the
 result, and the total operation latency applies from the issue of the operation to
 the completion of all parts of the operation.
- The translation latency is an IMPLEMENTATION DEFINED choice between:
-- The count of cycles for which at least one part of the operation is waiting
 for the MMU to complete an address translation, and no part of the operation is
 accessing memory.
-- The count of cycles for which at least one part of the operation is waiting for
 the MMU to complete an address translation.
```

The Example D9-2 (Sampling of micro-ops) is moved to section D9.6.1 (Information collected for micro-ops).

The following is added as an (Example of systematic bias when choosing part of a multi-access operation):

```
Example
A multiple register load operation is split into multiple accesses. The PE
 systematically chooses the first operation at the lower address for sampling
 translation latency and data source indicator.
This is not a valid implementation if it introduces systematic bias if, for example,
 the operation is executed in a loop with an incrementing address, meaning the first
 access has better spatial locality with preceding accesses than later accesses and
 is more likely to both:
- Hit in the TLB, giving a shorter translation latency.
- Return data from the Level 1 data cache.

Later accesses that actually incur long translation latencies or return data from
 further out from the PE might be systematically missed by sampling. Similarly,
 if the PE systematically chooses the operation at the higher address, it might
 systematically have worse spatial locality than preceding accesses, again leading
 to biased sampling.
```

2.24 D16571

In section C6.2.316 (SUBS (shifted register)), in the sub-section 'Alias conditions', there is ambiguity around which alias is preferred when: Rd == '11111' && Rn == '11111'. To resolve this ambiguity, the row for the 'NEGS' alias is corrected from:

```
Rn == '11111'
```

to

```
Rd != '11111' && Rn == '11111'
```

2.25 D16611

In section J1.3 (Shared pseudocode), the field EDSCR.INTdis is shown as 2-bits wide in the Pseudocode function ExternalDebugInterruptsDisabled(). When ARMv8.4 Debug is implemented EDSCR.INTdis is 1-bit wide.

The code that reads:

```
case target of
    when EL3
        int_dis = EDSCR.INTdis == '11' && ExternalSecureInvasiveDebugEnabled();
    when EL2
        int_dis = EDSCR.INTdis == '1x' && ExternalInvasiveDebugEnabled();
    when EL1
        if IsSecure() then
            int_dis = EDSCR.INTdis == '1x' && ExternalSecureInvasiveDebugEnabled();
        else
            int_dis = EDSCR.INTdis != '00' && ExternalInvasiveDebugEnabled();
```

Is updated to read:

```
if HaveV84Debug() then
    if target == EL3 || IsSecure() then
        int_dis = EDSCR.INTdis[0] == '1' && ExternalSecureInvasiveDebugEnabled();
    else
        int_dis = EDSCR.INTdis[0] == '1';
else
    case target of
        when EL3
            int_dis = EDSCR.INTdis == '11' && ExternalSecureInvasiveDebugEnabled();
        when EL2
            int_dis = EDSCR.INTdis == '1x' && ExternalInvasiveDebugEnabled();
        when EL1
            if IsSecure() then
                int_dis = EDSCR.INTdis == '1x' &&
 ExternalSecureInvasiveDebugEnabled();
            else
                int_dis = EDSCR.INTdis != '00' && ExternalInvasiveDebugEnabled();
```

2.26 D16625

In D5.10.2 (TLB maintenance instructions), subsection 'TLB maintenance instruction syntax', under 'Translation table level hints', the text that introduces Table D5-54 (TTL field encodings in TLB instructions that apply to multiple addresses) that reads:

```
The TTL field in TLB maintenance instructions that take a register argument that
 holds a VA or an IPA, and that do not apply to a range of addresses, use the
 encodings in Table D5-54.
```

is corrected to:

```
The TTL field in TLB maintenance instructions that take a register argument that
 holds a VA or an IPA, and apply to a range of addresses, use the encodings in Table
 D5-54.
```

2.27 C16672

In section D5.5.7 (Combining the stage 1 and stage 2 attributes, EL1&0 translation regime) in the subsection 'Combining the stage 1 and stage 2 memory type attributes', the bullet associated with the text

```
When the first stage of the translation regime specifies Device memory, HCR_EL2.FWB
 is set to 1, and the stage 2 page or block descriptor [4:2] is set to 0b110 , does
 not prevent:
```

that reads

```
A misaligned memory access generating a first stage alignment fault.
```

is deleted as the next paragraph explains that this is **CONSTRAINED UNPREDICTABLE**.

2.28 C16674

In section D13.2.113 (SCTLR_EL1, System Control Register (EL1)), in the ITFSB field, the text that currently states:

```
When synchronous exceptions are not being generated by Tag Check Faults which are
 generated for Loads and Stores in EL0 or EL1, controls the auto-synchronization of
 Tag Check Faults into TFSRE0_EL1 and TFSR_EL1.
```

is replaced by:

```
When synchronous exceptions are not being generated by Tag Check Faults, this
 field controls whether on exception entry into EL1, all Tag Check Faults due to
 instructions executed before exception entry, that are reported asynchronously, are
 synchronized into TFSRE0_EL1 and TFSR_EL1 registers.
```

In section D13.2.114 (SCTLR_EL2, System Control Register (EL2)), in the ITFSB field (both fieldsets) the text that currently states:

```
When synchronous exceptions are not being generated by Tag Check Faults which
 are generated for Loads and Stores in EL0, EL1 or EL2, controls the auto-
synchronization of Tag Check Faults into TFSRE0_EL1, TFSR_EL1 and TFSR_EL2.
```

is replaced by:

```
When synchronous exceptions are not being generated by Tag Check Faults, this
 field controls whether on exception entry into EL2, all Tag Check Faults due to
 instructions executed before exception entry, that are reported asynchronously, are
 synchronized into TFSRE0_EL1, TFSR_EL1 and TFSR_EL2 registers.
```

D13.2.115 (SCTLR_EL3, System Control Register (EL3)), in the ITFSB field the text that currently states:

```
When synchronous exceptions are not being generated by Tag Check Faults, which
 are generated for Loads and Stores at any exception level, controls the auto-
synchronization of Tag Check Faults into TFSRE0_EL1 and TFSR_ELx.
```

is clarified to:

```
When synchronous exceptions are not being generated by Tag Check Faults, this
 field controls whether on exception entry into EL3, all Tag Check Faults due to
 instructions executed before exception entry, that are reported asynchronously, are
 synchronized into TFSRE0_EL1, and the TFSR_ELx registers.
```

2.29 C16676

In D7.11.3 (Common event numbers, Subsection Common microarchitectural events), in the descriptions of the following Performance Monitors events `0x4024` MEM_ACCESS_CHECKED, `0x4025` MEM_ACCESS_CHECKED_RD, `0x4026` MEM_ACCESS_CHECKED_WR counters, the following text is added:

```
It is IMPLEMENTATION DEFINED whether the counter increments on a Tag Checked
 access made when Tag Check Faults are configured to be ignored by SCTLR_ELx.TCF or
 SCTLR_ELx.TCF0
```

2.30 D16688

In sections D13.6.5 ('PMSCR_EL1, Statistical Profiling Control Register (EL1)') and D13.6.6 ('PMSCR_EL2, Statistical Profiling Control Register (EL2)'), in field PCT, the text for the 0b11 value that reads:

```
0b11 When FEAT_ECV is implemented: Physical counter, CNTPCT_EL0 minus CNTPOFF_EL2 is
 collected.
```

is replaced by:

```
0b11 When FEAT_ECV is implemented: The profiling timestamp is the physical counter
 value minus a physical offset. If any of the following are true, the physical
 offset value is zero.
Otherwise, the physical offset value is the value of CNTPOFF_EL2.
- CNTHCTL_EL2.ECV is 0b0.
- SCR_EL3.ECVEn is 0b0.
```

In sections D13.3.29 ('TRFCR_EL1, Trace Filter Control Register (EL1)') and D13.3.30 ('TRFCR_EL2, Trace Filter Control Register (EL2)'), in field TS, the text for the 0b10 value that reads:

```
0b10 When FEAT_ECV is implemented Guest Physical timestamp. The traced timestamp is
 the physical counter value, minus the value of CNTPOFF_EL2.
```

is replaced by:

```
0b10 When FEAT_ECV is implemented: The trace timestamp is the physical counter value
 minus a physical offset. If any of the following are true, the physical offset
 value is zero.
Otherwise, the physical offset value is the value of CNTPOFF_EL2.
- CNTHCTL_EL2.ECV is 0b0.
- SCR_EL3.ECVEn is 0b0.
```

In section G8.3.36 ('TRFCR, Trace Filter Control Register'), in field TS, in the 0b10 value description, the text that reads:

```
0b10 When FEAT_ECV is implemented Guest Physical timestamp. The traced timestamp is
 the physical counter value, minus the value of CNTPOFF_EL2.
```

is replaced by:

```
0b10 When FEAT_ECV is implemented: The trace timestamp is the physical counter value
 minus a physical offset. If any of the following are true, the physical offset
 value is zero.
Otherwise, the physical offset value is the value of CNTPOFF_EL2.
- EL3 is using AArch32.
- EL2 is enabled in the current security state and is using AArch32.
- CNTHCTL_EL2.ECV is 0b0.
- SCR_EL3.ECVEn is 0b0.
```

In all of the above fields, the following text is deleted:

```
If FEAT_ECV is implemented, and EL2 is implemented and enabled in the current
 Security state, the physical counter uses a fixed physical offset of zero if any of
 the following are true:
- CNTHCTL_EL2.ECV is 0.
- SCR_EL3.ECVEn is 0.
- HCR_EL2.{E2H, TGE} is {1, 1}.
```

In addition: - Details of the physical timestamp offset are also missing from section G3.3 ('Self-hosted trace timestamps'). These are added, to cover the case where tracing is used by EL1 using AArch32, but EL2 is using AArch64. - In section D9.6.8 ('Controlling the data that is collected'), where these conditions are repeated as part of the Statistical Profiling specification, the mentions of EL3 or EL2 using AArch32 are removed.

2.31 D16694

In section J1.1 (Pseudocode for AArch64 operation), the Pseudocode for Mem[] incorrectly treats all NV2REGISTER accesses as Big-endian when EL1 is configured for Big-endian operation, regardless of the value of EL2. This is fixed by passing the AccType parameter to BigEndian and removing the NV2REGISTER tests in Mem[]. All other calls to BigEndian in the manual are updated to pass an appropriate parameter.

The following code in Mem[] - non-assignment (read) form that reads as:

```
if (HaveNV2Ext() && acctype == AccType_NV2REGISTER && SCTLR_EL2.EE == '1') ||
 BigEndian() then
```

Is corrected to read:

```
if BigEndian(acctype) then
```

The following code in Mem[] - assignment (write) form that reads as:

```
if (HaveNV2Ext() && acctype == AccType_NV2REGISTER && SCTLR_EL2.EE == '1') ||
 BigEndian() then
```

Is corrected to read:

```
if BigEndian(acctype) then
```

In section J1.3 (Shared pseudocode), the code for BigEndian() that reads as:

```
boolean BigEndian()
    boolean bigend;
    if UsingAArch32() then
```

Is corrected to read:

```
boolean BigEndian(AccType acctype)
    boolean bigend;
    if HaveNV2Ext() && acctype == AccType_NV2REGISTER then
        bigend = SCTLR_EL2.EE == '1';
    elsif UsingAArch32() then
```

2.32 D16698

In D13.3.3 (DBGBVR<n>_EL1, Debug Breakpoint Value Registers, n = 0 - 15), in the fieldset 'When DBGBCR<n>_EL1.BT == 0b000x' the text in field RESS:

```
Reserved, Sign extended. Software must treat this field as RES0 if the most
 significant bit of VA is 0 or RES0, and as RES1 if the most significant bit of VA
 is 1.
It is IMPLEMENTATION DEFINED whether:
* Reads return the value of the most significant bit of the VA for every bit in this
 field.
* Reads return the last value written.
```

is replaced by:

```
Reserved, Sign extended. Software must set all bits in this field to the same value
 as the most significant bit of the VA field. If all bits in this field are not the
 same value as the most significant bit of the VA field, then all of the following
 apply:
* It is CONSTRAINED UNPREDICTABLE whether the PE ignores this field when comparing
 an address.
* If the breakpoint is not context-aware, it is IMPLEMENTATION DEFINED whether the
 value read back in each bit of this field is a copy of the most significant bit of
 the VA field or the value written.
```

In D13.3.12 (DBGWVR<n>_EL1, Debug Watchpoint Value Registers, n = 0 - 15), the text in field RESS:

```
Reserved, Sign extended. Hardware and software must treat this field as RES0 if the
 most significant bit of VA is 0 or RES0, and as RES1 if the most significant bit of
 VA is 1.
Hardware always ignores the value of these bits and it is IMPLEMENTATION DEFINED
 whether:
* The bits are hardwired to a copy of the most significant bit of VA, meaning writes
 to these bits are ignored, and reads to the bits always return the hardwired value.
* The value in those bits can be written, and reads will return the last value
 written. The value held in those bits is ignored by hardware.
```

is replaced by:

```
Reserved, Sign extended. Software must set all bits in this field to the same value
 as the most significant bit of the VA field. If all bits in this field are not the
 same value as the most significant bit of the VA field, then all of the following
 apply:
* It is CONSTRAINED UNPREDICTABLE whether the PE ignores this field when comparing
 an address.
```

```
* If the breakpoint is not context-aware, it is IMPLEMENTATION DEFINED whether the
 value read back in each bit of this field is a copy of the most significant bit of
 the VA field or the value written.
```

In both DBGBVR<n>_EL1 and DBGWVR<n>_EL1, in the VA[48:2] field description, the text:

```
When FEAT_LVA is implemented, VA[52:49] forms the upper part of the address value.
 Otherwise, VA[52:49] are RESS.
```

is replaced by:

```
When FEAT_LVA is implemented, VA[52:49] forms the upper part of the address value.
 Otherwise, bits [52:49] are part of the RESS field.
```

In section J1.1 (Pseudocode for AArch64 operation), the Pseudocode function AArch64.BreakpointMatch() does not describe the CONSTRAINED UNPREDICTABLE case when matching on the top bits of a given virtual address. The code that reads:

```
top = AddrTop(vaddress, TRUE, PSTATE.EL);
BVR_match = vaddress<top:2> == DBGBVR_EL1[n]<top:2> && byte_select_match;
```

Is updated to read:

```
// If the DBGBVR<n>_EL1.RESS fields are not a sign extension of the MSB
// of DBGBVR<n>_EL1.VA, it is UNPREDICTABLE whether they appear to be
// included in the match.
// If 'vaddress' is outside of the current virtual address space, then the access
// generates a Translation fault.
top = (if Have52BitVAExt() then 52 else 48);
if !IsOnes(DBGBVR_EL1[n]<63:top>) && !IsZero(DBGBVR_EL1[n]<63:top>) then
    if ConstrainUnpredictableBool() then top = AddrTop(vaddress, TRUE,
 PSTATE.EL); // Unpredictable case
BVR_match = (vaddress<top:2> == DBGBVR_EL1[n]<top:2> && byte_select_match);
```

A similar change is made in AArch64.WatchpointByteMatch(). The code that reads:

```
el = if HaveNV2Ext() && acctype == AccType_NV2REGISTER then EL2 else PSTATE.EL;
top = AddrTop(vaddress, FALSE, el);
bottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3;           // Word or doubleword
mask = UInt(DBGWCR_EL1[n].MASK);
```

Is updated to read:

```
el = if HaveNV2Ext() && acctype == AccType_NV2REGISTER then EL2 else PSTATE.EL;
top = (if Have52BitVAExt() then 52 else 48);
if !IsOnes(DBGBVR_EL1[n]<63:top>) && !IsZero(DBGBVR_EL1[n]<63:top>) then
    if ConstrainUnpredictableBool() then top = AddrTop(vaddress, TRUE,
 PSTATE.EL); // Unpredictable case
bottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3;           // Word or doubleword
mask = UInt(DBGWCR_EL1[n].MASK);
```

2.33 R16700

The architecture has been relaxed to allow an implementation to change behavior on an access by access basis. To this end, the following text is added to section D6.7 ('PE handling of Tag Check Fault'):

```
It is CONSTRAINED UNPREDICTABLE whether the FFR element associated with the read of
 an Active element in an SVE Non-fault load, or an Active element which is not the
 first Active element in an SVE First-fault load, R2 , to location X , is set to
 FALSE if all of the following are true:
- Tag check faults are configured as asynchronous for both reads and writes.
- A read or write RW1 to location Y causes a tag check fault.
- Tag check faults for locations X and Y are reported in the same status bit
 TFSR(E0)_ELx.TFy.
- RW1 is in program order before R2, or is the first Active element in the first-
fault load instruction causing R2.
- There are no other faults caused by R2 that are reported in FFR.
- There is not a DSB and a direct write of 0b0 to that status bit appearing in
 program order between the instruction causing RW1 and the instruction causing R2
```

2.34 D16704

In section J1.1 (Pseudocode for AArch64 operation), within the Pseudocode function CollectRecord(), the check for UNPREDICTABLE cases does not account for the mask bits.

The code that reads:

```
if ((PMSFCR_EL1.FE == '1' && !IsZero(PMSEVFR_EL1)) ||
    (PMSFCR_EL1.FT == '1' && !IsZero(PMSFCR_EL1.<B,LD,ST>)) ||
    (PMSFCR_EL1.FL == '1' && !IsZero(PMSLATFR_EL1.MINLAT))) then
    return ConstrainUnpredictableBool();
```

Is updated to read:

```
if ((PMSFCR_EL1.FE == '1' && IsZero(PMSEVFR_EL1 AND mask)) ||
    (PMSFCR_EL1.FT == '1' && IsZero(PMSFCR_EL1.<B,LD,ST>)) ||
    (PMSFCR_EL1.FL == '1' && IsZero(PMSLATFR_EL1.MINLAT))) then
    return ConstrainUnpredictableBool();
```

2.35 D16707

In Section D7.11.3 (Common event numbers), subsection 'Common architectural events', in the 0x811E BR_INDNR_RETIRED event description, the text that reads:

```
The counter counts the instructions on the architecturally executed path counted by
 BR_RETIRED, but not counted by BR_RETURN_ANY_RETIRED. These are branch instructions
 but does not include returns or immediate instructions.
```

is corrected to:

```
The counter counts the instructions on the architecturally executed path counted
 by BR_IND_RETIRED, but not counted by BR_RETURN_ANY_RETIRED. These are branch
 instructions but does not include returns or immediate instructions.
```

2.36 D16708

In section J1.1 (Pseudocode for AArch64 operation), the function AArch64.TranslateAddressS1Off() did not always initialize variables.

The code that reads:

```
TLBRecord result;
```

Is corrected to read:

```
TLBRecord result;
result.descupdate.AF = FALSE;
result.descupdate.AP = FALSE;
```

Also, the code that reads:

```
result.descupdate.AF = FALSE;
result.descupdate.AP = FALSE;
result.descupdate.descaddr = result.addrdesc;
```

Is corrected to read:

```
result.descupdate.descaddr = result.addrdesc;
```

2.37 C16714

In section D5.10.2 (TLB maintenance instructions), in the subsection 'TLB range maintenance instructions', the line that reads

```
All TLB range maintenance instructions invalidate TLB entries that are within the
 address range determined by the formula
```

is clarified to read

```
All TLB range maintenance instructions invalidate TLB entries translating addresses
 that are within the address range determined by the formula
```

2.38 D16732

In I5.8.25 ERR<n>MISC0, I5.8.26 ERR<n>MISC1, I5.8.27 ERR<n>MISC2, I5.8.28, ERR<n>MISC3 registers, the 'Purpose', text that reads:

```
The miscellaneous syndrome registers might contain:
*  Information to identify the FRU in which the error was detected, and might
 contain enough information to locate the error within that FRU.
```

is replaced with:

```
The miscellaneous syndrome registers might contain:
* Information to locate where the error was detected.
* If the error was detected within an FRU, the identity of the FRU.
```

2.39 D16736

In K1.2.6 (The Performance Monitors Extension), the sub-section '**CONSTRAINED UNPREDICTABLE** behavior caused by MDCR_EL2.HPMN', the text that reads:

```
If MDCR_EL2.HPMN is set to 0, or to a value larger than PMCR_EL0.N, then the
 following CONSTRAINED UNPREDICTABLE behavior applies:
```

is corrected to:

```
If PMCR_EL0.N is nonzero, and MDCR_EL2.HPMN is set to 0 or to a value larger than
 PMCR_EL0.N, then the following CONSTRAINED UNPREDICTABLE behavior applies:
```

The equivalent constraint for AArch32 in K1.1.17 (The Performance Monitors Extension,) sub-section '**CONSTRAINED UNPREDICTABLE** behavior caused by HDCR.HPMN' is also updated.

2.40 D16737

In section J1.2 (Pseudocode for AArch32 operation), the Pseudocode function AArch32.TranslationTableWalkSD() has an incorrect check for the availability of second stage translation.

The code that reads:

```
if !HaveEL(EL2) || (IsSecure() && !IsSecureEL2Enabled()) then
    // if only 1 stage of translation
```

```
    l1descaddr2 = l1descaddr;
```

is updated to read:

```
if !HasS2Translation() then
    // if only 1 stage of translation
    l1descaddr2 = l1descaddr;
```

2.41 D16745

The following statement is added to section A2.7.1 ('Architectural features added by Armv8.4'), in the description for 'FEAT_Debugv8p4, Debug v8.4':

```
This feature is mandatory if FEAT_SEL2 is implemented.
```

2.42 D16753

In section G8.2.33 (CPSR, Current Program Status Register), in the DIT field, under the bullet beginning 'A subset of those instructions which use the SIMD&FP register file', the following instructions are removed: VABS, VACGE, VACGT, VACLE, VACLT, VCGE, VCGT, VCLE, VCLT, VCMP, VCMPE, VSELEQ, VSELGE, VSELGT, VSELVS, and VNEG. In addition, the VABD* instructions gain an '(integer)' qualifier, and a new bullet and list of instructions are added:

```
- Another subset of the instructions that use the SIMD&FP register file. For these
 instructions, the effects of CPSR.DIT apply only if they pass their condition
 execution check and apply only when the instructions are operating on integer
 vector elements. These instructions are:
-- VABS, VCGE, VCGT, VCLE, VCLT, VMLA (by scalar), VMLS (by scalar), and VNEG.
```

The equivalent changes are made to the 'Operational information' sections of all of the instruction definitions mentioned in this item.

2.43 D16761

In section F4.1.18 (Unconditional instructions), the op0 field is extended from covering bits[26:25] to bits[26:24].

The table lines:

op0	op1	Decode group or instruction page
00	x	Miscellaneous
01	x	Advanced SIMD data-processing
1x	1	Memory hints and barriers
10	0	Advanced SIMD element or structure load/store
11	0	UNALLOCATED

are updated to:

op0	op1	Decode group or instruction page
00x	x	Miscellaneous
01x	x	Advanced SIMD data-processing
1xx	1	Memory hints and barriers
100	0	Advanced SIMD element or structure load/store
101	x	UNALLOCATED
11x	0	UNALLOCATED

2.44 D16762

In section F3.1.7 (System register access, Advanced SIMD, and floating-point), the 'op2' field is extended to cover bits 11 and 10. The updated table is:

op0	op1	op2	op3	Instruction details
-	0x	0x	-	UNALLOCATED
-	10	0x	-	UNALLOCATED
-	11	-	-	Advanced SIMD data-processing
0	0x	1x	-	Advanced SIMD and System register load/store and 64-bit move

op0	op1	op2	op3	Instruction details
0	10	10	0	Floating-point data-processing
0	10	11	0	UNALLOCATED
0	10	1x	1	Advanced SIMD and System register 32-bit move
1	!= 11	1x	-	Additional Advanced SIMD and floating-point instructions

The equivalent changes are made in F4.1.12 (System register access, Advanced SIMD, floating-point, and Supervisor call), where the equivalent field is named 'op1', and the resulting table is:

cond	op0	op1	op2	Instruction details
-	0x	0x	-	UNALLOCATED
-	10	0x	-	UNALLOCATED
-	11	-	-	Supervisor call
1111	!= 11	1x	-	Unconditional Advanced SIMD and floating-point instructions
!= 1111	0x	1x	-	Advanced SIMD and System register load/store and 64-bit move
!= 1111	10	10	0	Floating-point data-processing
!= 1111	10	11	0	UNALLOCATED
!= 1111	10	1x	1	Advanced SIMD and System register 32-bit move

2.45 D16763

In section F4.1.19 (Miscellaneous), sub-section 'Change Process State', the table is incomplete and it does not identify that the values of I and F must be 0 for SETEND. Otherwise, the encoding is **UNDEFINED**.

2.46 D16766

In section D13.2.47 (HCR_EL2, Hypervisor Configuration Register), the NV1 field description under the condition 'When FEAT_NV2 is implemented' that incorrectly reads:

```
If HCR_EL2.{NV, NV1, NV2} are {0, 1, 0}, then the behavior is a CONSTRAINED
 UNPREDICTABLE choice of:
* Behaving as if HCR_EL2.NV is 1 and HCR_EL2.NV1 is 1 for all purposes other than
 reading than reading back the value of the HCR_EL2.NV bit.
* Behaving as if HCR_EL2.NV is 0 and HCR_EL2.NV1 is 0 for all purposes other than
 reading than reading back the value of the HCR_EL2.NV1 bit.
* Behaving with regard to the HCR_EL2.NV and HCR_EL2.NV1 bits behavior as defined in
 the rest of this description.
```

is corrected to:

```
If HCR_EL2.{NV, NV1} are {0, 1}, then the behavior is a CONSTRAINED UNPREDICTABLE
 choice of:
* Behaving as if HCR_EL2.NV is 1 and HCR_EL2.NV1 is 1 for all purposes other than
 reading than reading back the value of the HCR_EL2.NV bit.
* Behaving as if HCR_EL2.NV is 0 and HCR_EL2.NV1 is 0 for all purposes other than
 reading than reading back the value of the HCR_EL2.NV1 bit.
* Behaving with regard to the HCR_EL2.NV and HCR_EL2.NV1 bits behavior as defined in
 the rest of this description.
```

2.47 D16767

In the Glossary, the following text is added to the definition of Speculative:

```
* Memory effects (M2) generated by load, store or barrier instructions (LSB2)
 appearing in program order after load, store or barrier instructions LSB1) that
 generate memory effects (M1)  where all the following apply:
- M1 is locally-ordered-before M2.
- LSB1 has not been executed before LSB2.
```

2.48 D16769

In D1.14.3 (EL2 configurable controls), the title of the subsection 'Traps to EL2 of System register accesses to the trace registers' is updated to 'Traps to EL2 of EL2, EL1, and EL0 System register accesses to the trace registers'.

And the text:

```
CPTR_EL2.TTA traps System register accesses to the trace registers to EL2.
```

is updated to:

```
CPTR_EL2.TTA traps EL2, EL1 and EL0 System register accesses to the trace registers
 to EL2.
```

In D1.14.4 (EL3 configurable controls), Table D1-25, the row for MDCR_EL3.TTRF has a cross-reference to the subsection 'Traps to EL3 of System register accesses to the trace registers'. This cross-reference is updated to 'Traps to EL3 of all System register accesses to the filter trace control registers'. Note: the title of this subsection is changed later in this writeup.

In D1.14.4 (EL3 configurable controls), subsection 'Traps to EL3 of System register accesses to the trace registers', the row 'For EL0 and EL1, this trap control applies to accesses from both Security states.' is removed.

In D1.14.4 (EL3 configurable controls), the title of the subsection 'Traps to EL3 of all System register accesses to the filter trace control registers' is updated to 'Traps to EL3 of EL2 and EL1 System register accesses to the trace filter control registers'.

The line 'For EL0 and EL1, this trap control applies to accesses from both Security states.' is removed.

And the text:

```
MDCR_EL3.TTRF traps System register accesses to the trace registers, from all
 Exception levels, to EL3.
```

is updated to:

```
MDCR_EL3.TTRF traps System register accesses to the trace filter registers, from EL1
 and EL2, to EL3.
```

In D13.2.36 (ESR_EL1, Exception Syndrome Register (EL1)), D13.2.37 (ESR_EL2, Exception Syndrome Register (EL2)), and D13.2.38 (ESR_EL3, Exception Syndrome Register (EL3)), in the descriptions of the ISS encoding for an exception from MSR, MRS, or System instruction execution in AArch64 state, the phrase:

```
filter trace control registers
```

is replaced by:

```
trace filter control registers
```

2.49 R16773

In section D10.2.1 ('Address packet'), in the subsection 'Address packet payload', the following text is added to the CH bit description:

```
It is IMPLEMENTATION DEFINED whether this bit is 1 or 0 on a Tag Checked access
 made when Tag Check Faults are configured to be ignored by SCTLR_ELx.TCF or
 SCTLR_ELx.TCF0
```

2.50 D16774

In section A2.9.1 (Architectural features added by Armv8.6), the text that reads:

```
FEAT_FGT introduces additional traps to EL2 of EL1 and EL0 access to individual or
 small groups of System registers and instructions.
```

is changed to read:

```
FEAT_FGT introduces additional traps to EL2 of EL1 and EL0 access to individual or
 small groups of System registers and instructions, and traps to EL3 and EL2 of the
 Debug Communications Channel registers.
```

In section D13.2.61 (ID_AA64MMFR0_EL1, AArch64 Memory Model Feature Register 0), the text in the FGT field description that reads:

```
Indicates presence of the Fine-Grained Trap controls:
* HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGRTR_EL2, HFGITR_EL2 and HFGWTR_EL2
 registers, and their associated traps.
* MDCR_EL2.TDCC and MDCR_EL3.TDCC.
* SCR_EL3.FGTEn.
```

is corrected to read:

```
Indicates presence of the Fine-Grained Trap controls:
* If EL2 is implemented, the HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGRTR_EL2,
 HFGITR_EL2 and HFGWTR_EL2 registers, and their associated traps.
* If EL2 is implemented, MDCR_EL2.TDCC.
* If EL3 is implemented, MDCR_EL3.TDCC.
* If both EL2 and EL3 are implemented, SCR_EL3.FGTEn.
```

In section D13.2.112 (SCR_EL3, Secure Configuration Register), in the description of the FGTEn field, the start of both value descriptions are changed from:

```
EL2 Accesses to ...
```

to the following:

```
When EL2 is implemented, EL2 accesses to...
```

Additionally, the following Note is added to the field description:

```
If EL2 is not implemented but EL3 is implemented, FEAT_FGT implements the
 MDCR_EL3.TDDC traps.
```

2.51 D16776

In section F6.1 (T32 and A32 Advanced SIMD and Floating-point Instruction Descriptions), the Decode Pseudocode for the VRECPE and VRSQRTE instructions incorrectly imply that the Half precision extensions introduce a 16-bit integer reciprocal estimate or reciprocal square root estimate.

The code that reads:

```
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
```

Is corrected to read:

```
if (size == '01' && (!HaveFP16Ext() || F=='0')) || size IN {'00', '11'} then
 UNDEFINED;
```

2.52 D16778

In section E1.2.4 (Process state, PSTATE), in the subsection 'Use of PSTATE.IT', the following sentence is removed:

```
For performance reasons, Armv8 deprecates the use of IT other than with a single 16-
bit T32 instruction from a specified subset of the 16-bit T32 instructions, see ...
```

In addition, the text that reads:

```
In addition, implementations can provide a set of ITD control fields, SCTLR.ITD,
 SCTLR_EL1.ITD, and HSCTLR.ITD, to disable these deprecated uses, making them
 UNDEFINED.
```

is updated to

```
Implementations can provide a set of ITD control fields, SCTLR.ITD, SCTLR_EL1.ITD,
 and HSCTLR.ITD, to disable use of IT for some instructions, making them UNDEFINED.
```

The section F1.8.2 (Partial deprecation of IT) is now redundant and deleted.

2.53 D16779

In Section J1.1 (Pseudocode for AArch64 operation), the Pseudocode function AArch64.FaultSyndrome() does not correctly describe the Data Abort ISS of ESR_ELx.ISV when RAS is not implemented.

The code that reads:

```
if IsSecondStage(fault) && !fault.s2fs1walk then
    iss<24:14> = LSInstructionSyndrome();
```

Is updated to read:

```
if (IsSecondStage(fault) && !fault.s2fs1walk && (!IsExternalSyncAbort(fault) ||
    (!HaveRASExt() && fault.acctype == AccType_PTW &&
    boolean IMPLEMENTATION_DEFINED \"ISV on second stage page table walk\"))) then
    iss<24:14> = LSInstructionSyndrome();
```

A similar change is made in AArch32.FaultSyndrome() in section J1.2 (Pseudocode for AArch32 operation).

The code that reads:

```
if IsSecondStage(fault) && !fault.s2fs1walk then
    iss<24:14> = LSInstructionSyndrome();
```

Is updated to read:

```
if (IsSecondStage(fault) && !fault.s2fs1walk && (!IsExternalSyncAbort(fault) ||
    (!HaveRASExt() && fault.acctype == AccType_PTW &&
    boolean IMPLEMENTATION_DEFINED \"ISV on second stage page table walk\"))) then
    iss<24:14> = LSInstructionSyndrome();
```

Also in D13.2.36 (ESR_EL1, Exception Syndrome Register (EL1), D13.2.37 (ESR_EL2, Exception Syndrome Register (EL2)), D13.2.38 (ESR_EL3, Exception Syndrome Register (EL3)), under 'ISS encoding for an exception from a Data Abort', the description for ISV, bit[24], that reads:

```
When the RAS Extension is implemented, ISV is 0 for any synchronous External abort.
For ISS reporting, a stage 2 abort on a stage 1 translation table walk does not
 return a valid instruction syndrome, and therefore ISV is 0 for these aborts.
```

```
When the RAS Extension is not implemented, the value of ISV on a synchronous
 External abort on a stage 2 translation table walk is IMPLEMENTATION DEFINED.
```

is replaced with

```
When the RAS Extension is implemented, ISV is 0 for any synchronous External abort.
For ISS reporting, a stage 2 abort on a stage 1 translation table walk does not
 return a valid instruction syndrome, and therefore ISV is 0 for these aborts.
When the RAS Extension is not implemented, it is IMPLEMENTATION DEFINED whether ISV
 is set to 1 or 0 on a synchronous External abort on a stage 2 translation table
 walk.
```

2.54 D16780

In section J1.2.2 (aarch32/exceptions), the function AArch32.Abort() incorrectly uses HCR2 in an expression in which EL2 is using AArch64.

The code that reads:

```
if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
    route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
                        (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault))
 ||
                        (IsDebugException(fault) && MDCR_EL2.TDE == '1'));
```

Is corrected to read:

```
if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
    route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
                        (HaveRASExt() && HCR_EL2.TEA == '1' &&
 IsExternalAbort(fault)) ||
                        (IsDebugException(fault) && MDCR_EL2.TDE == '1'));
```

2.55 D16792

In section J1.1 (Pseudocode for AArch64 operation), in the Pseudocode function AArch64.ExceptionReturn(), the presence of SynchroniseContext() at the start of the function implies that changes to the SCTLR_ELx.IESB and other registers involved in error synchronisation, that have not been context synchronised before the Exception Return, must be context synchronised.

The code that reads:

```
AArch64.ExceptionReturn(bits(64) new_pc, bits(64) spsr)
    SynchronizeContext();

    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
```

```
    if HaveDoubleFaultExt() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' &&
 PSTATE.EL == EL3);
    if sync_errors then
        SynchronizeErrors();
        iesb_req = TRUE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
    // Attempts to change to an illegal state will invoke the Illegal Execution
 state mechanism
    bits(2) source_el = PSTATE.EL;
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();
```

Is updated to read:

```
AArch64.ExceptionReturn(bits(64) new_pc, bits(64) spsr)
    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
    if HaveDoubleFaultExt() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' &&
 PSTATE.EL == EL3);
    if sync_errors then
        SynchronizeErrors();
        iesb_req = TRUE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

    SynchronizeContext();

    // Attempts to change to an illegal state will invoke the Illegal Execution
 state mechanism
    bits(2) source_el = PSTATE.EL;
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();
```

2.56 C16796

In section D5.10.1 (General TLB maintenance requirements), in the subsection 'Using break-before-make when updating translation table entries', the bullet that reads:

```
* A change of the memory type.
```

is clarified to read:

```
* A change of the memory type, including shareability.
```

And a note is added :

```
Note: Changes to the OA include changing between Secure and Non-secure output
 addresses.
```

The equivalent edit is made for AArch32 in section G5.9.1 (General TLB maintenance requirements), in the sub-section 'Using break-before-make when updating translation table entries'.

2.57 D16804

In section J1.3 (Shared pseudocode) the Pseudocode function Halt() incorrectly assigns to DSPSR instead of the spsr variable.

As such, the code that reads:

```
if (HaveBTIExt() &&
    !(reason IN {DebugHalt_Step_Normal, DebugHalt_Step_Exclusive,
                 DebugHalt_Step_NoSyndrome,
                 DebugHalt_Breakpoint, DebugHalt_HaltInstruction}) &&
    ConstrainUnpredictableBool(Unpredictable_ZEROBTYPE)) then
    DSPSR<11:10> = '00';

if UsingAArch32() then
    DLR = preferred_restart_address<31:0>;
    DSPSR = spsr;
else
    DLR_EL0 = preferred_restart_address;
    DSPSR_EL0 = spsr;
```

Is corrected to read:

```
if (HaveBTIExt() &&
    !(reason IN {DebugHalt_Step_Normal, DebugHalt_Step_Exclusive,
                 DebugHalt_Step_NoSyndrome,
                 DebugHalt_Breakpoint, DebugHalt_HaltInstruction}) &&
    ConstrainUnpredictableBool(Unpredictable_ZEROBTYPE)) then
    if UsingAArch32() then
        spsr_32<11:10> = '00'
    else
        spsr_64<11:10> = '00'

if UsingAArch32() then
    DLR = preferred_restart_address<31:0>;
    DSPSR = spsr;
else
    DLR_EL0 = preferred_restart_address;
    DSPSR_EL0 = spsr;
```

2.58 D16816

In section J1.3 (Shared pseudocode), the function InterruptPending() did not account for all sources of virtual interrupts.

The code that reads:

```
pending_physical_interrupt = (IRQPending() || FIQPending() ||
 IsPhysicalSErrorPending());
pending_virtual_interrupt = !IsInHost() && ((HCR_EL2.<VSE,VI,VF> AND
HCR_EL2.<AMO,IMO,FMO>) != '000');
```

Is corrected to read:

```
bit vIRQstatus = (if (VirtualIRQPending()) then '1' else '0') OR HCR_EL2.VI;
bit vFIQstatus = (if (VirtualFIQPending()) then '1' else '0') OR HCR_EL2.VF;
bits(3) v_interrupts = (HCR_EL2.VSE : vIRQstatus : vFIQstatus);

pending_physical_interrupt = (IRQPending() || FIQPending() ||
 IsPhysicalSErrorPending());
pending_virtual_interrupt = !IsInHost() && ((v_interrupts AND
 HCR_EL2.<AMO,IMO,FMO>) != '000');
```

2.59 D16825

In section D7.11.3 (Common event numbers), in the `0x811D`, BR_IND_RETIRED event description, the text that reads:

```
These are all branch instructions that are not immediate,
```

is corrected to:

```
These are all branch instructions that are not immediate branch instructions.
```

2.60 D16826

In section D5.2.5 (Translation tables and the translation process), the sub-section 'Ordering of memory accesses from translation table walks', the text that reads:

```
Any writes to the translation tables are not observed by the translation table walks
 of an explicit memory access generated by a load or store that occurs in program
 order before the instruction that performs the write to the translation tables
```

is replaced with

```
Any writes to the translation tables are not seen by any translation table accesses
 associated with an explicit memory access generated by a load or store that occurs
 in program order before the instruction that performs the write to the translation
 tables.
```

2.61 D16835

In section J1.1.2 (aarch64/exceptions), the function AArch64.FPTrappedException() is passed an explicit argument 'element'.

The function signature that reads:

```
AArch64.FPTrappedException(boolean is_ase, integer element, bits(8)
 accumulated_exceptions)
```

Is updated to read:

```
AArch64.FPTrappedException(boolean is_ase, bits(8), accumulated_exceptions)
```

Consequently, the call to AArch64.FPTrappedException() from AArch32.FPTrappedException() is updated to match this change.

The code that reads:

```
if AArch32.GeneralExceptionsToAArch64() then
    is_ase = FALSE;
    element = 0;
    AArch64.FPTrappedException(is_ase, element, accumulated_exceptions);
```

Is updated to read:

```
if AArch32.GeneralExceptionsToAArch64() then
    is_ase = FALSE;
    AArch64.FPTrappedException(is_ase, accumulated_exceptions);
```

2.62 R16836

In section D9.6.5 (Additional information for each profiled Scalable Vector Extension operation), the 'Sampled SVE operation' definition that reads:

```
Means an instruction or micro-operation defined by the 'Arm Architecture Reference
 Manual Supplement: The Scalable Vector Extension (SVE), for Armv8-A' and sampled by
 the Statistical Profiling Extension that has a vector or a predicate as an input or
 output. This includes instructions
with scalar outputs, but excludes the Non-SIMD SVE instructions.
```

is relaxed to read:

```
If an implementation samples micro-operations, then it is IMPLEMENTATION DEFINED,
 and might vary between operation types, whether an operation for which all the
 following are true is treated as a Sampled SVE operation or the equivalent Advanced
 SIMD operation:
* The Accessible vector length is 128 bits.
```

```
* The operation is unpredicated, and does not have a predicate register as an input
 or output.
* The operation has an equivalent Advanced SIMD operation.

This includes SVE load and store operations where an equivalent Advanced SIMD
 operation is defined.
```

2.63 R16841

The architecture is relaxed to permit FEAT_RASv1p1 to be implemented from Armv8.2. That is, the features identified by ID_AA64PFR1_EL1.RAS_frac = 0b0001.

2.64 R16853

It is made **CONSTRAINED UNPREDICTABLE** whether traps on the following registers are ignored in Debug state: * AArch64: MDCCSR_EL0, OSDTRRX_EL1, OSDTRTX_EL1, MDCCINT_EL1. * AArch32: DBGDSCRint, DBGDIDR, DBGDSAR, DBGDRAR, DBGDTRRXext, DBGDTRTXext, DBGDCCINT.

This relaxation does not affect the lowest Exception levels these registers can be accessed at. The following registers are **UNDEFINED** at EL0: * AArch64: OSDTRRX_EL1, OSDTRTX_EL1, MDCCINT_EL1. * AArch32: DBGDTRRXext, DBGDTRTXext, DBGDCCINT.

2.65 D16854

In Table D12-2 (System instruction encodings for non-Debug System register accesses), the following missing registers are added:

```
ID_ISAR6_EL1 (op0 = 3, op1 = 0, CRn = 0, CRm = 2, op2 = 7)
ID_DFR1_EL1 (op0 = 3, op1 = 0, CRn = 0, CRm = 3, op2 = 5)
```

In Table G7-3 (VMSAv8-32 (coproc==0b1111) register summary, in MCR/MRC parameter order), the following missing registers are added:

```
ID_ISAR6 (CRn = c0, opc1 = 0, CRm = 2, opc2 = 7)
ID_DFR1 (CRn = c0, opc1 = 0, CRm = 3, opc2 = 5)
```

2.66 C16855

System instructions added by FEAT_MTE that access tags in memory, including DC ZVA and DC GZVA, are treated the same as instructions that access data in memory. To this end, in section D6.5 (PE access to Allocation Tags), the text:

```
An instruction that loads or stores an Allocation Tag:
- Is considered a load or store of data to each location associated with the
 Allocation Tag for the purpose of triggering Watchpoints and PMU events, other than
 for events which count bytes of data transferred.
```

is extended to read

```
An instruction that loads or stores an Allocation Tag:
- Is considered a load or store of data to each location associated with the
 Allocation Tag for the purpose of triggering Watchpoints and PMU events, other than
 for events which count bytes of data transferred.
- Is treated as a load or store for the purpose of Statistical profiling.
```

2.67 D16864

In section B2.3.1 (Basic definitions), the following definitions are added:

```
Tag-read
A Tag-read is a read of a Tag location generated by an LDG instruction.

Tag-write
A Tag-write is a write of a Tag location generated by an STG instruction.

Tag-Check-read
A Tag-Check-read is a read of a Tag location which is generated by a checked memory
 access. All other reads and writes are considered Data accesses.

Tag-Location-Ordered
Two Tag-Check-reads R1 and R2 are Tag-Location-Ordered if and only if all the
 following apply
- R1 is Tag-ordered-before a Checked data access RW3.
- R2 is Tag-ordered-before a Checked data access RW4.
- RW3 and RW4 are to the same location.
```

In addition, section B2.3.5 (Internal visibility requirement) is altered as follows:

```
For a Data- or Tag-read or a Data- or Tag write (RW1) that appears in program order
 before a Data- or Tag-read or a Data- or Tag-write (RW2) to the same location
 or two Tag-Check-reads R1 and R2 which are Tag-Location-Ordered, the internal
 visibility requirement requires that exactly one of the following statements is
 true:
- RW2 is a write (W2) that is coherence-after RW1.
- RW1 is a write (W1), RW2 is a read (R2) and either:
 -- R2 reads-from W1.
 -- R2 reads-from a write that is coherence-after W1.
- RW1 and RW2 are both reads (R1, R2), R1 reads-from a write (W3) and one of:
 -- R2 reads-from W3.
 -- R2 reads-from a write that is coherence-after W3.
```

2.68 C16873

In section D10.2.1 (Address packet), in the 'NS, byte 7 bit [7], when Instruction virtual address' description, the following Note is deleted:

```
For an Exception Return, the Security state at the target of the branch might be
 different to the Security state the instruction was executed in.
```

2.69 D16875

In section J1.3 (Shared Pseudocode), the Pseudocode function AArch64.CheckSystemAccess() is redundant as the information that this function describes is actually present as the accessibility Pseudocode for system registers. This function definition will therefore be removed. In section C6.2 the decode Pseudocode for A64 system instructions MRS, MSR (immediate), MSR (register), SYS, SYSL, are updated to remove the call to this function.

2.70 D16882

In section B2.3.1 (Basic definitions), in the definition of 'Memory effects', the following Note is added:

```
Note
Tag-Check-reads are read memory effects for the purpose of this specification.
```

In section B2.3.2 (Dependency definitions), in the definition of 'Address dependency', the following Note is added:

```
Note
An Address Dependency exists from a read R1 to a Tag-Check-read R2 if and only if
 there is a Dependency through registers from R1 to the address part of R2.
```

In section B2.3.3 (Ordering relations), in the definition of 'Barrier-ordered-before', the text that reads:

```
- RW1 is a write (W1) generated by an instruction with Release semantics and RW2 is
 a read (R2) generated by an instruction with Acquire semantics
[...]
- RW1 is a read (R1) generated by an instruction with Acquire or AcquirePC semantics
```

is corrected to read:

```
- RW1 is a write (W1) generated by an instruction with Release semantics and RW2
 is a read (R2), except a Tag-Check-read, generated by an instruction with Acquire
 semantics
[...]
- RW1 is a read (R1), except a Tag-Check-read, generated by an instruction with
 Acquire or AcquirePC semantics
```

2.71 D16888

In section B2.3.11 (Limited ordering regions), the line that currently reads:

```
The LORegion descriptors are programmed using the LORSA_EL1, LOREA_EL1, LORN_EL1,
 and LORC_EL1 registers in the System register space.
```

is augmented to read:

```
The LORegion descriptors are programmed using the LORSA_EL1, LOREA_EL1, LORN_EL1,
 and LORC_EL1 registers in the System register space. These registers only describe
 memory addresses in the Non-secure memory map. These registers are UNDEFINED if
 accessed when SCR_EL3.NS==0
```

2.72 D16889

In section J1.1, the Pseudocode functions AArch64.SecondStageTranslate() and AArch32.SecondStageTranslate() can incorrectly generate an Alignment Fault if the translation walk table entry is determined to have memory type attribute set to Device.

The code that reads:

```
// Check for unaligned data accesses to Device memory
if ((!wasaligned && acctype != AccType_IFETCH) || (acctype == AccType_DCZVA))
    && S2.addrdesc.memattrs.memtype == MemType_Device && !IsFault(S2.addrdesc) then
    S2.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);
```

Is updated to read:

```
// Check for unaligned data accesses to Device memory
if ((!wasaligned && acctype != AccType_IFETCH) || (acctype == AccType_DCZVA))
    && S2.addrdesc.memattrs.memtype == MemType_Device && !IsFault(S2.addrdesc) && !
s2fs1walk then
    S2.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);
```

2.73 D16891

In section D1.12.6 ('Floating-point exceptions and exception traps'), the text that reads:

```
When the execution of separate operations in separate SIMD elements causes multiple
 floating-point exceptions, the ESR_ELx reports one exception associated with one
 element that the instruction uses. The architecture does not specify which element
 is reported, however the element that is reported is identified in the ESR_ELx.
```

is updated to:

```
When the execution of separate operations in separate SIMD elements causes multiple
 floating-point exceptions, the ESR_ELx reports only the exceptions associated with
 one element that the instruction uses. The architecture does not specify which
 element is reported.
```

In addition, in the subsection 'Combinations of floating-point exceptions', the following Note is deleted:

```
An implementation might provide information about a lower priority or untrapped
 floating-point exceptions in an IMPLEMENTATION DEFINED way, for example using an
 IMPLEMENTATION DEFINED register.
```

2.74 D16892

In section J1.3 (Shared pseudocode), the Pseudocode function GetPSRFromPSTATE() does not copy PSTATE.SS to SPSR_ELx<21> on an exception taken from AArch32 to AArch64. The code that reads:

```
bits(32) GetPSRFromPSTATE()
    bits(32) spsr = Zeros();
    …
    if PSTATE.nRW == '1' then // AArch32 state
        if HaveDITExt() then spsr<21> = PSTATE.DIT;
```

Is updated to read:

```
bits(N) GetPSRFromPSTATE(ExceptionalOccurrenceTargetState target)
    if UsingAArch32() && (target IN {AArch32_NonDebugState, DebugState}) then
        assert N == 32;
    else
        assert N == 64;
    bits(N) spsr = Zeros();
    …
    if PSTATE.nRW == '1' then // AArch32 state
        if HaveDITExt() then
            if target == AArch32_NonDebugState then
                spsr<21> = PSTATE.DIT;
            else //AArch64_NonDebug or DebugState
                spsr<24> = PSTATE.DIT;
        if target IN {AArch64_NonDebugState, DebugState} then
```

```
            spsr<21> = PSTATE.SS;
```

Similarly, on an exception return from AArch64 to AArch32, the function SetPSTATEFromPSR() should copy SPSR_ELx<24> to PSTATE.DIT. The code that reads:

```
SetPSTATEFromPSR(bits(N) spsr)
    …
    if PSTATE.nRW == '1' then       // AArch32 state
        if HaveDITExt() then PSTATE.DIT = (if Restarting() then spsr<24> else
 spsr<21>);
    …
```

Is updated to read:

```
SetPSTATEFromPSR(bits(N) spsr)
    boolean from_aarch64 = FALSE;
    if UsingAArch32() then
        assert N == 32;
    else
        assert N == 64;
        from_aarch64 = TRUE;

    …
    if PSTATE.nRW == '1' then       // AArch32 state
          if HaveDITExt() then PSTATE.DIT = (if (Restarting() || from_aarch64)
 then spsr<24> else spsr<21>);
    …
```

2.75 C16894

In section D11.1.2 ('The system counter') and in section G6.1.2 ('The system counter') the text that reads:

```
From Armv8.6 the counter operates at a higher fixed frequency of 1GHz. This implies
 a resolution of 1ns
```

is clarified to read:

```
From Armv8.6, the counter operates at a higher fixed frequency of 1GHz.
```

2.76 D16900

In section C5.2.7 (FPCR, Floating-point Control Register), the text that reads:

```
0b1 Trapped exception handling selected. If the floating-point exception occurs, the
 PE does not update the FPSR.UFC bit. The trap handling software can decide whether
 to set the FPSR.UFC bit to 1.
```

is updated to read:

```
0b1 Trapped exception handling selected. If the floating-point exception occurs and
 FlushToZero is not enabled, the PE does not update the FPSR.UFC bit.
```

In addition, the statement for all the trapped exception enables, the following text is deleted:

```
The trap handling software can decide whether to set the FPSR.xxx bit to 1.
```

In section C5.2.8 (FPSR, Floating-point Status Register) in the definition of the UFC, bit [3], the text that currently reads:

```
How scalar and Advanced SIMD floating-point instructions update this bit depends on
 the value of the FPCR.UFE bit. This bit is only set to 1 to indicate a floating-
point exception if FPCR.UFE is 0, or if trapping software sets it.
```

is changed to read:

```
How scalar and Advanced SIMD floating-point instructions update this bit depends on
 the value of the FPCR.UFE bit. This bit is only set to 1 to indicate a floating-
point exception if FPCR.UFE is 0 or if the FlushToZero is enabled
```

In all the cumulative exception bits, the clause 'or if trapping software sets it.' is deleted, as that is describing software usage rather than architectural behaviour.

The equivalent edits for the UFE and UFC behaviour are made in G8.2.54 (FPSCR, Floating-Point Status and Control Register).

2.77 D16901

In section D5.10.2 (TLB maintenance instructions) in the subsection "Invalidation of TLB entries from stage 2 translations", the note that reads:

```
Depending on the invalidation required, software must use the entire sequence
 1, 2, or 3, even when Secure or Non-secure EL1&0, when EL2 is enabled, stage 1
 translation is disabled.
```

is clarified to read:

```
Software must use these entire sequences for an EL1&0 translation regime with stage
 2 translation enabled, even if stage 1 translation is disabled.
```

2.78 R16902

In section D7.11.7 (**IMPLEMENTATION DEFINED** event numbers), the following text is added:

```
The Arm architecture guarantees not to define any event prefixed with IMP_ as part
 of the standard Arm architecture.
```

2.79 C16906

In section D13.2.65 (ID_AA64PFR1_EL1, AArch64 Processor Feature Register 1), the text that reads:

```
FEAT_MTE implements the functionality identified by the value 0b0001.
```

is changed to read:

```
FEAT_MTE implements the functionality identified by the value 0b0001.
FEAT_MTE2 implements the functionality identified by the value 0b0010.
```

The values description that reads:

```
0b0001 Memory Tagging Extension instructions accessible at EL0 are implemented.
Instructions and System Registers defined by the extension not configurably
 accessible at EL0 are Unallocated and other System register fields defined by the
 extension are RES0.
0b0010 Memory Tagging Extension is implemented.
```

is changed to read:

```
0b0001 Instruction-only Memory Tagging Extension is implemented.
0b0010 Full Memory Tagging Extension is implemented.
```

Appropriate changes are made in A1.8.1 (Architectural features added by Armv8.5) and Chapter D6 (Memory Tagging Extension).

2.80 D16908

In section D2.10.1 (About Watchpoint exceptions) the text that reads:

```
DBGWCR2_EL2 and DBGWVR2_EL1 are for watchpoint number two.
```

is corrected to:

```
DBGWCR2_EL1 and DBGWVR2_EL1 are for watchpoint number two.
```

In addition the following text:

```
DBGWCR0_EL1 and DBGWVR0_EL1 are for watchpoint number zero.
```

is added, and the text that reads:

```
DBGWCR<n>_EL1 and DBGWVR<n>_EL1 are for watchpoint number n.
```

is changed to:

```
DBGWCR<n-1>_EL1 and DBGWVR<n-1>_EL1 are for watchpoint number (n-1).
```

Equivalent changes to these last two changes are also made in section D2.9.1 (About Breakpoint exceptions).

2.81 D16910

In section D2.9.5 ('Breakpoint context comparisons') the text that reads:

```
Context breakpoints do not generate Breakpoint exceptions when any of:
...
* The comparison uses the value of CONTEXTIDR_EL2 and any of:
 -- Neither ARMv8.1-VHE is implemented, nor ARMv8.2-Debug is implemented.
 -- If the PE is in Secure state, and either ARMv8.4-SecEL2 is not implemented, or
 Secure EL2 is disabled.
 -- EL2 is using AArch32.
 -- EL2 is not implemented.
* The comparison uses the current VMID value and any of:
 -- EL2 is not implemented.
 -- If the PE is in Secure state, and either ARMv8.4-SecEL2 is not implemented, or
 Secure EL2 is disabled.
 -- The PE is executing at EL2.
 -- ARMv8.1-VHE is implemented, EL2 is using AArch64, EL2 is enabled in the current
 Security state, and HCR_EL2.{E2H, TGE} == {1, 1}.
```

is corrected to:

```
Context breakpoints do not generate Breakpoint exceptions when any of:
...
* The comparison uses the value of CONTEXTIDR_EL2 and any of:
 -- Neither ARMv8.1-VHE is implemented, nor ARMv8.2-Debug is implemented.
 -- If the PE is in Secure state, and either ARMv8.4-SecEL2 is not implemented, or
 Secure EL2 is disabled.
 -- The PE is executing at EL3.
 -- EL2 is using AArch32.
 -- EL2 is not implemented.
* The comparison uses the current VMID value and any of:
```

```
-- EL2 is not implemented.
-- If the PE is in Secure state, and either ARMv8.4-SecEL2 is not implemented, or
Secure EL2 is disabled.
-- The PE is executing at EL2.
-- The PE is executing at EL3.
-- ARMv8.1-VHE is implemented, EL2 is using AArch64, EL2 is enabled in the current
Security state, and HCR_EL2.{E2H, TGE} == {1, 1}.
```

2.82 D16911

In section I5.2.1 (Performance Monitors external register views), in Table I5-1 'Performance Monitors external register views', the Offset value for the PMCID1SR register alias, which is incorrectly stated as '0x248', is corrected to '0x228'.

2.83 R16915

In section D1.11 (Exception return), the text that reads:

```
If FEAT_IESB is implemented, when the SCTLR_ELx.IESB bit at the Exception level the
 exception is returning from is 1, the PE inserts an error synchronization event
 before the ERET instruction.
```

is relaxed to:

```
If FEAT_IESB is implemented, when the SCTLR_ELx.IESB bit at the Exception level
 the exception is returning from is 1 and the exception return instruction does not
 generate an exception, the PE inserts an error synchronization event before the
 exception return instruction.
```

2.84 D16926

In section D1.12.4 (Synchronous exception prioritization for exceptions taken to AArch64 state), the following updates are made:

For item 13, a bullet is added:

```
* MRS or MSR instruction using a _EL12 register name when HCR_EL2.E2H ==0.
```

For item 17, the line that reads

```
* Any setting in HCR_EL2 other than the {TIDCP, NV} fields.
```

is changed to read:

```
* Any setting in HCR_EL2 other than the {TIDCP, NV} fields, and MRS/MSR instruction
 using an _EL12 register name with HCR_EL2.E2H==0.
```

In section D13.2.36 (ESR_EL1, Exception Syndrome Register (EL1)), D13.2.37 (ESR_EL2, Exception Syndrome Register (EL2)) and D13.2.38 (ESR_EL3, Exception Syndrome Register (EL3)), in the subsection 'ISS encoding for exceptions with an unknown reason' as part of the bullet list of exceptions that cause this exception code, the following bullet is added:

```
* MRS or MSR instruction using a _EL12 register name when HCR_EL2.E2H ==0
```

2.85 D16935

SPE defines a conceptual 'owning translation regime' for the profiling buffer, which is one of Secure EL2(&0), Secure EL1&0, Non-secure EL2(&0) or Non-secure EL1&0. Software must issue a PSB CSYNC operation to synchronize use of the owning translation regime MMU control register before changing any of these registers; e.g. when switching between guest operating systems in a hypervisor, changing TTBRn_EL1 values.

In addition to the VMSA control registers, the VMSA makes use of the current security state when performing a translation. For example, Secure translation regimes can access Secure and Non-secure memory, and there are additional controls at Secure stage 2 that are not part of Non-secure stage 2.

For accesses through these translation regimes at EL3, the SCR_EL3.NS bit is used.

It is somewhat unspecified whether the translations performed by SPE use the secure identity of the 'owning translation regime' for this purpose, or the SCR_EL3.NS bit. That is, whether a secure monitor must issue a PSB CSYNC operation to synchronize use of the owning translation regime MMU control register before changing SCR_EL3.NS, after entering EL3 from a security state that might be using SPE.

Given the number of places where the NS bit does affect the behavior of the translation regime, Arm proposes to clarify that such a PSB CSYNC operation *is* required, with the translation being UNPREDICTABLE if SCR_EL3.NS does not match the secure identity of the 'owning translation regime'.

2.86 R16945

Arm relaxes the operation of ESB such that it is only required to synchronize vSEIs if VSESR_EL2/VDFSR is writable. i.e. in a truly minimal implementation, ESB is permitted to be a **NOP**. (This is a relaxation for implementations, not a change to require the new behavior.)

2.87 D16957

In section D13.4.17 (PMUSERENR_EL0, Performance Monitors User Enable Register), the EN field description. the text that reads:

```
In AArch64 state, MRS or MSR accesses to the following registers are reported using
 EC syndrome value 0x18:
-  PMCR_EL0, PMOVSCLR_EL0, PMSELR_EL0, PMCEID0_EL0, PMCEID1_EL0, PMCCNTR_EL0,
 PMXEVTYPER_EL0, PMXEVCNTR_EL0, PMCNTENSET_EL0, PMCNTENSET_EL0, PMOVSSET_EL0,
 PMEVCNTR<n>_EL0, PMEVTYPER<n>_EL0, PMCCFILTR_EL0.
```

is corrected to read:

```
In AArch64 state, MRS or MSR accesses to the following registers are reported using
 EC syndrome value 0x18:
- PMCR_EL0, PMOVSCLR_EL0, PMSELR_EL0, PMCEID0_EL0, PMCEID1_EL0, PMCCNTR_EL0,
 PMXEVTYPER_EL0, PMXEVCNTR_EL0, PMCNTENSET_EL0, PMCNTENCLR_EL0, PMOVSSET_EL0,
 PMEVCNTR<n>_EL0, PMEVTYPER<n>_EL0, PMCCFILTR_EL0.
```

2.88 D16959

In section D13.2.60 (ID_AA64ISAR1_EL1, AArch64 Instruction Set Attribute Register 1) the list of instruction mnemonics associated with the BF16 field which reads:

```
BFDOT, BFMLAL, BFMLAL2, BFMMLA, BFCVT, and BFCVT2 instructions
```

is changed to read:

```
BFCVT, BFCVTN, BFCVTN2, BFDOT, BFMLALB, BFMLALT, and BFMMLA  instructions
```

In sections D13.2.75 (ID_ISAR6_EL1, AArch32 Instruction Set Attribute Register 6) and G8.2.91 (ID_ISAR6, Instruction Set Attribute Register 6), the list of instruction mnemonics associated with the BF16 field which reads:

```
VCVT, VCVTB, VCVTT, VDOT, VFMAL, and VMMLA instructions
```

is changed to read:

```
VCVT, VCVTB, VCVTT, VDOT, VFMAB, VFMAT, and VMMLA instructions
```

2.89 D16963

In section D9.7.5 (Effect on the exclusive monitors), the text that reads:

```
If an operation between Load-Exclusive and Store-Exclusive instructions is sampled,
 then the Store-Exclusive must be guaranteed not to fail, even though the sample
 record is written to an unrelated address.
```

is replaced by:

```
If a Load-exclusive instruction or an operation between Load-exclusive and Store-
exclusive instructions is sampled, and the sample record is written to an unrelated
 address, then to avoid a probe effect, Arm recommends that the Store-exclusive does
 not systematically fail on account of the sampled operation. If a Store-exclusive
 instruction is sampled, and the sample record is written to an unrelated address,
 then the Store-exclusive must not systematically fail on account of the instruction
 having been sampled.
```

2.90 D16971

In section F3.1.14 (Additional Advanced SIMD and floating-point instructions), sub-section 'Floating-point directed convert to integer', the table needs a column for the value of 'op'.

The value of 'op' is 0 for VRINTA, VRINTN, VRINTP and VRINTM.

The equivalent changes are made in section F4.1.14 (Unconditional Advanced SIMD and floating-point instructions), sub-section 'Floating-point directed convert to integer'.

2.91 C16981

In section D7.11.4 (Cycle counting on multi-threaded implementations) in the Performance Monitors Extension chapter, the text that reads:

```
When the PMU implementation supports multithreading, and the Effective value of
 PMEVTYPER<n>_EL0.MT bit is 0, the CPU_CYCLES event only counts cycles on which the
 thread was active. For the example multithreaded implementations, this means that:
```

is corrected to:

```
When the PMU implementation supports multithreading, and the Effective value of
 PMEVTYPER<n>_EL0.MT bit is 0, the CPU_CYCLES event does not count cycles on which
 the thread was not active. For the example multithreaded implementations, this
 means that, if the event counter is enabled and event counting is not prohibited:
```

And the Note that reads:

```
The PMCCNTR register counts every processor cycle.
```

is corrected to:

```
The cycle counter, PMCCNTR, is not affected by whether the thread is active or
 inactive. When enabled, PMCCNTR counts every processor cycle.
```

In addition, in section D7.1.3 (Time as measured by the Performance Monitors cycle counter), the Note that reads:

```
This means that, in an implementation where PEs are multithreaded, the counter
 continues to increment across all PEs, rather than only counting cycles for which
 the current PE is active.
```

is corrected to:

```
This means that, in an implementation where PEs are multithreaded, when enabled,
 the cycle counter continues to increment across all PEs, rather than only counting
 cycles for which the current PE is active.
```

And in section D7.5 (Prohibiting event counting), the text that reads:

```
The cycle counter, PMCCNTR, counts unless one of the following is true:
```

is corrected to:

```
The cycle counter, PMCCNTR, counts unless any of the following is true:
- The cycle counter is disabled by PMCR_EL0.E or PMCNTENSET_EL0[31].
```

2.92 C16983

In Section D7.11.3 (Common event numbers), in the subsection 'Common microarchitectural events', for each of the *_FIXED_OPS_SPEC events, a clarification is added to the event that

the event does *not* count the operation if the operation is counted by the corresponding *_SCALE_OPS_SPEC event.

This is implied by the list of operations in the event description, but is added to make this unambiguous.

2.93 C16984

In section D7.11.3 (Common event numbers), in the subsection 'Common microarchitectural events', the text that reads:

```
See Operation counts for dot-product and multiply-accumulate operations on page
 D7-2713 for information on counts for dot product, matrix multiplication, and
 BFloat16 multiply-accumulate instructions.
```

is removed from the descriptions of the following event descriptions: `0x80CA` LDST_SCALE_OPS_SPEC, `0x80CB` LDST_FIXED_OPS_SPEC `0x80CC` LD_SCALE_OPS_SPEC `0x80CD` LD_FIXED_OPS_SPEC `0x80CE` ST_SCALE_OPS_SPEC `0x80CF` ST_FIXED_OPS_SPEC

2.94 D16989

In J1.1 (Pseudocode for AArch64 operation) the Pseudocode function ProfilingBufferOwner() does not correctly reflect the owning exception level as described in D9.7.2 (The owning Exception level).

The code that reads:

```
(boolean, bits(2)) ProfilingBufferOwner()
    secure = if HaveEL(EL3) then (MDCR_EL3.NSPB<1> == '0') else IsSecure();
    el = if !secure && HaveEL(EL2) && MDCR_EL2.E2PB == '00' then EL2 else EL1;
    return (secure, el);
```

Is corrected to read:

```
(boolean, bits(2)) ProfilingBufferOwner()
    secure = if HaveEL(EL3) then (MDCR_EL3.NSPB<1> == '0') else IsSecure();
    el = if HaveEL(EL2) && (!secure || IsSecureEL2Enabled()) && MDCR_EL2.E2PB ==
 '00' then EL2 else EL1;
    return (secure, el);
```

2.95 D16990

In section J1.3, (Shared pseudocode) the Pseudocode functions GetPSRFromPSTATE() and SetPSTATEFromPSR() do not reflect that, in AArch64 state, SPSR_ELx and DSPSR_EL0 are 64-bits. As such, GetPSRFromPSTATE() is updated to return a variable length value and SetPSTATEFromPSR() is updated to accept a variable length parameter. Consequently, the functions DebugExceptionReturnSS() and Halt() are updated to reflect these changes.

The code that reads:

```
bits(32) GetPSRFromPSTATE(ExceptionalOccurrenceTargetState target)
    bits(32) spsr = Zeros();
    …
```

Is updated to read:

```
bits(N) GetPSRFromPSTATE(ExceptionalOccurrenceTargetState targetELState)
    bits(N) spsr = Zeros();
    …
```

The code that reads:

```
SetPSTATEFromPSR(bits(32) spsr)
```

Is updated to read:

```
SetPSTATEFromPSR(bits(N) spsr)
```

The code that reads:

```
Halt()
    bits(64) preferred_restart_address = ThisInstrAddr();
    spsr = GetPSRFromPSTATE();
    ...
```

Is updated to read:

```
Halt()
    bits(64) preferred_restart_address = ThisInstrAddr();
    integer N = if UsingAArch32() then 32 else 64;
    bits(N) spsr;
    ...
```

The code that reads:

```
bit DebugExceptionReturnSS(bits(32) spsr)
```

Is updated to read:

```
bit DebugExceptionReturnSS(bits(N) spsr)
```

2.96 D16994

The following text is added to section D6.5 (PE access to Allocation Tags):

```
DC GZVA and DC ZVA are instructions which store Allocation tags.

Instructions which load or store Allocation tags are considered to perform the
 access, irrespective of whether access to Allocation tags in memory is disabled due
 to Allocation tag access controls in SCR_EL3, HCR_EL2 and SCTLR_ELx, or due to the
 absence of the Tagged attribute on the locations being accessed, for the purpose
 of:
* Address translation
* Triggering watchpoints
* Generating PMU events
* Statistical profiling
```

The text in section D2.10.6 (Watchpoint behaviour on other instructions) which reads:

```
Watchpoint behavior on accesses by the DC IVAC instruction and the DC ZVA
 instruction

DC ZVA operations can generate Watchpoint exceptions. If the Point of Coherency
 is before any level of cache, it is IMPLEMENTATION DEFINED whether a > DC IVAC
 instruction can generate a Watchpoint exception. Otherwise, DC IVAC operations can
 generate Watchpoint exceptions.

DC IVAC and DC ZVA operations are treated as data stores by DBGWCR<n>_EL1.LSC.
```

is changed to read:

```
Watchpoint behavior on accesses by the DC IVAC instruction and the DC ZVA, DC GVA,
 and DC GZVA instructions

DC ZVA, DC GVA and DC GZVA operations can generate Watchpoint exceptions. If the
 Point of Coherency is before any level of cache, it is IMPLEMENTATION DEFINED
 whether a DC IVAC instruction can generate a Watchpoint exception. Otherwise, DC
 IVAC operations can generate Watchpoint exceptions.

DC IVAC, DC ZVA, DC GZVA and DC GVA operations are treated as data stores by
 DBGWCR<n>_EL1.LSC.
```

2.97 D17005

In section C5.5.24 (TLBI RIPAS2LE1OS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable), the text that reads:

```
The entry is a stage 2 only translation table entry, from any level of the
 translation table walk.
```

is replaced with

```
The entry is a stage 2 only translation table entry, from the final level of the
 translation table walk.
```

Also in section C5.5.46 (TLBI RVALE3, TLB Range Invalidate by VA, Last level, EL3), C5.5.47 (TLBI RVALE3IS, TLB Range Invalidate by VA, Last level, EL3, Inner Shareable), and C5.5.48 (TLBI RVALE3OS, TLB Range Invalidate by VA, Last level, EL3, Outer Shareable), the text that reads:

```
The entry is a stage 1 translation table entry, from any level of the translation
 table walk.
```

is corrected to:

```
The entry is a stage 1 translation table entry, from the final level of the
 translation table walk.
```

2.98 D17013

In section G8.3.31 (HDCR, Hyp Debug Control Register), in the TDCC field description, the text that reads:

```
On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets
 to an architecturally UNKNOWN value.
```

is corrected to:

```
On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets
 to 0.
```

In the same register, in the HLP field description, the text that reads:

```
On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets
 to 0.
```

is corrected to:

```
On a Warm reset, in a system where the PE resets into EL2 or EL3, this field resets
 to an architecturally UNKNOWN value.
```

2.99 D17015

In section F5.1 (Alphabetical list of T32 and A32 base instruction set instructions), in an implementation that includes EL2, the permitted LDC/STC access to DBGDTRTXint/ DBGDTRRXint can be trapped to Hyp mode. This is not shown for simplicity.

Furthermore, the pseudocode also does not show: - The possible trap to EL1, due to MDSCR_EL1.TDCC or DBGDSCRext.UDCCdis. (This will be routed to EL2 if TGE is set.)

- The possible trap to EL2 using AArch64, due to MDCR_EL2.TDA or (since Armv8.6) MDCR_EL2.TDCC. (This can happen in either Security state when Secure EL2 is implemented.)
- The possible trap to EL3 due to MDCR_EL3.TDA or (since Armv8.6) MDCR_EL3.TDCC.
- Since Armv8.6, the possible trap to Monitor mode due to SDCR.TDCC.

Accordingly, details of all the traps are added through the use of new LDC and STC accessibility pseudocode in the DBGDTRTXint and DBGDTRRXint register pages.

The STC execution pseudocode is changed to read:

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];

    // System register read from DBGDTRRXint.
    MemA[address,4] = AArch32.SysRegRead(cp, ThisInstr());

    if wback then R[n] = offset_addr;
```

The LDC execution pseudocode is changed to read:

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];

    // System register write to DBGDTRTXint.
    AArch32.SysRegWriteM(cp, ThisInstr(), address);

    if wback then R[n] = offset_addr;
```

2.100 D17018

In each of the following locations, the text 'if ... [expression] is greater than or equal to the number of accessible counters' (or similar) is corrected to 'if ... [expression] is greater than or equal to the number of accessible event counters'.

- 'Accessing the PMEVCNTR<n>_EL0' in D13.4.8 ('[expression]' is '<n>')
- 'Accessing the PMEVTYPER<n>_EL0' in D13.4.9 ('[expression]' is '<n>')
- 'Accessing the PMXEVCNTR_EL0' in D13.4.18 ('[expression]' is 'PMSELR_EL0.SEL')
- 'Accessing the PMXEVTYPER_EL0' in D13.4.19 ('[expression]' is 'PMSELR_EL0.SEL is not 31 and')
- 'Accessing the PMEVCNTR<n>' in G8.4.10 ('[expression]' is '<n>')
- 'Accessing the PMEVTYPER<n>' in G8.4.11 ('[expression]' is '<n>')
- 'Accessing the PMXEVCNTR' in G8.4.19 ('[expression]' is 'PMSELR.SEL')
- 'Accessing the PMXEVTYPER' in G8.4.20 ('[expression]' is 'PMSELR.SEL is not 31 and')
- '**CONSTRAINED UNPREDICTABLE** accesses to PMXEVTYPER or PMXEVCNTR' and '**CONSTRAINED UNPREDICTABLE** accesses to PMEVCNTR<n> and PMEVTYPER<n>' in section K1.1.17.
- '**CONSTRAINED UNPREDICTABLE** accesses to PMXEVTYPER_EL0 or PMXEVETYPER_EL0' and '**CONSTRAINED UNPREDICTABLE** accesses to PMEVCNTR<n>_EL0 and PMEVTYPER<n>_EL0' in section K1.2.6

In particular in some cases the value 31 for '[expression]' selects the cycle counter, which is always accessible. However these rules are specifically calling out values relating to event counters, and 31 is always greater than the number of accessible event counters.

Similarly 'implemented counters' is corrected to 'implemented event counters' in the applicable sections.

2.101 D17020

In section D13.3.18 (MDCR_EL3, Monitor Debug Configuration Register (EL3)), in the SDD field, the following additional text is added:

```
If Secure EL2 is implemented and enabled, and Secure EL1 is using AArch32 then:
* If debug exceptions from Secure EL1 are enabled, then debug exceptions from Secure
 EL0 are also enabled.
* Otherwise, debug exceptions from Secure EL0 are enabled only if the value of
 SDER32_EL3.SUIDEN is 0b1.
```

In section D13.3.18 (MDCR_EL3, Monitor Debug Configuration Register (EL3)), in the SPD32 field, the text that currently states:

```
This field is ignored if the PE is either:
* In Non-secure state.
```

```
* In Secure state and Secure EL1 is using AArch64.
```

is replaced by:

```
The SPD32 field is ignored unless both of the following are true:
* The PE is in Secure state.
* The Effective value of SCR_EL3.RW is 0b0.
```

In J1.2.1 (aarch32/debug), the code that reads:

```
spd = if ELUsingAArch32(EL3) then SDCR.SPD else MDCR_EL3.SPD32;
       if spd<1> == '1' then
           enabled = spd<0> == '1';
       else
           // SPD == 0b01 is reserved, but behaves the same as 0b00.
           enabled = AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled();
       if from == EL0 then enabled = enabled || SDER.SUIDEN == '1';
```

is updated to:

```
assert from != EL2; // Secure EL2 always uses AArch64
       if IsSecureEL2Enabled() then
           // Implies that EL3 and EL2 both using AArch64
           enabled = MDCR_EL3.SDD == '0';
       else
           spd = if ELUsingAArch32(EL3) then SDCR.SPD else MDCR_EL3.SPD32;
           if spd<1> == '1' then
               enabled = spd<0> == '1';
           else
               // SPD == 0b01 is reserved, but behaves the same as 0b00.
               enabled = AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled();

       if from == EL0 then enabled = enabled || SDER.SUIDEN == '1';
```

2.102 D17036

In section J1.1 (Pseudocode for AArch64 operation) AArch64.WatchpointMatch() needs to be updated to check the access of atomic memory operations as part of Armv8.1 Large System Extensions. The code that reads:

```
ls_match = (DBGWCR_EL1[n].LSC<(if iswrite then 1 else 0)> == '1');
```

Is updated to read:

```
   ls_match = FALSE;
   if acctype == AccType_ATOMICRW then
       ls_match = (DBGWCR_EL1[n].LSC != '00');
   else
       ls_match = (DBGWCR_EL1[n].LSC<(if iswrite then 1 else 0)> == '1');
```

2.103 D17045

In section J1.1 'Pseudocode for AArch64 operation', the Pseudocode function CollectTimeStamp() is missing reserved value checks.

The code that reads:

```
if EL2Enabled() then
      case PMSCR_EL2.PCT of
          when '00'
              return TimeStamp_Virtual;
          when '01'
              if el == EL2 then return TimeStamp_Physical;
          when '11'
              if (el == EL2 || PMSCR_EL1.PCT != '00') && HaveECVExt() then
                  return TimeStamp_OffsetPhysical;
          otherwise
              Unreachable();

  case PMSCR_EL1.PCT of
      when '00' return TimeStamp_Virtual;
      when '01' return TimeStamp_Physical;
      when '11' if HaveECVExt() then return TimeStamp_OffsetPhysical;
      otherwise Unreachable();
```

is updated to read:

```
  if !HaveECVExt() then
      PCT_el1 = '0':PMSCR_EL1.PCT<0>;       // PCT<1> is RES0
  else
      PCT_el1 = PMSCR_EL1.PCT;
      if PCT_el1 == '10' then
          // Reserved value
          (-, PCT_el1) = ConstrainUnpredictableBits(Unpredictable_PMSCR_PCT);
  if EL2Enabled() then
      if !HaveECVExt() then
          PCT_el2 = '0':PMSCR_EL2.PCT<0>;   // PCT<1> is RES0
      else
          PCT_el2 = PMSCR_EL2.PCT;
          if PCT_el2 == '10' then
              // Reserved value
              (-, PCT_el2) = ConstrainUnpredictableBits(Unpredictable_PMSCR_PCT);
      case PCT_el2 of
          when '00'
              return TimeStamp_Virtual;
          when '01'
              if el == EL2 then return TimeStamp_Physical;
          when '11'
              assert HaveECVExt();          // FEAT_ECV must be implemented
              if el == EL1 && PCT_el1 == '00' then
                  return TimeStamp_Virtual;
              else
                  return TimeStamp_OffsetPhysical;
          otherwise
              Unreachable();

  case PCT_el1 of
      when '00' return TimeStamp_Virtual;
      when '01' return TimeStamp_Physical;
      when '11'
          assert HaveECVExt();              // FEAT_ECV must be implemented
          return TimeStamp_OffsetPhysical;
      otherwise Unreachable();
```

2.104 R17047

In section B2.7.2 (Device memory), the text that reads:

```
For instruction fetches, if branches cause the program counter to point to an area
 of memory with the Device attribute which is not marked as Execute-never for the
 current Exception level, an implementation can either:
* Treat the instruction fetch as if it were to a memory location with the Normal
 Non-cacheable attribute.
* Take a Permission fault.
```

is relaxed to read:

```
For instruction fetches, if the program counter points to an area of memory with the
 Device attribute which is not marked as Execute-never for the current Exception
 level, an implementation can either:
* Treat the instruction fetch as if it were to a memory location with the Normal
 Non-cacheable attribute.
* Take a Permission fault.
```

2.105 D17050

In section J1.2 (Shared pseudocode), in the Pseudocode function AArch32.GenerateDebugExceptionsFrom(), calling AArch64.GenerateDebugExceptionsFrom() when target exception level is EL2 using AArch64 is not implemented.

The code that reads:

```
if from == EL0 && !ELStateUsingAArch32(EL1, secure) then
        mask = bit UNKNOWN;                          // PSTATE.D mask, unused for
 EL0 case
        return AArch64.GenerateDebugExceptionsFrom(from, secure, mask);
```

Is updated to read:

```
if !ELUsingAArch32(DebugTargetFrom(secure)) then
    mask = '0';                  // No PSTATE.D in AArch32 state
    return AArch64.GenerateDebugExceptionsFrom(from, secure, mask);
```

2.106 D17052

In section C6.2.82 (DSB), the encoding shows the CRm field as being restricted with the condition != `0x00`. This is intended to cover encodings 0b0000 and 0b0100, which are used for the SSBB

and PSSBB instructions. This description is clarified by making SSBB and PSSBB as architectural aliases of DSB.

2.107 D17067

Section D5.4.13 (Restriction on memory types for hardware updates on translation tables) does not fully describe what happens upon execution of an Address translation instruction. To this end, the following paragraphs are added at the end of the section:

```
The execution of an Address translation instruction can report an Unsupported atomic
 hardware update fault, in PAR_EL1, using the Fault status code of 0x110001, as
 follows:
* On an address translation instruction executed at EL1, if hardware updates to
 the translation tables are enabled for stage 1, and the stage 1 translation tables
 are held in memory with a memory type that means that hardware updates of the
 translation tables are not atomic as observed by other agents that can access
 memory, then the architecture permits, but does not require, that the PAR_EL1
 reports a Translation table hardware update fault.
* On an address translation instruction executed at EL2 or EL3, if hardware
 updates to the translation tables used by the instruction are enabled, and those
 translation tables are held in memory with a memory type that means that hardware
 updates of the translation tables are not atomic as observed by other agents that
 can access memory, then the architecture permits, but does not require, that the
 PAR_EL1 reports a Translation table hardware fault.
```

2.108 D17075

In section J1.3.5 (shared/translation), the Pseudocode function CombineS1S2AttrHints() did not take into account Device memory when S2FWB is enabled. This is fixed by passing the MemType s1desc.memattrs.memtype parameter to CombineS1S2AttrHints.

The code in CombineS1S2AttrHints() that reads:

```
elsif apply_force_writeback then
    if s1desc.attrs != MemAttr_NC then
        result.hints = s1desc.hints;
    else
        result.hints = MemHint_RWA;
```

Is corrected to read:

```
elsif apply_force_writeback then
    if s1desc.attrs == MemAttr_NC || memtype == MemType_Device  then
        result.hints = MemHint_RWA;
    else
        result.hints = s1desc.hints;
```

2.109 D17079

In G8.3.13 (DBGDSAR, Debug Self Address Register), the accessibility pseudocode does not account for a trap to EL3 by MDCR_EL3.TDA.

To this end, the following code is added:

```
if PSTATE_EL == EL0 then
    …
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGDSAR;
elsif PSTATE_EL == EL1 then
    …
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGDSAR;
elsif PSTATE_EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED
 \"EL3 trap priority when SDD == '1'\" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA ==
 '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGDSAR;
...
```

An equivalent change is made in G8.3.12 (DBGDRAR, Debug ROM Address Register).

2.110 D17088

In section D13.2.96 (MPIDR_EL1, Multiprocessor Affinity Register), the text in the MT field description that reads:

```
0b0 Performance of PEs at the lowest affinity level, or PEs with MPIDR_EL1.MT set to
 1, different affinity level 0 values, and the same values for affinity level 1 and
 higher, is largely independent.
0b1 Performance of PEs at the lowest affinity level, or PEs with MPIDR_EL1.MT set to
 1, different affinity level 0 values, and the same values for affinity level 1 and
 higher, is very interdependent.
```

is changed to read:

```
0b0 Performance of PEs with different affinity level 0 values, and the same values
 for affinity level 1 and higher, is largely independent.
0b1 Performance of PEs with different affinity level 0 values, and the same values
 for affinity level 1 and higher, is very interdependent.
```

The equivalent edit is made for AArch32 in G8.2.113 (MPIDR, Multiprocessor Affinity Register).

2.111 D17091

In section C6.2 (Alphabetical list of A64 base instructions), the Pseudocode for instructions BLR, BLRAA, BLRAAZ, BLRAB, BLRABZ, BR, BRAA, BRAAZ, BRAB, BRABZ, RET, RETAA, RETAB was improperly trimmed due to a tooling issue. This is corrected now.

The code in BLR and BLRAA, BLRAAZ, BLRAB, BLRABZ that reads:

```
BranchTo(target, BranchType_INDCALL);
```

is corrected to read:

```
// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '10';
BranchTo(target, BranchType_INDCALL);
```

The code in BR and BRAA, BRAAZ, BRAB, BRABZ that reads:

```
BranchTo(target, BranchType_INDIR);
```

is corrected to read:

```
// Value in BTypeNext will be used to set PSTATE.BTYPE
if InGuardedPage then
    if n == 16 || n == 17 then
        BTypeNext = '01';
    else
        BTypeNext = '11';
else
    BTypeNext = '01';
BranchTo(target, BranchType_INDIR);
```

The code in RET and RETAA, RETAB that reads:

```
BranchTo(target, BranchType_RET);
```

is corrected to read:

```
// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00';

BranchTo(target, BranchType_RET);
```

2.112 D17093

In G8.3.34 (SDCR, Secure Debug Control Register), the text in the TTRF field description that reads:

```
Trap Trace Filter controls. Controls whether accesses at EL2 and EL1 to the trace
 filter control registers are trapped to EL3.
```

is changed to read:

```
Trap Trace Filter controls. Controls whether accesses in modes other than Monitor
 mode to the trace filter control registers generate a Monitor Trap exception.
```

2.113 D17119

In the following sections: * F3.1.10 (Advanced SIMD shifts and immediate generation), sub-section 'Advanced SIMD two registers and shift amount' * F4.1.22 (Advanced SIMD shifts and immediate generation), sub-section 'Advanced SIMD two registers and shift amount'

The entry under 'imm3H:L' for VMOVL is corrected to read:

```
* 'L'  must be '0'.
* 'imm3H' cannot be '000'.
```

2.114 D17120

In section F3.1.16 (Branches and miscellaneous control), sub-section 'Exception return', the following row is added:

Rn | imm8 | Instruction page |
!= 1110 | 00000000 | SUB. SUBS (immediate) - T5 variant |

2.115 R17126

In the 'Glossary' section, in the entry for **RES0**, the line that reads:

```
Whether RES0 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION
 DEFINED on a field-by-field basis.
```

is relaxed to read:

```
Whether RES0 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION
 DEFINED on a bit-by-bit basis.
```

The equivalent edit is made to the definition of RES1.

2.116 D17128

In section J1.1 (Pseudocode for AArch64 operation) the routine AArch64.SoftwareStepException is updated to include Instruction Fault Status Code as Debug Exception.

The code that reads:

```
exception = ExceptionSyndrome(Exception_SoftwareStep);
if SoftwareStep_DidNotStep() then
    exception.syndrome<24> = '0';
else
    exception.syndrome<24> = '1';
    exception.syndrome<6> = if SoftwareStep_SteppedEX() then '1' else '0';
```

is updated to read:

```
exception = ExceptionSyndrome(Exception_SoftwareStep);
if SoftwareStep_DidNotStep() then
    exception.syndrome<24> = '0';
else
    exception.syndrome<24> = '1';
    exception.syndrome<6> = if SoftwareStep_SteppedEX() then '1' else '0';
exception.syndrome<5:0> = '100010';               // IFSC = Debug Exception
```

2.117 D17130

In section C6.2 (Alphabetical list of A64 base instructions), the STGP operational Pseudocode does not check access is aligned to a tag granule (16 bytes).

The pseudocode for STGP which reads:

```
Mem[address, 8, AccType_NORMAL] = data1;
Mem[address+8, 8, AccType_NORMAL] = data2;

AArch64.MemTag[address, AccType_NORMAL] = AArch64.AllocationTagFromAddress(address);
```

Is changed to read:

```
if address != Align(address, TAG_GRANULE) then
AArch64.Abort(address, AArch64.AlignmentFault(AccType_NORMAL, TRUE, FALSE));

Mem[address, 8, AccType_NORMAL] = data1;
Mem[address+8, 8, AccType_NORMAL] = data2;

AArch64.MemTag[address, AccType_NORMAL] = AArch64.AllocationTagFromAddress(address);
```

2.118 D17131

PSTATE.SS was normally cleared after executing an instruction and there is no existing mechanism to prevent PSTATE.SS being cleared immediately after being restored by an Exception Return. The new variable 'ShouldAdvanceSS' is added to indicate this, similar to how 'ShouldAdvanceIT' is used for PSTATE.IT

Software Step state machine is extended by introducing a new global variable 'ShouldAdvanceSS'.

```
  boolean ShouldAdvanceSS;
```

The Pseudocode function SetPSTATEFromPSR() is modified to clear 'ShouldAdvanceSS' variable when PSTATE.SS is modified.

The pseudocode that reads:

```
SetPSTATEFromPSR(bits(32) spsr)
    PSTATE.SS = DebugExceptionReturnSS(spsr);
    if IllegalExceptionReturn(spsr) then
        PSTATE.IL = '1';
```

is updated to read:

```
SetPSTATEFromPSR(bits(32) spsr)
    PSTATE.SS = DebugExceptionReturnSS(spsr);
    ShouldAdvanceSS = FALSE;
```

```
    if IllegalExceptionReturn(spsr) then
        PSTATE.IL = '1';
```

2.119 D17148

In section, D1.11.2 (Illegal return events from AArch64 state), in the list that describes the situations that cause an illegal return event, the text that reads:

```
* A return to EL2 when EL3 is implemented and the value of the SCR_EL3.NS bit is 0
 if FEAT_SEL2 is not implemented.
* A return to EL1 when EL2 is implemented and the value of the HCR_EL2.TGE bit is 1.
```

is updated to read

```
* If FEAT_SEL2 is not implemented or if SCR_EL3.EEL2 is 0, a return to EL2 when EL3
 is implemented and the value of the SCR_EL3.NS bit is 0.
* A return to EL1 when EL2 is implemented and enabled in the current Security state,
 and the value of the HCR_EL2.TGE bit is 1.
```

2.120 C17164

In section D13.2.100 (PAR_EL1, Physical Address Register), in the ATTR, bits[63:56] field, the following Note is added:

```
Note: The attributes presented are consistent with the stages of translation
 applied in the address translation instruction. If the instruction performed a
 stage 1 translation only, the attributes are from the stage 1 translation. If the
 instruction performed a stage 1 and stage 2 translation, the attributes are from
 the combined stage 1 and stage 2 translation.
```

2.121 D17165

In section H2.4.2 (Executing instructions in Debug state), the two references to:

```
CSDB, when FEAT_SSBS is implemented.
```

are updated to read:

```
CSDB.
```

2.122 R17166

In C6.2 (Alphabetical list of A64 base instructions), the CSEL execution code that reads:

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

if ConditionHolds(cond) then
    result = operand1;
else
result = operand2;

X[d] = result;
```

is updated to read:

```
bits(datasize) result;
if ConditionHolds(cond) then
    result = X[n];
else
    result = X[m];

X[d] = result;
```

An equivalent change is made for all Conditional Select instructions, such as CSINC, CSINV, CSNEG, CCMP, CCMN.

2.123 R17167

In section D4.4.6 (Non-cacheable accesses and instruction caches), the text that reads:

```
In a multiprocessor system, the IC IVAU is broadcast to all PEs within the Inner
 Shareable domain of the PE running this sequence, but additional software steps
 might be required to synchronize the threads with other PEs.
```

is clarified to read:

```
In a multiprocessor system, the IC IVAU for a non-cacheable location is broadcast
 to all PEs within the Inner Shareable domain of the PE running this sequence. This
 is despite non-cacheable normal memory locations being treated as Outer Shared in
 other parts of the architecture.

Additional software steps might be required to synchronize the threads with other
 PEs. This might be necessary so that the PEs executing the modified instructions
 can execute an ISB after completing the invalidation, and to avoid issues
 associated with concurrent modification and execution of instruction sequences.
 See also Concurrent modification and execution of instructions on page B2-130 and
 Concurrent modification and execution of instructions on page E2-4060.
```

2.124 D17168

In sections D13.3.4 (DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear register), D13.3.5 (DBGCLAIMSET_EL1, Debug CLAIM Tag Set register), G8.3.5 (DBGCLAIMCLR, Debug CLAIM Tag Clear register), G8.3.6 (DBGCLAIMSET, Debug CLAIM Tag Set register), H9.2.4 (DBGCLAIMCLR_EL1, Debug CLAIM Tag Clear register), and H9.2.5 (DBGCLAIMSET_EL1, Debug CLAIM Tag Set register), the access to bits [31:8] is changed from:

```
RAZ/SBZ
```

to:

```
RAZ/WI
```

Similar changes to the accesses are made in sections H9.3.12 (CTICLAIMCLR, CTI CLAIM Tag Clear register) and H9.3.13 (CTICLAIMSET, CTI CLAIM Tag Set register), in the CLAIM<x> fields.

2.125 D17169

In section J1.1 (Pseudocode for AArch64 operation), in the Pseudocode function AArch64.CheckPermission(), the value of initial priv_xn should be determined using the value of priv_w as derived from the page table. However, it is possible for the value of priv_w to be set to FALSE as a result of a PAN check before priv_xn is initialised. An equivalent change is made to AArch32.CheckPermission().

The code that reads:

```
        ispriv = AArch64.AccessUsesEL(acctype) != EL0;

        pan = if HavePANExt() then PSTATE.PAN else '0';
        if (EL2Enabled() && ((PSTATE.EL == EL1 && HaveNVExt() && HCR_EL2.<NV, NV1>
== '11') ||
            (HaveNV2Ext() && acctype == AccType_NV2REGISTER && HCR_EL2.NV2 == '1')))
then
            pan = '0';
        is_ldst = !(acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_AT,
AccType_IFETCH});
        is_ats1xp = (acctype == AccType_AT && AArch64.ExecutingATS1xPInstr());
        if pan == '1' && user_r && ispriv && (is_ldst || is_ats1xp) then
            priv_r = FALSE;
            priv_w = FALSE;

        user_xn = perms.xn == '1' || (user_w && wxn);
        priv_xn = perms.pxn == '1' || (priv_w && wxn) || user_w;
```

is updated to read:

```
        ispriv = AArch64.AccessUsesEL(acctype) != EL0;

        user_xn = perms.xn == '1' || (user_w && wxn);
```

```
        priv_xn = perms.pxn == '1' || (priv_w && wxn) || user_w;

        pan = if HavePANExt() then PSTATE.PAN else '0';
        if (EL2Enabled() && ((PSTATE.EL == EL1 && HaveNVExt() && HCR_EL2.<NV, NV1>
== '11') ||
            (HaveNV2Ext() && acctype == AccType_NV2REGISTER && HCR_EL2.NV2 == '1')))
then
            pan = '0';
        is_ldst = !(acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_AT,
AccType_IFETCH});
        is_ats1xp = (acctype == AccType_AT && AArch64.ExecutingATS1xPInstr());
        if pan == '1' && user_r && ispriv && (is_ldst || is_ats1xp) then
            priv_r = FALSE;
            priv_w = FALSE;
```

2.126 D17178

In section J1.2.2 (aarch32/exceptions), in the pseudo-code function 'aarch32/exceptions/exceptions/AArch32.TakeReset', the lines that read:

```
PSTATE.T = SCTLR.TE; // Instruction set: TE=0: A32, TE=1: T32. PSTATE.J is RES0.
PSTATE.E = SCTLR.EE; // Endianness: EE=0: little-endian, EE=1: big-endian
```

are updated to:

```
if HaveEL(EL2) && !HaveEL(EL3) then
PSTATE.T = HSCTLR.TE; // Instruction set: TE=0: A32, TE=1: T32. PSTATE.J is RES0.
PSTATE.E = HSCTLR.EE; // Endianness: EE=0: little-endian, EE=1: big-endian
else
PSTATE.T = SCTLR.TE; // Instruction set: TE=0: A32, TE=1: T32. PSTATE.J is RES0.
PSTATE.E = SCTLR.EE; // Endianness: EE=0: little-endian, EE=1: big-endian
```

In section G8.2.72 (HSCTLR, Hyp System Control Register), in the TE field description, the line that reads:

```
In a system where the PE resets into EL2, this field resets to an architecturally
 UNKNOWN value.
```

is changed to read:

```
In a system where the PE resets into EL2, this field resets to an IMPLEMENTATION
 DEFINED value.
```

2.127 D17184

In section C6.2 (Alphabetical list of A64 base instructions), the STGP instruction is conditional on the implementation of FEAT_MTE. This check is missing from the pre-index and post-index forms of STGP.

The following code is added to the decode Pseudocode:

```
if !HaveMTEExt() then UNDEFINED;
```

2.128 D17185

In section I5.8.30 (ERR<n>PFGCTL, Pseudo-fault Generation Control Register), in the MV field description, the text that reads:

```
This bit reads-as-one if the node always records some syndrome in ERR<n>MISC<m>,
 setting ERR<n>STATUS.MV to 1, when an injected error is recorded.
```

is corrected to:

```
This bit reads-as-one and ignores writes if the node always records some syndrome in
 ERR<n>MISC<m>, setting ERR<n>STATUS.MV to 1, when an injected error is recorded.
```

A similar correction is made in I5.8.30 (ERR<n>PFGCTL, Pseudo-fault Generation Control Register), in the AV field description.

2.129 D17188

In C5.2.4 (ELR_EL1, Exception Link Register (EL1)), C5.2.5 (ELR_EL2, Exception Link Register (EL2)), C5.2.17 (SPSR_EL1, Saved Program Status Register (EL1)), D13.2.124 (TFSR_EL1, Tag Fault Status Register (EL1)), and D13.2.137 (VBAR_EL1, Vector Base Address Register (EL1)), the EL1 access pseudocode text:

```
if EL2Enabled() && HCR_EL2.<NV2,NV1> == '01' then
    AArch64.SystemAccessTrap(EL2, 0x18);
```

is replaced with:

```
if EL2Enabled() && HCR_EL2.<NV2,NV1, NV> == '011' then
    AArch64.SystemAccessTrap(EL2, 0x18);
```

The change also applies to the same access pseudocode where it is rendered in the following Special-purpose registers: C5.2.18 (SPSR_EL2, Saved Program Status Register (EL2)), D13.2.125 (TFSR_EL2, Tag Fault Status Register (EL2)), and D13.2.138 (VBAR_EL2, Vector Base Address Register (EL2)).

The change is also applied to D13.2.117 SCXTNUM_EL1, as part of D15648 when the trap is introduced for that register.

2.130 D17190

In section H2.4.8 (Accessing registers in Debug state), in Figures H2-1 and H2-2 that show example sequences for reading and writing general-purpose registers, in the step that shows the debugger writing to EDITR to execute an instruction, within the right-hand 'writing' sequence, the text that reads:

```
Sets TXfull to 0
```

is corrected to:

```
Sets TXfull to 1
```

2.131 D17193

In section D13.2.48 (HCR_EL2, Hypervisor Configuration Register), in the TID2 field, the text that currently states:

```
If the value of SCTLR_EL1.UCT is 0 then EL0 reads of CTR_EL0 are UNDEFINED and any
 resulting exception takes precedence over this trap.
```

is replaced by:

```
If the value of SCTLR_EL1.UCT is 0 then EL0 reads of CTR_EL0 are trapped to EL1 and
 the resulting exception takes precedence over this trap.
```

2.132 D17198

In Section E2.6.2 (Unaligned data access), in Table E2-3 'Alignment requirements of load/store instructions', the following row:

Instructions	Alignment check	SCTLR.A or HSCTLR.A is 0	SCTLR.A or HSCTLR.A is 1
VLDM, VLDR, VPOP, VPUSH, VSTM, VSTR	Word	Alignment fault	Alignment fault

is replaced by:

Instructions	Alignment check	SCTLR.A or HSCTLR.A is 0	SCTLR.A or HSCTLR.A is 1
VLDM, VPOP, VPUSH, VSTM	Word	Alignment fault	Alignment fault
VLDR, VSTR - single-precision scalar and double-precision scalar	Word	Alignment fault	Alignment fault
VLDR, VSTR - half-precision scalar	Halfword	Alignment fault	Alignment fault

2.133 D17199

In AArch32.CheckAdvSIMDOrFPEnabled(), the code that reads:

```
if PSTATE.EL == EL0 && (!HaveEL(EL2) || (!ELUsingAArch32(EL2) && HCR_EL2.TGE ==
 '0')) && !ELUsingAArch32(EL1) then
    // The PE behaves as if FPEXC.EN is 1
    AArch64.CheckFPAdvSIMDEnabled();
elsif PSTATE.EL == EL0 && HaveEL(EL2) && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1'
 && !ELUsingAArch32(EL1) then
    if fpexc_check && HCR_EL2.RW == '0' then
        fpexc_en = bits(1) IMPLEMENTATION_DEFINED  \"FPEXC.EN value when TGE==1 and
 RW==0\";
        if fpexc_en == '0' then UNDEFINED;
    AArch64.CheckFPAdvSIMDEnabled();
```

is corrected to:

```
if PSTATE.EL == EL0 && (!EL2Enabled() || (!ELUsingAArch32(EL2) && HCR_EL2.TGE ==
 '0')) && !ELUsingAArch32(EL1) then
    // The PE behaves as if FPEXC.EN is 1
    AArch64.CheckFPAdvSIMDEnabled();
elsif PSTATE.EL == EL0 && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1'
 then
    if (fpexc_check && HCR_EL2.RW == '0' && boolean IMPLEMENTATION_DEFINED \"Use
 FPEXC32_EL2.EN value when {TGE,RW} == {1,0}\") then
```

```
        if FPEXC32_EL2.EN == '0' then UNDEFINED;
    AArch64.CheckFPAdvSIMDEnabled();
```

That is, the following corrections are made to align with the definition of FPEXC.EN:

- The tests that include HCR_EL2 are updated to check that EL2 is enabled in the current security state.
- The "!ELUsingAArch32(EL1)" check in the "HCR_EL2.TGE == '1'" case is not correct and is removed.
- For the case when HCR_EL2.{RW,TGE} == {0,1}, the IMPLEMENTATION DEFINED choice is between the value in FPEXC32_EL2.EN and '1', not '0' and '1' as previously.

Note: There will be a further update on this issue.

2.134 D17200

In section J1.3 (Shared pseudocode), the Pseudocode function TraceTimeStamp() incorrectly treats the value '10' for TRFCR_EL1.TS and TRFCR_EL2.TS as UNPREDICTABLE in all cases. This value is defined when FEAT_ECV is implemented.

The code that reads:

```
if TS_el2 == '10' then (-, TS_el2) = ConstrainUnpredictableBits(); // Reserved value
```

Is corrected to:

```
if !HaveECVExt() && TS_el2 == '10' then
    // Reserved value
    (-, TS_el2) = ConstrainUnpredictableBits();
```

The code that reads:

```
if TS_el1 == 'x0' then (-, TS_el1) = ConstrainUnpredictableBits(); // Reserved
 values
```

is corrected to:

```
if TS_el1 == '00' || (!HaveECVExt() && TS_el1 == '10') then
    // Reserved value
    (-, TS_el1) = ConstrainUnpredictableBits();
```

In addition the code that deals with the '10' values checks again whether FEAT_ECV is implemented. The Unpredictable code removes this possibility.

The two instances where the code reads:

```
when '10' if HaveECVExt() then return TimeStamp_OffsetPhysical;
```

Are changed to:

```
when '10'
    assert(HaveECVExt());  // Otherwise ConstrainUnpredictableBits removes this case
    return TimeStamp_OffsetPhysical;
```

2.135 C17205

The following Note is added at the end of section B2.3.7 (Completion and Endpoint ordering):

```
Note:
Arm expects that, in most systems with early acknowledgements, those acknowledgement
 will come from a point at or after the point which establishes global visibility.
 This is expected in such systems to enable the acknowledgements to be used as part
 of the mechanisms to implement the ordering requirements of the Arm memory model.
```

2.136 R17206

In sections C5.2.17 (SPSR_EL1, Saved Program Status Register (EL1)), C5.2.18 (SPSR_EL2, Saved Program Status Register (EL2)), C5.2.19 (SPSR_EL3, Saved Program Status Register (EL3)) and D13.3.14 (DSPSR_EL0, Debug Saved Program Status Register), the following text is added to the TCO field description:

```
When FEAT_MTE2 is not implemented it is Constrained UNPREDICTABLE whether this field
 is RES0 or behaves as if FEAT_MTE is implemented.
```

In section C5.2.24 (TCO, Tag Check Override), the following statement is added to the description:

```
When FEAT_MTE2 is not implemented it is Constrained UNPREDICTABLE whether this
 register is RES0 or behaves as if FEAT_MTE is implemented.
```

In section D1.7 (Process state, PSTATE), the following statement is added to the description:

```
When FEAT_MTE2 is not implemented it is Constrained UNPREDICTABLE whether this bit
 is RES0 or behaves as if FEAT_MTE is implemented.
```

2.137 D17210

In section C3.2.12 (Atomic instructions), in the subsection 'Compare and Swap', the following paragraph is deleted:

```
All Compare and Swap instructions generate an Alignment fault if the address being
 accessed is not aligned to the size of the data structure being accessed.
```

2.138 D17216

In Section J1.1 (Pseudocode for AArch64 operation), AArch64.CheckWatchpoint() is updated to set the WnR bit for watchpoint match on an Atomic Read-Write access to read if DBGWCR has Read/Read-Write as LSC bit.

The code that reads:

```
    for i = 0 to UInt(DBGDIDR_(WRPs))
        match = match || AArch64.WatchpointMatch(i, vaddress, size, ispriv, acctype,
iswrite);
```

is updated to read:

```
    match_on_read = FALSE;

    for i = 0 to UInt(DBGDIDR_(WRPs))
        if AArch64.WatchpointMatch(i, vaddress, size, ispriv, acctype, iswrite) then
            match = TRUE;
            if DBGWCR_(i,LSC)<0> == '1' then
                match_on_read = TRUE;

    if match && acctype == AccType_ATOMICRW then
        iswrite = !match_on_read;
```

(Where "iswrite" is then used when reporting the Watchpoint exception).

2.139 D17218

In section H2.4.3 (Decode tables), Table H2-4 'A64 instructions that are unchanged in Debug state' is updated with the following instructions:

```
LDAPR , LDAPRB , LDAPRH .
LDAPURH , LDAPURSH , LDAPUR , LDAPURSW , LDAPURSB , LDAPURB.
STLUR , STLURH , STLURB.
```

2.140 R17220

In sections A2.2.1 (Additional functionality added to Armv8.0 in later releases), D13.2.64 (ID_AA64PFR0_EL1, AArch64 Processor Feature Register 0), D13.2.82 (ID_PFR0_EL1, AArch32 Processor Feature Register 0), and G8.2.98 (ID_PFR0, Processor Feature Register 0), the text in the CSV2 field description that reads:

```
affect speculative execution
```

is updated to:

```
control speculative execution
```

In section B2.3.9 (Restrictions on the effects of speculation), in subsection 'Restrictions on the effects of speculation from Armv8.5', and in section E2.3.9 (Restrictions on the effects of speculation), in subsection 'Further restrictions on the effects of speculation from Armv8.5', the line that reads:

```
For all execution prediction resources that predict address or register values,
 speculative execution at one hardware defined context should be separated in a
 hard-to-determine manner from the predictions trained in a different hardware
 defined context.
```

is clarified to read:

```
For all execution prediction resources that predict address or register values,
 speculative execution at one hardware defined context should be separated in a
 hard-to-determine manner from control by a different hardware defined context.
```

Also in section B2.3.9 (Restrictions on the effects of speculation), in subsection 'Restrictions on the effects of speculation from Armv8.5', the bullet that reads:

```
When in AArch64 state, the current SCXTNUM_ELx value.
```

is updated to read:

```
When in AArch64 state, the current SCXTNUM_ELx value if SCXTNUM_ELx is implemented.
```

In Section A2.2.1 (Additional functionality added to Armv8.0 in later releases), the line that reads:

```
In AArch64, the feature also adds the SCXTNUM_EL0, SCXTNUM_EL1, SCXTNUM_EL2, and
 SCXTNUM_EL3 registers,
```

is clarified to read:

```
In AArch64, the feature also optionally adds the SCXTNUM_EL0, SCXTNUM_EL1,
 SCXTNUM_EL2, and SCXTNUM_EL3 registers,
```

2.141 R17229

In sections B2.7.2 (Device memory) and E2.8.2 (Device memory), the text that reads:

```
For instruction fetches, if branches cause the program counter to point to an area
 of memory with the Device attribute which is not marked as Execute-never for the
 current Exception level, an implementation can either:
- Treat the instruction fetch as if it were to a memory location with the Normal Non
 cacheable attribute.
- Take a Permission fault.
```

is relaxed to read:

```
For an instruction fetch from a memory location with the Device attribute that is
 not marked as execute-never for the current Exception level, an implementation can
 either:
- Treat the instruction fetch as if it were to a memory location with the Normal
 Non-cacheable attribute.
- Take a Permission fault.
```

2.142 D17230

In section G8.3 (Debug Registers), the MRC pseudocode for DBGDSCRint checks EL using AArch64 with the checks for AArch32 registers.

The code that reads:

```
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCC == '1' then
    AArch32.TakeHypTrapException(0x05);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR_EL2.TGE == '1' ||
 MDCR_EL2.<TDE,TDA> != '00') then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR.TGE == '1' || HDCR.<TDE,TDA> !=
 '00') then
    AArch32.TakeHypTrapException(0x05);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
```

Is corrected to read:

```
elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.TDCC == '1' then
    AArch32.TakeHypTrapException(0x05);
elsif EL2Enabled() && !ELUsingAArch32(EL2) && (HCR_EL2.TGE == '1' ||
 MDCR_EL2.<TDE,TDA> != '00') then
    AArch64.AArch32SystemAccessTrap(EL2, 0x05);
elsif EL2Enabled() && ELUsingAArch32(EL2) && (HCR.TGE == '1' || HDCR.<TDE,TDA> !=
 '00') then
    AArch32.TakeHypTrapException(0x05);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDCC == '1' then
    AArch64.AArch32SystemAccessTrap(EL3, 0x05);
```

2.143 D17233

In section D13.2.144 (VSTTBR_EL2, Virtualization Secure Translation Table Base Register), the following text is deleted:

```
Any of the bits in VSTTBR_EL2 are permitted to be cached in a TLB.
```

In the same section, in the CnP field, the following text is added:

```
This field is permitted to be cached in a TLB.
```

2.144 D17236

In D5.10.2 (TLB maintenance instructions), in subsection 'TLB maintenance instruction syntax', the descriptions of NS, bit[63], and IPA[51:48], bits [39:36] are missing from the section covering VMSAv8-64 TLB maintenance instructions that take a register argument that holds an IPA. The text that reads:

```
VMSAv8-64 TLB maintenance instructions that take a register argument that holds
 an IPA, and that do not apply to a range of addresses, use the register argument
 format:
Bits[63:48]     RES0.
Bits[47:44]     TTL. Indicates the level of the translation table walk that holds
 the leaf entry for the address being invalidated, see Translation table level
 hints. This field is RES0 if the instruction does not require an IPA argument, or
 if FEAT_TTL is not implemented.
Bits[43:36]     RES0.
Bits[35:0]       IPA[47:12]. For an instruction that requires a VA argument, the
 treatment of the low-order bits of this field depends on the translation granule
 size, as follows:
```

is updated to read:

```
VMSAv8-64 TLB maintenance instructions that take a register argument that holds
 an IPA, and that do not apply to a range of addresses, use the register argument
 format:
Bit[63]           NS. Specifies the Secure or Non-secure IPA space. This field is
 RES0 if the instruction is executed in Non-secure state, or when FEAT_SEL2 is not
 implemented or is disabled in the current Security state.
Bits[62:48]     RES0.
Bits[47:44]     TTL. Indicates the level of the translation table walk that holds
 the leaf entry for the address being invalidated, see Translation table level
 hints. This field is RES0 if the instruction does not require an IPA argument, or
 if FEAT_TTL is not implemented.
Bits[43:40]     RES0.
Bits[39:36]    IPA[51:48]. Extension to IPA[47:12]. When 52-bit addresses are
 in use, forms the upper part of the address value. This field is RES0 if 52-bit
 addresses are not in use.
```

```
Bits[35:0]       IPA[47:12]. For an instruction that requires a VA argument, the
 treatment of the low-order bits of this field depends on the translation granule
 size, as follows:
```

2.145 C17238

In section F3.1.12 (Floating-point data-processing), in sub-section 'Floating-point data-processing (two registers)' the following changes are made:

Row '0 010 - 0 VCVTB - Half-precision to double-precision variant -' is replaced by the following rows:

o1	opc2	size	o3	Instruction page	Architecture version
0	010	10	0	VCVTB - Half-precision to single-precision variant	-
0	010	11	0	VCVTB - Half-precision to double-precision variant	-

and Row '0 010 - 1 VCVTT - Half-precision to double-precision variant -' is replaced by the following rows:

o1	opc2	size	o3	Instruction page	Architecture version
0	010	10	1	VCVTT - Half-precision to single-precision variant	-
0	010	11	1	VCVTT - Half-precision to double-precision variant	-

In section F4.1.16 (Floating-point data-processing), in sub-section 'Floating-point data-processing (two registers)' the following changes are made:

Row '0 010 - 0 VCVTB - Half-precision to double-precision variant -' is replaced by the following rows:

o1	opc2	size	o3	Instruction page	Architecture version
0	010	10	0	VCVTB - Half-precision to single-precision variant	-
0	010	11	0	VCVTB - Half-precision to double-precision variant	-

and Row '0 010 - 1 VCVTT - Half-precision to double-precision variant -' is replaced by the following rows:

o1	opc2	size	o3	Instruction page	Architecture version
0	010	10	1	VCVTT - Half-precision to single-precision variant	-
0	010	11	1	VCVTT - Half-precision to double-precision variant	-

2.146 D17240

In section D13.4.7 (PMCR_EL0, Performance Monitors Control Register (EL0)), the text in the description of the 'C' fields that reads:

```
The value of PMCR_EL0.LC is ignored, and bits [63:0] of all affected event counters
 are reset.
```

is updated to read

```
The value of PMCR_EL0.LC is ignored, and bits [63:0] of the cycle counter are reset.
```

An equivalent change is made in section G8.4.9 (AArch32 PMCR, Performance Monitors Control Register).

The sentence is also added in section I5.3.17 (PMCR_EL0, Performance Monitors Control Register (EL0)).

2.147 D17247

In section J1.3 (Shared pseudocode), the function IsSErrorEdgeTriggered() does not check the IDS bit of the Instruction Specific Syndrome to determine if the SError is **IMPLEMENTATION DEFINED**. Furthermore, in the case where an exception is routed to AArch64 in AArch32.TakePhysicalSErrorException(), IsSErrorEdgeTriggered() will be called from AArch32 state and the incorrect test will occur.

The code that reads:

```
AArch64.TakePhysicalSErrorException()
    ...
    if IsSErrorEdgeTriggered(syndrome) then
        ClearPendingPhysicalSError();
```

Is updated to read:

```
AArch64.TakePhysicalSErrorException()
    ...
    bits(2) target_el;
    if PSTATE.EL == EL3 || route_to_el3 then
        target_el = EL3;
    elsif PSTATE.EL == EL2 || route_to_el2 then
        target_el = EL2;
    else
        target_el = EL1;

    if IsSErrorEdgeTriggered(target_el, syndrome) then
        ClearPendingPhysicalSError();
```

The code that reads:

```
boolean IsSErrorEdgeTriggered(bits(24) syndrome)
    ...
        if UsingAArch32() && syndrome<11:10> != '00' then
            // AArch32 and not Uncontainable.
            return TRUE;
        if !UsingAArch32() && syndrome<23> == '0' && syndrome<5:0> != '000000' then
            // AArch64 and neither IMPLEMENTATION DEFINED syndrome nor
 Uncategorized.
            return TRUE;
    return boolean IMPLEMENTATION_DEFINED \"Edge-triggered SError\";
```

Is corrected to read:

```
boolean IsSErrorEdgeTriggered(bits(2) target_el, bits(25) syndrome)
    ...
        if ELUsingAArch32(target_el) then
            if syndrome<11:10> != '00' then
                // AArch32 and not Uncontainable.
                return TRUE;
        else
            if syndrome<24> == '0' && syndrome<5:0> != '000000' then
                // AArch64 and neither IMPLEMENTATION DEFINED syndrome nor
 Uncategorized.
                return TRUE;
    return boolean IMPLEMENTATION_DEFINED \"Edge-triggered SError\";
```

2.148 D17249

In section D5.10.2 (TLB maintenance instructions), subsection 'Ordering and completion of TLB maintenance instructions', the text that reads:

```
In an implementation that implements FEAT_ETS:
- A TLB maintenance instruction that applies only to translations without execute
 permission executed by a PE, PEx, can complete at any time after it is issued, but
 is only guaranteed to be finished for a PE, PEx, after the execution of DSB.
```

is updated to read:

```
In an implementation that implements FEAT_ETS:
- A TLB maintenance instruction that applies only to translations without execute
 permission and where the later translations also do not have execute permission,
 executed by a PE, PEx, can complete at any time after it is issued, but is only
 guaranteed to be finished for a PE, PEx, after the execution of DSB.
```

2.149 D17252

In section D4.4.8 (A64 Cache maintenance instructions), in table D4-7 'Effects of virtualization and security on the maintenance instructions', the text in the entry for 'Invalidate All: IC IALLU, IC IALLUIS' that reads:

```
* EL1 when the Effective value of SCR_EL3.{EEL2, NS} is {0,0}, EL2 when SCR_EL3.EEL2
 is 1, or EL3, all instruction cache lines.
```

is corrected to:

```
EL1 when the Effective value of SCR_EL3.{EEL2, NS} is {0,0}, EL2 when the SCR_EL3.
{EEL2, NS} is {1, 0}, or EL3, all instruction cache lines.
```

2.150 D17256

In section F5.1.117 (MRS), the Pseudocode for MRS operation incorrectly masks SSBS, bit(23). The mask computation is also simplified to align with the intent.

The code that reads:

```
       // CPSR has same bit assignments as SPSR, but with the IT, J, SS, IL, and T
bits masked out.
       bits(32) mask = '11111000 00001111 00000011 11011111';
       if HavePANExt() then
           mask<22> = '1';
       if HaveDITExt() then
           mask<21> = '1';
       psr_val = GetPSRFromPSTATE(AArch32_NonDebugState) AND mask;
```

is updated to read:

```
       // CPSR has same bit assignments as SPSR, but with the IT, J, SS, IL, and T
bits masked out.
       bits(32) mask = '11111000 11101111 00000011 11011111';
       psr_val = GetPSRFromPSTATE(AArch32_NonDebugState) AND mask;
```

2.151 C17257

In section D2.12.10 (Additional Considerations), in subsection 'Synchronization and the software step state machine', the text that currently states:

```
Any of the following can cause transitions between software step states:
 - A direct write to a System register.
 - A direct write to a Special-purpose register.
 - A write to an external debug register that affects the routing of debug
 exceptions.

Because the software step state machine indirectly reads these registers, it is not
 guaranteed to observe any new values until after a Context synchronization event
 has occurred.

In the time between a write to one of these registers and the next Context
 synchronization event, it is CONSTRAINED UNPREDICTABLE whether software step uses
 the state of the PE before the write, or the state of the PE after the write.
```

is updated to read:

```
Any of the following can cause transitions between software step states:
 - A direct write to a System register.
 - A direct write to a Special-purpose register.
 - A write to an external debug register.

The software step state machine indirectly reads some of these registers and so
 is not guaranteed to observe any new values until after a Context synchronization
 event has occurred.

Between a write to the register and the next Context synchronization event, it is
 CONSTRAINED UNPREDICTABLE whether software step uses the state of the PE before the
 write, or the state of the PE after the write.
```

2.152 D17258

In section D13.2.47 (HCR_EL2, Hypervisor Configuration Register), in the definition of the NV2 bit, the following sentence is added:

```
When HCR_EL2.NV==0, this bit is treated as 0 for all purposes other than direct
 reads and writes of this bit.
```

2.153 D17262

In section D11.2.1 (The physical counter), in subsection 'The physical offset register', the text that reads:

```
When EL2 is not enabled for the current Security state, or when CNTHCTL_EL2.ECV is
 0, then the behavior of the counters and timers is as described for Armv8.5 and the
 optional physical offset is not used.
```

is updated to read:

```
When EL2 is not enabled for the current Security state, or when CNTHCTL_EL2.ECV is
 0, then:
- An MRS to CNTPCT_EL0 from either EL0 or EL1 that is not trapped will return the
 value PCount<63:0>.
- The Physical Offset is treated as zero for all timer and counter calculations
 involving the Physical Offset.
```

2.154 R17265

In sections D3.1 (About self-hosted trace) and G3.1 (About self-hosted trace), the text that reads:

```
If an Armv8.4-compliant PE implements an ETM Architecture PE Trace Unit, FEAT_TRF
 extension must be implemented.

If an Armv8.4-compliant PE implements a Trace Unit that is not an ETM Architecture
 PE Trace Unit, Arm recommends that FEAT_TRF extension is implemented, but this is
 not mandatory.

If the self-hosted trace extensions are implemented, the PE Trace Unit must
 implement the system register interface.
```

is relaxed to read:

```
If an Armv8.4-compliant PE implements an ETM Architecture PE Trace Unit that
 includes the ETM System register interface, FEAT_TRF must be implemented.

If an Armv8.4-compliant PE implements a Trace Unit that is either not an ETM
 Architecture PE Trace Unit or does not implement the ETM System register interface,
 Arm recommends that FEAT_TRF is implemented, but this is not mandatory.
```

Similarly in section A2.7.1 (Architectural features added by Armv8.4), in subsection 'FEAT_TRF, Self-hosted Trace Extensions', the text that reads:

```
If an ETM Architecture PE Trace Unit is implemented, this feature is mandatory,
 and the ETM PE Trace Unit must implement System register access to its control
 registers. If a different PE Trace Unit is implemented, this feature is OPTIONAL.
```

is relaxed to read:

```
If an ETM Architecture PE Trace Unit is implemented and the ETM PE Trace Unit
 includes System register access to its control registers, this feature is
 mandatory. If a different PE Trace Unit is implemented or the ETM PE Trace Unit
 does not include System register access to its control registers, this feature is
 OPTIONAL.
```

2.155 D17282

In section C6.2 (Alphabetical list of A64 base instructions), the STG and SUBP(S) instructions are conditional on the implementation of FEAT_MTE. This check is missing from the pre-index and post-index forms of STG and also from SUBP(S).

The following code is added to the decode Pseudocode:

```
if !HaveMTEExt() then UNDEFINED;
```

2.156 D17285

In section I5.8.32 (ERR<n>STATUS, Error Record Primary Status Register, n = 0 - 65534), `0x1A` is added to the SERR field to cover any other error detected in the internal state of the component.

2.157 D17287

In Section D13.2.62 (ID_AA64MMFR1_EL1, AArch64 Memory Model Feature Register 1), a new ID field, nTLBPA, is added at bits[51:48]. The new field has the following definition:

0b0000 The intermediate caching of translation table walks might include non-coherent caches of previous valid translation table entries since the last completed relevant TLBI applicable to the PE where either:

- The caching is indexed by the physical address of the location holding the translation table entry.
- The caching is used for stage 1 translations and is indexed by the intermediate physical address of the location holding the translation table entry.

0b0001 The intermediate caching of translation table walks does not include non-coherent caches of previous valid translation table entries since the last completed TLBI applicable to the PE where either:

- The caching is indexed by the physical address of the location holding the translation table entry.
- The caching is used for stage 1 translations and is indexed by the intermediate physical address of the location holding the translation table entry.

All other values are reserved.

The equivalent field is allocated in ID_MMFR5[7:4]/ID_MMFR5_EL1[7:4] for AArch32.

In section D5.10.1 (General TLB maintenance requirements), the text that reads:

```
Such entries might be held in intermediate TLB caching structures that are used
 during a translation table walk and that are distinct from the data caches in that
 they are not required to be invalidated as the result of writes of the data. The
 architecture makes no restriction of the form of these intermediate TLB caching
 structures.
```

is enhanced to read:

```
Such entries might be held in intermediate TLB caching structures that are used
 during a translation table walk and that are distinct from the data caches in that
 they are not required to be invalidated as the result of writes of the data. The
 architecture makes no restriction on the form of these intermediate TLB caching
 structures when these caches are indexed by their input address. The architecture
 does not restrict having either:
- Translation table entry caching that is indexed by the physical address of the
 location holding the translation table entry.
- Translation table entry caching that is used for stage 1 translations and
 is indexed by the intermediate physical address of the location holding the
 translation table entry. However, FEAT_nTLBPA allows software discoverability of
 whether such caches exist, such that if FEAT_nTLBPA is implemented, such caching is
 not implemented.

If all of the following are true, a TLB maintenance instruction will ensure that
 any physical address or intermediate physical address indexed cached copies of
 translation table entries are invalidated for a PE:
- The TLB maintenance instruction applies to that PE with the context information
 that is relevant to translation table entry caching that is either:
-- Indexed by the physical address of the location holding the translation table
 entry.
-- Stage 1 translation information that is indexed by the intermediate physical
 address of the location holding the translation table entry.
- FEAT_nTLBPA is not implemented.
```

and the following Note is added:

```
Any TLB caching based on the physical address or intermediate physical address obeys
 the other rules regarding the caching to TLB entries described in this manner,
 including restrictions on types of entries that cannot be held in a TLB, and a
 requirement that entries held in a TLB are distinguished by context information
 such as translation regime, VMID, and ASID.
```

The equivalent text is added for AArch32 in section G5.9.5 (The scope of TLB maintenance instructions).

2.158 C17288

In section D9.4 (Enabling profiling), the text that currently reads:

```
Profiling is disabled if the Profiling Buffer is disabled, including when:
 - PMBLIMITR_EL1.E is cleared to 0 or PMBSR_EL1.S is set to 1.
 - Executing at a higher Exception level than the Profiling Buffer owning Exception
 level.
 - Executing in the Security state that is not the Security state of the owning
 Exception level.
 - The PE is in Debug state.
```

is updated to read:

```
Profiling is enabled when all of the following are true:
 - The PE is in AArch64 state.
 - PMBLIMITR_EL1.E is 1 and PMBSR_EL1.S is 0.
 - The PE is executing at either the Profiling Buffer owning Exception level or any
 lower Exception level.
 - The PE is executing in the Security state of the owning Exception level.
 - The PE is in Non-debug state.
 - PMSCR_EL1.{E1SPE, E0SPE} and PMSCR_EL2.{E2SPE, E0HSPE} enable profiling at the
 current Exception level.
```

2.159 D17292

In a future release, Arm will introduce specific reset domains for the following register specifications:

- Timer reset domain for external Timer registers. These are currently IMPLEMENTATION DEFINED, and referenced generically as reset in the registers.

- AMU reset domain, for AMU registers that are currently indicated as reset on Cold reset in the registers, or IMPLEMENTATION DEFINED in D8.2.3 (Power and reset domains).

- GIC reset domain for GICD, GICR, GITS registers. These are currently generically referenced as reset in the registers.

- MSC reset domain for MPAM registers prefixed with: MPAMCFG, MPAMF, and MSMON.

Additionally, RW fields that currently do not specify a reset domain and reset value will be updated with a specific reset domain and reset value (typically architecturally **UNKNOWN**).

2.160 D17297

In B2.3.9 (Restrictions on the effects of speculation), a new subsection is added, titled 'Restrictions on exploitative control of speculative execution'

```
The execution of some code (code1) can 'exploitatively control speculative
 execution' of some other code (code2) if and only if all of the following apply:
* The actions of code1 can influence the speculative execution of code2 to cause an
 irreversible change to the microarchitectural state of the PE that is indicative of
 some architectural state accessible to the execution context of code2.
* Code1 has control in determining the choice of the architecture state that causes
 the irreversible change to the microarchitectural state.
* The irreversible changes to the microarchitectural state of the PE can be measured
 by code executing in an execution context other than that of code2 to allow the
 retrieval of the architectural state in a computationally feasible manner.
```

In B2.3.9 (Restrictions on the effects of speculation), in subsection 'Restrictions on the effects of speculation from Armv8.5', the text that reads:

```
- For all execution prediction resources that predict address or register values,
 speculative execution at one hardware defined context should be separated in a
 hard-to-determine manner from the predictions trained in a different hardware
 defined context.
```

is updated to read:

```
- Code running in one hardware-defined context cannot exploitatively control
 speculative execution of code in a different hardware-defined context as a result
 of the behavior of any execution prediction resources that predict address or
 register values.
```

The text:

```
- When in AArch64 state, the current SCXTNUM_ELx value.
```

is updated to read:

```
- When in AArch64 state, the current SCXTNUM_ELx value if SCXTNUM_ELx is implemented
 and the hardware identifies that SCXTNUM_ELx is part of the context. Where
 SCXTNUM_ELx is not included as part of the hardware-indicated context, an
 implementation can further identify that branch targets trained for branches
 situated at one address can control speculative execution of branches situated at
 different addresses only in a hard-to-determine way.
```

In D13.2.64 (ID_AA64PFR0_EL1, AArch64 Processor Feature Register 0), the CSV2 values:

```
0b0000 This Device does not disclose whether branch targets trained in one hardware
 described context can affect speculative execution in a different hardware
 described context.

0b0001 Branch targets trained in one hardware described context can only affect
 speculative execution in a different hardware described context in a hard-to-
determine way. Contexts do not include the SCXTNUM_ELx register contexts, and these
 registers are not supported.
```

```
0b0010 Branch targets trained in one hardware described context can only affect
 speculative execution in a different hardware described context in a hard-to-
determine way. Contexts include the SCXTNUM_ELx register contexts, and these
 registers are supported.
```

are replaced with:

```
0b0000 This device does not disclose whether branch targets trained in one hardware-
described context can exploitatively control speculative execution in a different
 hardware-described context.

0b0001 Branch targets trained in one hardware-described context can exploitatively
 control speculative execution in a different hardware-described context only in a
 hard-to-determine way. Contexts do not include the SCXTNUM_ELx register contexts,
 and these registers are not supported.

0b0010 Branch targets trained in one hardware-described context can exploitatively
 control speculative execution in a different hardware-described context only in
 a hard-to-determine way. Contexts include the SCXTNUM_ELx register contexts, and
 these registers are supported.
```

In D13.2.65 (ID_AA64PFR1_EL1, AArch64 Processor Feature Register 1), a new field, CSV2_frac, is added using bits 35:32. CSV2_frac is valid only when ID_AA64PFR0_EL1.CSV2== 0001. The values of this new field are:

```
0b0000: Branch targets trained in one hardware-described context can exploitatively
 control speculative execution in a different hardware-described context only in a
 hard-to-determine way. Contexts do not include the SCXTNUM_ELx register contexts,
 and these registers are not supported.

0b0001: Branch targets trained in one hardware-described context can exploitatively
 control speculative execution in a different hardware-described context only in a
 hard-to-determine way. Within a hardware-described context, branch targets trained
 for branches situated at one address can control speculative execution of branches
 situated at different addresses only in a hard-to-determine way. Contexts do not
 include the SCXTNUM_ELx register contexts, and these registers are not supported.

0b0010: Branch targets trained in one hardware-described context can exploitatively
 control speculative execution in a different hardware-described context only in a
 hard-to-determine way. Within a hardware-described context, branch targets trained
 for branches situated at one address can control speculative execution of branches
 situated at different addresses only in a hard-to-determine way. Contexts do not
 include the SCXTNUM_ELx register contexts, but these registers are supported in
 hardware.

All other values are reserved.
```

In sections D13.2.82 (ID_PFR0_EL1, AArch32 Processor Feature Register 0) and G8.2.98 (ID_PFR0, Processor Feature Register 0), the CSV2 field values:

```
0b0000 This Device does not disclose whether branch targets trained in one hardware
 described context can affect speculative execution in a different hardware
 described context.

0b0001 Branch targets trained in one hardware described context can only affect
 speculative execution in a different hardware described context in a hard-to-
determine way.
```

are replaced with:

```
0b0000 This device does not disclose whether branch targets trained in one hardware-
described context can exploitatively control speculative execution in a different
 hardware-described context.

0b0001 Branch targets trained in one hardware-described context can exploitatively
 control speculative execution in a different hardware-described context only in a
 hard-to-determine way.

0b0010 Branch targets trained in one hardware-described context can exploitatively
 control speculative execution in a different hardware-described context only in a
 hard-to-determine way.  Within a hardware-described context, branch targets trained
 for branches situated at one address can control speculative execution of branches
 situated at different addresses only in a hard-to-determine way.
```

2.161 R17302

In section D6.4.1 (Virtual address translation), the following text is added after the section starting 'If a memory location is marked as Untagged, a data...':

```
If a memory location is marked both as Tagged and as Non-shared, it is
 IMPLEMENTATION DEFINED whether the memory location is treated as Tagged or
 Untagged.
```

2.162 D17308

In section D10.2.6 (Events packet), in subsection 'Events packet payload', the field description that reads:

```
E[11], byte 1, bit [11], when SZ == 0b10, or SZ == 0b11

Alignment.
<< definition of event >>

Byte 1 bit [3], when SZ == 0b01
This bit reads-as-zero.
```

is corrected to:

```
E[11], byte 1 bit [3], when SZ == 0b01, when SZ == 0b10, or when SZ == 0b11

Alignment.
<< definition of event >>
```

2.163 R17309

In section D6.5 (PE access to Allocation Tags), the following relaxation is added:

```
A read of an Allocation Tag that returns zero due to access to Allocation tags being
 disabled by HCR_EL2.ATA, SCR_EL3.ATA or SCTLR_ELx.{ATA, ATA0}, or due to the memory
 type not having the Tagged attribute, is permitted to generate an External abort if
 a read of data from the same address would generate an External abort.
```

2.164 D17318

In section D1.12.4 (Synchronous exception prioritization for exceptions taken to AArch64 state), for priority number 13, the text that reads:

```
Attempting to execute an instruction that is defined never to be accessible at the
 current Exception level regardless of any enables or traps.
```

is updated to read:

```
Attempting to execute an instruction that is defined never to be accessible at the
 current Exception level and Security state regardless of any enables or traps.
```

2.165 D17323

In light of D17297, which added ID_AA64PFR1_EL1.CSV2_frac and updated ID_AA64PFR0_EL1.CSV2, the following features are redefined:

- FEAT_CSV2 when ID_AA64PFR0_EL1.CSV2 is 0b0001 and ID_AA64PFR1_EL1.CSV2_frac is 0b0000.
- FEAT_CSV2_1p1 when ID_AA64PFR0_EL1.CSV2 is 0b0001 and ID_AA64PFR1_EL1.CSV2_frac is 0b0001.
- FEAT_CSV2_1p2 when ID_AA64PFR0_EL1.CSV2 is 0b0001 and ID_AA64PFR1_EL1.CSV2_frac is 0b0010.
- FEAT_CSV2_2 when ID_AA64PFR0_EL1.CSV2 is 0b0010.

These feature descriptions are added to section A2.2 (Architectural features within Armv8.0 architecture).

2.166 D17330

In section D4.4.8 (A64 Cache maintenance instructions), in the subsection 'Effects of virtualization and Security state on the cache maintenance instructions', the text that currently reads:

```
Table D4-7 shows the effects of virtualization and security on the cache maintenance
 instructions. In the table, the Specified entries are entries that the architecture
 requires the instruction to affect. The rules described in 'General behavior of the
 caches on page D4-2493' mean that an instruction might also affect other entries.
```

is clarified to read:

```
Table D4-7 shows the effects of virtualization and security on the cache maintenance
 instructions. In the table, the Specified entries are entries that the architecture
 requires the instruction to affect.

Note: The rules described in 'General behavior of the caches on page D4-2493' mean
 that an instruction might also cause changes to other entries consistent with those
 rules, and which do not cause loss of dirty data.
```

2.167 R17331

In section D5.4.11 (Hardware management of the Access flag and dirty state), in the subsection 'Hardware management of dirty state', in the bullet list under:

```
The architecture does not permit updates to AP[2] and S2AP[1] by the hardware
 management of the dirty state mechanism to occur as a result of speculative
 accesses by the PE that are not performed architecturally, except that for
 translation table entries for which the value of DBM is 1:
```

A new bullet is added:

```
* The dirty state information for a stage of translation can be updated to indicate
 dirty even if the store performing the access has an exception which has a lower
 priority than a Permission fault from that stage of translation, as determined
 by sections D1.12.4 (Synchronous exception prioritization for exceptions taken to
 AArch64 state) and D5.8.3 (AArch64 state prioritization of synchronous aborts from
 a single stage of address translation).
```

2.168 D17335

In D13.2.112 (SCR_EL3, Secure Configuration Register), the text in the FGTEn field description that reads:

```
0b EL2 accesses to HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGRTR_EL2, HFGITR_EL2 and
 HFGWTR_EL2 registers are trapped to EL3, and those registers behave as if all bits
 are set to 0.
```

is updated to read:

```
0b EL2 accesses to HAFGRTR_EL2, HDFGRTR_EL2, HDFGWTR_EL2, HFGRTR_EL2, HFGITR_EL2 and
 HFGWTR_EL2 registers are trapped to EL3, and the traps to EL2 controlled by those
 registers are disabled.
```

2.169 D17341

In section D5.10.2 (TLB maintenance instructions), in subsection 'Scope of the A64 TLB maintenance instructions', the text that reads:

```
The entries that the invalidations apply to are not affected by the state of any
 other control bits involved in the translation process. Therefore, the following
 is a non-exhaustive list of control bits that do not affect how a TLB  maintenance
 instruction updates the TLB entries
```

is simplified to read:

```
The entries that the invalidations apply to are not affected by the state of any
 other control bits involved in the translation process.

Note: In particular, in response to a commonly asked question, TLB maintenance
 applies when memory translation is disabled.
```

2.170 D17342

In D13.2.118 (SCXTNUM_EL2, EL2 Read/Write Software Context Number), the HCR_EL2.<NV2,NV1,NV> == '011' trap is added at EL1 for the accessor to SCXTNUM_EL1.

The MRS accessor at EL1:

```
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED
 \"EL3 trap priority when SDD == '1'\" && SCR_EL3.EnSCXT == '0' then
      UNDEFINED;
  elsif EL2Enabled() && HCR_EL2.EnSCXT == '0' then
      AArch64.SystemAccessTrap(EL2, 0x18);
```

```
   elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
 HFGWTR_EL2.SCXTNUM_EL1 == '1' then
       AArch64.SystemAccessTrap(EL2, 0x18);
   elsif HaveEL(EL3) && SCR_EL3.EnSCXT == '0' then
       if Halted() && EDSCR.SDD == '1' then
           UNDEFINED;
       else
           AArch64.SystemAccessTrap(EL3, 0x18);
   elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
       NVMem[0x188] = X[t];
   else
       SCXTNUM_EL1 = X[t];
```

is updated to:

```
elsif PSTATE.EL == EL1 then
   if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED
 \"EL3 trap priority when SDD == '1'\" && SCR_EL3.EnSCXT == '0' then
       UNDEFINED;
   elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '011' then
       AArch64.SystemAccessTrap(EL2, 0x18);
   elsif EL2Enabled() && HCR_EL2.EnSCXT == '0' then
       AArch64.SystemAccessTrap(EL2, 0x18);
   elsif EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
 HFGWTR_EL2.SCXTNUM_EL1 == '1' then
       AArch64.SystemAccessTrap(EL2, 0x18);
   elsif HaveEL(EL3) && SCR_EL3.EnSCXT == '0' then
       if Halted() && EDSCR.SDD == '1' then
           UNDEFINED;
       else
           AArch64.SystemAccessTrap(EL3, 0x18);
   elsif EL2Enabled() && HCR_EL2.<NV2,NV1,NV> == '111' then
       NVMem[0x188] = X[t];
   else
       SCXTNUM_EL1 = X[t];
```

The equivalent change is made for the MSR accessor for SCXTNUM_EL2 at EL1.

2.171 D17359

In section B2.7.2 (Device memory) in the subsection 'Early Write Acknowledgement', the text that reads:

```
For memory system endpoints where the system architecture in which the PE is
 operating requires that acknowledgement of a write comes from the endpoint,
 assigning the No Early Write Acknowledgement attribute to a Device memory location
 guarantees that:
- Only the endpoint of the write access returns a write acknowledgement of the
 access.
- No earlier point in the memory system returns a write acknowledgement.
```

is clarified to read:

```
If the No Early Write Acknowledgement attribute is assigned for a Device memory
 location then:
```

```
- For memory system endpoints where the system architecture in which the PE is
 operating requires that acknowledgement of a write comes from the endpoint, it is
 guaranteed that:
-- Only the endpoint of the write access returns a write acknowledgement of the
 access.
-- No earlier point in the memory system returns a write acknowledgement.
- For memory system endpoints where the system architecture in which the PE is
 operating does not require that acknowledgement of a write comes from the endpoint,
 the acknowledgement of write is not required to come from the endpoint.

 Note - it is not expected that a write with the No Early Write Acknowledgement
 attribute assigned for a Device memory location where the system architecture in
 which the PE is operating does not require that acknowledgement of a write will
 generate an abort if the equivalent write to the same location without the No Early
 Write Acknowledgement attribute assigned does not generate an abort.
```

2.172 D17367

In sections D13.2.36-8 (ESR_EL1-3, Exception Syndrome Register (EL1-3)), in the subsection 'ISS encoding for an exception from a trapped floating-point exception', in the definition of the TFV bit, the line that reads:

```
It is IMPLEMENTATION DEFINED whether this field is set to 0 on an exception
 generated by a trapped floating point exception from a vector instruction.
```

is clarified to read:

```
It is IMPLEMENTATION DEFINED whether this field is set to 0 on an exception
 generated by a trapped floating point exception from an instruction that is
 performing floating-point operations on more than one lane of a vector.
```

2.173 D17387

In section H9.2.42 (EDSCR, External Debug Status and Control Register), the following text is added in the definition of INTdis, bits [23:22] when FEAT_Debugv8p4 is implemented:

```
When FEAT_Debugv8p4 is implemented, bit[23] of the register is RES0.
```

and references to External(Secure)DebugEnabled are corrected to External(Secure)InvasiveDebugEnabled.

The following text is added when FEAT_Debugv8p4 is not implemented:

```
Support for the values 0b01 and 0b10 is IMPLEMENTATION DEFINED. If these values
 are not supported, they are reserved. If programmed with a reserved value, the PE
 behaves as if INTdis has been programmed with a defined value, other than for a
 direct read of EDSCR, and the value returned by a read of EDSCR.INTdis is UNKNOWN.
```

and the following condition is added to the value definitions:

```
This field is ignored by the PE and treated as zero when
 ExternalInvasiveDebugEnabled() == FALSE.
```

2.174 D17396

In section D1.14.3 (EL2 configurable controls), the text that reads:

```
These controls are ignored in Secure state.
```

is corrected to read:

```
If Secure EL2 is implemented and enabled, configurable instruction controls
 available at EL2 apply in Secure state. If Secure EL2 is not implemented or not
 enabled, the configurable instruction controls available at EL2 are ignored in
 Secure state.
```

2.175 D17401

In section H3.2.4 (Detailed Halting Step state machine behavior), the text that reads:

```
The PE enters the active-not-pending state:
* By exiting Debug state with EDECR.SS == 1.
```

is corrected to read:

```
The PE enters the active-not-pending state:
* By exiting Debug state to a state where halting is allowed with EDECR.SS == 1.
```

and the text that reads:

```
When the PE is in the active-not-pending state it does one of the following:
* It executes one instruction and does one of the following:
-- Completes it without generating a synchronous exception.
-- Generates a synchronous exception.
-- Generates a debug event that causes entry to Debug state.
```

is clarified to read:

```
When the PE is in the active-not-pending state it does one of the following:
* It executes one instruction and does one of the following:
-- Completes it without taking a synchronous exception.
-- Takes a synchronous exception generated by the instruction.
```

```
-- Generates a debug event that causes entry to Debug state.
```

The same change from 'generated' to 'taken' is made in the paragraphs following this change.

2.176 D17403

In section D13.2.42 (FPEXC32_EL2, Floating-Point Exception Control register), the text in the DEX field description that reads:

```
0b1 The exception was generated during the execution of an unallocated encoding.
FPEXC32_EL2.TFV is valid and indicates the cause of the exception.
```

is corrected to read:

```
0b1 The exception was generated during the execution of an allocated encoding.
FPEXC32_EL2.TFV is valid and indicates the cause of the exception.
```

The equivalent edit is made in section G8.2.53 (FPEXC, Floating-Point Exception Control register).

2.177 D17405

In section A2.7.1 (Architectural features added by Armv8.4), the text that reads:

```
FEAT_TLBIOS provides TLBI maintenance instructions that extend to the Outer
 Shareable domain and TLBI invalidation instructions that apply to a range of input
 addresses.
```

is changed to read:

```
FEAT_TLBIOS provides TLBI maintenance instructions that extend to the Outer
 Shareable domain.
```

And the text that reads:

```
FEAT_TLBIRANGE provides TLBI maintenance instructions that extend to the Outer
 Shareable domain and TLBI invalidation instructions that apply to a range of input
 addresses.
```

is changed to read:

```
FEAT_TLBIRANGE provides TLBI maintenance instructions that apply to a range of
 input addresses. FEAT_TLBIRANGE being implemented implies that FEAT_TLBIOS is
 implemented.
```

2.178 D17417

In section A2.7.1 (Architectural features added by Armv8.4), in the subsection titled 'FEAT_RASv1p1, RAS Extension v1.1', the text that currently reads:

```
FEAT_RASv1p1 implements RAS System Architecture v1.1 and adds support for:

- FEAT_DoubleFault.
- Simplifications to ERR<n>STATUS.
- Additional ERR<n>MISC<m> registers.
- The OPTIONAL RAS Common Fault Injection Model Extension.
```

is corrected to read:

```
FEAT_RASv1p1 implements RAS System Architecture v1.1 and adds support for:

- Simplifications to ERR<n>STATUS.
- Additional ERR<n>MISC<m> registers.
- The OPTIONAL RAS Common Fault Injection Model Extension.
```

In section D13.2.64 (ID_AA64PFR0_EL1, AArch64 Processor Feature Register 0), in the RAS field, the text that currently reads:

```
0b0010 FEAT_RASv1p1 present. As 0b0001, and adds support for:

- If EL3 is implemented, FEAT_DoubleFault.
```

is corrected to read:

```
0b0010 FEAT_RASv1p1 and, if EL3 is implemented, FEAT_DoubleFault present. As 0b0001,
 and adds support for:

- If EL3 is implemented, FEAT_DoubleFault.
```

2.179 R17420

In section D13.2.115 (SCTLR_EL3, System Control Register (EL3)), in the IESB field, the text that currently reads:

```
When the PE is in Debug state, the effect of this field is CONSTRAINED
 UNPREDICTABLE, and its Effective value might be 0 or 1 regardless of the value of
 the field.
```

is corrected to read:

```
When the PE is in Debug state, the effect of this field is CONSTRAINED
 UNPREDICTABLE, and its Effective value might be 0 or 1 regardless of the value of
 the field and, if implemented, SCR_EL3.NMEA.
```

Within the same section, the text that currently reads:

```
When FEAT_DoubleFault is implemented, and the Effective value of SCR_EL3.NMEA is 1,
 this field is ignored and its Effective value is 1.
```

is corrected to read:

```
When FEAT_DoubleFault is implemented, the PE is in Non-debug state, and the
 Effective value of SCR_EL3.NMEA is 1, this field is ignored and its Effective value
 is 1.
```

2.180 D17423

In section D5.10.2 (TLB maintenance instructions), in the subsection 'Scope of the A64 TLB maintenance instructions', for each of the entries VA, VAL, VAA, VAAL, the bullet list that currently reads:

```
- The Security state specified by SCR_EL3.NS and SCR_EL3.EEL2.
- For the Secure or Non-secure EL1&0, when EL2 is enabled, translation regime, the
 current VMID.
```

is changed to read:

```
- The Security state specified by SCR_EL3.NS and SCR_EL3.EEL2.
- For the Secure or Non-secure EL1&0, when EL2 is enabled, translation regime, the
 current VMID.
- For instructions specifying the EL2 Exception level, the current distinguishing
 of the translation regime between EL2&0 or EL2, as determined by the setting of
 HCR_EL2.E2H.
```

2.181 D17433

In section D10.1.3 (Byte Order), the text that currently reads:

```
Header bytes and payload bytes are written in ascending address order. Within a
 payload value, values are written in little-endian byte order.
```

is corrected to read:

```
This chapter describes header bytes and payload bytes in ascending memory address
 order. Within a payload value, values are in little-endian byte order.
```

Additionally, the following note is removed:

```
Note: This means that if the memory type accessed is non-Gathering Device, the
 architecture does not require a specific access granule size at the end device.
```

2.182 R17435

In section H9.2.25 (EDECCR, External Debug Exception Catch Control Register), the text in the description for each field that currently reads:

```
A value of the (NSR, SR, NSE, SE) field that enables an Exception Catch debug event
 for an Exception level that is not implemented is reserved. If the (NSR, SR, NSE,
 SE) field is programmed with a reserved value then:
- The PE behaves as if it is programmed with a defined value, other than for a read
 of EDECCR.
- The value returned for (NSR, SR, NSE, SE) by a read of EDECCR is UNKNOWN.
```

is changed to the following:

in the NSR field:

```
If EL<n> is not implemented then NSR<n> is RES0.
```

in the SR field:

```
If FEAT_SEL2 is not implemented then SR<2> is RES0. If EL<n> is not implemented then
 SR<n> is RES0.
```

in the NSE fields:

```
NSE<0> is RES0. If EL<n> is not implemented then NSE<n> is RES0.
```

in the SE fields:

```
SE<0> is RES0. If FEAT_SEL2 is not implemented then SE<2> is RES0. If EL<n> is not
 implemented then SE<n> is RES0.
```

2.183 C17438

In section D13.2.50 (HFGITR_EL2, Hypervisor Fine-Grained Instruction Trap Register), in the DCZVA field, bit [11], the following note is added to the specification:

```
Note: Unlike the HCR_EL2.TDZ bit, this bit does not have an impact on the
 DCZID_EL0.DZP bit.
```

2.184 D17441

In section D13.2.24 (CCSIDR2_EL1, Current Cache Size ID Register 2), the text that currently reads:

```
In an AArch64 only implementation, it is IMPLEMENTATION DEFINED whether reading this
 register gives an UNKNOWN value or is UNDEFINED.
```

is relaxed to read:

```
In an implementation which doesn't support AArch32 at EL1, it is IMPLEMENTATION
 DEFINED whether reading this register gives an UNKNOWN value or is UNDEFINED.
```

Similarly, in section G8.2.25 (CCSIDR2, Current Cache Size ID Register 2), the text that reads:

```
This register is present only when AArch32 is supported at any Exception level and
 FEAT_CCIDX is implemented. Otherwise, direct accesses to CCSIDR2 are UNDEFINED.
```

is relaxed to read:

```
This register is present only when AArch32 is supported at EL1 and FEAT_CCIDX is
 implemented. Otherwise, direct accesses to CCSIDR2 are UNDEFINED.
```

2.185 D17464

In section C5.6.1 (CFP RCTX, Control Flow Prediction Restriction by Context), the paragraph that reads:

```
When this instruction is complete and synchronized, control flow prediction does
 not permit later speculative execution within the target execution context to be
 observable through side channels.
```

is replaced by the following paragraph:

```
Control flow predictions determined by the actions of code in the target execution
 context(s) appearing in program order before the instruction cannot exploitatively
 control speculative execution occurring after the instruction is complete and
 synchronized.
```

The equivalent edits are made to sections C5.6.2 (CPP RCTX, Cache Prefetch Prediction Restriction by Context), C5.6.3 (DVP RCTX, Data Value Prediction Restriction by Context), C6.2.51 (CFP), C6.2.65 (CPP), C6.2.83 (DVP) and the corresponding AArch32 sections G8.2.26 (CFPRCTX, Control Flow Prediction Restriction by Context), G8.2.34 (CPPRCTX, Cache Prefetch Prediction Restriction by Context), and G8.2.50 (DVPRCTX, Data Value Prediction Restriction by Context).

2.186 D17478

In section D4.4.13 (Execution and data prediction restriction System instructions), the text that reads:

```
When FEAT_SPECRES is implemented, the System instructions listed in A64 System
 instructions for prediction restriction on page C5-756 prevent predictions based on
 information gathered from earlier execution within a particular execution context
 from affecting the later Speculative execution within that context, to the extent
 that the speculative execution is observable through side-channels.

The prediction restriction System instructions being used by a particular execution
 context apply to:
```

is clarified to read:

```
When FEAT_SPECRES is implemented, the System instructions listed in A64 System
 instructions for prediction restriction on page C5-756 prevent predictions based
 on information gathered from earlier execution within a particular execution
 context, termed for these instructions as an CTX,  from affecting the later
 Speculative execution within that CTX, to the extent that the speculative execution
 is observable through side-channels.

The prediction restriction System instructions being used by a particular CTX apply
 to:
```

Within the same section, the text that reads:

```
For these System instructions, the execution context is defined by:
```

is clarified to read:

```
For these System instructions, the CTX is defined by:
```