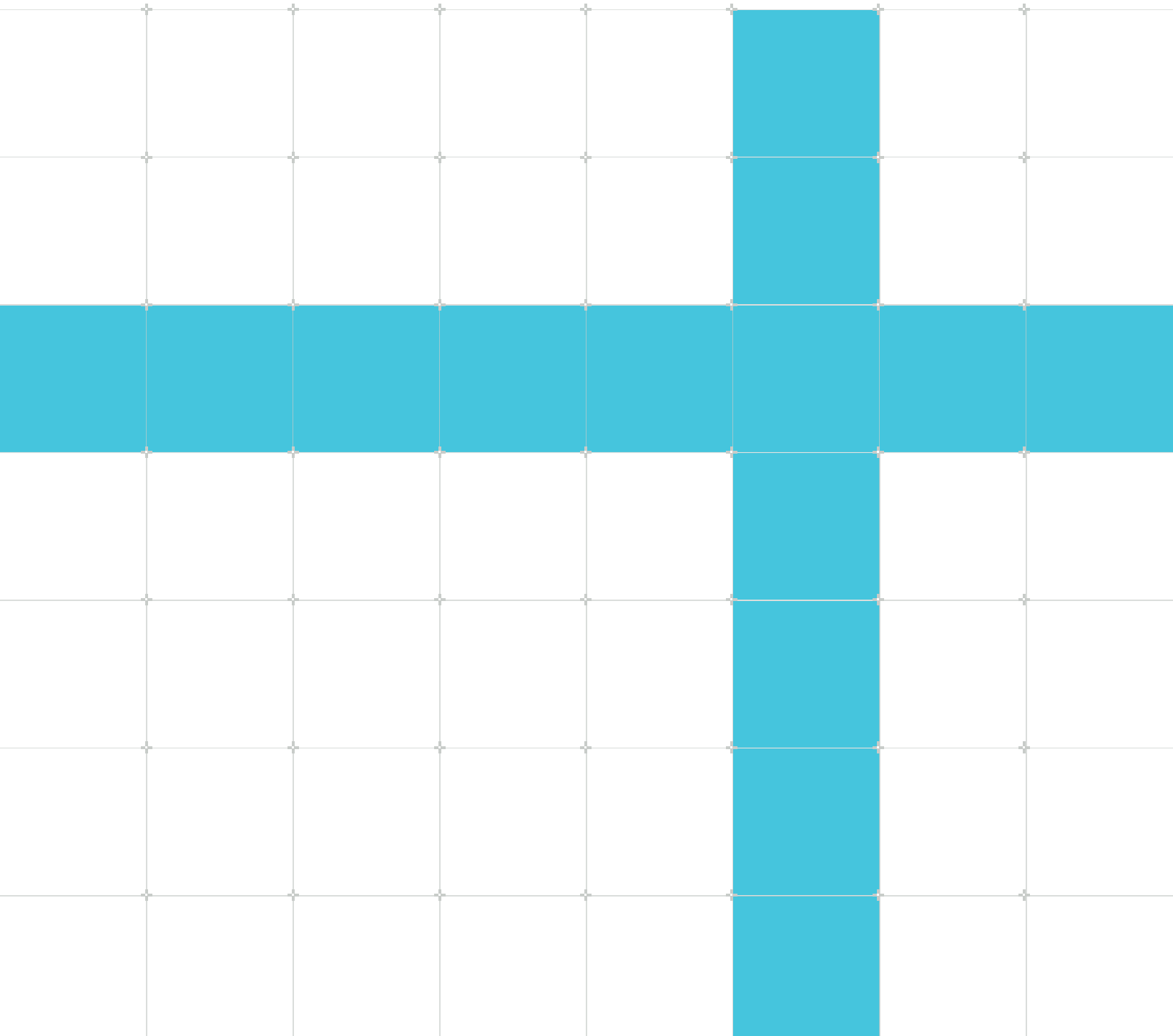
Arm® Architecture Reference Manual for A-profile architecture

Known issues in Issue I.a

Issue 04
102105_I.a_04_en



Arm® Architecture Reference Manual for A-profile architecture

Known issues in Issue I.a

Release information

Document history

Issue	Date	Confidentiality	Change
F.c-04	18 December 2020	Non-Confidential	Known Issues in Arm® Architecture Reference Manual, Issue F.c, as of 18 December 2020
G.b-05	31 January 2022	Non-Confidential	Known Issues in Arm® Architecture Reference Manual, Issue G.b, as of 7 January 2022
H.a-06	22 July 2022	Non-Confidential	Known Issues in Arm® Architecture Reference Manual, Issue H.a, as of 22 July 2022
I.a-00	5 August 2022	Non-Confidential	Known Issues in Arm® Architecture Reference Manual, Issue I.a, as of 5 August 2022
I.a-01	30 September 2022	Non-Confidential	Known Issues in Arm® Architecture Reference Manual, Issue I.a, as of 23 September 2022
I.a-02	31 October 2022	Non-Confidential	Known Issues in Arm® Architecture Reference Manual, Issue I.a, as of 21 October 2022
I.a-03	30 November 2022	Non-Confidential	Known Issues in Arm® Architecture Reference Manual, Issue I.a, as of 18 November 2022
I.a-04	6 January 2023	Non-Confidential	Known Issues in Arm® Architecture Reference Manual, Issue I.a, as of 16 December 2022

Proprietary Notice

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.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

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

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1 Introduction

1.1 Conventions

The following subsections describe conventions used in Arm documents.

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.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace underline`	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	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.
Caution	Recommendations. Not following these recommendations might lead to system failure or damage.
Warning	Requirements for the system. Not following these requirements might result in system failure or damage.
Danger	Requirements for the system. Not following these requirements will result in system failure or damage.
Note	An important piece of information that needs your attention.

Convention	Use
 Tip	A useful tip that might make it easier, better or faster to perform a task.
 Remember	A reminder of something important that relates to the information you are reading.

1.2 Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Access to Arm documents depends on their confidentiality:

- Non-Confidential documents are available at developer.arm.com/documentation. Each document link in the following tables goes to the online version of the document.
- Confidential documents are available to licensees only through the product package.

Arm product resources	Document ID	Confidentiality
Arm® Architecture Reference Manual for A-profile architecture, Issue I.a	DDI 0487I.a	Non-Confidential


Note

Arm tests its PDFs only in Adobe Acrobat and Acrobat Reader. Arm cannot guarantee the quality of its documents when used with any other PDF reader.

Adobe PDF reader products can be downloaded at http://www.adobe.com

1.3 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 for A-profile architecture (DDI 0487), Issue I.a.

Key

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

2.1 C15788

In section D8.13.5 (TLB maintenance instructions), in the subsection 'TLB maintenance instructions that apply to a range of addresses', the following text is added:

> It is possible for a TLB range maintenance instruction for a translation regime that supports two VA ranges to be issued with an address in the TTBR1 half of the virtual address space, and SCALE and NUM values such that the range exceeds the top of the address space. In this scenario, the address is not considered to wrap on overflow and the PE is not required to invalidate any entries inserted for the TTBR0 half of the VA space.

2.2 C16212

In section D17.2.156 (VSTTBR_EL2, Virtualization Secure Translation Table Base Register) and D17.2.158 (VTTBR_EL2, Virtualization Translation Table Base Register), in the field 'BADDR, bits [47:1]', the references to:

> stage 1 translation table base

are corrected to read:

> stage 2 translation table base

2.3 D16424

In section D7.3 (Mixed-endian support), the following footnote is added beneath Table D7-2 'Endianness support':

> When HCR_EL2.{E2H,TGE} == {1, 1}, the control is from SCTLR_EL2.E0E.

This footnote is linked to from the 'SCTLR_EL1.E0E' entry in the 'Explicit data accesses' column of the table.

Within the same section, the text that reads:

> - All Exception levels support mixed-endianness:
>   - SCTLR_ELx.EE is RW and SCTLR_EL1.E0E is RW.
> - Only EL0 supports mixed-endianness and EL1, EL2, and EL3 support only little-endianness:
>   - SCTLR_ELx.EE is **RES0** and SCTLR_EL1.E0E is RW.
> - Only EL0 supports mixed-endianness and EL1, EL2, and EL3 support only big-endianness:
>   - SCTLR_ELx.EE is RES1 and SCTLR_EL1.E0E is RW.
> - All Exception levels support only little-endianness:
>   - SCTLR_ELx.EE is **RES0** and SCTLR_EL1.E0E is **RES0**.
> - All Exception levels support only big-endianness:
>   - SCTLR_ELx.EE is RES1 and SCTLR_EL1.E0E is RES1.

is changed to read:

> - All Exception levels support mixed-endianness:
>   - SCTLR_ELx.EE, SCTLR_EL1.E0E, and SCTLR_EL2.E0E are RW.
> - Only EL0 supports mixed-endianness and EL1, EL2, and EL3 support only little-endianness:
>   - SCTLR_ELx.EE is **RES0**, and SCTLR_EL1.E0E and SCTLR_EL2.E0E are RW.
> - Only EL0 supports mixed-endianness and EL1, EL2, and EL3 support only big-endianness:
>   - SCTLR_ELx.EE is RES1, and SCTLR_EL1.E0E and SCTLR_EL2.E0E are RW.
> - All Exception levels support only little-endianness:
>   - SCTLR_ELx.EE, SCTLR_EL1.E0E, and SCTLR_EL2.E0E are **RES0**.
> - All Exception levels support only big-endianness:
>   - SCTLR_ELx.EE, SCTLR_EL1.E0E, and SCTLR_EL2.E0E are RES1.

A corresponding change is made in section B2.6.3 (Data endianness), where the text that reads:

> SCTLR_EL1.E0E, configurable at EL1 or higher, determines the data endianness for execution at EL0.

is changed to read:

> SCTLR_EL1.E0E, configurable at EL1 or higher, determines the data endianness for execution at EL0. When HCR_EL2.{E2H,TGE} == {1, 1}, the control is from SCTLR_EL2.E0E.

2.4 D16504

In section B2.3.11 (Memory Barriers), subsection 'Load-Acquire, Load-AcquirePC, and Store-Release', the text that reads:

> Load-Acquire, Load-AcquirePC and Store-Release, other than Load-Acquire Exclusive Pair and Store-Release-Exclusive Pair, access only a single data element. This access is single-copy atomic. The address of the data object must be aligned to the size of the data element being accessed, otherwise the access generates an Alignment fault.
>
> Load-Acquire Exclusive Pair and Store-Release Exclusive Pair access two data elements. The address supplied to the instructions must be aligned to twice the size of the element being loaded, otherwise the access generates an Alignment fault.

is corrected to read:

> Load-Acquire, Load-AcquirePC and Store-Release, other than Load-Acquire Exclusive Pair and Store-Release-Exclusive Pair, access only a single data element.
>
> Load-Acquire Exclusive Pair and Store-Release Exclusive Pair access two data elements.

2.5 D16716

In the Glossary definition of 'Context synchronization event', the list 'The effects of a Context synchronization event are:' has the following bullet points added:

> - The effect of the completion of any of the instructions added by FEAT_SPECRES is synchronized to the current execution context.
> - Restrictions on the effects of speculation (as described in B2.3.10 Restrictions on the effects of speculation) are observed.
> - Ensuring that the TSB CSYNC instruction is executed in the necessary order with respect to other instructions.
> - Profiling operations for all instructions that are executed in program order are synchronized by execution of a PSB CSYNC instruction before the Context synchronization event.

2.6 D16729

In section D17.2.43 (FPEXC32_EL2, Floating-Point Exception Control register), in the field 'EN, bit [30]', the text that reads:

> - For Advanced SIMD instructions only:
>   - CPACR.ASEDIS.
>   - If executing in Non-secure state, HCPTR.TASE and NSACR.NSTRCDIS.

is replaced with:

> - For Advanced SIMD instructions only:
>   - CPACR.ASEDIS.
>   - If executing in Non-secure state, HCPTR.TASE and NSACR.NSASEDIS.

2.7 D17015

Details of traps will be added through the use of new LDC and STC accessibility pseudocode in sections G8.3.17 (DBGDTRRXint, Debug Data Transfer Register, Receive) and G8.3.19 (DBGDTRTXint, Debug Data Transfer Register, Transmit). This accessibility pseudocode is the same as for the equivalent MRC and MCR instructions, except that:

- The reported exception syndrome value, if applicable, is `0x06`.
- For LDC instructions the accessibility pseudocode loads the value to be written to the System register from 'MemA[address, 4]', where 'address' is the virtual address calculated by the LDC instruction.

2.8 D17119

In sections F3.1.10 (Advanced SIMD shifts and immediate generation), subsection 'Advanced SIMD two registers and shift amount' and F4.1.22 (Advanced SIMD shifts and immediate generation), subsection 'Advanced SIMD two registers and shift amount', the following constraints are added to VMOVL:

- 'L' must be '0'.
- 'imm3H' cannot be '000'.

2.9 D17556

In section D7.4.8 (A64 Cache maintenance instructions), in the subsection 'Effects of All and set/way maintenance instructions', the text that reads:

> The IC IALLU and DC set/way instructions apply only to the caches of the PE that performs the instruction.

is corrected to read:

> The DC set/way instructions apply only to the caches of the PE that performs the instruction. IC IALLU instructions apply only to the caches of the PE that performs the instruction, unless HCR_EL2.FB=1, which causes the instructions to be broadcast within the Inner Shareable domain when executed from EL1.

In the subsection 'Effects of virtualization and Security state on the cache maintenance instructions', the text that reads:

> TLB and instruction cache invalidate instructions executed at EL1 are broadcast across the Inner Shareable domain when all of the following is true:
>
> - When the value of HCR_EL2.FB is 1.
> - EL3 is not implemented, or EL3 is implemented and either SCR_EL3.NS == 1 or SCR_EL3.EEL2 == 1.
>
> When EL1 is using AArch64, this applies to the IC IALLU instruction. This means the instruction performs the invalidation that would be performed by the corresponding Inner Shareable instruction IC IALLUIS.

is corrected to read:

> TLB invalidate instructions and IC IALLU instructions executed at EL1 are broadcast across the Inner Shareable domain when all of the following are true:
>
> - EL2 is implemented and enabled in the current Security state.
> - The value of HCR_EL2.FB is 1.

2.10 R17661

In section D9.2 (Allocation Tags), the following Notes are removed:

> Note: The value `0b1111` may incur a higher performance overhead than other Allocation Tag encodings.
>
> Note: Arm recommends that software does not use instructions which write `0b1111` as an Allocation Tag to memory.

2.11 E17792

In section J1.3.3 (shared/functions), the AccType enumeration is refactored, such that the AccessDescriptor type is repurposed to hold information captured by the AccType enumeration and replaces the occurrences of AccType throughout the pseudocode in chapter J1 (Armv8 Pseudocode).

The enumeration AccType that reads:

```
enumeration AccType {AccType_NORMAL,          // Normal loads and stores
                     AccType_STREAM,          // Streaming loads and stores
                     AccType_VEC,             // Vector loads and stores
                     AccType_VECSTREAM,       // Streaming vector loads and stores
                     AccType_SVE,             // Scalable vector loads and stores
                     AccType_SVESTREAM,       // Scalable vector streaming loads and
 stores
```

```
                      AccType_SME,              // Scalable matrix loads and stores
                      AccType_SMESTREAM,        // Scalable matrix streaming loads and
 stores
                      AccType_UNPRIVSTREAM,     // Streaming unprivileged loads and
 stores
                      AccType_A32LSMD,          // Load and store multiple
                      AccType_ATOMIC,           // Atomic loads and stores
                      AccType_ATOMICRW,
                      AccType_ORDERED,          // Load-Acquire and Store-Release
                      AccType_ORDEREDRW,
                      AccType_ORDEREDATOMIC,    // Load-Acquire and Store-Release with
 atomic access
                      AccType_ORDEREDATOMICRW,
                      AccType_ATOMICLS64,       // Atomic 64-byte loads and stores
                      AccType_LIMITEDORDERED,   // Load-LOAcquire and Store-LORelease
                      AccType_UNPRIV,           // Load and store unprivileged
                      AccType_IFETCH,           // Instruction fetch
                      AccType_TTW,              // Translation table walk
                      AccType_NONFAULT,         // Non-faulting loads
                      AccType_CNOTFIRST,        // Contiguous FF load, not first
 element
                      AccType_NV2REGISTER,      // MRS/MSR instruction used at EL1 and
 which is converted to a memory access that uses the EL2 translation regime
                      AccType_TRBE,             // TRBE memory access
                     // Other operations
                      AccType_DC,               // Data cache maintenance
                      AccType_IC,               // Instruction cache maintenance
                      AccType_DCZVA,            // DC ZVA instructions
                      AccType_ATPAN,            // Address translation with PAN
 permission checks
                      AccType_AT};              // Address translation
```

Is replaced with:

```
// AccessType
// ==========
enumeration AccessType {
    AccessType_IFETCH,   // Instruction FETCH
    AccessType_GPR,      // Software load/store to a General Purpose Register
    AccessType_ASIMD,    // Software ASIMD extension load/store instructions
    AccessType_SVE,      // Software SVE load/store instructions
    AccessType_SME,      // Software SME load/store instructions
    AccessType_IC,       // Sysop IC
    AccessType_DC,       // Sysop DC (not DC {Z,G,GZ}VA)
    AccessType_DCZero,   // Sysop DC {Z,G,GZ}VA
    AccessType_AT,       // Sysop AT
    AccessType_NV2,      // NV2 memory redirected access
    AccessType_TRBE,     // Trace Buffer access
    AccessType_GPTW,     // Granule Protection Table Walk
    AccessType_TTW       // Translation Table Walk
};
```

The AccessDescriptor type that reads:

```
type AccessDescriptor is (
        boolean transactional,
        MPAMinfo mpam,
        AccType acctype)
```

Is updated to read:

```
// AccessDescriptor
// ================
```

```
// Memory access or translation invocation attributes that steer architectural
 behavior
type AccessDescriptor is (
    AccessType acctype,
    bits(2) el,              // Acting EL for the access
    SecurityState ss,        // Acting Security State for the access
    boolean acqsc,           // Acquire with Sequential Consistency
    boolean acqpc,           // FEAT_LRCPC: Acquire with Processor Consistency
    boolean relsc,           // Release with Sequential Consistency
    boolean limitedordered,  // FEAT_LOR: Acquire/Release with limited ordering
    boolean exclusive,       // Access has Exclusive semantics
    boolean atomicop,        // FEAT_LSE: Atomic read-modify-write access
    MemAtomicOp modop,       // FEAT_LSE: The modification operation in the
 'atomicop' access
    boolean nontemporal,     // Hints the access is non-temporal
    boolean read,            // Read from memory or only require read permissions
    boolean write,           // Write to memory or only require write permissions
    CacheOp cacheop,         // DC/IC: Cache operation
    CacheOpScope opscope,    // DC/IC: Scope of cache operation
    CacheType cachetype,     // DC/IC: Type of target cache
    boolean pan,             // FEAT_PAN: The access is subject to PSTATE.PAN
    boolean transactional,   // FEAT_TME: Access is part of a transaction
    boolean nonfault,        // SVE: Non-faulting load
    boolean firstfault,      // SVE: First-fault load
    boolean first,           // SVE: First-fault load for the first active element
    boolean contiguous,      // SVE: Contiguous load/store not gather load/scatter
 store
    boolean streamingsve,    // SME: Access made by PE while in streaming SVE mode
    boolean ls64,            // FEAT_LS64: Accesses by accelerator support loads/
stores
    boolean mops,            // FEAT_MOPS: Memory operation (CPY/SET) accesses
    boolean a32lsmd,         // A32 Load/Store Multiple Data access
    boolean tagchecked,      // FEAT_MTE2: Access is tag checked
    boolean tagaccess,       // FEAT_MTE: Access targets the tag bits
    MPAMinfo mpam            // FEAT_MPAM: MPAM information
)
```

2.12 C17811

In section I5.8.32 (ERR<n>STATUS, Error Record Primary Status Register, n = 0 - 65534), under the heading 'Accessing the ERR<n>STATUS', the text that reads:

> To ensure correct and portable operation, when software is clearing the valid fields in the register to allow new errors to be recorded, Arm recommends that software:
>
> - Read ERR<n>STATUS and determine which fields need to be cleared to zero.
> - Write ones to all the W1C fields that are nonzero in the read value.
> - Write zero to all the W1C fields that are zero in the read value.
> - Write zero to all the RW fields.

is clarified to read:

> To ensure correct and portable operation, when software is clearing the valid fields in the register to allow new errors to be recorded, Arm recommends that software:
>
> - Read ERR<n>STATUS and determine which fields need to be cleared to zero.

- In a single write to ERR<n>STATUS:
  - Write ones to all the W1C fields that are nonzero in the read value.
  - Write zero to all the W1C fields that are zero in the read value.
  - Write zero to all the RW fields.
- Read back ERR<n>STATUS after the write to confirm no new fault has been recorded.

2.13 E17996

In section J1.2.3 (aarch32/functions) and J1.1.3 (aarch64/functions), the previous stub functions AArch32.PhysicalSErrorSyndrome() and AArch64.PhysicalSErrorSyndrome() respectively are now defined as:

```
// AArch32.PhysicalSErrorSyndrome()
// ================================
// Generate SError syndrome.
bits(16) AArch32.PhysicalSErrorSyndrome()
    bits(32) syndrome = Zeros(32);
    FaultRecord fault = GetSavedFault();
    boolean long_format = TTBCR.EAE == '1';
    syndrome = AArch32.CommonFaultStatus(fault, long_format);
    return syndrome<15:0>;
// AArch64.PhysicalSErrorSyndrome()
// ================================
// Generate SError syndrome.
bits(25) AArch64.PhysicalSErrorSyndrome(boolean implicit_esb)
    bits(25) syndrome = Zeros(25);
    FaultRecord fault = GetSavedFault();
    ErrorState errorstate = AArch64.PEErrorState(fault);
    if errorstate == ErrorState_Uncategorized then
        syndrome = Zeros(25);
    elsif errorstate == ErrorState_IMPDEF then
        syndrome<24> = '1';                                                 // IDS
        syndrome<23:0> = bits(24) IMPLEMENTATION_DEFINED "IMPDEF ErrorState";
    else
        syndrome<24> = '0';                                                 // IDS
        syndrome<13> = (if implicit_esb then '1' else '0');                 // IESB
        syndrome<12:10> = AArch64.EncodeAsyncErrorSyndrome(errorstate); // AET
        syndrome<5:0> = '010001';                                           // DFSC
    return syndrome;
```

A new enumeration ErrorState is added in the same section, which is used instead of the errortype member of FaultRecord and PhysMemRetStatus:

```
enumeration ErrorState {ErrorState_UC,            // Uncontainable
                        ErrorState_UEU,           // Unrecoverable state
                        ErrorState_UEO,           // Restartable state
                        ErrorState_UER,           // Recoverable state
                        ErrorState_CE,            // Corrected
                        ErrorState_Uncategorized,
                        ErrorState_IMPDEF};
```

A new function AArch32.CommonFaultStatus() is added to section J1.2.2 (aarch32/exceptions):

```
// AArch32.CommonFaultStatus()
// ===========================
```

```
// Return the common part of the fault status on reporting a Data
// or Prefetch Abort.
bits(32) AArch32.CommonFaultStatus(FaultRecord fault, boolean long_format)
    bits(32) target = Zeros(32);
    if HaveRASExt() && IsAsyncAbort(fault) then
        ErrorState errstate = AArch32.PEErrorState(fault);
        target<15:14> = AArch32.EncodeAsyncErrorSyndrome(errstate);   // AET
    if IsExternalAbort(fault) then target<12> = fault.extflag;        // ExT
    target<9> = if long_format then '1' else '0';                     // LPAE
    if long_format then                                 // Long-descriptor format
        target<5:0>    = EncodeLDFSC(fault.statuscode, fault.level);  // STATUS
    else                                                // Short-descriptor format
        target<10,3:0> = EncodeSDFSC(fault.statuscode, fault.level);  // FS
    return target;
```

A new function GetSavedFault() is added to section J1.3.3 (shared/functions):

```
// GetSavedFault()
// ===========
// Return the saved asynchronous fault.
FaultRecord GetSavedFault();
```

2.14 D18330

Arm® Architecture Reference Manual for A-profile architecture, Issue I.a is somewhat inconsistent in its use of 'prefetch' and 'preload' to describe the bringing in of items into caches either by hardware prediction or as a result of some prefetch or preload instructions.

In future versions of *Arm® Architecture Reference Manual for A-profile architecture*, this will be cleaned up. The term 'prefetch' will be used for this functionality, with 'hardware prefetch' used where the prefetch is predicted by hardware, and 'software prefetch' used where the prefetch is prompted by particular instructions (such as the AArch64 PRFM or AArch32 PLD instructions).

2.15 D18465

In section D17.2.119 (SCTLR_EL2, System Control Register (EL2)), for all of the bits that are described as having a function when HCR_EL2.E2H==1 && HCR_EL2.TGE==1 and being **RES0** otherwise, it is clarified that these bits:

- Are **RES0** when HCR_EL2.E2H==0, so software should write the value 0.
- Are ignored by hardware when HCR_EL2.E2H==1 && HCR_EL2.TGE==0, but software doesn't have to set the value 0.
- Have their described effect when HCR_EL2.E2H==1 && HCR_EL2.TGE==1.

2.16 R18485

In section I5.8.8 (ERRDEVAFF, Device Affinity Register), the following text is added to the end of the Purpose section:

> Depending on the **IMPLEMENTATION DEFINED** nature of the system, it might be possible that ERRDEVAFF is read before system firmware has configured the group of error records or the PE or group of PEs that the group of error records has affinity with. When this is the case, ERRDEVAFF reads as zero.

2.17 D18520

In section I5.8.31 (ERR<n>PFGF, Pseudo-fault Generation Feature Register, n = 0 - 65534), the text in MV, bit [12] that reads:

> `0b0` When an injected error is recorded, the node might update ERR<n>MISC<m>. If any syndrome is recorded by the node in ERR<n>MISC<m>, then ERR<n>STATUS.MV is set to `0b1`. ERR<n>PFGCTL.MV is **RES0**.

is updated to read:

> `0b0` ERR<n>PFGCTL.MV not supported. When an injected error is recorded, the node might update ERR<n>MISC<m>. If any syndrome is recorded by the node in ERR<n>MISC<m>, then ERR<n>STATUS.MV is set to `0b1`. If the node always sets ERR<n>.STATUS.MV to `0b1` when recording an injected error, then ERR<n>PFGCTL.MV might be RAO/WI. Otherwise, ERR<n>PFGCTL.MV is **RES0**.

Corresponding updates are made to section I5.8.30 (ERR<n>PFGCTL, Pseudo-fault Generation Control Register, n = 0 - 65534), for bit [12] 'when the node supports this control'. Similar corrections are made for the ERR<n>PFGF.AV and ERR<n>PFGCTL.AV controls.

2.18 D18736

In section I5.8.5 (ERRCRIC0, Critical Error Interrupt Configuration Register 0), under the heading 'Accessing the ERRCRICR0', the following text is added:

> If the implementation does not use the recommended layout for the ERRIRQCR<n> registers, accesses to ERRCRICR0 are IMPLEMENTATION DEFINED.
>
> ERRCRICR0 ignores writes if all of the following are true:
>
> - The implementation uses the recommended layout for the ERRIRQCR<n> registers.
> - ERRCRICR2.NSMSI configures the physical address space for message signaled interrupts as Secure.
> - Accessed as a Non-secure access.

The equivalent changes are made in the following sections:

- I5.8.6 (ERRCRICR1, Critical Error Interrupt Configuration Register 1).
- I5.8.7 (ERRCRICR2, Critical Error Interrupt Configuration Register 2).
- I5.8.11 (ERRERICR0, Error Recovery Interrupt Configuration Register 0).
- I5.8.12 (ERRERICR1, Error Recovery Interrupt Configuration Register 1).
- I5.8.13 (ERRERICR2, Error Recovery Interrupt Configuration Register 2).
- I5.8.14 (ERRFHICR0, Fault Handling Interrupt Configuration Register 0).
- I5.8.15 (ERRFHICR1, Faulting Handling Interrupt Configuration Register 1).
- I5.8.16 (ERRFHICR2, Faulting Handling Interrupt Configuration Register 2).

In section I5.8.7 (ERRCRICR2, Critical Error Interrupt Configuration Register 2), the text in the description of NSMSI, bit [6], that reads:

> When the component supports configuring the Security attribute for messaged signaled interrupts and the component does not allow Non-secure writes to ERRCRICR2:
>
> Security attribute. Defines the physical address space for message signaled interrupts.
>
> `0b0` Secure.
>
> `0b1` Non-secure.
>
> The reset behavior of this field is:
>
> - On a Error recovery reset,this field resets to an IMPLEMENTATION DEFINED VALUE.
>
> When the component allows Non-secure writes to ERRCRICR2:
>
> Reserved, **RES0**. Security attribute. Defines the physical address space for message signaled interrupts. The Security attribute used for message signaled interrupts is Non-secure.

is changed to read:

> When the component supports configuring the physical address space for message signaled interrupts:
>
> Non-secure message signaled interrupt. Defines the physical address space for message signaled interrupts.
>
> `0b0` Secure physical address space.
>
> `0b1` Non-secure physical address space.
>
> The reset behavior of this field is:
>
> - On an Error recovery reset, this field resets to an IMPLEMENTATION DEFINED VALUE.

> Accessing this field has the following behavior:
>
> - If accessed as a Non-secure access, access to this field is RES1.
> - Otherwise, access to this field is RW.

The equivalent changes are made in the following sections:

- I5.8.13 (ERRERICR2, Error Recovery Interrupt Configuration Register 2).
- I5.8.16 (ERRFHICR2, Faulting Handling Interrupt Configuration Register 2).

2.19 R18746

In section B2.7.2 (Device memory), in the subsection 'Multi-register loads and stores that access Device memory', the following paragraph is added:

> The architecture permits that the non-speculative execution of an instruction that loads or stores more than one general-purpose or SIMD and floating-point register might result in repeated accesses to the same address.

The equivalent edit is made in section E2.8.2 (Device Memory), in the subsection 'Multi-register loads and stores that access Device memory'.

2.20 D18800

In section D17.5.17 (PMUSERENR_EL0, Performance Monitors User Enable Register), the EN, bit [0] description is updated to read:

> Enable. Enables EL0 read/write access to PMU registers.
>
> `0b0` EL0 accesses to the specified PMU System registers are trapped, unless enabled by PMUSERENR_EL0.{ER,CR,SW}.
>
> `0b1` EL0 accesses to the specified PMU System registers are enabled, unless trapped by another control.
>
> In AArch64 state, the register accesses affected by this control are:
>
> - MRS or MSR accesses to PMCCFILTR_EL0, PMCCNTR_EL0, PMCNTENCLR_EL0, PMCNTENSET_EL0, PMCR_EL0, PMEVCNTR<n>_EL0, PMEVTYPER<n>_EL0, PMOVSCLR_EL0, PMOVSSET_EL0, PMSELR_EL0, PMXEVCNTR_EL0, PMXEVTYPER_EL0.
> - MRS reads of PMCEID0_EL0 and PMCEID1_EL0.
> - MSR writes to PMSWINC_EL0.
>
> In AArch32 state, the register accesses affected by this control are:

- MRC or MCR accesses to PMCCFILTR, PMCCNTR, PMCNTENCLR, PMCNTENSET, PMCR, PMEVCNTR<n>, PMEVTYPER<n>, PMOVSR, PMOVSSET, PMSELR, PMXEVCNTR, PMXEVTYPER.
- MRC reads of the following registers:
  - PMCEID0 and PMCEID1.
  - If FEAT_PMUv3p1 is implemented, PMCEID2 and PMCEID3.
- MCR writes to PMSWINC.
- MRRC or MCRR accesses to PMCCNTR.

When trapped, reads and writes generate an exception to EL1, or to EL2 when EL2 is implemented and enabled for the current Security state and HCR_EL2.TGE is 1, and:

- AArch64 MRS and MSR accesses are reported using EC syndrome value `0x18`.
- AArch32 MRC and MCR accesses are reported using EC syndrome value `0x03`.
- AArch32 MRRC and MCRR accesses are reported using EC syndrome value `0x04`.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

Equivalent changes are made to the {ER, CR, SW} fields, and to the PMUSERENR.{ER, CR, SW, EN} fields in section G8.4.18 (PMUSERENR, Performance Monitors User Enable Register).

2.21 D18823

In section J1.1.3 (aarch64/functions), the function CalculateBottomPACBit(), reading:

```
integer CalculateBottomPACBit(bit top_bit)
    integer tsz_field;
    boolean using64k;
    Constraint c;
    if PtrHasUpperAndLowerAddRanges() then
        assert S1TranslationRegime() IN {EL1, EL2};
        if S1TranslationRegime() == EL1 then
            // EL1 translation regime registers
            tsz_field = if top_bit == '1' then UInt(TCR_EL1.T1SZ) else
 UInt(TCR_EL1.T0SZ);
            using64k = if top_bit == '1' then TCR_EL1.TG1 == '11' else TCR_EL1.TG0
 == '01';
        else
            // EL2 translation regime registers
            assert HaveEL(EL2);
            tsz_field = if top_bit == '1' then UInt(TCR_EL2.T1SZ) else
 UInt(TCR_EL2.T0SZ);
            using64k = if top_bit == '1' then TCR_EL2.TG1 == '11' else TCR_EL2.TG0
 == '01';
    else
        tsz_field = if PSTATE.EL == EL2 then UInt(TCR_EL2.T0SZ) else
 UInt(TCR_EL3.T0SZ);
        using64k = if PSTATE.EL == EL2 then TCR_EL2.TG0 == '01' else TCR_EL3.TG0 ==
 '01';
    max_limit_tsz_field = (if !HaveSmallTranslationTableExt() then 39 else if
 using64k then 47 else 48);
    if tsz_field > max_limit_tsz_field then
```

```
        // TCR_ELx.TySZ is out of range
        c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_NONE};
        if c == Constraint_FORCE then tsz_field = max_limit_tsz_field;
    tszmin = if using64k && AArch64.VAMax() == 52 then 12 else 16;
    if tsz_field < tszmin then
        c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_NONE};
        if c == Constraint_FORCE then tsz_field = tszmin;
    return (64-tsz_field);
```

is updated to read:

```
integer CalculateBottomPACBit(bit top_bit)
    Regime regime;
    S1TTWParams walkparams;
    integer bottom_PAC_bit;
    // There is no distinction between AccType_NORMAL and AccType_IFETCH
    // when determining the translation regime
    regime = TranslationRegime(PSTATE.EL, AccType_NORMAL);
    walkparams = AArch64.GetS1TTWParams(regime, Replicate(top_bit, 64));
    bottom_PAC_bit = 64 - UInt(AArch64.PACEffetiveTxSZ(walkparams));
    return bottom_PAC_bit;
```

In section J1.1.3 (aarch64/functions), the function AArch64.PACEffectiveTxSZ() is added:

```
// AArch64.PACEffectiveTxSZ()
// ==========================
// Compute the effective value for TxSZ used to determine the placement of the PAC
 field
bits(6) AArch64.PACEffetiveTxSZ(S1TTWParams walkparams)
    constant integer s1maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    constant integer s1mintxsz = AArch64.S1MinTxSZ(walkparams.ds, walkparams.tgx);
    if AArch64.S1TxSZFaults(walkparams) then
        if ConstrainUnpredictable(Unpredictable_RESTnSZ) == Constraint_FORCE then
            if UInt(walkparams.txsz) < s1mintxsz then
                return s1mintxsz<5:0>;
            if UInt(walkparams.txsz) > s1maxtxsz then
                return s1maxtxsz<5:0>;
    elsif UInt(walkparams.txsz) < s1mintxsz then
        return s1mintxsz<5:0>;
    elsif UInt(walkparams.txsz) > s1maxtxsz then
        return s1maxtxsz<5:0>;
    return walkparams.txsz;
```

In section J1.1.5 (aarch64/translation), the code within the function AArch64.GetS1TTWParams(), reading:

```
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    mintxsz = AArch64.S1MinTxSZ(walkparams.ds, walkparams.tgx);
    if UInt(walkparams.txsz) > maxtxsz then
        if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum")
 then
            walkparams.txsz = maxtxsz<5:0>;
    elsif !Have52BitVAExt() && UInt(walkparams.txsz) < mintxsz then
        if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum")
 then
            walkparams.txsz = mintxsz<5:0>;
```

is removed.

In section J1.1.5 (aarch64/translation), the code within the function AArch64.GetS2TTWParams(), reading:

```
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    mintxsz = AArch64.S2MinTxSZ(walkparams.ds, walkparams.tgx, s1aarch64);
    if UInt(walkparams.txsz) > maxtxsz then
        if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum")
 then
            walkparams.txsz = maxtxsz<5:0>;
    elsif !Have52BitPAExt() && UInt(walkparams.txsz) < mintxsz then
        if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum")
 then
            walkparams.txsz = mintxsz<5:0>;
```

is removed.

In section J1.1.5 (aarch64/translation), the function AArch64.S1InvalidTxSZ(), reading:

```
boolean AArch64.S1InvalidTxSZ(S1TTWParams walkparams)
    mintxsz = AArch64.S1MinTxSZ(walkparams.ds, walkparams.tgx);
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    return UInt(walkparams.txsz) < mintxsz || UInt(walkparams.txsz) > maxtxsz;
```

is updated to read:

```
boolean AArch64.S1TxSZFaults(S1TTWParams walkparams)
    mintxsz = AArch64.S1MinTxSZ(walkparams.ds, walkparams.tgx);
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    if UInt(walkparams.txsz) < mintxsz then
        return (Have52BitVAExt() ||
                boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum");
    if UInt(walkparams.txsz) > maxtxsz then
        return boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum";
    return FALSE;
```

In section J1.1.5 (aarch64/translation), the function AArch64.S2InvalidTxSZ(), reading:

```
boolean AArch64.S2InvalidTxSZ(S2TTWParams walkparams, boolean s1aarch64)
    mintxsz = AArch64.S2MinTxSZ(walkparams.ds, walkparams.tgx, s1aarch64);
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    return UInt(walkparams.txsz) < mintxsz || UInt(walkparams.txsz) > maxtxsz;
```

is updated to read:

```
boolean AArch64.S2TxSZFaults(S2TTWParams walkparams, boolean s1aarch64)
    mintxsz = AArch64.S2MinTxSZ(walkparams.ds, walkparams.tgx, s1aarch64);
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    if UInt(walkparams.txsz) < mintxsz then
        return (Have52BitPAExt() ||
                boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum");
    if UInt(walkparams.txsz) > maxtxsz then
        return boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum";
    return FALSE;
```

In section J1.1.5 (aarch64/translation), the code within the function AArch64.S1Translate(), reading:

```
if (AArch64.S1InvalidTxSZ(walkparams) ||
           (!ispriv && walkparams.e0pd == '1') ||
           (!ispriv && walkparams.nfd == '1' && IsDataAccess(acctype) &&
TSTATE.depth > 0) ||
           (!ispriv && walkparams.nfd == '1' && acctype == AccType_NONFAULT) ||
           AArch64.VAIsOutOfRange(va, acctype, regime, walkparams)) then
       fault.statuscode = Fault_Translation;
       fault.level      = 0;
       return (fault, AddressDescriptor UNKNOWN);
```

is updated to read:

```
   constant integer s1mintxsz = AArch64.S1MinTxSZ(walkparams.ds, walkparams.tgx);
   constant integer s1maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
   if AArch64.S1TxSZFaults(walkparams) then
       fault.statuscode = Fault_Translation;
       fault.level      = 0;
       return (fault, AddressDescriptor UNKNOWN);
   elsif UInt(walkparams.txsz) < s1mintxsz then
       walkparams.txsz = s1mintxsz<5:0>;
   elsif UInt(walkparams.txsz) > s1maxtxsz then
       walkparams.txsz = s1maxtxsz<5:0>;
   if AArch64.VAIsOutOfRange(va, acctype, regime, walkparams) then
       fault.statuscode = Fault_Translation;
       fault.level      = 0;
       return (fault, AddressDescriptor UNKNOWN);
   if !ispriv && walkparams.e0pd == '1' then
       fault.statuscode = Fault_Translation;
       fault.level      = 0;
       return (fault, AddressDescriptor UNKNOWN);
   if !ispriv && walkparams.nfd == '1' && IsDataAccess(acctype) && TSTATE.depth > 0
then
       fault.statuscode = Fault_Translation;
       fault.level      = 0;
       return (fault, AddressDescriptor UNKNOWN);
   if !ispriv && walkparams.nfd == '1' && acctype == AccType_NONFAULT then
       fault.statuscode = Fault_Translation;
       fault.level      = 0;
       return (fault, AddressDescriptor UNKNOWN);
```

In section J1.1.5 (aarch64/translation), the code within the function AArch64.S2Translate(), reading:

```
   if (AArch64.S2InvalidTxSZ(walkparams, s1aarch64) ||
           AArch64.S2InvalidSL(walkparams) ||
           AArch64.S2InconsistentSL(walkparams) ||
           AArch64.IPAIsOutOfRange(ipa.paddress.address, walkparams)) then
       fault.statuscode = Fault_Translation;
       fault.level      = 0;
       return (fault, AddressDescriptor UNKNOWN);
```

is updated to read:

```
   constant integer s2mintxsz = AArch64.S2MinTxSZ(walkparams.ds, walkparams.tgx,
s1aarch64);
   constant integer s2maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
   if AArch64.S2TxSZFaults(walkparams, s1aarch64) then
       fault.statuscode = Fault_Translation;
       fault.level      = 0;
       return (fault, AddressDescriptor UNKNOWN);
   elsif UInt(walkparams.txsz) < s2mintxsz then
```

```
        walkparams.txsz = s2mintxsz<5:0>;
    elsif UInt(walkparams.txsz) > s2maxtxsz then
        walkparams.txsz = s2maxtxsz<5:0>;
    if AArch64.S2InvalidSL(walkparams) || AArch64.S2InconsistentSL(walkparams) then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN);
    if AArch64.IPAIsOutOfRange(ipa.paddress.address, walkparams) then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN);
```

2.22 C18842

In section I5.5.14 (AMDEVARCH, Activity Monitors Device Architecture Register), the text in the ARCHID, bits [15:0] description that reads:

> For AMU:
>
> - Bits [15:12] are the architecture version, `0x0`.
> - Bits [11:0] are the architecture part number, `0xA66`.
>
> This corresponds to AMU architecture version AMUv1.

is changed to read:

> For AMU:
>
> - Bits [19:16] are the minor architecture version, `0x0`.
> - Bits [15:12] are the major architecture version, `0x0`.
> - Bits [11:0] are the architecture part number, `0xA66`.
>
> This corresponds to a generic AMU, version 1.0.

2.23 C18843

The current description of FEAT_LPA2 in *Arm® Architecture Reference Manual for A-profile architecture, Issue I.a* lacks clarity between the ability to describe the size of the output address as having 52 bits, and there being 52 bits of physical address. This will be rectified in a future release of *Arm® Architecture Reference Manual for A-profile architecture*.

2.24 D18853

In section D17.2.107 (RGSR_EL1, Random Allocation Tag Seed Register), the field descriptions are changed to read:

> When GCR_EL1.RRND == 0:

Bits [63:24]

Reserved, RES0.

SEED, bits [23:8]

Seed register used for generating values returned by RandomAllocationTag(). The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

Bits [7:4]

Reserved, RES0.

TAG, bits [3:0]

Tag generated by the most recent IRG instruction. The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

When GCR_EL1.RRND == 1:

Bits [63:56]

Reserved, RES0.

SEED, bits [55:8]

IMPLEMENTATION DEFINED

Note: Software is recommended to avoid writing SEED[15:0] with a value of zero, unless this has been generated by the PE in response to an earlier value with SEED being non-zero. The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

Bits [7:4]

Reserved, RES0.

TAG, bits [3:0]

Tag generated by the most recent IRG instruction. The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

2.25 D18889

In section C5.2.18 (SPSR_EL1, Saved Program Status Register (EL1)), in the 'TCO, bit [25]' field, the text that reads:

> When FEAT_MTE is not implemented, it is CONSTRAINED UNPREDICTABLE whether this field is **RES0** or behaves as if FEAT_MTE is implemented.

is corrected to read:

> When FEAT_MTE2 is not implemented, it is CONSTRAINED UNPREDICTABLE whether this field is **RES0** or behaves as if FEAT_MTE2 is implemented.

The equivalent changes are made in the following sections:

- C5.2.19 (SPSR_EL2, Saved Program Status Register (EL2)).
- C5.2.20 (SPSR_EL3, Saved Program Status Register (EL3)).
- D13.3.14 (DSPSR_EL0, Debug Saved Program Status Register).

In section C5.2.26 (TCO, Tag Check Override), in the subsection 'Purpose', the text that reads:

> When FEAT_MTE is implemented, this register allows tag checks to be disabled globally.
>
> When FEAT_MTE is not implemented, it is CONSTRAINED UNPREDICTABLE whether this register is **RES0** or behaves as if FEAT_MTE is implemented.

is corrected to read:

> Allows tag checks to be disabled globally.
>
> When FEAT_MTE2 is not implemented, it is CONSTRAINED UNPREDICTABLE whether this register is **RES0** or behaves as if FEAT_MTE2 is implemented.

In section D1.4.1 (PSTATE fields that are meaningful in AArch64 state), rule R_{PCDTX}, the text in the 'Additional details' column for the TCO table entry that reads:

> If FEAT_MTE2 is not implemented, it is CONSTRAINED UNPREDICTABLE whether the PSTATE.TCO bit is **RES0** or behaves as if FEAT_MTE is implemented.

is corrected to read:

> If FEAT_MTE2 is not implemented, it is CONSTRAINED UNPREDICTABLE whether the PSTATE.TCO bit is **RES0** or behaves as if FEAT_MTE2 is implemented.

In section H2.4.1 (PSTATE in Debug state), the text that reads:

> When FEAT_MTE is implemented, if Memory-access mode is enabled and PSTATE.TCO is 0, reads and writes to the external debug interface DTR registers are CONSTRAINED UNPREDICTABLE, with the following permitted behaviors:

- The PE behaves as if PSTATE.TCO is 0. That is, the load or store operation performs the tag check if required.
- The PE behaves as if PSTATE.TCO is 1. That is, the load or store operation does not perform the tag check.

is corrected to read:

When FEAT_MTE2 is implemented, if Memory-access mode is enabled and PSTATE.TCO is 0, reads and writes to the external debug interface DTR registers are CONSTRAINED UNPREDICTABLE, with the following permitted behaviors:

- The PE behaves as if PSTATE.TCO is 0. That is, the load or store operation performs the tag check if required.
- The PE behaves as if PSTATE.TCO is 1. That is, the load or store operation does not perform the tag check.

2.26 C19027

In section D11.11.3 (Common event numbers), subsection 'Common microarchitectural events', the following text is added to the descriptions of MEM_ACCESS_CHECKED (`0x4024`), MEM_ACCESS_CHECKED_RD (`0x4025`), and MEM_ACCESS_CHECKED_WR (`0x4026`):

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.

2.27 C19047

In section D17.2.27 (CLIDR_EL1, Cache Level ID Register), the following Note is added to the descriptions of LoUU, bits [29:27], and LoUIS, bits [23:21]:

Note: This field does not describe the requirements for instruction cache invalidation. See CTR_EL0.DIC.

The equivalent changes are made in section G8.2.27 (CLIDR, Cache Level ID Register).

2.28 D19116

In section D17.11.21 (CNTPS_CTL_EL1, Counter-timer Physical Secure Timer Control register), the following text is added under 'Configurations':

This register is present only when EL3 is implemented. Otherwise, direct accesses to CNTPS_CTL_EL1 are **UNDEFINED**.

Equivalent changes are made in the following sections:

- D17.11.23 (CNTPS_CVAL_EL1, Counter-timer Physical Secure Timer CompareValue register).
- D17.11.24 (CNTPS_TVAL_EL1, Counter-timer Physical Secure Timer TimerValue register).

2.29 D19121

In section D17.2.118 (SCTLR_EL1, System Control Register (EL1)), in field 'C, bit [2]', the text that reads:

> When the value of the HCR_EL2.DC bit is 1, the PE ignores SCTLR.C. This means that Non-secure EL0 and Non-secure EL1 data accesses to Normal memory are Cacheable.
>
> When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

is changed to read:

> When the Effective value of the HCR_EL2.DC bit in the current Security state is 1, the PE ignores SCTLR_EL1.C. This means that EL0 and EL1 data accesses to Normal memory are Cacheable.
>
> When FEAT_VHE is implemented, and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

Similarly in field 'M, bit [0]', the text that reads:

> If the value of HCR_EL2.{DC, TGE} is not {0, 0} then in Non-secure state the PE behaves as if the value of the SCTLR_EL1.M field is 0 for all purposes other than returning the value of a direct read of the field.
>
> When FEAT_VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

is changed to read:

> If the Effective value of HCR_EL2.{DC, TGE} in the current Security state is not {0, 0} then the PE behaves as if the value of the SCTLR_EL1.M field is 0 for all purposes other than returning the value of a direct read of the field.
>
> When FEAT_VHE is implemented, and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}, this bit has no effect on the PE.

The equivalent changes are made in section G8.2.126 (SCTLR, System Control Register).

2.30 D19162

In section B2.3.10 (Restrictions on the effects of speculation), in the subsection 'Restrictions on the effects of speculation from Armv8.5', the text that reads:

> Any System register read under speculation to a register that is not architecturally accessible from the current Exception level cannot be used to form an address, to generate condition codes, or to generate SVE predicate values to be used by other instructions in the speculative sequence.

is updated to read:

> Any read under speculation from a register that is not architecturally accessible from the current Exception level cannot be used to form an address, to generate condition codes, or to generate SVE predicate values to be used by other instructions in the speculative sequence.

The equivalent change is made in section E2.3.9 (Restrictions on the effects of speculation), in the subsection 'Further restrictions on the effects of speculation from Armv8.5'.

2.31 D19178

In section J1.1.3 (aarch64/functions), the function AddressSupportsLS64(), that reads as:

```
boolean AddressSupportsLS64(bits(64) address)
```

Is updated to read as:

```
boolean AddressSupportsLS64(bits(52) paddress);
```

The following changes are also made in the same section:

In MemStore64B(), the code that reads:

```
MemStore64B(bits(64) address, bits(512) value, AccType acctype)
    boolean iswrite = TRUE;
    constant integer size = 64;
    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    if !AddressSupportsLS64(address) then
        c = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
        assert c IN {Constraint_LIMITED_ATOMICITY, Constraint_FAULT};
        if c == Constraint_FAULT then
            ...
        else
            // Accesses are not single-copy atomic above the byte level.
            for i = 0 to 63
                AArch64.MemSingle[address+8*i, 1, acctype, aligned] = value<7+8*i :
 8*i>;
    else
        -= MemStore64BWithRet(address, value, acctype);  // Return status is ignored
 by ST64B
return;
```

Is updated to read:

```
MemStore64B(bits(64) address, bits(512) value, AccType acctype)
    boolean iswrite = TRUE;
    constant integer size = 64;
    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, acctype,
 iswrite,
                                                             istagaccess, aligned,
 size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);
    // Effect on exclusives
    if memaddrdesc.mematttrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);
    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    if !AddressSupportsLS64(memaddrdesc.paddress.address) then
        c = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
        assert c IN {Constraint_LIMITED_ATOMICITY, Constraint_FAULT};
        if c == Constraint_FAULT then
            ...
        else
            // Accesses are not single-copy atomic above the byte level.
            accdesc.acctype = AccType_ATOMIC;
            for i = 0 to size-1
                memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, value<8*i+7:8*i>);
                if IsFault(memstatus) then
                    HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
                memaddrdesc.paddress.address = memaddrdesc.paddress.address+1;
    else
        memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
    return;
```

In MemLoad64B(), the code that reads:

```
bits(512) MemLoad64B(bits(64) address, AccType acctype)
    ...
    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    if !AddressSupportsLS64(address) then
        c = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
        assert c IN {Constraint_LIMITED_ATOMICITY, Constraint_FAULT};
        if c == Constraint_FAULT then
            // Generate a stage 1 Data Abort reported using the DFSC code of 110101.
            boolean secondstage = FALSE;
            boolean s2fs1walk = FALSE;
            FaultRecord fault = AArch64.ExclusiveFault(acctype, iswrite,
 secondstage, s2fs1walk);
            AArch64.Abort(address, fault);
        else
            // Accesses are not single-copy atomic above the byte level
            for i = 0 to 63
                data<7+8*i : 8*i> = AArch64.MemSingle[address+8*i, 1, acctype,
 aligned];
            return data;
    AddressDescriptor memaddrdesc;
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, istagaccess,
 aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        ...
    accdesc = CreateAccessDescriptor(acctype);
    PhysMemRetStatus memstatus;
    (memstatus, data) = PhysMemRead(memaddrdesc, size, accdesc);
```

```
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
    return data;
```

Is updated to read as:

```
bits(512) MemLoad64B(bits(64) address, AccType acctype)
    ...
    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    AddressDescriptor memaddrdesc;
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, istagaccess,
 aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        ...
    accdesc = CreateAccessDescriptor(acctype);
    if !AddressSupportsLS64(memaddrdesc.paddress.address) then
        c = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
        assert c IN {Constraint_LIMITED_ATOMICITY, Constraint_FAULT};
        if c == Constraint_FAULT then
            // Generate a stage 1 Data Abort reported using the DFSC code of 110101.
            boolean secondstage = FALSE;
            boolean s2fs1walk = FALSE;
            FaultRecord fault = AArch64.ExclusiveFault(acctype, iswrite,
 secondstage, s2fs1walk);
            AArch64.Abort(address, fault);
        else
            // Accesses are not single-copy atomic above the byte level.
            accdesc.acctype = AccType_ATOMIC;
            for i = 0 to size-1
                PhysMemRetStatus memstatus;
                (memstatus, data<8*i+7:8*i>) = PhysMemRead(memaddrdesc, 1, accdesc);
                if IsFault(memstatus) then
                    HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
                memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
    else
        PhysMemRetStatus memstatus;
        (memstatus, data) = PhysMemRead(memaddrdesc, size, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
    return data;
```

2.32 C19183

In section H3.5.1 (Synchronization and External Debug Request debug events) the following text:

> An External Debug Request debug event that is asserted before a Context synchronization event is taken and the PE enters Debug state before the first instruction following the Context synchronization event completes its execution, provided that halting is allowed after completion of the Context synchronization event.

is replaced by:

> For all Context synchronization events, if an External Debug Request debug event is asserted before the Context synchronization event, and the External Debug Request debug event remains asserted and halting is allowed after the Context synchronization event, then the debug event is taken and the PE enters Debug state before the first instruction following the Context synchronization event completes its execution.

2.33 C19202

In section A2.2.1 (Additional functionality added to Armv8.0 in later releases), in the definition 'FEAT_CSV2, FEAT_CSV2_2, and FEAT_CSV2_3, Cache Speculation Variant 2', the text that reads:

> FEAT_CSV2 adds a mechanism to identify if hardware cannot disclose information about whether branch targets trained in one hardware described context can control speculative execution in a different hardware described context.

is updated to read:

> FEAT_CSV2 adds a mechanism to identify if hardware cannot disclose information about whether branch targets, including those used by return instructions, trained in one hardware described context can control speculative execution in a different hardware described context.

In section B2.3.10 (Restrictions on the effects of speculation), in the subsection 'Restrictions on the effects of speculation from Armv8.5', the text that reads:

> If FEAT_CSV2 is implemented:
>
> - Code running in one hardware-defined context (context1) cannot either exploitatively control, or predictively leak to, the speculative execution of code in a different hardware-defined context (context2), as a result of the behavior of any of the following resources:
>   - Branch target prediction based on the branch targets used in context1.
>   - This applies to both direct and indirect branches, but excludes the prediction of the direction of a conditional branch.

is updated to read:

> If FEAT_CSV2 is implemented:
>
> - Code running in one hardware-defined context (context1) cannot either exploitatively control, or predictively leak to, the speculative execution of code in a different hardware-defined context (context2), as a result of the behavior of any of the following resources:
>   - Branch target prediction based on the branch targets used in context1.
>   - This applies to both direct and indirect branches, including those used by return instructions, but excludes the prediction of the direction of a conditional branch.

2.34 D19239

In section D17.2.49 (HCRX_EL2, Extended Hypervisor Configuration Register), the text in the fields MSCEN, MCE2, CMOW, and SMPME that reads:

> On a Warm reset, this field resets to an architecturally **UNKNOWN** value.

is corrected to read:

> On a Warm reset:
> - When EL3 is not implemented and EL2 is implemented, this field resets to 0.
> - Otherwise, this field resets to an architecturally **UNKNOWN** value.

In the same register, the text in the fields VFNMI, VINMI, TALLINT, FGTnXS, FnXS, EnASR, EnALS, and EnAS0 that reads:

> On a Warm reset, when EL3 is not implemented and EL2 is implemented, this field resets to 0.

is corrected to read:

> On a Warm reset:
> - When EL3 is not implemented and EL2 is implemented, this field resets to 0.
> - Otherwise, this field resets to an architecturally **UNKNOWN** value.

2.35 D19275

In section D17.2.48 (HCR_EL2, Hypervisor Configuration Register), in the description of FWB, bit [46], the following Note is removed:

> When FEAT_MTE2 is implemented, if the stage 1 page or block descriptor specifies the Tagged attribute, the final memory type is Tagged only if the final cacheable memory type is Inner and Outer Write-back cacheable and the final allocation hints are Read-Allocate, Write-Allocate.

2.36 D19323

In section J1.1.2 (aarch64/exceptions), the function AArch64.TakeException() that reads:

```
AArch64.TakeException(bits(2) target_el, ExceptionRecord exception_in,
                                         bits(64) preferred_exception_return,
                                         integer vect_offset_in)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >=
 UInt(PSTATE.EL);
    ExceptionRecord exception = exception_in;
    ...
```

Is updated to read:

```
AArch64.TakeException(bits(2) target_el, ExceptionRecord exception_in,
                                         bits(64) preferred_exception_return,
                                         integer vect_offset_in)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >=
 UInt(PSTATE.EL);
    if Halted() then
        AArch64.TakeExceptionInDebugState(target_el, exception_in);
        return;
```

```
    ExceptionRecord exception = exception_in;
    ...
```

In section J1.2.2 (aarch32/exceptions), the function AArch32.EnterMonitorMode() that reads:

```
AArch32.EnterMonitorMode(bits(32) preferred_exception_return, integer lr_offset,
                                                   integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = CurrentSecurityState() == SS_Secure;
    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState, 32);
    ...
```

Is updated to read:

```
AArch32.EnterMonitorMode(bits(32) preferred_exception_return, integer lr_offset,
                                                   integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = CurrentSecurityState() == SS_Secure;
    if Halted() then
        AArch32.EnterMonitorModeInDebugState();
        return;
    bits(32) s
```

In section J1.2.2 (aarch32/exceptions), similar changes are made such that function calls to AArch32.EnterHypMode() and AArch32.EnterMode() are redirected to AArch32.EnterHypModeInDebugState() and AArch32.EnterModeInDebugState() functions, respectively.

In section J1.3.3 (shared/functions), a new function EffectiveEA() is added:

```
bit EffectiveEA()
    if Halted() && EDSCR.SDD == '0' then
        return '0';
    else
        return if HaveAArch64() then SCR_EL3.EA else SCR.EA;
```

2.37 C19346

In section A2.6.3 (Features added to the Armv8.3 extension in later releases), the description of 'FEAT_CONSTPACFIELD, PAC algorithm enhancement' that reads:

> FEAT_CONSTPACFIELD introduces functionality that permits an implementation with pointer authentication to use the value of bit[55] in the virtual address to determine the size of the PAC field, even when the top byte is not being ignored.

is updated to read:

> FEAT_CONSTPACFIELD introduces functionality that permits an implementation with pointer authentication to use the value of bit[55] in the virtual address to determine the size of the PAC field when adding a PAC to the virtual address, even when the top byte is not being ignored.

In section D8.8 (Pointer authentication), rule R_{NQZWG} that reads:

> If FEAT_CONSTPACFIELD is implemented, then an implementation is permitted to use the value in Xn[55] to determine the size of the PAC field, even when address tagging is not used.

is updated to read:

> If FEAT_CONSTPACFIELD is implemented, then an implementation is permitted to use the value in Xn[55] to determine the size of the PAC field when adding a PAC to Xn, even when address tagging is not used.

2.38 R19370

In section E2.8.1 (Normal memory), after the text that reads:

> Writes to a memory location with the Normal memory type that is either Non-cacheable or Write-Through cacheable for both the Inner and Outer Cacheability must reach the endpoint for that location in the memory system in finite time. Two writes to the same location, where at least one is using the Normal memory type, might be merged before they reach the endpoint unless there is an ordered-before relationship between the two writes.

The following text is added:

> For the purposes of this requirement, the endpoint for a location in Conventional memory is the PoC.

2.39 D19372

In section D17.2.107 (RGSR_EL1, Random Seed Allocation Tag Seed Register), the following text is added under ‘Configurations’:

> When GCR_EL1.RRND==0b0, direct and indirect reads and writes to the register appear to occur in program order relative to other instructions, without the need for any explicit synchronization.

2.40 E19440

In section H9.2.42 (EDSCR, External Debug Status and Control Register), in the fields RXfull, TXfull, RXO, TXU, TDA, SC2, HDE, and ERR, the following text is added:

> When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through the following System registers:
>
> - MDSCR_EL1.
> - DBGDSCRext.

2.41 D19451

In section C6.2.378 (TLBI), in the 'Assembler symbols' subsection, the following statements are added to the definition of '<tlbi_op>':

> When FEAT_RME is implemented, the following values are also valid:

PAALLOS	when op1 = 110, CRn = 1000, CRm = 0001, op2 = 100
RPAOS	when op1 = 110, CRn = 1000, CRm = 0100, op2 = 011
RPALOS	when op1 = 110, CRn = 1000, CRm = 0100, op2 = 111
PAALL	when op1 = 110, CRn = 1000, CRm = 0111, op2 = 100

2.42 D19452

Following the update communicated as D18736, in section I5.8.7 (ERRCRICR2, Critical Error Interrupt Configuration Register 2), the text in the NSMSI, bit [6] field that reads:

> If accessed as a Non-secure access, access to this field is **RES1**.

is updated to read:

> Accessing this field has the following behavior:
>
> - Access is RO if any of the following are true:
> - an access is Non-secure
> - an access is Realm
> - Otherwise, access to this field is RW.

The equivalent changes are made in the following sections:

- I5.8.13 (ERRERICR2, Error Recovery Interrupt Configuration Register 2).
- I5.8.16 (ERRFHICR2, Faulting Handling Interrupt Configuration Register 2).

2.43 D19494

In section J1.3.3 (shared/functions/externalaborts) the function IsSErrorEdgeTriggered(), that reads as

```
boolean IsSErrorEdgeTriggered(bits(2) target_el, bits(25) syndrome)
    if HaveRASExt() then
        if HaveDoubleFaultExt() then
            return TRUE;
        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";
```

Is updated to read:

```
 boolean IsSErrorEdgeTriggered()
    if HaveDoubleFaultExt() then
        return TRUE;
    else
        return boolean IMPLEMENTATION_DEFINED "Edge-triggered SError";
```

In section J1.1.2 (aarch64/exceptions/async), the function AArch64.TakePhysicalSErrorException(), that reads as:

```
AArch64.TakePhysicalSErrorException(boolean implicit_esb)
    ...
    bits(25) syndrome = Zeros(25);
    syndrome = AArch64.PhysicalSErrorSyndrome(implicit_esb);
    if IsSErrorEdgeTriggered(target_el, exception.syndrome) then
        ClearPendingPhysicalSError();
    ...
```

Is updated to read:

```
AArch64.TakePhysicalSErrorException(boolean implicit_esb)
  ...
    bits(25) syndrome = AArch64.PhysicalSErrorSyndrome(implicit_esb);
    if IsSErrorEdgeTriggered() then
        ClearPendingPhysicalSError();
    ...
```

In section J1.2.2 (aarch32/exceptions/async), similar changes are made to the function AArch32.TakePhysicalSErrorException().

2.44 R19519

In section B2.3.10 (Restrictions on the effects of speculation), in the subsection 'Restrictions on the effects of speculation from Armv8.5', the sub-bullet point that reads:

- Data Value predictions based on data value from execution in context1.

is updated to include the following Note:

Note: PSTATE.{N,Z,C,V} values from context1 are not considered a data value for this purpose.

The equivalent change is made in section E2.3.9 (Restrictions on the effects of speculation), in the subsection 'Further restrictions on the effects of speculation from Armv8.5'.

In section C5.6.3 (DVP RCTX, Data Value Prediction Restriction by Context), the following Note is added:

Note: The prediction of the PSTATE.{N,Z,C,V} values is not considered a data value for this purpose.

The equivalent change is made in section G8.2.50 (DVPRCTX, Data Value Prediction Restriction by Context).

2.45 D19521

In section C5.2.25 (SVCR, Streaming Vector Control Register), for the field ZA, bit [1], the text that reads:

When a write to SVCR.ZA changes the value of PSTATE.ZA, the following applies:

When changed from 0 to 1, all implemented bits of the storage are set to zero. When changed from 1 to 0, there is no observable change to the storage.

Changes to this field do not have an affect on the SVE vector and predicate registers and FPSR.

is corrected to read:

When a write to SVCR.ZA changes the value of PSTATE.ZA from 0 to 1, all implemented bits of the storage are set to zero.

Changes to this field do not have an effect on the SVE vector and predicate registers and FPSR.

2.46 D19549

In section D11.11.3 (Common event numbers), in the subsection 'Common microarchitectural events', for each TRCEXTOUT<n> event, where <n> is 0 to 3, the text that reads:

> This event must be implemented if FEAT_ETE is implemented.

is updated to read:

> This event must be implemented if FEAT_ETE is implemented and the ETE implements External output <n>.

2.47 D19560

In section D17.2.26 (CCSIDR_EL1, Current Cache Size Register), the text in LineSize, bits [2:0] when FEAT_CCIDX is implemented that reads:

> When FEAT_MTE is implemented and enabled, where a cache only holds Allocation tags, this field is **RES0**.

is changed to read:

> When FEAT_MTE is implemented, where a cache only holds Allocation tags, this field is **RES0**.

The following text is added to LineSize, bits [2:0] when FEAT_CCIDX is not implemented:

> When FEAT_MTE is implemented, where a cache only holds Allocation tags, this field is **RES0**.

2.48 D19561

In section D17.2.107 (RGSR_EL1, Random Allocation Tag Seed Register), the text that reads:

> When GCR_EL1.RRND=0, direct and indirect reads and writes to the register appear to occur in program order relative to other instructions, without the need for any explicit synchronization.

is changed to read:

> Direct and indirect reads and writes to the register appear to occur in program order relative to other instructions, without the need for any explicit synchronization.

2.49 D19581

In the function AArch64.RestrictPrediction() in section J1.1.4 (aarch64/instrs), the code that reads:

```
// If the instruction is executed at an EL lower than the specified
// level, it is treated as a NOP.
if UInt(target_el) > UInt(PSTATE.EL) then return;
```

Is updated to read:

```
// If the target EL is not implemented or the instruction is executed at an
// EL lower than the specified level, the instruction is treated as a NOP.
if !HaveEL(target_el) || UInt(target_el) > UInt(PSTATE.EL) then EndOfInstruction();
```

This affects the A64 System instructions in the following sections:

- C5.6.1 (CFP RCTX, Control Flow Prediction Restriction by Context).
- C5.6.2 (CPP RCTX, Cache Prefetch Prediction Restriction by Context).
- C5.6.3 (DVP RCTX, Data Value Prediction Restriction by Context).

An equivalent change is made in AArch32.RestrictPrediction() affecting the AArch32 System Registers in the following sections:

- G8.2.26 (CFPRCTX, Control Flow Prediction Restriction by Context).
- G8.2.34 (CPPRCTX, Cache Prefetch Prediction Restriction by Context).
- G8.2.50 (DVPRCTX, Data Value Prediction Restriction by Context).

2.50 D19583

In section D1.3.8 (Configurable instruction controls), rule R_{JTXTF} that reads:

> It is UNPREDICTABLE / **CONSTRAINED UNPREDICTABLE** whether configurable instruction controls generate an exception when the instruction is UNPREDICTABLE or **CONSTRAINED UNPREDICTABLE** in the PE state in which the instruction is executed.

is updated to read:

> It is **CONSTRAINED UNPREDICTABLE** whether configurable instruction controls generate an exception when the instruction is UNPREDICTABLE or **CONSTRAINED UNPREDICTABLE** in the PE state in which the instruction is executed, with all of the following constraints:
>
> - If the instruction description explicitly states that the configurable instruction control is applied with higher priority than the **CONSTRAINED UNPREDICTABLE** behavior, then the configurable instruction control generates an exception.
> - The **CONSTRAINED UNPREDICTABLE** behaviors cannot lead to any behavior that is prohibited by the general definition of UNPREDICTABLE.

2.51 D19642

In section D11.11.3 (Common event numbers), subsection 'Common microarchitectural events', the PMU events that read:

> `0x4025`, MEM_ACCESS_RD_CHECKED, Checked data memory access, read
>
> `0x4026`, MEM_ACCESS_WR_CHECKED, Checked data memory access, write

are corrected to read:

> `0x4025`, MEM_ACCESS_CHECKED_RD, Checked data memory access, read
>
> `0x4026`, MEM_ACCESS_CHECKED_WR, Checked data memory access, write

2.52 C19644

In section D11.11.3 (Common event numbers), subsection 'Common microarchitectural events', the text in the descriptions of MEM_ACCESS_CHECKED_RD (`0x4025`) and MEM_ACCESS_CHECKED_WR (`0x4026`) that reads:

> Implementation of this optional event requires that FEAT_MTE is implemented.

is corrected to read:

> Implementation of this optional event requires that FEAT_MTE2 is implemented.

This text is also added to the MEM_ACCESS_CHECKED (`0x4024`) event description.

2.53 D19647

In section D8.2.3 (Translation table base address register), the following text is added:

> Direct writes to TTBR0_ELx and TTBR1_ELx occur in program order relative to one another, without the need for explicit synchronization. For any one translation, all indirect reads of TTBR0_ELx and TTBR1_ELx made as part of the translation observe only one point in that order of direct writes. Consistent with the general requirements for direct writes to System registers, direct writes to TTBRn_ELx are not required to be observed by indirect reads until completion of a Context synchronization event.

A new subsection, 'Example sequences for changing TTBRn_ELx for AArch64', is added after this text:

Example D8-1 Example software sequence for changing translation table base address and ASID value when TCR_EL1.A1=1

```
Change TTBR0 to point to no valid entries
Change TTBR1 (includes changing the ASID)
Change TTBR0 to have valid entries in it
ISB
```

Example D8-2 Example software sequence for changing translation table base address and ASID value when TCR_EL1.A1=0

```
Change TTBR1 to point only at global entries
Change TTBR0 (includes changing the ASID)
Change TTBR1 to point at new tables, containing non-global entries
ISB
```

2.54 C19649

In section B2.7.2 (Device Memory), in subsection 'Reordering', the bullet point in the note that reads:

> The non-Reordering property is only required by the architecture to apply the order of arrival of accesses to a single memory-mapped peripheral of an **IMPLEMENTATION DEFINED** size, and is not required to have an impact on the order of observation of memory accesses to SDRAM. For this reason, there is no effect of the non-Reordering attribute on the ordering relations between accesses to different locations described in Ordering relations on page B2-165 as part of the formal definition of the memory model.

is updated to read:

> The non-Reordering property is only required by the architecture to apply the order of arrival of accesses to a single memory-mapped peripheral of an **IMPLEMENTATION DEFINED** size, and is not required to have an impact on the order of observation of memory accesses to SDRAM. For this reason, there is no effect of the non-Reordering attribute on the ordering relations between accesses to different locations described in B2.3.3 Ordering relations on page B2-165 as part of the formal definition of the memory model. It does have an effect on the Peripheral Coherence Order described in section B2.3.7 (Completion and endpoint ordering).

2.55 D19680

In section C5.5.62 (TLBI VAE2, TLBI VAE2NXS, TLB Invalidate by VA, EL2), the accessibility pseudocode that reads:

```
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE,
 Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
```

```
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[],
 Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '1' then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE,
 Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID[],
 Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
```

is corrected to read:

```
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE,
 Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE,
 Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
elsif PSTATE.EL == EL3 then
    if !EL2Enabled() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '1' then
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL20, VMID_NONE,
 Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
    else
        AArch64.TLBI_VA(SecurityStateAtEL(EL2), Regime_EL2, VMID_NONE,
 Shareability_NSH, TLBILevel_Any, TLBI_AllAttr, X[t, 64]);
```

The same change, from VMID[] to VMID_NONE, is made in all the TLBI VAE2*, TLBI VAE3*, TLBI VALE2*, and TLBI VALE3* System instructions.

2.56 E19713

In section J1.3.3 (shared/functions), the contents of the HaveXXX() functions are updated to reflect the official feature names. For example:

```
boolean Have16bitVMID()
    return (HasArchVersion(ARMv8p1) && HaveEL(EL2) &&
        boolean IMPLEMENTATION_DEFINED "Has 16-bit VMID");
```

Is updated to read:

```
boolean Have16bitVMID()
    return IsFeatureImplemented(FEAT_VMID16);
```

2.57 D19741

In the function AArch64.WatchpointByteMatch() in section J1.1.1 (aarch64/debug), the code that reads:

```
if mask > bottom then
    ...
    if !IsOnes(DBGBVR_EL1[n]<63:top>) && !IsZero(DBGBVR_EL1[n]<63:top>) then
        if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
```

Is updated to read as:

```
if mask > bottom then
    ...
    if !IsOnes(DBGWVR_EL1[n]<63:top>) && !IsZero(DBGWVR_EL1[n]<63:top>) then
        if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
```

In the function AArch32.WatchpointByteMatch() in section J1.2.1 (aarch32/debug), the code that reads:

```
if mask > bottom then
    // If the DBGxVR<n>_EL1.RESS field bits 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 !IsOnes(DBGBVR_EL1[n]<63:top>) && !IsZero(DBGBVR_EL1[n]<63:top>) then
        if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
            top = 63;
    WVR_match = (vaddress<top:mask> == DBGWVR[n]<top:mask>);
```

Is updated to read as:

```
if mask > bottom then
    WVR_match = (vaddress<top:mask> == DBGWVR[n]<top:mask>);
```

2.58 D19753

In section J1.3.1 (shared/debug), the function Halt(), that reads as:

```
Halt(bits(6) reason, boolean is_async)
   CTI_SignalEvent(CrossTriggerIn_CrossHalt);  // Trigger other cores to halt
   ...
```

Is updated to read:

```
Halt(bits(6) reason, boolean is_async)
    if HaveTME() && TSTATE.depth > 0 then
        FailTransaction(TMFailure_DBG, FALSE);
    CTI_SignalEvent(CrossTriggerIn_CrossHalt);  // Trigger other cores to halt
    ...
```

2.59 C19772

In section C5.5.10 (TLBI ASIDE1, TLBI ASIDE1NXS, TLB Invalidate by ASID, EL1), in the subsection 'Executing TLBI ASIDE1, TLBI ASIDE1NXS instruction', the EL1 accessibility pseudocode that reads:

```
elsif EL2Enabled() && HCR_EL2.FB == '1' then
    if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) &&
 HCRX_EL2.FnXS == '1' then
       AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[],
 Shareability_ISH, TLBI_ExcludeXS, X[t, 64]);
```

is updated to read:

```
elsif EL2Enabled() && HCR_EL2.FB == '1' then
    if IsFeatureImplemented(FEAT_XS) && IsFeatureImplemented(FEAT_HCX) &&
 IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1' then
       AArch64.TLBI_ASID(SecurityStateAtEL(EL1), Regime_EL10, VMID[],
 Shareability_ISH, TLBI_ExcludeXS, X[t, 64]);
```

The same edits are made in the following sections:

- C5.5.29 (TLBI RVAAE1, TLBI RVAAE1NXS).
- C5.5.32 (TLBI RVAALE1, TLBI RAAVLE1NXS).
- C5.5.35 (TLBI RVAE1, TLBI RVAE1NXS).
- C5.5.44 (TLBI RVALE1, TLBI RAVLE1NXS).
- C5.5.53 (TLBI VAAE1, TLBI VAAE1NXS).
- C5.5.56 (TLBI VAALE1, TLBI VAALE1NXS).
- C5.5.59 (TLBI VAE1, TLBI VAE1NXS).
- C5.5.68 (TLBI VALE1, TLBI VALE1NXS).
- C5.5.77 (TLBI VMALLE1, TLBI VMALLE1NXS).
- G8.2.136 (TLBIALL, TLB Invalidate All).
- G8.2.142 (TLBIASID, TLB Invalidate by ASID match).
- G8.2.148 (TLBIMVA, TLB Invalidate by VA).
- G8.2.149 (TLBIMVAA, TLB Invalidate by VA, All ASID).
- G8.2.151 (TLBIMVAAL, TLB Invalidate by VA, All ASID, Last level).
- G8.2.156 (TLBIMVAL, TLB Invalidate by VA, Last level).

2.60 C19793

In section C5.5.25 (TLBI RIPAS2LE1IS, TLBI RIPAS2LE1ISNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable), in the subsection 'Purpose', the text that reads:

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

is updated to read:

> - The entry is a stage 2 only translation table entry, from the leaf level of the translation table walk, indicated by the TTL hint.

Equivalent changes are made in the following sections:

- C5.5.24 (TLBI RIPAS2LE1, TLBI RIPAS2LE1NXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1).
- C5.5.26 (TLBI RIPAS2LE1OS, TLBI RIPAS2LE1OSNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable).
- C5.5.32 (TLBI RVAALE1, TLBI RVAALE1NXS, TLB Range Invalidate by VA, All ASID, Last level, EL1).
- C5.5.33 (TLBI RVAALE1IS, TLBI RVAALE1ISNXS, TLB Range Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable).
- C5.5.34 (TLBI RVAALE1OS, TLBI RVAALE1OSNXS, TLB Range Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable).

In section C5.5.35 (TLBI RVAE1, TLBI RVAE1NXS, TLB Range Invalidate by VA, EL1), in the subsection 'Purpose', the text that reads:

> - The entry is a stage 1 translation table entry.

is updated to read:

> - The entry is a stage 1 translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.

Equivalent changes are made in the following sections:

- C5.5.36 (TLBI RVAE1IS, TLBI RVAE1ISNXS, TLB Range Invalidate by VA, EL1, Inner Shareable).
- C5.5.37 (TLBI RVAE1OS, TLBI RVAE1OSNXS, TLB Range Invalidate by VA, EL1, Outer Shareable).
- C5.5.38 (TLBI RVAE2, TLBI RVAE2NXS, TLB Range Invalidate by VA, EL2).
- C5.5.39 (TLBI RVAE2IS, TLBI RVAE2ISNXS, TLB Range Invalidate by VA, EL2, Inner Shareable).
- C5.5.40 (TLBI RVAE2OS, TLBI RVAE2OSNXS, TLB Range Invalidate by VA, EL2, Outer Shareable).
- C5.5.44 (TLBI RVALE1, TLBI RVALE1NXS, TLB Range Invalidate by VA, Last level, EL1).

- C5.5.45 (TLBI RVALE1IS, TLBI RVALE1ISNXS, TLB Range Invalidate by VA, Last level, EL1, Inner Shareable).
- C5.5.46 (TLBI RVALE1OS, TLBI RVALE1OSNXS, TLB Range Invalidate by VA, Last level, EL1, Outer Shareable).
- C5.5.47 (TLBI RVALE2, TLBI RVALE2NXS, TLB Range Invalidate by VA, Last level, EL2).
- C5.5.48 (TLBI RVALE2IS, TLBI RVALE2ISNXS, TLB Range Invalidate by VA, Last level, EL2, Inner Shareable).
- C5.5.49 (TLBI RVALE2OS, TLBI RVALE2OSNXS, TLB Range Invalidate by VA, Last level, EL2, Outer Shareable).

In section C5.5.21 (TLBI RIPAS2E1, TLBI RIPAS2E1NXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1), in the subsection 'Purpose', the text that reads:

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

is updated to read:

- The entry is a stage 2 only translation table entry, from any level of the translation table walk up to the level indicated in the TTL hint.

Equivalent changes are made in the following sections:

- C5.5.22 (TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable).
- C5.5.23 (TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable).
- C5.5.29 (TLBI RVAAE1, TLBI RVAAE1NXS, TLB Range Invalidate by VA, All ASID, EL1).
- C5.5.30 (TLBI RVAAE1IS, TLBI RVAAE1ISNXS, TLB Range Invalidate by VA, All ASID, EL1, Inner Shareable).
- C5.5.31 (TLBI RVAAE1OS, TLBI RVAAE1OSNXS, TLB Range Invalidate by VA, All ASID, EL1, Outer Shareable).
- C5.5.41 (TLBI RVAE3, TLBI RVAE3NXS, TLB Range Invalidate by VA, EL3).
- C5.5.42 (TLBI RVAE3IS, TLBI RVAE3ISNXS, TLB Range Invalidate by VA, EL3, Inner Shareable).
- C5.5.43 (TLBI RVAE3OS, TLBI RVAE3OSNXS, TLB Range Invalidate by VA, EL3, Outer Shareable).
- C5.5.50 (TLBI RVALE3, TLBI RVALE3NXS, TLB Range Invalidate by VA, Last level, EL3).
- C5.5.51 (TLBI RVALE3IS, TLBI RVALE3ISNXS, TLB Range Invalidate by VA, Last level, EL3, Inner Shareable).
- C5.5.52 (TLBI RVALE3OS, TLBI RVALE3OSNXS, TLB Range Invalidate by VA, Last level, EL3, Outer Shareable).

Also in section C5.5.25 (TLBI RIPAS2LE1IS, TLBI RIPAS2LE1ISNXS, TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable), in the field 'TTL, bits [38:37]', the text that reads:

> TTL Level hint. The TTL hint is only guaranteed to invalidate entries in the range that match the level described by the TTL hint.
>
> `0b00` The entries in the range can be using any level for the translation table entries.
>
> `0b01` All entries to invalidate are Level 1 translation table entries.
>
> If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.
>
> `0b10` All entries to invalidate are Level 2 translation table entries.
>
> `0b11` All entries to invalidate are Level 3 translation table entries.

is updated to read:

> TTL Level hint. The TTL hint is only guaranteed to invalidate:
>
> - Non-leaf-level entries in the range up to but not including the level described by the TTL hint.
> - Leaf-level entries in the range that match the level described by the TTL hint.
>
> `0b00` The entries in the range can be using any level for the translation table entries.
>
> `0b01` The TTL hint indicates level 1.
>
> If FEAT_LPA2 is not implemented, when using a 16KB translation granule, this value is reserved and hardware should treat this field as `0b00`.
>
> `0b10` The TTL hint indicates level 2.
>
> `0b11` The TTL hint indicates level 3.

Equivalent changes are made in all of the sections listed above.

2.61 D19800

In section J1.1.3 (aarch64/function), the function IsHCRXEL2Enabled(), that reads as:

```
boolean IsHCRXEL2Enabled()
    assert(HaveFeatHCX());
   ...
```

Is updated to read:

```
boolean IsHCRXEL2Enabled()
    if !HaveFeatHCX() then return FALSE;
    ...
```

2.62 D19804

In section D9.4.1 (Virtual address translation), the following text is added:

> If a tag write by an STG instruction that does not also write data is translated by a writeable-clean descriptor, but the tag write effect is IGNORED due to a stage 1 descriptor not having the Tagged memory attribute, or because Allocation tag access is disabled for the instruction by SCR_EL3.ATA, HCR_EL2.ATA, SCTLR_ELx.ATA or SCTLR_ELx.ATA0, it is **CONSTRAINED UNPREDICTABLE** whether hardware updates the dirty state of that descriptor.

2.63 R19810

In section B2.3.3 (Ordering relations), the definition of 'Tag-ordered-before' is updated to read:

> If FEAT_MTE2 is implemented, a Memory Tag-Check-read R1 is Tag-ordered-before a Checked Memory Write effect W2 generated by the same instruction if and only if all of the following apply:
>
> - There is an Intrinsic data dependency from R1 to a Conditional-Branching effect B3 generated by the same instruction as R1.
> - There is an Intrinsic control dependency from the Conditional-Branching effect B3 to W2.

2.64 D19817

In section G8.3.33 (PMMIR, Performance Monitors Machine Identification Register) in the BUS_SLOTS, bits [15:8] field, the text that reads:

> Bus count. The largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle. When this field is nonzero, the largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle is BUS_SLOTS. This field has an **IMPLEMENTATION DEFINED** value. Access to this field is RO.

is corrected to read:

> Bus count. The largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle. When this field is nonzero, the largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle is BUS_SLOTS. If the information is not available, this field will read as zero. This field has an **IMPLEMENTATION DEFINED** value. Access to this field is RO.

The equivalent changes are made in section D17.5.12 (PMMIR_EL1, Performance Monitors Machine Identification Register) and I5.3.30 (PMMIR, Performance Monitors Machine Identification Register).

2.65 D19829

In section D17.2.63 (ID_AA64ISAR2_EL1, AArch64 Instruction Set Attribute Register 2), in the 'RPRES, bits [7:4]' field, the following text is removed:

> From Armv8.7, if Advanced SIMD and floating-point is implemented, the only permitted value is `0b0001`.

2.66 E19831

In section K7.2 (Gray-count scheme for timer distribution scheme), the following pseudocode for Gray code encoding and decoding:

```
Gray[N] = Count[N]
Gray[i] = (XOR(Gray[N:i+1])) XOR Count[i] for N-1 >= i >= 0
Count[i] = XOR(Gray[N:i]) for N >= i >= 0
```

is updated to read:

```
Gray = Count EOR ('0':Count<N:1>)
Count<N> = Gray<N>
for i = N-1 downto 0
   Count<i> = Gray<i> EOR Count<i+1>
```

2.67 D19833

In section K7.2 (Gray-count scheme for timer distribution scheme) the following Note is removed:

> This scheme has the advantage of being relatively simple to switch, in either direction, between operating with low-frequency and low-precision, and operating with high-frequency and high-precision. To achieve this, the ratio of the frequencies must be 2^n, where n is an integer. A switch-over can occur only on the 2 n+1 boundary to avoid losing the Gray-coding property on a switch-over.

2.68 C19835

In section B2.3.12 (Limited ordering regions), after the following text:

> A memory location lies within the LORegion identified by the LORegion Number if the PA lies between the Start Address and the End Address, inclusive. The Start Address must be defined to be aligned to 64KB and the End Address must be defined as the top byte of a 64KB block of memory.

the following statement is added:

> It is permitted for multiple LORegion descriptors with non-overlapping address ranges to be configured with the same LORegion Number.

2.69 D19887

In section J1.1.3 (aarch64/functions), the write accessor Mem[] (assignment form) reading:

```
Mem[bits(64) address, integer size, AccType acctype, boolean ispair] = bits(size*8)
 value_in
    ...
    if !atomic && ispair && address == Align(address, halfsize) then
        single_is_aligned = TRUE;
        <highhalf, lowhalf> = value;
        AArch64.MemSingle[address, halfsize, acctype,
                          single_is_aligned, ispair] = lowhalf;
        AArch64.MemSingle[address + halfsize, halfsize, acctype,
                          single_is_aligned, ispair] = highhalf;
    elsif atomic && ispair then
        AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
    ...
```

Is updated to read:

```
Mem[bits(64) address, integer size, AccType acctype, boolean ispair] = bits(size*8)
 value_in
    ...
    if !atomic && ispair && address == Align(address, halfsize) then
        single_is_pair = FALSE;
        single_is_aligned = TRUE;
        <highhalf, lowhalf> = value;
        AArch64.MemSingle[address, halfsize, acctype,
                          single_is_aligned, single_is_pair] = lowhalf;
        AArch64.MemSingle[address + halfsize, halfsize, acctype,
                          single_is_aligned, single_is_pair] = highhalf;
    elsif atomic && ispair then
        AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
    ...
```

2.70 E19892

In section J1.1.5 (aarch64/translation), the function S1HasPermissionsFault() that reads:

```
boolean AArch64.S1HasPermissionsFault(
                Regime regime,
                SecurityState ss,
                TTWState walkstate,
                S1TTWParams walkparams,
                boolean ispriv,
                AccType acctype,
                boolean iswrite
)
```

Is replaced by S1CheckPermissions():

```
FaultRecord AArch64.S1CheckPermissions(
                Regime regime,
                SecurityState ss,
                TTWState walkstate,
                S1TTWParams walkparams,
                boolean ispriv,
                AccType acctype,
                boolean iswrite,
                FaultRecord fault_in
)
```

In section J1.1.5 (aarch64/translation), the function S2HasPermissionsFault() that reads:

```
boolean AArch64.S2HasPermissionsFault(
                boolean s2fs1walk,
                TTWState walkstate,
                SecurityState ss,
                S2TTWParams walkparams,
                boolean ispriv,
                AccType acctype,
                boolean iswrite
)
```

Is replaced by S2CheckPermissions():

```
FaultRecord AArch64.S2CheckPermissions(
                boolean s2fs1walk,
                TTWState walkstate,
                SecurityState ss,
                S2TTWParams walkparams,
                boolean ispriv,
                AccType acctype,
                boolean iswrite,
                FaultRecord fault
)
```

Appropriate changes are made in the pseudocode where these functions are called.

In section D8.15 (Pseudocode description of VMSAv8-64 address translation), the subsection 'Fault detection' is updated to take these changes into account.

2.71 D19917

In section D17.2.36 (DCZID_EL0, Data Cache Zero ID register), in the definition of 'BS, bits [3:0]', the following text is added:

> If FEAT_MTE2 is implemented, the minimum size supported is 16 bytes (value == 2).

2.72 D19918

In section J1.1.3 (aarch64/functions), in the AArch64.CheckAlignment() function, the code that reads:

```
    if SCTLR[].A == '1' then check = TRUE;
    elsif HaveLSE2Ext() then
        check = (UInt(address<3:0>) + alignment > 16) && ((ordered && SCTLR[].nAA ==
'0') || atomic);
    else check = atomic || ordered;
```

Is updated to read:

```
    if SCTLR[].A == '1' then check = TRUE;
    elsif HaveLSE2Ext() then
        // For ordered pair operation check whether entire access is within 16-byte
        integer accsize = if ispair then alignment * 2 else alignment;
        check = (UInt(address<3:0>) + accsize > 16) && ((ordered && SCTLR[].nAA ==
'0') || atomic);
    else check = atomic || ordered;
```

In section J1.1.3 (aarch64/functions), in the Mem[] non-assignment (read) accessor function, the code that reads:

```
bits(size*8) Mem[...]
    ...
    if ispair then
        // check alignment on size of element accessed, not overall access size
        aligned = AArch64.CheckAlignment(address, halfsize, acctype, iswrite);
    else
        aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
```

Is updated to read:

```
bits(size*8) Mem[...]
    ...
    integer align_size = if ispair then halfsize else size;
    aligned = AArch64.CheckAlignment(address, align_size, acctype, iswrite, ispair);
```

Equivalent changes are made to the Mem[] assignment (write) accessor function.

2.73 D19928

In sections D17.2.118 (SCTLR_EL1, System Control Register (EL1)) and D17.2.119 (SCTLR_EL2, System Control Register (EL2)), in the 'EPAN, bit [57]' field, the text that reads:

> Any speculative data accesses that would generate a Permission fault if the accesses were not speculative will not cause an allocation into a cache.

is corrected to read:

> Any speculative data accesses that would generate a Permission fault as a result of PSTATE.PAN=1 if the accesses were not speculative will not cause an allocation into a cache.

2.74 D19936

In section D17.5.9 (PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30), the description of 'T, bit [23]' that reads:

> When FEAT_TME is implemented:
>
> Transactional state filtering bit. Controls counting in Transactional state.
>
> `0b0` This bit has no effect on filtering of events.
>
> `0b1` Do not count events in Transactional state.

is updated to read:

> When FEAT_TME is implemented:
>
> Transactional state filtering bit. Controls counting of Attributable events in Non-transactional state.
>
> `0b0` This bit has no effect on filtering of events.
>
> `0b1` Do not count Attributable events in Non-transactional state.
>
> For each Unattributable event, it is **IMPLEMENTATION DEFINED** whether the filtering applies.

Equivalent changes are made in the following sections:

- D17.5.1 (PMCCFILTR_EL0, Performance Monitors Cycle Count Filter Register).
- I5.3.24 (PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30).

The updated definition of 'T, bit [23]' is added to section I5.3.2 (PMCCFILTR_EL0, Performance Monitors Cycle Counter Filter Register).

2.75 C19956

In section D11.11.3 (Common event numbers), in the description of PMU event '`0x0012`, BR_PRED', the following text is added:

> If no program-flow prediction resources are implemented, this event is optional, but Arm recommends that BR_PRED counts all branches.

> It is **IMPLEMENTATION DEFINED** when the branch is counted. Arm recommends that it is counted when the branch is resolved, that is, at the same point in the instruction pipeline as when the BR_MIS_PRED event would be counted if the branch resolves as mispredicted. This means that (BR_PRED - BR_MIS_PRED) is the number of correctly predicted branches and the ratio (BR_MIS_PRED ÷ BR_PRED) can be calculated in a meaningful way.
>
> PMCEID0_EL0[18] reads as `0b1` if this event is implemented and `0b0` otherwise.

2.76 D19961

In section C7.2.227 (SABDL, SABDL2), the text that reads:

> This instruction subtracts the vector elements of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the results into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

is corrected to read:

> This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the results into a vector, and writes the vector to the destination SIMD&FP register.

2.77 C20009

In section D17.2.40 (FAR_EL1, Fault Address Register (EL1)), the Note that reads:

> The address held in this field is an address accessed by the instruction fetch or data access that caused the exception that actually gave rise to the instruction or data abort. It is the lower address that gave rise to the fault. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores an unaligned address that crosses a page boundary, the architecture does not prioritize between those different faults.

is updated to read:

> The address held in this field is an address accessed by the instruction fetch or data access that caused the exception that actually gave rise to the Instruction or Data Abort. It is the lower address that gave rise to the fault that is reported. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores an unaligned address that crosses a page boundary, the architecture does not prioritize which fault is reported.

Equivalent changes are made in the following sections:

- D17.2.41 (FAR_EL2, Fault Address Register (EL2)).

- D17.2.42 (FAR_EL3, Fault Address Register (EL3)).
- D17.2.55 (HPFAR_EL2, Hypervisor IPA Fault Address Register).

2.78 D20011

In section D11.11.3 (Common event numbers), subsection 'Common microarchitectural events', in the '`0x0074`, ASE_SPEC, Operation speculatively executed, Advanced SIMD' definition, the bullet points that read:

> - Cryptographic operations other than PMULL, in AArch64 state.
> - VMULL, in AArch32 state.

are changed to read:

> - Cryptographic operations, other than PMULL, PMULL2 (1Q variant) in AArch64 state and VMULL (P64 variant) in AArch32 state.

In the same event definition, the text that reads:

> In AArch64 state, PMULL, and in AArch32 state, VMULL are counted as Advanced SIMD operations.

is changed to read:

> Advanced SIMD PMULL, PMULL2 (1Q variant) in AArch64 state and VMULL (P64 variant) in AArch32 state are counted as Advanced SIMD operations.

In the same section, in the '`0x0077`, CRYPTO_SPEC, Operation speculatively executed, Cryptographic instruction' definition, the text that reads:

> The counter counts each operation counted by INST_SPEC that is a cryptographic operation other than PMULL or VMULL.
>
> See The Cryptographic Extension on page C3-333.

is changed to read:

> The counter counts each operation counted by INST_SPEC that is a cryptographic operation, other than Advanced SIMD PMULL, PMULL2 (1Q variant) and SVE2 PMULLB, PMULLT (Q variant) in AArch64 state, and Advanced SIMD VMULL (P64 variant) in AArch32 state.
>
> See The Armv8 Cryptographic Extension on page A2-80 and SVE2 Crypto Extensions on page C4-485.

2.79 D20053

In section F2.11 (Advanced SIMD and floating-point load/store instructions), in Table F2-17 'SIMD and floating-point register file load/store instructions', the 'Operation' description for Vector Load Multiple that reads:

> Load 1-16 consecutive 32-bit registers, floating-point only.

is corrected to read:

> Load 1-32 consecutive 32-bit registers, floating-point only.

In the same table, the 'Operation' description for Vector Store Multiple that reads:

> Store 1-16 consecutive 32-bit registers, floating-point only.

is corrected to read:

> Store 1-32 consecutive 32-bit registers, floating-point only.

2.80 E20075

In section A2.2.1 (Additional functionality added to Armv8.0 in later releases), the definition 'FEAT_ETS, Enhanced Translation Synchronization' is deleted, and replaced with a definition of FEAT_ETS2.

In section D8.2.6 (Translation table walk properties), the rule R_{LTJGW} is deleted, and is replaced with the following rule:

> If FEAT_ETS2 is implemented, E1 is an Explicit Memory Effect, E2 is an Implicit Read of a PTE and all of the following apply, then E1 is Ordered-before E2:
>
> - E1 is program-order-before a Fault Effect E3.
> - E2 is Translation-intrinsically-before E3.

In the following sections:

- D17.2.65 (ID_AA64MMFR1_EL1, AArch64 Memory Model Feature Register 1), field 'ETS, bits [39:36]'.
- D17.2.86 (ID_MMFR5_EL1, AArch32 Memory Model Feature Register 5), field 'ETS, bits [3:0]'.
- G8.2.97 (ID_MMFR5, Memory Model Feature Register 5), field 'ETS, bits [3:0]'.

The field definition is updated to read:

> Indicates support for Enhanced Translation Synchronization. Defined values are:
>
> `0b0000` FEAT_ETS2 is not implemented.

> `0b0001` FEAT_ETS2 is not implemented.
>
> `0b0010` FEAT_ETS2 is implemented.
>
> All other values are reserved. FEAT_ETS2 implements the functionality identified by the value `0b0010`. In Armv8.0, the permitted values are `0b0000`, `0b0001`, and `0b0010`.

In section E2.4 (Ordering of translation table walks), the text that reads:

> If FEAT_ETS is implemented, and a memory access RW1 is Ordered-before a second memory access RW2, then RW1 is also Ordered-before any translation table walk generated by RW2 that generates any of the following:
>
> - A Translation fault.
> - An Address size fault.
> - An Access flag fault.

is updated to read:

> If FEAT_ETS2 is implemented, E1 is an Explicit Memory Effect, E2 is an Implicit Read of a PTE and all of the following apply, then E1 is Ordered-before E2:
>
> - E1 is program-order-before a Fault Effect E3.
> - E2 is Translation-intrinsically-before E3.

References to FEAT_ETS are replaced with FEAT_ETS2 throughout the document.

2.81 D20128

In section D13.6.3 (Additional information for each profiled memory access operation), the bullet list that reads:

> The sampled data physical address packet is not output if any of the following are true:
>
> - The PE does not translate the address, for example because it does not perform the access or the address translation generates a Translation fault.
> - The sampled data virtual address packet is not output.
> - Sampling of physical addresses is prohibited by System register controls.

is changed to read:

> The sampled data physical address packet is not output if any of the following are true:
>
> - The sampled operation operates on a virtual address and any of the following are true:
>   - The PE does not translate the address, for example because it does not perform the access or the address translation generates a Translation fault.
>   - The sampled data virtual address packet is not output.

- Sampling of physical addresses is prohibited by System register controls.

If AArch64.ExclusiveMonitorPass() or AArch32.ExclusiveMonitorPass() returns FALSE for a Store-Exclusive instruction, it is **IMPLEMENTATION DEFINED** whether or not the physical address packet is output when permitted by the above rules.

2.82 D20315

In section I5.8.32 (ERR<n>STATUS, Error Record <n> Primary Status Register, n = 0 - 65534), in the 'SERR, bits [7:0]' field, the value descriptions that read:

`0x10` Internal data register. For example, parity on a SIMD&FP register. For a PE, all general-purpose, stack pointer, SIMD&FP, and SVE registers are data registers.

`0x11` Internal control register. For example, Parity on a System register. For a PE, all registers other than general-purpose, stack pointer, SIMD&FP, and SVE registers are control registers.

are updated to read:

`0x10` Internal data register. For example, parity on a SIMD&FP register. For a PE, all general-purpose, stack pointer, SIMD&FP, SVE, and SME registers are data registers.

`0x11` Internal control register. For example, Parity on a System register. For a PE, all registers other than general-purpose, stack pointer, SIMD&FP, SVE, and SME registers are control registers.

2.83 C20158

In section D11.11.3 (Common event numbers), in the subsection 'Common microarchitectural events', the text in the description of '`0x4024`, MEM_ACCESS_CHECKED, Checked data memory access' that reads:

The counter counts each memory access counted by MEM_ACCESS that is checked by the Memory Tagging Extension.

is updated to read:

The counter counts each memory access counted by MEM_ACCESS that accesses an Allocation Tag due to a Tag Check operation.

2.84 D20163

In section J1.3.5 (shared/translation), the code in function S2CombineS1MemAttrs() which reads:

```
MemoryAttributes S2CombineS1MemAttrs(MemoryAttributes s1_mematttrs, MemoryAttributes
 s2_mematttrs)
    MemoryAttributes mematttrs;
    ...
    mematttrs.xs = s2_mematttrs.xs
```

is updated to read:

```
MemoryAttributes S2CombineS1MemAttrs(MemoryAttributes s1_mematttrs, MemoryAttributes
 s2_mematttrs)
    MemoryAttributes mematttrs;
    ...
    if (mematttrs.memtype == MemType_Normal &&
            mematttrs.inner.attrs == MemAttr_WB &&
            mematttrs.outer.attrs == MemAttr_WB) then
        mematttrs.xs = '0';
    else
        mematttrs.xs = s2_mematttrs.xs AND s1_mematttrs.xs;
```

In section J1.1.5 (aarch64/translation) the code in function AArch64.S2ApplyFWBMemAttrs() which reads:

```
MemoryAttributes AArch64.S2ApplyFWBMemAttrs(MemoryAttributes s1_mematttrs,

    bits(4) s2_attr, bits(2) s2_sh)
    MemoryAttributes mematttrs;
    if s2_attr<2> == '0' then // S2 Device, S1 any
        ...
    elsif s2_attr<1:0> == '11' then // S2 attr = S1 attr
        mematttrs = s1_mematttrs;
    elsif s2_attr<1:0> == '10' then // Force writeback
        ...
    else // Non-cacheable unless S1 is device
        ...
         if s1_mematttrs.memtype == MemType_Device then
             mematttrs = s1_mematttrs;
        else
            ...
    ...
    mematttrs.shareability = EffectiveShareability(mematttrs);
    return mematttrs;
```

is updated to read:

```
MemoryAttributes AArch64.S2ApplyFWBMemAttrs(MemoryAttributes s1_mematttrs,

    bits(4) s2_attr, bits(2) s2_sh)
    s2_attr = descriptor<5:2>;
    s2_sh = if walkparams.ds == '1' then walkparams.sh else descriptor<9:8>;
    s2_fnxs = descriptor<11>;
    MemoryAttributes mematttrs;
    if s2_attr<2> == '0' then // S2 Device, S1 any
        ...
        mematttrs.xs = s1_mematttrs.xs;
    elsif s2_attr<1:0> == '11' then // S2 attr = S1 attr
        mematttrs = s1_mematttrs;
```

```
    elsif s2_attr<1:0> == '10' then // Force writeback
        ...
        memattrs.xs = '0';
    else // Non-cacheable unless S1 is device
        ...
        if s1_memattrs.memtype == MemType_Device then
            memattrs = s1_memattrs;
        else
            ...
            memattrs.xs = s1_memattrs.xs;
    ...
    if s2_fnxs == '1' then
        memattrs.xs = '0';
    memattrs.shareability = EffectiveShareability(memattrs);
    return memattrs;
```

2.85 R20165

In section D11.7.2 (Accuracy of event filtering), subsection 'Software increment events', the text that reads:

> Software increment events must also be counted without the need for explicit synchronization. For example, two software increments executed without an intervening Context synchronization event must increment the event counter twice.

is updated to read:

> If the PE performs two architecturally executed writes to the PMSWINC_EL0 or PMSWINC register without an intervening Context synchronization event, then the counter is incremented twice.

2.86 D20171

In section C6.2.43 (CASH, CASAH, CASALH, CASLH), the bullet that reads:

- CAS has neither acquire nor release semantics.

is corrected to read:

- CASH has neither acquire nor release semantics.

In section C6.2.44 (CASP, CASPA, CASPAL, CASPL), the bullet that reads:

- CAS has neither acquire nor release semantics.

is corrected to read:

- CASP has neither acquire nor release semantics.

2.87 D20207

In sections C7.2.9 (AESIMC), C7.2.10 (AESMC), F6.1.3 (AESIMC), and F6.1.4 (AESMC), the instructions' dependency on FEAT_AES is added.

2.88 R20208

In section D11.11.3 (Common event numbers), in the subsection 'Common microarchitectural events', the text in the description of '0x0024, STALL_BACKEND, No operation sent for execution due to the backend' that reads:

> The counter counts each cycle counted by CPU_CYCLES where no Attributable instruction or operation was sent for execution and either:
>
> - The backend is unable to accept any of the instruction operations available for the PE.
> - The backend is unable to accept any operations for the PE.
>
> Note: In a single cycle, both the STALL_BACKEND and STALL_FRONTEND events might be counted, if both the backend is unable to accept any operations and there are no operations available to issue from the frontend.

is updated to read:

> The counter counts each cycle counted by CPU_CYCLES where Attributable instructions or operations are available to dispatch for the PE from the frontend, but no Attributable instruction or operation is sent for execution because the backend is unable to accept any of the instructions or operations available for the PE.
>
> It is **IMPLEMENTATION DEFINED** whether the counter also counts each cycle counted by CPU_CYCLES where no Attributable instructions or operations are available to dispatch for the PE from the frontend and the backend is unable to accept any instructions or operations for the PE.
>
> Note: This means that it is **IMPLEMENTATION DEFINED** whether both the STALL_BACKEND and STALL_FRONTEND events can be counted in the same cycle.

Equivalent changes are made to the following event descriptions:

- 0x003D, STALL_SLOT_BACKEND, No operation sent for execution on a Slot due to the backend.
- 0x003E, STALL_SLOT_FRONTEND, No operation sent for execution on a Slot due to the frontend.

An equivalent change is made to the Note in the description of '0x0023, STALL_FRONTEND, No operation sent for execution due to the frontend' as described above for the 0x0024, STALL_BACKEND event description.

2.89 D20210

In section J1.1.3 (aarch64/functions), the function AArch64.PhysicalSErrorSyndrome() that reads as:

```
bits(25) AArch64.PhysicalSErrorSyndrome(boolean implicit_esb)
    bits(25) syndrome = Zeros(25);
...
    if errorstate == ErrorState_Uncategorized then
    ...
    elsif errorstate == ErrorState_IMPDEF then
    ...
    else
        syndrome<24> = '0';                                              // IDS
        syndrome<13> = (if implicit_esb then '1' else '0');              // IESB
        syndrome<12:10> = AArch64.EncodeAsyncErrorSyndrome(errorstate); // AET
        syndrome<5:0> = '010001';
```

is changed to:

```
bits(25) AArch64.PhysicalSErrorSyndrome(boolean implicit_esb)
...
    if errorstate == ErrorState_Uncategorized then
    ...
    elsif errorstate == ErrorState_IMPDEF then
    ...
    else
        syndrome<24> = '0';                                              // IDS
        syndrome<13> = (if implicit_esb then '1' else '0');              // IESB
        syndrome<12:10> = AArch64.EncodeAsyncErrorSyndrome(errorstate); // AET
        syndrome<9> = fault.extflag;                                     // EA
        syndrome<5:0> = '010001';                                        // DFSC
```

Similarly in section J1.2.3 (aarch32/functions), the function AArch32.PhysicalSErrorSyndrome() that reads as:

```
bits(16) AArch32.PhysicalSErrorSyndrome()
    bits(32) syndrome = Zeros(32);
    FaultRecord fault = GetPendingPhysicalSError();
    boolean long_format = TTBCR.EAE == '1';
    syndrome = AArch32.CommonFaultStatus(fault, long_format);
    return syndrome<15:0>;
```

is updated to:

```
bits(16) AArch32.PhysicalSErrorSyndrome()
    bits(32) syndrome = Zeros(32);
    FaultRecord fault = GetPendingPhysicalSError();
    if PSTATE.EL == EL2 then
        ErrorState errstate = AArch32.PEErrorState(fault);
        syndrome<11:10> = AArch32.EncodeAsyncErrorSyndrome(errstate);   // AET
        syndrome<9>     = fault.extflag;                                // EA
        syndrome<5:0>   = '010001';                                     // DFSC
    else
        boolean long_format = TTBCR.EAE == '1';
        syndrome = AArch32.CommonFaultStatus(fault, long_format);
    return syndrome<15:0>;
```

2.90 C20220

In section D1.3.6 (Asynchronous exception types), the rule R_{PFDGT} is added:

> If an interrupt was pending and its Superpriority attribute changes, it is **CONSTRAINED UNPREDICTABLE** whether the interrupt uses the previous or current value of the Superpriority attribute when evaluating masking conditions. If the interrupt is taken using the previous value of the Superpriority attribute, it is taken before the first Context synchronization event after the Superpriority attribute changed.

2.91 C20237

In section H9.2.11 (EDACR, External Debug Auxiliary Control Register), the 'Configuration' text that reads:

> If FEAT_DoPD is implemented, this register is implemented in the Core power domain.
>
> If FEAT_DoPD is not implemented, the power domain that this register is implemented in is **IMPLEMENTATION DEFINED**.

> If the EDACR contains any control bits that must be preserved over power down, then these bits must be accessible by the external debug interface when the OS Lock is locked, OSLSR_EL1.OSLK == 1, and when the Core is powered off.

is updated to read:

> If FEAT_DoPD is implemented:
>
> - This register is implemented in the Core power domain.
> - Any mechanism to preserve control bits in EDACR over power down is optional and **IMPLEMENTATION DEFINED**.
>
> If FEAT_DoPD is not implemented:
>
> - The power domain that this register is implemented in is **IMPLEMENTATION DEFINED**.
> - If the EDACR contains any control bits that must be preserved over power down, then these bits must be accessible by the external debug interface when the OS Lock is locked, OSLSR_EL1.OSLK == 1, and, when the Core is powered off.

2.92 D20268

In section J1.1.4 (aarch64/instrs), the function AArch64.RestrictPrediction() that reads:

```
if EL2Enabled() && !IsInHost() then
     if PSTATE.EL IN {EL0, EL1} then
        c.is_vmid_valid = TRUE;
        c.all_vmid = FALSE;
```

```
        c.vmid = VMID[];
    elsif target_el IN {EL0, EL1} then
        c.is_vmid_valid = TRUE;
        c.all_vmid = val<48> == '1';
        c.vmid = val<47:32>; // Only valid if val<48> == '0';
    else
        c.is_vmid_valid = FALSE;
```

Is updated to read:

```
if EL2Enabled() then
     if (PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1 then
        c.is_vmid_valid = TRUE;
        c.all_vmid = FALSE;
        c.vmid = VMID[];
    elsif (target_el == EL0 && !ELIsInHost(target_el)) || target_el == EL1 then
        c.is_vmid_valid = TRUE;
        c.all_vmid = val<48> == '1';
        c.vmid = val<47:32>; // Only valid if val<48> == '0'
    else
        c.is_vmid_valid = FALSE;
```

2.93 C20275

In section D11.11.3 (Common event numbers), in the subsection 'Common architectural events', the event descriptions that read:

> `0x000B`, CID_WRITE_RETIRED, Instruction architecturally executed, Condition code check pass, write to CONTEXTIDR
>
> The counter counts each MSR write to CONTEXTIDR_EL1 and each MCR write to CONTEXTIDR.
>
> If the PE performs two architecturally-executed writes to CONTEXTIDR without an intervening Context synchronization event, it is **CONSTRAINED UNPREDICTABLE** whether the first write is counted.
>
> When FEAT_VHE is implemented, the counter:
>
> - Counts each architecturally-executed instruction accessing the named register CONTEXTIDR_EL1, including when executing at EL2 when HCR_EL2.E2H is `0b1`.
> - Does not count instructions accessing the named register CONTEXTIDR_EL12.
>
> Note: The event is defined by the name used to access the register. The counter does not count writes to the named register CONTEXTIDR_EL2.
>
> `0x001C`, TTBR_WRITE_RETIRED, Instruction architecturally executed, Condition code check pass, write to TTBR
>
> The counter counts MSR writes to TTBR0_EL1 and TTBR1_EL1 in AArch64 state and MCR and MCRR writes to TTBR0 and TTBR1 in AArch32 state. When EL3 is implemented and using AArch32, this includes counting writes to both banked copies of TTBR0 and TTBR1.

If the PE executes two writes to the same TTBR, without an intervening Context synchronization event, it is **CONSTRAINED UNPREDICTABLE** whether the first write to the TTBR, is counted.

If EL3 is implemented and using AArch64, the counter does not count writes to TTBR0_EL3.

If EL2 is implemented and using AArch64, the counter does not count writes to TTBR0_EL2 and VTTBR_EL2.

If EL2 is implemented and using AArch32, the counter does not count writes to HTTBR and VTTBR.

When FEAT_VHE is implemented, the counter:

- Counts each architecturally-executed instruction accessing the named registers TTBR0_EL1 and TTBR1_EL1, including when executing at EL2 when HCR_EL2.E2H is `0b1`.
- Does not count instructions accessing the named registers TTBR0_EL12 and TTBR1_EL12.

are updated to read:

`0x000B`, CID_WRITE_RETIRED, Instruction architecturally executed, Condition code check pass, write to CONTEXTIDR

The counter counts each MSR write to CONTEXTIDR_EL1 and each MCR write to CONTEXTIDR.

If the PE performs two architecturally executed writes to CONTEXTIDR without an intervening Context synchronization event, it is **CONSTRAINED UNPREDICTABLE** whether the first write is counted.

Note: The counter counts only writes to these named registers. For example:

- When FEAT_VHE or FEAT_Debugv8p2 is implemented, the counter does not count writes to the named register CONTEXTIDR_EL2.
- When FEAT_VHE is implemented, the counter:
  - Counts each architecturally executed instruction accessing the named register CONTEXTIDR_EL1, including when executing at EL2 when HCR_EL2.E2H is `0b1`.
  - Does not count instructions accessing the named register CONTEXTIDR_EL12.
- When FEAT_NV2 is implemented, the counter counts each write to the named register CONTEXTIDR_EL1, including when executing at EL1 when HCR_EL2 {NV2,NV1,NV} is {`0b1`,`0b1`,`0b1`}.

`0x001C`, TTBR_WRITE_RETIRED, Instruction architecturally executed, Condition code check pass, write to TTBR

The counter counts MSR writes to TTBR0_EL1 and TTBR1_EL1 in AArch64 state and MCR and MCRR writes to TTBR0 and TTBR1 in AArch32 state. When EL3 is implemented and using AArch32, this includes counting writes to both banked copies of TTBR0 and TTBR1.

If the PE executes two writes to the same TTBR, without an intervening Context synchronization event, it is **CONSTRAINED UNPREDICTABLE** whether the first write to the TTBR, is counted.

Note: The counter counts only writes to these named registers. For example:

- If EL3 is implemented and using AArch64, the counter does not count writes to TTBR0_EL3.
- If EL2 is implemented and using AArch64, the counter does not count writes to TTBR0_EL2 and VTTBR_EL2.
- If EL2 is implemented and using AArch32, the counter does not count writes to HTTBR and VTTBR.
- When FEAT_VHE is implemented, the counter:
  - Counts each write to the named registers TTBR0_EL1 and TTBR1_EL1, including when executing at EL2 when HCR_EL2.E2H is `0b1`.
  - Does not count instructions accessing the named registers TTBR0_EL12 and TTBR1_EL12.
- When FEAT_NV2 is implemented, the counter counts each write to the named registers TTBR0_EL1 and TTBR1_EL1, including when executing at EL1 when HCR_EL2.{NV2,NV1,NV} is {`0b1`,`0b1`,`0b1`}.

2.94 D20283

In section G8.2.65 (HCR2, Hyp Configuration Register 2), in field TOCU, bit [20], the text that reads:

Trap cache maintenance instructions that operate to the Point of Unification. Traps execution of those cache maintenance instructions at EL1 or EL0 using AArch64, and at EL1 using AArch32, to EL2.

This applies to the following instructions:

- When Non-secure EL0 is using AArch64, IC IVAU, DC CVAU. However, if the value of SCTLR_EL1.UCI is 0 these instructions are **UNDEFINED** at EL0 and any resulting exception is higher priority than this trap to EL2.
- When EL1 is using AArch64, IC IVAU, IC IALLU, DC CVAU.
- When Non-secure EL1 is using AArch32, ICIMVAU, ICIALLU, DCCMVAU.

— Note — An exception generated because an instruction is **UNDEFINED** at EL0 is higher priority than this trap to EL2. In addition:

- IC IALLUIS and IC IALLU are always **UNDEFINED** at EL0 using AArch64.
- ICIMVAU, ICIALLU, ICIALLUIS, and DCCMVAU are always **UNDEFINED** at EL0 using AArch32.

is changed to read:

Trap cache maintenance instructions that operate to the Point of Unification. Traps execution of ICIMVAU, ICIALLU, DCCMVAU at EL1 using AArch32, to EL2.

2.95 E20288

In section D17.2.49 (HCRX_EL2, Extended Hypervisor Configuration Register), the 'TALLINT, bit [6]' field description is updated from:

> Traps MSR writes of ALLINT at EL1 using AArch64 to EL2, when EL2 is implemented and enabled in the current Security state, reported using EC syndrome value `0x18`.

to:

> Traps the following writes at EL1 using AArch64 to EL2, when EL2 is implemented and enabled:
>
> - MSR (register) writes of ALLINT.
> - MSR (immediate) writes of ALLINT with a value of 1.

2.96 D20303

In section D1.3.1 (Exception entry terminology), in the subsection 'Definition of a precise exception and imprecise exception', the bullet within rule R_{TNVSL} that reads:

> - For a synchronous exception that is taken from AArch64 state during an instruction that performs more than one single-copy atomic memory access, the values in registers or memory affected by the instructions can be **UNKNOWN**, if all of the following apply:

is updated to read:

> - For a precise exception that is taken from AArch64 state during an instruction that performs more than one single-copy atomic memory access, the values in registers or memory affected by the instructions can be **UNKNOWN**, if all of the following apply:

In section D1.3.6 (Asynchronous exception types), in the subsection 'Taking an interrupt during a multi-access load or store', rule R_{ZBFSL} that reads:

> If in AArch64 state, interrupts can be taken during a sequence of memory accesses caused by a single load or store instruction. This is true regardless of the memory type being accessed.

is updated to read:

> In AArch64 state, interrupts can be taken during a sequence of memory accesses caused by a single load or store instruction. This is true regardless of the memory type being accessed.
>
> In this situation, the behavior is consistent with the requirements described in R_{TNVSL} in Definition of a precise exception and imprecise exception on page D1-4630.

2.97 D20310

In section D1.6.2 (Wait for Interrupt mechanism), the following rule R_{ZWCCZ} is deleted:

> If a WFI or WFIT instruction put a PE into low-power state, the PE remains in that low power state until it receives a WFE wake-up event.

2.98 D20317

In section D15.3 (Branch record buffer operation), the text in rule R_{QKQZL} that reads:

> If any of the following are true, the physical offset is zero, otherwise the physical offset is the value of CNTPOFF_EL2:
>
> - EL3 is implemented and SCR_EL3.ECVEn is 0.
> - EL2 is implemented and CNTHCTL_EL2.ECV is 0.

is updated to read:

> If any of the following are true, the physical offset is zero, otherwise the physical offset is the value of CNTPOFF_EL2:
>
> - FEAT_ECV is not implemented.
> - EL2 is not implemented.
> - EL3 is implemented and SCR_EL3.ECVEn is 0.
> - CNTHCTL_EL2.ECV is 0.

Additionally, the following text is added after Table D15-11 'Captured timestamp':

> If EL2 is not implemented, then the Effective value of BRBCR_EL2.TS is `0b00`.

2.99 D20319

In section J1.1.5 (aarch64/translation), the function AArch64.TLBContextEL20 that reads:

```
TLBContext AArch64.TLBContextEL20(SecurityState ss, bits(64) va, TGx tg)
    ...
    tlbcontext.asid   = if TCR_EL2.A1 == '0' then TTBR0_EL2.ASID else
 TTBR1_EL2.ASID;
    tlbcontext.tg     = tg;
    ...
```

Is updated to read:

```
TLBContext AArch64.TLBContextEL20(SecurityState ss, bits(64) va, TGx tg)
    ...
```

```
    tlbcontext.asid   = if TCR_EL2.A1 == '0' then TTBR0_EL2.ASID else
TTBR1_EL2.ASID;
    if TCR_EL2.AS == '0' then
        tlbcontext.asid<15:8> = Zeros(8);
    tlbcontext.tg     = tg;
    ...
```

In section J1.1.5 (aarch64/translation), the function AArch64.TLBContextEL10 that reads:

```
TLBContext AArch64.TLBContextEL10(SecurityState ss, bits(64) va, TGx tg)
    ...
    tlbcontext.asid   = if TCR_EL1.A1 == '0' then TTBR0_EL1.ASID else
TTBR1_EL1.ASID;
    tlbcontext.tg     = tg;
    ...
```

Is updated to read:

```
TLBContext AArch64.TLBContextEL10(SecurityState ss, bits(64) va, TGx tg)
    ...
    tlbcontext.asid   = if TCR_EL1.A1 == '0' then TTBR0_EL1.ASID else
TTBR1_EL1.ASID;
    if TCR_EL1.AS == '0' then
        tlbcontext.asid<15:8> = Zeros(8);
    tlbcontext.tg     = tg;
    ...
```

2.100 D20330

In section D8.8.3 (PAC instructions), rule I_{KBKGF} that reads:

> An instruction that extracts the PAC from the upper register bits and checks that the value is correct does all of the following:
>
> - The check is based on the value of the register and one other 64-bit value.
> - When the value is correct, the PAC is replaced with the extension bits.
> - When the value is incorrect, all of the following occur:
>   - The PAC is replaced with the extension bits.
>   - Two extension bits are set to a unique, fixed value.
>   - When the register is used as an indirect branch target, a Translation fault is generated because the VA is not mapped.

is updated to read:

> An instruction that extracts the PAC from the upper register bits and checks that the value is correct does all of the following:
>
> - The check is based on the value of the register and one other 64-bit value.
> - When the value is correct, the PAC is replaced with the extension bits.

- When the value is incorrect, all of the following occur:
  - The PAC is replaced with the extension bits.
  - Two extension bits are set to a unique, fixed value, such that the 64-bit value represents a non-canonical VA. This is referred to as making the VA non-canonical.

In section D8.8.4 (Faulting on pointer authentication), the following rules are deleted:

R_{CQRDJ} All statements in this section require implementation of FEAT_FPAC.

I_{JPFQK} If an instruction is a combined instruction that includes pointer authentication, then when the PAC is incorrect, one of the following **IMPLEMENTATION DEFINED** behaviors occur:

- A Translation fault is generated due to the authentication failure.
- The address is modified in a way that generates a Translation fault when the address is accessed.

I_{TVCPL} When an authentication failure occurs at EL0, the exception is taken at one of the following:

- If HCR_EL2.TGE is 0, the exception is taken at EL1.
- If HCR_EL2.TGE is 1, the exception is taken at EL2.

I_{MKMGV} If the current Exception level is not EL0, then when an authentication failure occurs, the exception is taken at the current Exception level.

I_{FBGSH} When an exception is generated due to an authentication failure, the ESR_ELx.EC code is set to `0x1C`.

Within the same section, the rules I_{XBGSY} and R_{MLWGL} are updated to read:

I_{XBGSY} A PAC authentication failure for a given VA can cause a fault to be generated in the following three manners, according to the type of the instruction and whether FEAT_FPAC and FEAT_FPACCOMBINE are implemented:

- The PAC instruction makes the VA non-canonical, such that a subsequent use of the VA generates a fault. In this case, the PAC instruction does not directly generate the fault.
- The PAC instruction makes the VA non-canonical and uses that VA such that a fault is generated by that instruction.
- The PAC instruction directly generates a Pointer Authentication instruction authentication failure exception, with EC code `0b011100`.

R_{MLWGL} If an instruction is a combined instruction that includes pointer authentication, then when the PAC is incorrect in a given VA, one of the following behaviors occurs:

- For a combined authenticate and load instruction, then:
  - If FEAT_FPACCOMBINE is not implemented, the VA is made non-canonical and then used as the address for the load.
  - If FEAT_FPACCOMBINE is implemented, then the instruction generates a Pointer Authentication instruction authentication failure exception, with EC code `0b011100`.

- For a combined authenticate and branch instruction, then:
  - If FEAT_FPACCOMBINE is not implemented, the VA is made non-canonical and the PC is updated to this non-canonical value.
  - If FEAT_FPACCOMBINE is implemented, then the instruction generates a Pointer Authentication instruction authentication failure exception, with EC code `0b011100`.

And the following rule is added to the section:

For a PAC authentication instruction, AUT*, then when the PAC is incorrect for a given VA, one of the following behaviors occurs:

- If FEAT_FPAC is not implemented, the VA is made non-canonical.
- If FEAT_FPAC is implemented, then the instruction generates a Pointer Authentication instruction authentication failure exception, with EC code `0b011100`.

2.101 D20332

In the following sections:

- C6.2.20 (AUTDA, AUTDZA).
- C6.2.21 (AUTDB, AUTDZB).
- C6.2.22 (AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA).
- C6.2.23 (AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB).

The text that reads:

If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

is updated to read:

For information on behavior if the authentication fails, see Faulting on pointer authentication on page D8-5159.

In the following sections:

- C6.2.36 (BLRAA, BLRAAZ, BLRAB, BLRABZ).
- C6.2.38 (BRAA, BRAAZ, BRAB, BRABZ).
- C6.2.122 (ERETAA, ERETAB).
- C6.2.169 (LDRAA, LDRAB).
- C6.2.255 (RETAA, RETAB).

The text that reads:

If the authentication fails, a Translation fault is generated.

is updated to read:

> For information on behavior if the authentication fails, see Faulting on pointer authentication on page D8-5159.

2.102 C20333

In section D8.11.1 (MMU fault types), in the subsection 'Translation fault', the rule R_{MLNTS} is updated to read:

> When a translation table entry generates a Translation fault, that translation table entry is not cached in a TLB.

2.103 D20334

In section D8.11.1 (MMU fault types), in the subsection 'Translation fault', the following bullet point in rule R_{VZZSZ} is deleted:

> - When FEAT_SVE is implemented, the corresponding TCR_ELx.NFDy field prevents non-faulting unprivileged accesses to an address translated by TTBRy_ELx.

2.104 D20335

In section B2.7.1 (Normal memory), the following bullet point is added under the list that begins 'The Normal memory type has the following properties:':

> - Where a load or store instruction performs a sequence of memory accesses, as opposed to one single-copy atomic access as defined in the rules for single-copy atomicity, these accesses might occur multiple times as a result of executing the load or store instruction.

The following Note is also added to the same section:

> Note:
> - Write speculation that is visible to other observers is prohibited for all memory types.

In section B2.3.10 (Restrictions on the effects of speculation), the following bullet point is added under the list that begins 'The Arm architecture places certain restrictions on the effects of speculation. These are:':

> - Write speculation that is visible to other observers is prohibited for all memory types.

2.105 D20340

In section D11.11.3 (Common event numbers), in the subsection 'Common microarchitectural events', the event definition '`0x8174`, CAS_SPEC, Atomic memory Operation speculatively executed, Compare and Swap' that reads:

> The counter counts each load atomic operation counted by LSE_LD_SPEC that is a Compare and Swap operation.

is updated to read:

> The counter counts each Compare and Swap operation.

2.106 C20341

In section D11.11.3 (Common events numbers), in the subsection 'Common microarchitectural events', the definition of '`0x8194`, DSNP_HIT_RD, Snoop hit, demand data read' that reads:

> The counter counts each snoop generated in response to a demand Memory-read operation counted by DSNP_HIT_RW that hits in a cache outside of the cache hierarchy of this PE.

is updated to read:

> The counter counts each snoop generated by the PE in response to a demand Memory-read operation counted by DSNP_HIT_RW that hits in and returns data from a cache outside of the cache hierarchy of this PE.
>
> Note: The event is counted by the PE generating the snoop, not the PE being snooped.

Equivalent changes are made to the ISNP_* and DSNP_* event descriptions throughout this section, although the Note is only added in the description of '`0x8190`, ISNP_HIT_RD, Snoop hit, demand instruction fetch'.

2.107 D20346

In section D7.4.13 (Execution, data prediction and prefetching restriction System instructions), the text that reads:

> If the System instruction is specified to apply to Exception levels that are not implemented, or which are higher than the Exception level that the System instruction is executed at, then the System instruction is treated as a **NOP**.

is updated to read:

> If the System instruction is specified to apply to a combination of Security state and Exception level that is not implemented, or an Exception level which is higher than the Exception level that the System instruction is executed at, then the System instruction is treated as a **NOP**.

Equivalent changes are made in the following sections:

- C5.6.1 (CFP RCTX, Control Flow Prediction Restriction by Context), in the description of 'EL, bits [25:24]'.
- C5.6.2 (CPP RCTX, Cache Prefetch Prediction Restriction by Context), in the description of 'EL, bits [25:24]'.
- C5.6.3 (DVP RCTX, Data Value Prediction Restriction by Context), in the description of 'EL, bits [25:24]'.
- G4.4.8 (Execution and data prediction restriction System instructions).
- G8.2.26 (CFPRCTX, Control Flow Prediction Restriction by Context), in the description of 'EL, bits [25:24]'.
- G8.2.34 (CPPRCTX, Cache Prefetch Prediction Restriction by Context), in the description of 'EL, bits [25:24]'.
- G8.2.50 (DVPRCTX, Data Value Prediction Restriction by Context), in the description of 'EL, bits [25:24]'.
- J1.1.4 (aarch64/instrs), in the function 'RestrictPrediction()'.

2.108 D20363

In section D17.2.111, (RNDR, Random Number), the accessibility pseudocode omits the effects of Debug state and the EDSCR.SDD field on the traps to EL3.

The MRS access pseudocode:

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_RNG_TRAP) && SCR_EL3.TRNDR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !IsFeatureImplemented(FEAT_RNG) then
        UNDEFINED;
    else
        X[t, 64] = RNDR;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_RNG_TRAP) && SCR_EL3.TRNDR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !IsFeatureImplemented(FEAT_RNG) then
        UNDEFINED;
    else
        X[t, 64] = RNDR;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_RNG_TRAP) && SCR_EL3.TRNDR == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !IsFeatureImplemented(FEAT_RNG) then
        UNDEFINED;
    else
        X[t, 64] = RNDR;
```

is corrected to:

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented(FEAT_RNG_TRAP) && SCR_EL3.TRNDR == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !IsFeatureImplemented(FEAT_RNG) then
        UNDEFINED;
    else
        X[t, 64] = RNDR;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented(FEAT_RNG_TRAP) && SCR_EL3.TRNDR == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !IsFeatureImplemented(FEAT_RNG) then
        UNDEFINED;
    else
        X[t, 64] = RNDR;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented(FEAT_RNG_TRAP) && SCR_EL3.TRNDR == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !IsFeatureImplemented(FEAT_RNG) then
        UNDEFINED;
    else
        X[t, 64] = RNDR;
```

A similar change is also made to the MSR access pseudocode and the accessors for section D17.2.112 (RNDRRS, Reseeded Random Number).

2.109 D20365

In section D15.1.4 (BRBE Prohibited regions), rule R_{JWWFY} which reads:

> When FEAT_BRBEv1p1 and EL3 are implemented:
>
> When MDCR_EL3.{E3BREC, E3BREW} is {0b01, 0b01} or MDCR_EL3.{E3BREC, E3BREW} is {0b10, 0b10}, self-hosted EL3 branch recording is enabled. When MDCR_EL3.{E3BREC, E3BREW} is {0b00, 0b00} or MDCR_EL3.{E3BREC, E3BREW} is {0b11,0b11}, self-hosted EL3 branch recording is disabled.

is corrected to read:

> When FEAT_BRBEv1p1 and EL3 are implemented:
>
> When MDCR_EL3.{E3BREC, E3BREW} is {0b0, 0b1} or MDCR_EL3.{E3BREC, E3BREW} is {0b1, 0b0}, self-hosted EL3 branch recording is enabled. When MDCR_EL3.{E3BREC, E3BREW} is {0b0, 0b0} or MDCR_EL3.{E3BREC, E3BREW} is {0b1,0b1}, self-hosted EL3 branch recording is disabled.

2.110 D20378

In section E1.3.5 (Flushing denormalized numbers to zero), the text that reads:

- If FPSCR.FZnstructions that convert from single-precision floating-point values to BF16 format flush denormalized outputs to zero.

is corrected to read:

- If FPSCR.FZ is 1, instructions that convert from single-precision floating-point values to BF16 format flush denormalized outputs to zero.

2.111 D20380

In J1.3.5 (shared/translation) the function EncodePARAttrs() does not account for encodings in which the xs attribute is 0.

That code that reads:

```
    if memattrs.memtype == MemType_Device then
        ...
        if memattrs.device == DeviceType_nGnRnE then
            ...
        else // DeviceType_GRE
            ...
    else
        if memattrs.outer.attrs == MemAttr_WT then
            ...
```

Is updated to read:

```
    if memattrs.memtype == MemType_Device then
        ...
        if memattrs.device == DeviceType_nGnRnE then
            ...
        else // DeviceType_GRE
            ...
        result<0> = NOT memattrs.xs;
    else
        if memattrs.xs == '0' then
            if (memattrs.outer.attrs == MemAttr_WT && memattrs.inner.attrs ==
MemAttr_WT &&
                    !memattrs.outer.transient && memattrs.outer.hints == MemHint_RA)
then
                return '10100000';
            elsif memattrs.outer.attrs == MemAttr_NC && memattrs.inner.attrs ==
MemAttr_NC then
                return '01000000';
        if memattrs.outer.attrs == MemAttr_WT then
            ...
```

In the same section, the function S2CombineS1MemAttrs() incorrectly combines the stage 1 and stage 2 xs attributes when stage 2 is in AArch32 Execution state.

The code that reads:

```
MemoryAttributes S2CombineS1MemAttrs(MemoryAttributes s1_mematttrs,
                                     MemoryAttributes s2_mematttrs)
    ...
    if (mematttrs.memtype == MemType_Normal &&
            mematttrs.inner.attrs == MemAttr_WB &&
            mematttrs.outer.attrs == MemAttr_WB) then
        mematttrs.xs = '0';
    else
        mematttrs.xs = s2_mematttrs.xs AND s1_mematttrs.xs;
```

Is updated to read:

```
MemoryAttributes S2CombineS1MemAttrs(MemoryAttributes s1_mematttrs, MemoryAttributes
 s2_mematttrs,
                                     boolean s2aarch64)
    ...
    if (mematttrs.memtype == MemType_Normal &&
            mematttrs.inner.attrs == MemAttr_WB &&
            mematttrs.outer.attrs == MemAttr_WB) then
        mematttrs.xs = '0';
    elsif s2aarch64 then
        mematttrs.xs = s2_mematttrs.xs AND s1_mematttrs.xs;
    else
        mematttrs.xs = s1_mematttrs.xs;
```

In J1.2.4 (aarch32/translation) the function AArch32.S1DisabledOutput() does not initialize the value of xs.

The code that reads:

```
AArch32.S1DisabledOutput(...)
    ...
    if default_cacheable == '1' then
        ...
    elsif accdesc.acctype == AccessType_IFETCH then
        ...
    else
        ...
    ...
```

Is updated to read:

```
AArch32.S1DisabledOutput(...)
    ...
    if default_cacheable == '1' then
        ...
        mematttrs.xs           = '0';
    elsif accdesc.acctype == AccessType_IFETCH then
        ...
        mematttrs.xs           = '1';
    else
        ...
        mematttrs.xs           = '1';
    ...
```

2.112 D20389

In section D8.4.6 (Hardware management of the dirty state), in the subsection 'Implications of enabling the dirty state management mechanism', the rule I_{NSYYW} that reads:

> For stage 1 translations, if the corresponding SCTLR_ELx.WXN is 1, then all of the following apply:
>
> - For a translation regime that supports a single privilege level, translations using a writeable-clean descriptor are treated as execute-never.
> - For a translation regime that supports two privilege levels, translations using a privileged writable-clean descriptor are treated as privileged execute-never.
> - For a translation regime that supports two privilege levels, translations using a writeable-clean descriptor are treated as unprivileged execute never.

is updated to read:

> For stage 1 translations, if the corresponding SCTLR_ELx.WXN is 1, then all of the following apply:
>
> - For a translation regime that supports a single privilege level, translations using a writeable-clean descriptor are treated as execute-never.
> - For a translation regime that supports two privilege levels, translations using a privileged writable-clean descriptor are treated as privileged execute-never.
> - For a translation regime that supports two privilege levels, translations using an unprivileged writeable-clean descriptor are treated as unprivileged execute-never.

2.113 D20398

In section D17.2.144 (TTBR0_EL1, Translation Table Base Register 0 (EL1)), in the 'BADDR[47:1], bits [47:1]' field, the text that reads:

> Address bit x is the minimum address bit required to align the translation table to the size of the table. The smallest permitted value of x is 6. The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR_EL1.T0SZ, the translation stage, and the translation granule size.
>
> Note: A translation table is required to be aligned to the size of the table. If a table contains fewer than eight entries, it must be aligned on a 64 byte address boundary.

is updated to read:

> Address bit x is the minimum address bit required to align the translation table to the size of the table. The AArch64 Virtual Memory System Architecture chapter describes how x is calculated based on the value of TCR_EL1.T0SZ, the translation stage, and the translation granule size.

> Note: If an OA size of more than 48 bits is in use, and the translation table has fewer than eight entries, the table must be aligned to 64 bytes. Otherwise the translation table must be aligned to the size of the table.

Within the same field description, the text that reads:

> If FEAT_LPA is implemented and the value of TCR_EL1.IPS is `0b110`, then:
>
> - Bits A[51:48] of the stage 1 translation table base address bits are in register bits[5:2].
> - Register bit[1] is **RES0**.
> - When x>6, register bits[(x-1):6] are **RES0**.

is updated to read:

> If FEAT_LPA is implemented and the value of TCR_EL1.IPS is `0b110`, then:
>
> Bits A[51:48] of the stage 1 translation table base address bits are in register bits[5:2]. Register bit[1] is **RES0**. The smallest value of x is 6. When x>6, register bits[(x-1):6] are **RES0**.

Similar changes are made in the following sections:

- D17.2.145 (TTBR0_EL2, Translation Table Base Register 0 (EL2)).
- D17.2.146 (TTBR0_EL3, Translation Table Base Register 0 (EL3)).
- D17.2.147 (TTBR1_EL1, Translation Table Base Register 1 (EL1)).
- D17.2.148 (TTBR1_EL2, Translation Table Base Register 1 (EL2)).

In section D8.2.5 (Translation table and translation table lookup properties), rule R_{KBLCR} that reads:

> A translation table is required to be aligned to one of the following:
>
> - If the translation table has eight or more entries, then it is aligned to the translation table size.
> - If the translation table has fewer than eight entries, then it is aligned to 64 bytes.

is updated to read:

> A translation table is required to be aligned to one of the following:
>
> - If the translation table has fewer than eight entries, and an OA size of greater than 48 bits is in use, then the table is aligned to 64 bytes.
> - Otherwise, the translation table is aligned to the size of that translation table.

2.114 D20433

In section D1.3.5 (Synchronous exception types), in the subsection 'Prioritization of Synchronous exceptions taken to AArch64 state', the table rows in I_{ZFGJP} that read:

> 7 Instruction Abort exceptions, including exceptions generated by a Translation Table Walk not prioritized as 29. See MMU fault prioritization from a single address translation stage on page D8-5182.

> 31 Any Data Abort Exception not defined by Priority 33. It is **IMPLEMENTATION DEFINED** whether a Data Abort Exceptions generated by synchronous External Aborts are prioritized here or as Priority 33. See MMU fault prioritization from a single address translation stage on page D8-5182.

> 33 Any of the following Data Abort Exceptions:
>
> - An External abort that was not generated by a translation table walk and therefore not prioritized as 7.
> - An External abort that was not generated by a translation table entry update.
> - If FEAT_MTE2 is implemented, any Tag Check Fault.
>
> It is **IMPLEMENTATION DEFINED** whether synchronous External Aborts are prioritized here or as Priority 31. See External aborts on page D7-5064 and PE handling of Tag Check Fault on page D9-5219.

are updated to read:

> 7 Instruction Abort exceptions, including exceptions generated by an MMU fault for the translation of an instruction fetch. See MMU fault prioritization from a single address translation stage on page D8-5182.

> 31 Any Data Abort Exception not defined by Priority 33. It is **IMPLEMENTATION DEFINED** whether Data Abort Exceptions generated by synchronous External Aborts on explicit accesses are prioritized here or as Priority 33. See MMU fault prioritization from a single address translation stage on page D8-5182.

> 33 Data Abort Exceptions generated for any of the following reasons:
>
> - An External abort that was not generated on a translation table walk and not generated on a translation table entry update.
> - If FEAT_MTE2 is implemented, any Tag Check Fault.
>
> It is **IMPLEMENTATION DEFINED** whether synchronous External Aborts are prioritized here or as Priority 31. See External aborts on page D7-5064 and PE handling of Tag Check Fault on page D9-5219.

2.115 C1186: SME

In section D17.2.70 (ID_AA64ZFR0_EL1, SVE Feature ID register 0), the following text:

> Irrespective of the value of this field, when the PE is in Streaming SVE mode and it is not known whether FEAT_SME_FA64 is implemented and enabled at the current Exception level, software should not attempt to execute the instructions described by non-zero values of this field.

is added to the descriptions of the following fields:

- F64MM, bits [59:56].
- F32MM, bits [55:52].
- SM4, bits [43:40].
- SHA3, bits [35:32].
- BitPerm, bits [19:16].
- AES, bits [7:4].

The following text is added to the description of I8MM, bits [47:44]:

> Irrespective of the value of this field, when the PE is in Streaming SVE mode and it is not known whether FEAT_SME_FA64 is implemented and enabled at the current Exception level, software should not attempt to execute the SVE instructions SMMLA, UMMLA, and USMMLA.

The following text is added to the description of BF16, bits [23:20]:

> Irrespective of the value of this field, when the PE is in Streaming SVE mode and it is not known whether FEAT_SME_FA64 is implemented and enabled at the current Exception level, software should not attempt to execute the SVE instruction BFMMLA.

2.116 D1386: SME

In the following sections:

- C7.2.15 BFDOT (by element).
- C7.2.16 BFDOT (vector).
- C7.2.19 BFMMLA.
- C8.2.35 BFDOT (indexed).
- C8.2.36 BFDOT (vectors).
- C8.2.41 BFMMLA.

The text that reads:

> If FEAT_EBF16 is implemented and FPCR.EBF is 1, then this instruction:

> - Performs a fused sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the specified pair of elements in the second source vector. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
> - Generates only the default NaN, as if FPCR.DN is 1.
> - Follows all other floating-point behaviors that apply to single-precision arithmetic, as controlled by the effective value of the FPCR in the current execution mode, and captured in the FPSR.

is corrected to read:

> If FEAT_EBF16 is implemented and FPCR.EBF is 1, then this instruction:
>
> - Performs a fused sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the specified pair of elements in the second source vector. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
> - Generates only the default NaN, as if FPCR.DN is 1.
> - Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
> - Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
> - Follows all other floating-point behaviors that apply to single-precision arithmetic, as governed by FPCR.RMode, FPCR.FZ, FPCR.AH, and FPCR.FIZ controls in the current execution mode.

2.117 D494: SVE2

In section C8.2 (Alphabetical list of SVE instructions), the following text is added to the 'Operation Information' subsection of all predicated SVE load/store (vector) instructions, except for the first-fault (FF) and non-fault (NF) loads:

> If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored when its governing predicate register contains the same value for each execution.

The following text is added to the 'Operational Information' subsection of all unpredicated SVE load/store (vector and predicate) instructions:

> If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

In section C8.2.82 (CNT), the following text is added to the 'Operational Information' subsection:

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

2.118 D504: SVE2

In section C8.2 (Alphabetical list of SVE instructions), in the descriptions of the 'shift by immediate' instructions, the description of the <const> assembler symbol that reads:

> Is the immediate shift amount, in the range ..., encoded in "tsz:imm3".

is corrected to read:

> Is the immediate shift amount, in the range ..., encoded in "tszh:tszl:imm3".

2.119 C215: SVE

In section A1.4 (Supported data types), the text that reads:

> - An SVE scalable vector register has an **IMPLEMENTATION DEFINED** width that is a multiple of 128 bits, up to a maximum of 2048 bits.

is changed to read:

> - An SVE scalable vector register has an **IMPLEMENTATION DEFINED** width that is a power of two, from a minimum of 128 bits up to a maximum of 2048 bits.

Within the same section, the text that reads:

> - An SVE predicate vector register has an **IMPLEMENTATION DEFINED** width that is a multiple of 16 bits, up to a maximum of 256 bits.

is changed to read:

> - An SVE predicate vector register has an **IMPLEMENTATION DEFINED** width that is a power of two, from a minimum of 16 bits up to a maximum of 256 bits.

In section A1.4.2 (SVE vector format), in the subsection 'SVE configurable vector length', the rules R_{RYQYY} and I_{CPZLW} are deleted.

In section B1.2.2 (SVE vector registers), the rule R_{KCWQB} is deleted.

In section B1.2.3 (SVE predicate registers), the rule R_{NKRJV} that reads:

> The size of an SVE predicate register is an **IMPLEMENTATION DEFINED** multiple of 16 bits.

is changed to read:

> The size of an SVE predicate register is an **IMPLEMENTATION DEFINED** power of two.

Within the same section, rules R_{MFPXG} and R_{BBTXX} are deleted.

In section D13.6.5 (Additional information for each profiled Scalable Vector Extension operation), in the definition of 'Effective vector length', the Note that reads:

> The Accessible vector length is always quantized into multiples of 128 bits. However, the Sampled operation vector can be any size down to the element size of the operation.

is changed to read:

> The Accessible vector length is always quantized into a power of two. However, the Sampled operation vector can be any size down to the element size of the operation.

Similarly, in section D14.2.7 (Operation Type packet), subsection 'Operation Type packet payload (Other)', the text in the description of 'EVL, byte 0 bits [6:4], when SVE operation' that reads:

> The accessible vector length is always quantized into multiples of 128 bits. However, the effective vector length can be any size down to the element size of the operation.

is changed to read:

> The Accessible vector length is always quantized into a power of two. However, the Effective vector length can be any size down to the element size of the operation.

Within the same section, the text that reads:

> If the effective vector length is not a power of two, or is less than 32 bits, the value is rounded up before it is encoded in this field.

is changed to read:

> If the Effective vector length is less than 32 bits, the value is rounded up before it is encoded in this field.

The same changes are made in the subsection 'Operation Type packet payload (load/store)', in the description of 'EVL, byte 0 bits [6:4], when SVE load/store'.

In section D17.2.159 (ZCR_EL1, SVE Control Register (EL1)), in the LEN, bits [3:0] field, the text that reads:

> The Non-streaming SVE vector length can be any multiple of 128 bits, from 128 bits to 2048 bits inclusive.

is changed to read:

> The Non-streaming SVE vector length can be any power of two from 128 bits to 2048 bits inclusive.

The same change is made in the following sections:

- D17.2.160 (ZCR_EL2, SVE Control Register (EL2)).
- D17.2.161 (ZCR_EL3, SVE Control Register (EL3)).

In section J1.1.3 (aarch64/functions), the code within the function ImplementedSVEVectorLength() that reads:

```
// Reduce SVE vector length to a supported value (e.g. power of two)
integer ImplementedSVEVectorLength(integer nbits_in)
    integer nbits = Min(nbits_in, MaxImplementedVL());
    assert 128 <= nbits && nbits <= 2048 && Align(nbits, 128) == nbits;
    while nbits > 128 do
        if IsPow2(nbits) || SupportedNonPowerTwoVL(nbits) then return nbits;
        nbits = nbits - 128;
    return nbits;
```

is changed to read:

```
// Reduce SVE vector length to a supported value (power of two)
integer ImplementedSVEVectorLength(integer nbits_in)
    integer maxbits = MaxImplementedVL();
    assert 128 <= maxbits && maxbits <= 2048 && IsPow2(maxbits);
    integer nbits = Min(nbits_in, maxbits);
    assert 128 <= nbits && nbits <= 2048 && Align(nbits, 128) == nbits;
    while nbits > 128 do
        if IsPow2(nbits) then return nbits;
        nbits = nbits - 128;
    return nbits;
```

Within the same section, the function SupportedNonPowerTwoVL() is removed.

In the Glossary, the definition of ‘Predicate register’ that reads:

> An SVE predicate register, P0-P15, having a length that is a multiple of 16 bits, in the range 16 to 256, inclusive.

is changed to read:

> An SVE predicate register, P0-P15, having a length that is a power of two, in the range 16 bits to 256 bits, inclusive.

Also in the Glossary, the definition of ‘Scalable vector register’ that reads:

> An SVE vector register, Z0-Z31, having a length that is a multiple of 128 bits, in the range 128 bits to 2048 bits, inclusive.

is changed to read:

> An SVE vector register, Z0-Z31, having a length that is a power of two, in the range 128 bits to 2048 bits, inclusive.

2.120 C225: SVE

In section D7.2.1 (Virtual address space overflow), the following text is added:

> The **UNKNOWN** virtual address behavior also applies to the set of bytes addressed by SVE and SME predicated, contiguous loads and stores that cross the `0xFFFF_FFFF_FFFF_FFFF` boundary, even if all of the virtual addresses below the boundary correspond to Inactive elements. Conversely, for SVE gather loads and scatter stores, the **UNKNOWN** address behavior applies only to accesses corresponding to an individual Active element that crosses the boundary.

2.121 C256: SVE

In section H2.4.2 (Executing instructions in Debug state), in the subsection 'A64 instructions that are unchanged in Debug state', the list that reads:

> SVE instructions
>
> When FEAT_SVE is implemented, these instructions are:
> - CPY.
> - DUP (scalar).
> - EXT.
> - INSR (scalar).
> - PTRUE with ALL constraint and byte element size.
> - RDFFR (unpredicated).
> - RDVL.
> - WRFFR.

is changed to read:

> SVE instructions
>
> When FEAT_SVE is implemented, these instructions are:
> - CPY.
> - DUP (scalar).
> - EXT, destructive variant.
> - INSR (scalar).

- PTRUE with ALL constraint and byte element size.
- RDFFR (unpredicated).
- RDVL.
- WRFFR.

2.122 C279: SVE

In section B1.2.4 (FFR, First Fault Register), rule R_{WZJVT} that reads:

> Bits in the FFR are indirectly set to 0 as a result of a suppressed access or fault generated in response to an Active element of an SVE First-fault or Non-fault vector load.

is clarified to read:

> Bits in the FFR are indirectly set to 0 as a result of a suppressed access or suppressed fault corresponding to an Active element of an SVE First-fault or Non-fault vector load.

2.123 C301: SVE

In section D17.2.131 (TCR_EL1, Translation Control Register (EL1)), in the NFD0, bit [53] field, the text that reads:

> Non-fault translation table walk disable for stage 1 translations using TTBR0_EL1.
>
> This bit controls whether to perform a stage 1 translation table walk in response to a non-fault unprivileged access for a virtual address that is translated using TTBR0_EL1.

is changed to read:

> Non-fault translation timing disable for stage 1 translations using TTBR0_EL1.
>
> This bit controls how a TLB miss is reported in response to a non-fault unprivileged access for a virtual address that is translated using TTBR0_EL1.

The following text is deleted:

> For more information, see 'The Scalable Vector Extension (SVE)'.

The value descriptions that read:

> `0b0` Does not disable stage 1 translation table walks using TTBR0_EL1.
>
> `0b1` A TLB miss on a virtual address that is translated using TTBR0_EL1 due to the specified access types causes the access to fail without taking an exception. No stage 1 translation table walk is performed.

are changed to read:

> `0b0` Does not affect the handling of a TLB miss on accesses translated using TTBR0_EL1.
>
> `0b1` A TLB miss on a virtual address that is translated using TTBR0_EL1 due to the specified access types causes the access to fail without taking an exception. The failure should take the same amount of time to be handled as a Permission fault on a TLB entry that is present in the TLB, to mitigate attacks that use fault timing.

The equivalent changes are made to the NFD1, bit [54] field, and to the NFD0, bit [53] and NFD1, bit [54] fields in section D17.2.132 (TCR_EL2, Translation Control Register (EL2)).

2.124 D302: SVE

In section C1.2.6 (Register names), in the subsection 'SIMD vector register list', the text that reads:

> Where an instruction operates on multiple SIMD and floating-point registers, for example vector load/store structure and table lookup operations, the registers are specified as a list enclosed by curly braces. This list consists of either a sequence of registers separated by commas, or a register range separated by a hyphen. The registers must be numbered in increasing order, modulo 32, in increments of one. The hyphenated form is preferred for disassembly if there are more than two registers in the list and the register number are increasing.

is updated to read:

> Where an instruction operates on multiple SIMD&FP or SVE vector registers, for example vector load/store structure and table lookup operations, the registers are specified as a list enclosed by curly braces. This list consists of either a sequence of registers separated by commas, or a register range separated by a hyphen. The registers must be numbered in increasing order, modulo 32, in increments of one. The hyphenated form is preferred for disassembly if there are more than two registers in the list and the register numbers are increasing.

Similar updates are made throughout section C1.2 (Structure of the A64 assembler language) to account for the SVE assembler syntax.

2.125 C313: SVE

In section A1.5.4 (Flushing denormalized numbers to zero), in the subsection 'Flushing denormalized outputs to zero', the text that reads:

> If FPCR.FZ16 == 1, for floating-point instructions other than FABS, FNEG, FMAX*, and FMIN*, if the instruction processes half-precision numbers, flushing denormalized output numbers to zero can be controlled as follows:

is clarified to read:

If FPCR.FZ16 == 1, for floating-point instructions other than FABS, FNEG, FMAX, FMAXP, FMAXV, FMIN, FMINP, and FMINV, if the instruction processes half-precision numbers, flushing denormalized output numbers to zero can be controlled as follows:

In the same section, the bullet that reads:

- For FABS, FNEG, FMAX*, and FMIN*, denormalized output operands are not flushed to zero.

is clarified to read:

For FABS, FNEG, FMAX, FMAXP, FMAXV, FMIN, FMINP, and FMINV, denormalized output operands are not flushed to zero.

2.126 C314: SVE

In sections C8.2.143 (FMAX (vectors)) and C8.2.151 (FMIN (vectors)), the following text is removed:

If either element value is NaN then the result is NaN.

In the following sections:

- C7.2.101 (FMAX (vector)).
- C7.2.102 (FMAX (scalar)).
- C7.2.108 (FMAXP (scalar)).
- C7.2.109 (FMAXP (vector)).
- C7.2.110 (FMAXV).
- C7.2.111 (FMIN (vector)).
- C7.2.112 (FMIN (scalar)).
- C7.2.118 (FMINP (scalar)).
- C7.2.119 (FMINP (vector)).
- C7.2.120 (FMINV).

The following text is added:

When FPCR.AH == 0, the behavior is as follows:

- Negative zero compares less than positive zero.
- When FPCR.DN is 0, if either input is a NaN, the result is a Quiet NaN.
- When FPCR.DN is 1, if either input is a NaN, the result is Default NaN.

When FPCR.AH == 1, the behavior is as follows:

- If both inputs are zeros, regardless of the sign of either zero, the result is the second input.

- If either input is a NaN, regardless of the value of FPCR.DN, the result is the second input.

In sections C8.2.142 (FMAX (immediate)) and C8.2.150 (FMIN (immediate)), the text that reads:

> If the element value is NaN then the result is NaN.

is updated to read:

> When FPCR.AH == 0, the behavior is as follows:
> - Negative zero compares less than positive zero.
> - When FPCR.DN is 0, if the input is a NaN, the result is a Quiet NaN.
> - When FPCR.DN is 1, if the input is a NaN, the result is Default NaN.
>
> When FPCR.AH == 1, the behavior is as follows:
> - If both the input and the immediate are zeros, regardless of the sign of input zero, the result is the immediate.
> - If the input is a NaN, regardless of the value of FPCR.DN, the result is the immediate.

In the following sections:

- C7.2.103 (FMAXNM (vector)).
- C7.2.104 (FMAXNM (scalar)).
- C7.2.106 (FMAXNMP (vector)).
- C7.2.107 (FMAXNMV).
- C7.2.113 (FMINNM (vector)).
- C7.2.114 (FMINNM (scalar)).
- C7.2.116 (FMINNMP (vector)).
- C7.2.117 (FMINNMV).

The text that reads:

> NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result that is placed in the vector is the numerical value, otherwise the result is identical to ...

is updated to read:

> Regardless of the value of FPCR.AH, the behavior is as follows:
> - Negative zero compares less than positive zero.
> - If one input is numeric and the other is a NaN, the result is the numeric value.
> - When FPCR.DN == 0, if either input is a signaling NaN or if both inputs are NaNs, the result is a Quiet NaN.
> - When FPCR.DN == 1, if either input is a signaling NaN or if both inputs are NaNs, the result is Default NaN.

This updated text also replaces the following text in sections C8.2.145 (FMAXNM (vectors)), C8.2.146 (FMAXNMP), C8.2.153 (FMINNM (vectors)), and C8.2.154 (FMINNMP):

> If one element value is numeric and the other is a quiet NaN, then the result is the numeric value.

In sections C8.2.144 (FMAXNM (immediate)) and C8.2.152 (FMINNM (immediate)), the text that reads:

> If the element value is a quiet NaN, then the result is the immediate.

is updated to read:

> Regardless of the value of FPCR.AH, the behavior is as follows:
> - Negative zero compares less than positive zero.
> - If the input is a Quiet NaN, the result is the immediate value.
> - When FPCR.DN == 0, if the input is a signaling NaN, the result is a Quiet NaN.
> - When FPCR.DN == 1, if the input is a signaling NaN, the result is Default NaN.

2.127 C318: SVE

In section D17.2.131 (TCR_EL1, Translation Control Register (EL1)), in the 'NFD0, bit [53]' field, the `0b1` value description that reads:

> `0b1` A TLB miss on a virtual address that is translated using TTBR0_EL1 due to the specified access types causes the access to fail without taking an exception. The failure should take the same amount of time to be handled as a Permission fault on a TLB entry that is present in the TLB, to mitigate attacks that use fault timing.

is updated to read:

> `0b1` A TLB miss on a virtual address that is translated using TTBR0_EL1 due to the specified access types causes the access to fail without taking an exception. The amount of time that the failure takes to be handled should not predictively leak whether it was caused by a TLB miss or a Permission fault, to mitigate attacks that use fault timing.

Equivalent changes are made to the 'NFD1, bit [54]' field description, and to the 'NFD0, bit [53]' and 'NFD1, bit [54]' field descriptions in section D17.2.132 (TCR_EL2, Translation Control Register (EL2)).

2.128 C1206: Armv9 Debug

In section D4.4.5 (Exceptions to Exception element encoding), Table D4-19 'Exception mapping for exceptions taken to AArch64 state' within the rule R_{GZQKS} is updated to indicate that:

- Exceptions taken due to HFGITR_EL2.SVC_EL0 and HFGITR_EL2.SVC_EL1 are traced or recorded using the 'Trap' exception type.
- Exceptions taken due to HCR_EL2.TSC are traced or recorded using the 'Trap' exception type.

2.129 D1383: Armv9 Debug

In section D4.5.9 (Element Generation), in the subsections 'Exception element' and 'Target Address element', statements are added to recommend that when a branch occurs to an invalid address and the resultant exception is taken from that address, the addresses reported by the Target Address element and the Exception element are the same value.

2.130 D1461: Armv9 Debug

In section D4.6.12 (External Outputs), the statement I_{BZHDF} that reads:

> The ETE architecture supports between one and four External Outputs. The number of outputs that a trace unit has is **IMPLEMENTATION DEFINED**, but at least one output is always implemented.

is updated to read:

> The ETE architecture supports between zero and four External Outputs. The number of outputs that a trace unit has is **IMPLEMENTATION DEFINED**, and Arm recommends that at least one output is implemented.

2.131 D1466: Armv9 Debug

In section D11.11.3 (Common event numbers), in the subsection 'Common microarchitectural events', the description for each CTI_TRIGOUT<n> event, where <n> is in the range 4 to 7, that reads:

> This event must be implemented if FEAT_ETE is implemented.

is updated to read:

> This event must be implemented if FEAT_ETE is implemented and TRCIDR5.NUMEXTINSEL > (n - 4).

2.132 D1493: Armv9 Debug

In section D4.5.3 (Trace unit behavior while the PE is in Debug state), rule R_{DPKSC} that reads:

> While the PE is in Debug state, the trace unit does not trace instructions that are executed.

is updated to read:

> While the PE is in Debug state, the trace unit:
> - Does not trace instructions that are executed.
> - Does not trace the effects of instructions that are executed.
> - Does not trace Exceptional occurrences.

Additionally, in section D4.5.8 (Filtering trace generation), in the subsection 'Rules for tracing Exceptional occurrences', rule R_{DPMBQ} that reads:

> When an Exceptional occurrence occurs and TRCRSR.TA is `0b1`, the Exceptional occurrence is traced.

is updated to read:

> When an Exceptional occurrence occurs and the PE is not in Debug state and TRCRSR.TA is `0b1`, the Exceptional occurrence is traced.

2.133 D1023: RME

A new function HaveSecureState() is added in section J1.3.3 (shared/functions):

```
boolean HaveSecureState()
    if !HaveEL(EL3) then
        return SecureOnlyImplementation();
    if HaveRME() && !HaveSecureEL2Ext() then
        return FALSE;
    return TRUE;
```

New functions EffectiveSCR_EL3_NS() and EffectiveSCR_EL3_NSE() are added in section J1.3.3 (shared/functions):

```
bit EffectiveSCR_EL3_NS()
    if !HaveSecureState() then
        return '1';
    elsif !HaveEL(EL3) then
        return '0';
    else
        return SCR_EL3.NS;
bit EffectiveSCR_EL3_NSE()
    return if !HaveRME() then '0' else SCR_EL3.NSE;
```

The function CheckValidStateMatch() in section J1.3.1 (shared/debug) is changed from:

```
(Constraint, bits(2), bit, bit, bits(2)) CheckValidStateMatch(bits(2) ssc_in, bit
 ssce_in,
                                                               bit hmc_in, bits(2)
 pxc_in,
                                                               boolean isbreakpnt)
....
    // Values that are not allocated in any architecture version
    case hmc:ssce:ssc:pxc of
        when '0 0 11 10' reserved = TRUE;
        when '1 0 00 x0' reserved = TRUE;
```

to:

```
(Constraint, bits(2), bit, bit, bits(2)) CheckValidStateMatch(bits(2) ssc_in, bit
 ssce_in,
                                                               bit hmc_in, bits(2)
 pxc_in,
                                                               boolean isbreakpnt)
....
    // Values that are not allocated in any architecture version
    case hmc:ssce:ssc:pxc of
        when '0 0 11 10' reserved = TRUE;
        when '0 0 1x xx' reserved = !HaveSecureState();
        when '1 0 00 x0' reserved = TRUE;
```

The corresponding code change for the SCR_EL3.SIF Effective value is made as part of section J1.2.4 (aarch32/translation) for AArch32.S1TTWParamsEL10(), and section J1.1.5 (aarch64/translation) for AArch64.S1TTWParamsEL3(), AArch64.S1TTWParamsEL2(), AArch64.S1TTWParamsEL20(), and AArch64.S1TTWParamsEL10().

In section J1.1.1 (aarch64/debug), the function ProfilingBufferOwner() is changed from:

```
(SecurityState, bits(2)) ProfilingBufferOwner()
    SecurityState owning_ss;
    if HaveEL(EL3) then
        bits(3) state_bits;
        if HaveRME() then
            state_bits = MDCR_EL3.<NSPBE,NSPB>;
            if state_bits IN {'10x'} then
....
```

to:

```
(SecurityState, bits(2)) ProfilingBufferOwner()
    SecurityState owning_ss;
    if HaveEL(EL3) then
        bits(3) state_bits;
        if HaveRME() then
            state_bits = MDCR_EL3.<NSPBE,NSPB>;
            if (state_bits IN {'10x'} ||
                (!HaveSecureEL2Ext() && state_bits IN {'00x'})) then
....
```

To account for the Effective value of the SCR_EL3.NS field, the function SecurityStateAtEL() in section J1.3.3 (shared/functions) is changed from:

```
SecurityState SecurityStateAtEL(bits(2) EL)
    if HaveRME() then
        if EL == EL3 then return SS_Root;
        case SCR_EL3.<NSE, NS> of
            when '00' return SS_Secure;
....
```

to:

```
SecurityState SecurityStateAtEL(bits(2) EL)
    if HaveRME() then
        if EL == EL3 then return SS_Root;
        effective_nse_ns = SCR_EL3.NSE : EffectiveSCR_EL3_NS();
        case effective_nse_ns of
            when '00' if HaveSecureEL2Ext() then return SS_Secure; else
 Unreachable();
....
```

Similar Effective value checks of SCR_EL3.NS (**RES1** in case Secure state is not implemented) and SCR_EL3.NSE are added in functions ELFromM32(), ELFromSPSR(), AArch64.AT(), ProfilingBufferEnabled() using the new functions EffectiveSCR_EL3_NS() and EffectiveSCR_EL3_NSE().

In section J1.3.1 (shared/debug), the function ExternalRootInvasiveDebugEnabled() is changed from:

```
boolean ExternalRootInvasiveDebugEnabled()
    if !HaveRME() then return FALSE;
    return (ExternalInvasiveDebugEnabled() &&
            ExternalSecureInvasiveDebugEnabled() &&
....
```

to:

```
boolean ExternalRootInvasiveDebugEnabled()
    if !HaveRME() then return FALSE;
    return (ExternalInvasiveDebugEnabled() &&
            (!HaveSecureEL2Ext() || ExternalSecureInvasiveDebugEnabled()) &&
....
```

The function SelfHostedTraceEnabled() in section J1.3.4 (shared/trace) is refactored to account for the Effective value of MDCR_EL3.STE.

The function TraceBufferOwner() in section J1.3.4 (shared/trace) is changed to take the Effective value of MDCR_EL3.<NSTBE,NSTB>. The function is changed from:

```
(SecurityState, bits(2)) TraceBufferOwner()
    assert HaveTraceBufferExtension() && SelfHostedTraceEnabled();
    SecurityState owning_ss;
    if HaveEL(EL3) then
        bits(3) state_bits;
```

```
        if HaveRME() then
            state_bits = MDCR_EL3.<NSTBE,NSTB>;
            if state_bits IN {'10x'} then
....
```

to:

```
(SecurityState, bits(2)) TraceBufferOwner()
    assert HaveTraceBufferExtension() && SelfHostedTraceEnabled();
    SecurityState owning_ss;
    if HaveEL(EL3) then
        bits(3) state_bits;
        if HaveRME() then
            state_bits = MDCR_EL3.<NSTBE,NSTB>;
            if (state_bits IN {'10x'} ||
                (!HaveSecureEL2Ext() && state_bits IN {'00x'})) then
....
```

The function GPIValid() in section J1.3.5 (shared/translation) is updated to account for the definition of GPI encoding `0b1000`. The function is changed from:

```
boolean GPIValid(bits(4) gpi)
    return gpi IN {GPT_NoAccess,
                   GPT_Secure,
                   GPT_NonSecure,
....
```

to:

```
boolean GPIValid(bits(4) gpi)
    if gpi == GPT_Secure then
        return HaveSecureEL2Ext();
    return gpi IN {GPT_NoAccess,
                   GPT_NonSecure,
....
```

Similarly the function GPICheck() in section J1.3.5 (shared/translation) is changed from:

```
boolean GPICheck(PASpace paspace, bits(4) gpi)
    case gpi of
        when GPT_NoAccess  return FALSE;
        when GPT_Secure    return paspace == PAS_Secure;
....
```

to:

```
boolean GPIValid(bits(4) gpi)
 boolean GPICheck(PASpace paspace, bits(4) gpi)
    case gpi of
        when GPT_NoAccess  return FALSE;
        when GPT_Secure    assert HaveSecureEL2Ext();return paspace == PAS_Secure;
....
```

The function AArch64.S1NextWalkStateLeaf() in section J1.1.5 (aarch64/translation) is changed from:

```
TTWState AArch64.S1NextWalkStateLeaf(TTWState currentstate,THEARG( boolean s2fs1mro)
 Regime regime,
                                      SecurityState ss, S1TTWParams walkparams,
 bits(N) descriptor)
....
        case <nse,ns> of
            when '00' baseaddress.paspace = PAS_Secure;
            when '01' baseaddress.paspace = PAS_NonSecure;
            when '10' baseaddress.paspace = PAS_Root;
            when '11' baseaddress.paspace = PAS_Realm;
....
```

to:

```
TTWState AArch64.S1NextWalkStateLeaf(TTWState currentstate,THEARG( boolean s2fs1mro)
 Regime regime,
                                      SecurityState ss, S1TTWParams walkparams,
 bits(N) descriptor)
....
        case <nse,ns> of
            when '00'
                baseaddress.paspace = if HaveSecureEL2Ext() then PAS_Secure else
 PAS_NonSecure;
            when '01'
                baseaddress.paspace = PAS_NonSecure;
            when '10'
                baseaddress.paspace = PAS_Root;
            when '11'
                baseaddress.paspace = PAS_Realm;
```

2.134 C1277: RME

In section D17.2.133 (TCR_EL3, Translation Control Register (EL3)), in the 'DS' field description, the following text:

> Otherwise:
>
> Reserved, **RES0**.

Is clarified to read:

> Otherwise:
>
> Reserved, **RES0**, and the Effective value of this bit is `0b0`.

The equivalent change is made in sections D17.2.132 (TCR_EL2, Translation Control Register (EL2)), and D17.2.131 (TCR_EL1, Translation Control Register (EL1)).

In section D17.2.117 (SCR_EL3, Secure Configuration Register), in the 'NSE' field description, the following text:

> Otherwise:
>
> Reserved, **RES0**.

Is clarified to read:

> Otherwise:
>
> Reserved, **RES0**, and the Effective value of this bit is `0b0`.

2.135 C1283: RME

In section C5.4.1 (AT S12E0R, Address Translate Stages 1 and 2 EL0 Read), in the 'Purpose' section, the following text:

> When EL2 is implemented and enabled in the Security state described by the current value of SCR_EL3.NS.

Is clarified to read:

> When EL2 is implemented and enabled in the Security state described by the current Effective value of SCR_EL3.{NSE, NS}.

The equivalent change is also made in the following sections:

- C5.4.2 (AT S12E0W).
- C5.4.3 (AT S12E1R).
- C5.4.4 (AT S12E1W).
- C5.4.5 (AT S1E0R).
- C5.4.6 (AT S1E0W).
- C5.4.7 (AT S1E1R).
- C5.4.8 (AT S1E1RP).
- C5.4.9 (AT S1E1W).
- C5.4.10 (AT S1E1WP).

2.136 D1284: RME

In section D17.2.38 (ESR_EL2, Exception Syndrome Register (EL2)), the following are removed:

- In the 'EC' field description, EC value `0b011110` and its associated text is removed.
- In the 'ISS' field description, the 'ISS encoding for an exception from a Granule Protection Check' subsection is removed.

The equivalent changes are also made in section D17.2.37 (ESR_EL1, Exception Syndrome Register (EL1)).

In section D17.2.39 (ESR_EL3, Exception Syndrome Register (EL3)), in the 'EC' field description, the following text for EC value `0b011110`:

> Exception from a Granule Protection Check. See ISS encoding for an exception from a Granule Protection Check.

is corrected to read:

> Granule Protection Check exception. See ISS encoding for a Granule Protection Check exception.

Correspondingly, in the 'ISS' field description, the subsection titled 'ISS encoding for an exception from a Granule Protection Check' is corrected to 'ISS encoding for a Granule Protection Check exception'.