Arm® Architecture Reference Manual for A-profile architecture

Known issues in Issue I.a

Non-Confidential

Issue 01
102105_I.a_01_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

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` <u>`underline`</u>	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 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 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.

2.4 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'.

2.5 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.

2.6 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 storesArmv8 Pseudocode
                     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
)
```

2.7 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();
```

2.8 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).

2.9 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'.

2.10 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);
```

2.11 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*.

2.12 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.

2.13 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 'purpose' field, 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) the text in the 'Additional details' column for the TCO entry of the table in R PCDTX 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.

2.14 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.

2.15 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).

2.16 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'.

2.17 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.memattrs.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;
```

2.18 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.

2.19 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.

2.20 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.

2.21 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.

2.22 E19440

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

2.23 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

2.24 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).

2.25 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.

2.26 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>.

2.27 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.

2.28 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).

2.29 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

2.30 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.

2.31 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).

2.32 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);
```

2.33 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>);
```

2.34 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
    ...
```

2.35 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).

2.36 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.

2.37 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;
    ...
```

2.38 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.

2.39 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.

2.40 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).

2.41 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`.

2.42 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>
```

2.43 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 2n, 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.

2.44 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.

2.45 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;
    ...
```

2.46 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.

2.47 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).

2.48 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), 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.

2.49 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.

2.50 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.

2.51 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.

2.52 C215: SVE

Arm is making a retrospective change to the SVE architecture to remove the capability of selecting a non-power-of-two vector length in non-Streaming SVE as well as in Streaming SVE mode. Specific updates as a result of this change will be communicated in due course.

2.53 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.

2.54 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.

2.55 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).

2.56 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.