Arm® Architecture Reference Manual Supplement, Transactional Memory Extension (TME), for A-profile architecture

Document number	DDI0617
Document version	A.a
Document confidentiality	Non-confidential

Release information

Date	Version	Changes
2022/Aug/19	EAC	• EAC release.

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349 version 21.0

Contents

Arm® Architecture Reference Manual Supplement, Transactional Memory Extension (TME), for A-profile architecture

Contents

Part D Glossary

Chapter D1 Glossary

Part A
Preface

About this Supplement

This document is the *Arm ® Architecture Reference Manual Supplement, Transactional Memory Extension (TME), for A-profile architecture*. This Supplement describes the changes and additions to A-profile architecture, and therefore must be read with the *Arm ® Architecture Reference Manual, for A-profile architecture*.

This Supplement is organized into the following parts:

- Part A

  Preface to the Supplement.

- Part B

  Describes the Transactional Memory Extension.

- Part C

  Appendixes to the Supplement.

- Part D

  Glossary that defines terms used in this document that have a specialized meaning.

Conventions

Typographical conventions

The typographical conventions are:

italic

Introduces special terminology, and denotes citations.

bold

Denotes signal names, and is used for terms in descriptive lists, where appropriate.

`monospace`

Used for assembler syntax descriptions, pseudocode, and source code examples.

Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.

SMALL CAPITALS

Used for some common terms such as IMPLEMENTATION DEFINED.

Used for a few terms that have specific technical meanings, and are included in the Glossary.

Colored text

Indicates a link. This can be:

- A URL, for example http://developer.arm.com
- A cross-reference to another location within the document
- A link, to a chapter or appendix, or to a glossary entry, or to the section of the document that defines the colored term.

{ and }

Braces, { and }, have two distinct uses:

Optional items

In syntax descriptions braces enclose optional items. In the following example they indicate that the <shift> parameter is optional:

```
ADD <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

Similarly they can be used in generalized field descriptions, for example TCR_ELx.{I}PS refers to a field in the TCR_ELx registers that is called either IPS or PS.

Sets of items

Braces can be used to enclose sets. For example, HCR_EL2.{E2H, TGE} refers to a set of two register fields, HCR_EL2.E2H and HCR_EL2.TGE

Notes

Notes are formatted as:

Note

This is a note.

In this Manual, Notes are used only to provide additional information, usually to help understanding of the text. While a Note may repeat architectural information given elsewhere in the Manual, a Note never provides any part of the definition of the architecture.

Signals

In general this specification does not define hardware signals, but it does include some signal examples and recommendations. The signal conventions are:

Signal level

The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:

- HIGH for active-HIGH signals.
- LOW for active-LOW signals.

Lower-case n

At the start or end of a signal name denotes an active-LOW signal.

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by `0b`, and hexadecimal numbers by `0x`. In both cases, the prefix and the associated value are written in a monospace font, for example `0xFFFF0000`. To improve readability, long numbers can be written with an underscore separator between every four characters, for example `0xFFFF_0000_0000_0000`. Ignore any underscores when interpreting the value of a number.

Pseudocode descriptions

This book uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in a monospace font. The pseudocode language is described in the Arm Architecture Reference Manual.

Assembler syntax descriptions

This book contains numerous syntax descriptions for assembler instructions and for components of assembler instructions.

These are shown in a `monospace` font.

Rules-based writing

This specification consists of a set of individual *content items*. A content item is classified as one of the following:

- Declaration
- Rule
- Goal
- Information
- Rationale
- Implementation note
- Software usage

Declarations and Rules are normative statements. An implementation that is compliant with this specification must conform to all Declarations and Rules in this specification that apply to that implementation.

Declarations and Rules must not be read in isolation. Where a particular feature is specified by multiple Declarations and Rules, these are generally grouped into sections and subsections that provide context. Where appropriate, these sections begin with a short introduction.

Arm strongly recommends that implementers read *all* chapters and sections of this document to ensure that an implementation is compliant.

Content items other than Declarations and Rules are informative statements. These are provided as an aid to understanding this specification.

Content item identifiers

A content item may have an associated identifier which is unique among content items in this specification.

After this specification reaches beta status, a given content item has the same identifier across subsequent versions of the specification.

Content item rendering

In this document, a content item is rendered with a token of the following format in the left margin: L_{iiiii}

- L is a label that indicates the content class of the content item.
- *iiiii* is the identifier of the content item.

Content item classes

Declaration

A Declaration is a statement that either

- introduces a concept, or
- introduces a term, or
- describes the structure of data, or
- describes the encoding of data.

A Declaration does not describe behaviour.

A Declaration is rendered with the label *D*.

Rule

A Rule is a statement that describes the behaviour of a compliant implementation.

A Rule explains what happens in a particular situation.

A Rule does not define concepts or terminology.

A Rule is rendered with the label *R*.

Goal

A Goal is a statement about the purpose of a set of rules.

A Goal explains why a particular feature has been included in the specification.

A Goal is comparable to a “business requirement” or an “emergent property.”

A Goal is intended to be upheld by the logical conjunction of a set of rules.

A Goal is rendered with the label *G*.

Information

An Information statement provides information and guidance as an aid to understanding the specification.

An Information statement is rendered with the label *I*.

Rationale

A Rationale statement explains why the specification was specified in the way it was.

A Rationale statement is rendered with the label *X*.

Implementation note

An Implementation note provides guidance on implementation of the specification.

An Implementation note is rendered with the label *U*.

Software usage

A Software usage statement provides guidance on how software can make use of the features defined by the specification.

A Software usage statement is rendered with the label *S*.

Additional reading

This section lists publications by Arm and by third parties.

See Arm Developer, http://developer.arm.com, for access to Arm documentation.

[1] *Arm® Architecture Reference Manual, for A-profile architecture*. (ARM DDI 0487).

This supplement should also be read with the following System register and ISA descriptions:

- *Arm® Architecture Registers, for A-profile architecture*.
- *Arm® A64 Instruction Set, for A-profile architecture*.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this Supplement

If you have comments on the content of this supplement, send an e-mail to errata@arm.com. Give:

- The title, Arm® Architecture Reference Manual Supplement, Transactional Memory Extension (TME), for A-profile architecture.
- The number, DDI0617 A.a.
- The page numbers to which your comments apply.
- The rule identifiers to which your comments apply, if applicable.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests PDFs only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the appearance or behavior of any document when viewed with any other PDF reader.

Progressive terminology commitment

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

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

Part B
The Transactional Memory Extension

Chapter B1
Transactional Memory Extension

B1.1 Introduction

The *Transactional Memory Extension* (TME), FEAT_TME, introduces the TCANCEL, TCOMMIT, TSTART, and TTEST instructions. These instructions support hardware transactional memory, which means a group of instructions can appear to be collectively executed as a single atomic operation.

FEAT_TME is OPTIONAL.

This feature is supported in AArch64 state only.

The ID_AA64ISAR0_EL1.TME field identifies the presence of FEAT_TME.

B1.2 Transactions

R_{TJXB} A *transaction* is a group of instructions executing in *Transactional state*.

R_{YQLB} Instructions outside a transaction execute in *Non-transactional state*.

B1.2.1 Transactional state

B1.2.1.1 Entering transactional state: starting a transaction (TSTART)

R_{ZYKL} When a TSTART instruction is committed for execution in Non-transactional state, it starts an *outer transaction*.

R_{BMPK} When starting an outer transaction, the PE enters Transactional state.

R_{ZFWF} When a TSTART instruction is committed for execution in Transactional state it starts a transaction *nested* within the pre-existing transaction, or simply a *nested transaction*.

R_{DCDQ} The *transactional nesting depth* indicates the degree of nesting of a transaction.

R_{DNMF} The architecture requires the maximum transactional nesting depth to be 255.

R_{RRJX} In Non-transactional state, the transactional nesting depth is 0.

R_{XKHY} When starting a transaction, the transactional nesting depth is incremented by 1.

I_{WTLG} In the rest of the document, unless explicitly prefixed with *outer* or *nested*, the term *transaction* will refer to an outer transaction and all the nested transactions contained within.

B1.2.1.2 Exiting transactional state by committing a transaction (TCOMMIT)

R_{HXMY} A transaction *commits* when a TCOMMIT instruction is committed for execution in Transactional state.

R_{DDFV} Transactional state is exited when committing an outer transaction.

R_{WYQK} When committing a transaction, the transactional nesting depth is decremented by 1.

B1.2.1.3 Exiting transactional state by cancelling (TCANCEL) or failing a transaction

R_{LJYW} A transaction *is canceled* when a TCANCEL instruction is committed for execution in Transactional state.

R_{CSXK} A transaction *fails* when the PE exits transactional state for any reason other than the execution of a TCOMMIT instruction or the execution of a TCANCEL instruction.

R_{HVFD} When a transaction fails or is canceled, Transactional state is exited, and execution continues at the instruction that follows the TSTART instruction of the outer transaction.

R_{MNVC} The result of the TSTART instruction of the outer transaction encodes the cause of the failure. For more information, see B1.3.1 *Failure causes*).

R_{VRLY} When a transaction fails or is canceled, the transactional nesting depth is set to 0.

B1.2.2 Transactional reservation granule, read and write sets

R_{XCXC} The *transactional reservation granule* is defined as a contiguous memory block of size 2^a bytes, formed by ignoring the least significant bits of a memory access.

R_{DVWQ} The size of the memory block is IMPLEMENTATION DEFINED in the range 4 – 512 words.

R_{NYST} The Exclusive Reservation Granule CTR_EL0.ERG identifies the transactional reservation granule.

Below the notions of Location and read or write memory effects are as described in [1] *Basic definitions*.

B1.2.2.1 Transactional read set

R_{RRZY} The *transactional read set* of a transaction is defined to be the set of transactional reservation granules containing all Locations accessed by memory reads inside the transaction.

R_{KJCC} The reads in the transactional read set are referred to as *transactional reads.*

B1.2.2.2 Transactional write set

R_{XJNK} The *transactional write set* of a transaction is defined to be the set of transactional reservation granules containing all Locations accessed by memory writes inside the transaction.

R_{HWVK} The writes in the transactional write set are referred to as *transactional writes.*

R_{BLGB} Limits to the transactional read set size and the transactional write set size are IMPLEMENTATION DEFINED.

B1.3 Transaction failure

B1.3.1 Failure causes

R_{BWYQ} When a transaction fails or is canceled, the destination register of the TSTART instruction of the outer transaction encodes the cause of the failure as follows.

R_{SJJT} For causes that are due to direct or attempted execution of an instruction, only the cause generated by the instruction that appears first in program order is reported.

R_{PTGZ} For causes that are not due to direct or attempted execution of an instruction, any number of causes may be reported.

R_{YXLQ} When more than one cause is reported, then RTRY is set to the logical AND of the prescribed or expected RTRY value of each identified failure cause.

R_{YSWY} **RTRY, bit [15]**

When this bit is set it signifies that the transaction may commit on retry.

When this bit is clear the software should assume that the transaction will not commit on retry.

RTRY is not a failure cause.

R_{YDHF} **REASON, bits [14:0]**

This field holds the 15 low order bits of the TCANCEL operand value when CNCL is 1 else this field is 0.

Bits [63:25]

Reserved, RES0.

R_{TYHM} **TRIVIAL, bit [24]**

When this bit is set it signifies that the system is currently running the trivial implementation enabled by the bits described in B3.1.5 *ISS encoding for an exception from a TSTART instruction*.

The prescribed RTRY value is 0.

R_{BXPB} **INT, bit [23]**

When IMP=1, this bit indicates whether or not an unmasked interrupt was delivered in transactional state but not subsequently taken in non-transactional state due to being masked by the PE. See Section B1.8 *Interrupt masking* for more information.

The prescribed RTRY value is 0.

R_{TTKV} **DBG, bit [22]**

When this bit is set it signifies that a debug-related exception was encountered but not raised.

The prescribed RTRY value is 0.

R_{STJB} **NEST, bit [21]**

When this bit is set it signifies that the maximum transactional nesting depth was exceeded.

The prescribed RTRY value is 0.

R_{RVBK} **SIZE, bit [20]**

When this bit is set it signifies that the transaction failed because the transactional read set limit or the transactional write set limit was exceeded.

The prescribed RTRY value is 0.

R_{SZFF} **ERR, bit [19]**

When this bit is set it signifies that an operation was attempted which is not architecturally permitted in Transactional state. This includes but is not limited to attempting to raise a synchronous exception, attempting to execute an instruction not permitted in Transactional state, or attempting to change Exception level.

The prescribed RTRY value is 0.

R_{XZDB} **IMP, bit [18]**

When this bit is set it signifies a failure cause that does not fall under any of the other cases.

The expected RTRY value is 1 if the transaction may commit on retrying and 0 otherwise.

RTRY must not systematically be set to 1 with IMP cause. This is because it could prevent the forward progress in finite time of at least one the threads that is accessing a location within the transactional read or write sets.

R_{RMJK} **MEM, bit [17]**

When this bit is set it signifies that the transaction failed because a transactional memory conflict was detected.

The expected RTRY value is 1.

R_{KWHH} **CNCL, bit [16]**

When this bit is set it signifies that the transaction was canceled by a `TCANCEL` instruction.

The RTRY value is the most significant bit of the `TCANCEL` immediate operand.

B1.3.2 Transaction checkpoint

R_{PJGV} The *transaction checkpoint* defines the following subset of the AArch64 state:

- Registers in AArch64 execution state: R0-R30, SP, ICC_PMR_EL1.
- AArch64 process state: NZCV, DAIF.
- If both floating-point and SVE are enabled: Z0-Z31, P0-P15, FFR, FPCR, FPSR.
- If floating-point is enabled and SVE is disabled or trapped: V0-V31, FPCR, FPSR.

R_{GKRW} It is IMPLEMENTATION DEFINED if any of the System registers encoded with op0==0b11 and CRn==0b1x11 are included in the transaction checkpoint.

R_{QJYL} No other System registers are included in the transaction checkpoint.

R_{BFYL} When a transaction fails or is canceled, the subset of the AArch64 state defined by the transaction checkpoint is reverted to a state that is consistent with the PE having executed all of the instructions up to but not including the point in the instruction stream where Transactional state was entered, and none afterwards, with the following exceptions:

- The destination register of the `TSTART` instruction of the outer transaction is updated to encode the transaction failure cause.
- When executing at an Exception level that is constrained to use a vector length that is less than the maximum implemented vector length, the bits beyond the constrained length of Z0-Z31, P0-P15, and FFR are restored to a value of either zero or the value they had when Transactional state was entered. The choice between these options is IMPLEMENTATION DEFINED and can vary dynamically.

R_{FFQR} Writes by a failed or canceled transaction do not generate write Memory effects. For the definition of Memory effects, see [1] *Basic definitions*.

R_{KBBS} If SVE is disabled or trapped, the current vector length is considered to be constrained to 128 bits (see [1] *SVE Configurable vector length*).

R_{VQGT} SPSel cannot be modified in Transactional state. For more information, see Section B1.9 *A64 instruction behavior in Transactional state*.

B1.4 Memory model

Transactional Memory Extension (TME) proposes the following additions to the memory ordering and observability rules described in the [1] *Definition of the Armv8 memory model*.

B1.4.1 External visibility

Adding the following definitions:

R_{TCXC} **Locally-ordered-before**

A read or a write RW1 is *Locally-ordered-before* a read or a write RW2 from the same Observer if and only if any of the following cases apply:

- RW1 is Dependency-ordered-before RW2.
- RW1 is Atomic-ordered-before RW2.
- RW1 is Barrier-ordered-before RW2.
- RW1 is Locally-ordered-before a read or a write that is Locally-ordered-before RW2.

R_{KDFQ} **Transactionally-observed-by**

A read or a write RW1 from an Observer is *Transactionally-observed-by* a read or a write RW2 from a different Observer if and only if any of the following cases apply:

- There is a read or a write RW3 in the same transaction as RW1, and RW3 is Observed-by RW2.
- There is a read or a write RW3 in the same transaction as RW2, and RW1 is Observed-by RW3.

Changing the definition of Barrier-ordered-before to the following:

R_{RBRW} **Barrier-ordered-before**

Barrier instructions order prior Memory effects before subsequent Memory effects generated by the same Observer. A read or a write RW1 is *Barrier-ordered-before* a read or a write RW2 from the same Observer if and only if RW1 appears in program order before RW2 and any of the following cases apply:

- RW1 appears in program order before a `DMB FULL` that appears in program order before RW2.
- RW1 is a write W1 generated by an instruction with Release semantics and RW2 is a read R2 generated by an instruction with Acquire semantics.
- RW1 is generated by an instruction with Acquire semantics.
- RW2 is generated by an instruction with Release semantics.
- RW1 is a read R1 appearing in program order before a `DMB LD` that appears in program order before RW2.
- RW2 is a write W2 and either:
  - RW1 is a write W1 appearing in program order before a `DMB ST` that appears in program order before W2.
  - RW1 appears in program order before a write W3 generated by an instruction with Release semantics and W2 is Coherence-after W3.
- RW1 and RW2 are not in the same transaction, and at least one of RW1 or RW2 is in the read or write set of a committed transaction.
- RW1 appears in program order before a committed transaction that appears in program order before RW2.

Changing the definition of Ordered-before to the following:

R_{BXFQ} **Ordered-before**

An arbitrary pair of Memory effects is ordered if it can be linked by a chain of ordered accesses consistent with external observation. A read or a write RW1 is Ordered-before a read or a write RW2 if and only if any of the following cases apply:

- RW1 is Observed-by RW2.
- RW1 is Transactionally-observed-by RW2.
- RW1 is Locally-ordered-before RW2.
- RW1 is Ordered-before a read or a write that is Ordered-before RW2.

I_{FMHN} Conflicts are a natural consequence of the pre-existing External visibility requirement. For more information, see [1] *Ordering constraint*. A cycle in Ordered-before that involves a Transactionally-observed-by relation indicates a conflict.

R_{ZRBP} A transaction is said to be conflicting if and only if committing the transaction would violate the external visibility requirement, in which case the transaction fails with MEM cause.

S_{JCSK} In the event of repeated transactional conflicts the architecture does not guarantee forward progress for any transactions involved, and the software must take appropriate measures for example by setting a threshold after which the software takes a specific fallback path.

B1.4.2 Atomicity

I_{LWMY} This section documents the behavior of the A64 Load-Exclusive and Store-Exclusive instructions, and all A64 atomic instructions (`CAS`, `CASP`, `LD<OP>`, and `SWP`) in Transactional state.

R_{LJDF} Transactional writes generated as side-effects from the above instructions follow the ordering and observability rules described in the previous section.

R_{PNXP} A transactional store to an address marked for exclusive access in the global monitor for any other PE:

- Clears the marking if the transaction commits.
- May clear the marking if the transaction fails or is canceled.

R_{JFLH} When entering Transactional state or exiting Transactional state by committing, canceling or failing a transaction:

- The local monitor state of the executing PE transitions to the Open access state.
- The final state of the global monitor state machine for the executing PE is IMPLEMENTATION DEFINED.
- The global monitor state machine for any other PE is not affected.

R_{NBWK} If the global monitor state for a PE changes from Exclusive access to Open access because of entering or exiting Transactional state, an event is generated and held in the Event register for that PE.

S_{HBSY} Inserting any of the A64 atomic primitive instructions inside a transaction does not provide any extra functionality to software. Sharing code among the transaction and its fallback path may lead to such instructions being executed in transactional state.

B1.5 Transactions and memory attributes

I_{CBHX} Some system implementations might not support transactional accesses for all regions of the memory. This can apply to:

- Any type of memory in the system that does not support hardware cache coherency.
- Device memory, Non-cacheable memory, or memory that is treated as Non-cacheable, in an implementation that does support hardware cache coherency.

R_{FVGF} In such implementations, it is defined by the system which address ranges or memory types support transactional accesses.

R_{NZRZ} The memory types for which it is architecturally guaranteed that transactional accesses are supported are:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hints and Write allocation hints and not transient.
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hints and Write allocation hints and not transient.

R_{LMVV} If transactional accesses are not supported for an address range or memory type, then performing a transactional load or a transactional store to such a location fails the transaction with IMP cause.

I_{JXXM} Memory accesses generated by different instructions inside a transaction can have different shareability attributes.

R_{LSXT} When accesses to any two Locations generated by the same instruction inside a transaction have different shareability attributes then the results are CONSTRAINED UNPREDICTABLE. For more information, see [1] *Memory access restrictions*.

R_{DCWP} Accesses, including transactional accesses, by multiple PEs to a Location with mismatched attributes leads to CONSTRAINED UNPREDICTABLE behavior. For more information, see [1] *Mismatched memory attributes*.

B1.6 Address translation

B1.6.1 Transactional translation table walks

R_{YCTD} Transactional memory accesses to a given address are permitted to perform translation table walks, except when the transactional memory access originates from EL0 and either of the following cases holds:

- The address is translated using TTBR0_EL1, and TCR_EL1.NFD0==1.
- The address is translated using TTBR1_EL1, and TCR_EL1.NFD1==1.
- The address is translated using TTBR0_EL2, and TCR_EL2.NFD0==1.
- The address is translated using TTBR1_EL2, and TCR_EL2.NFD1==1.

R_{MZBV} A transactional memory access that is not permitted to perform a translation table walk and would otherwise generate an exception in Non-transactional state fails the transaction with ERR cause without generating an exception.

X_{WRWD} This scheme addresses timing attacks on Kernel Address Space Layout Randomization. If TCR_EL1.NFD1 is set, an EL0 transaction that attempts to probe the kernel address space will always fail with the same timing and the same failure cause because either there is a TLB miss and the transaction fails with ERR cause, or there is a TLB hit and a suppressed MMU permission fault (assuming TTBR1_EL1 address range is protecting itself from EL0 accesses) fails the transaction with ERR cause. This way the malicious software should not be able to distinguish between the two cases.

B1.6.2 Hardware management of the Access flag and dirty state

R_{ZNNY} TME requires that the implementation supports hardware management of the Access flag and dirty state. For more information, see [1] *Hardware management of the Access flag and dirty state*.

R_{KGPQ} Transactional memory accesses follow the rules for updating the Access flag and dirty state as described in [1] *Hardware management of the Access flag and dirty state* and [1] *Ordering of hardware updates to the translation tables*.

R_{NCHW} When hardware updating of the Access flag is enabled, updates to the stage 1 and stage 2 Access flag generated by memory accesses in Transactional state may become observable even if the transaction fails or is canceled.

R_{SSNZ} When hardware updating of the dirty state is enabled, updates to the stage 1 and stage 2 dirty state generated by memory accesses in Transactional state may become observable even if the transaction fails or is canceled.

I_{LZDF} Arm requires hardware management of the Access flag and dirty state for performance reasons.

S_{JKDY} Software management of the Access flag would mean that when a page is accessed for the first time inside a transaction, the transaction fails and is re-executed in Non-transactional state.

I_{TMBP} Arm requires allowing transactional dirty state updates to become observable even if the responsible transaction fails or is canceled for performance reasons. Otherwise, every time a page is written for the first time inside a transaction, then either the transaction fails which is bad for performance, or the hardware must manage the dirty state updates until the PE exits Transactional state which increases implementation complexity.

B1.6.3 TLB shoot-down

R_{XVMC} A `TLBI` by another PE that applies to a Location in the transactional read set or the transactional write set of the currently executing transaction causes that transaction to fail.

I_{YJQH} In order to provide this functionality, an implementation needs to either track the Virtual to Physical Address mappings for the Locations in the transactional read or write sets of the transaction that is currently executing, or fail the transaction on any invalidation by another PE. In the former case, if a transaction exceeds the IMPLEMENTATION DEFINED tracking limit of Virtual to Physical Address mappings, then the transaction fails.

I_{RFLJ} For performance reasons, Arm recommends that the implementation does not fail the transaction if the ASID and VMID of an invalidation by another PE, does not match the one of the currently executing transaction.

B1.6.4 Translation table modifications inside transactions

I_{KXRF} The required break-before-make sequence described in the[1] *General TLB maintenance requirements* for updating translation table entries cannot be executed inside a transaction, since the required `TLBI` and `DSB` instructions lead to transaction failure (see Table B1.4 and Table B1.6).

B1.7 Modification of instructions in Transactional state

I_{ZMVM} The Arm architecture does not require the hardware to ensure coherency between instruction caches and memory, even for shared memory locations. For more information, see [1] *Implication of caches for the application programmer*.

R_{RLTS} TME follows the rules for concurrent modification and execution of instructions as explained in [1] *Concurrent modification and execution of instructions*.

R_{CLBS} TME does not guarantee that a transactional thread of execution T is isolated from a non-transactional thread of execution making modifications to the instruction stream of T.

See also Table B1.6 for the behavior of `ISB` and `DSB` instructions in Transactional state.

I_{CVKD} This implies that a transactional thread of execution cannot modify its own instruction stream, or other instruction streams using the mechanism suggested in [1] *Concurrent modification and execution of instructions*, since transactional writes are not observable until a transaction commits and the `DSB` instruction required for synchronization of the modifications fails the transaction.

B1.8 Interrupt masking

R_{TTSQ} In Transactional state, interrupts are pended, and unmasked interrupts are taken when Transactional state is exited.

R_{CHZK} In the absence of a specific requirement to take an interrupt, it is IMPLEMENTATION DEFINED if the delivery of an unmasked interrupt fails the transaction, but the architecture requires that the interrupt is taken in finite time. For more information, see [1] *Prioritization and recognition of interrupts*.

R_{BFMS} If the delivery of an unmasked interrupt fails the transaction, the failure cause reported is IMP.

I_{LHSC} Transactional code with sufficient privileges can change the value of DAIF or ICC_PMR_EL1 to mask or unmask interrupts.

R_{NXXN} A transaction fails with IMP cause and INT set if both of the following happen:

- an unmasked interrupt delivered to a PE leads to the currently executing transaction on the PE to fail, and
- upon restoring DAIF and ICC_PMR_EL1 the interrupt becomes masked again and will not be taken.

X_{MLVZ} If the transaction fails or is canceled the DAIF and ICC_PMR_EL1 registers are restored to the values they held before entering Transactional state. This action will affect the masking or unmasking of interrupts before the first non-transactional instruction executes. If the implementation decides to fail the transaction when the interrupt is delivered, then after the values of DAIF and ICC_PMR_EL1 are restored to their pre-transactional state, the interrupt will be masked and will not be taken. But if the transaction restarts then, as soon as interrupts are transactionally re-enabled, the transaction will fail because there is a pending interrupt. To avoid a livelock this is reported as a non-restartable failure. For more information, see Section B1.3.2 *Transaction checkpoint*.

B1.9 A64 instruction behavior in Transactional state

R_{NHND} Transactional state changes the execution of some A64 instructions.

This section includes the affected instructions and their expected behavior in Transactional state.

R_{QHYS} Any instruction not included in this section behaves the same in Transactional state as in Non-transactional state.

R_{SHQC}
- Exception level changes cannot occur. Executing an instruction that would otherwise generate an Exception level change fails the transaction with ERR cause as described in this document.

R_{LXPV}
- Synchronous exceptions are suppressed and fail the transaction with ERR cause. See Sections B2.1.1 *Breakpoint Instruction exceptions*, B2.1.2 *Breakpoint exceptions*, and B2.1.3 *Watchpoint exceptions* for details.

Table B1.1: Exception generating instructions

Mnemonic	Instruction	Behavior
BRK	Breakpoint Instruction	See B2.1.1 *Breakpoint Instruction exceptions*
HLT	Halt Instruction	See B2.2.2 *Halting Instruction debug event*
HVC	Generate exception targeting EL2	Transaction fails with ERR cause
SMC	Generate exception targeting EL3	Transaction fails with ERR cause
SVC	Generate exception targeting EL1	Transaction fails with ERR cause

Table B1.2: Exception return instructions

Mnemonic	Instruction	Behavior
ERET	Exception return using current ELR and SPSR	Transaction fails with ERR cause
ERETAA, ERETAB	Exception return with pointer authentication	Transaction fails with ERR cause

Table B1.3: System register instructions

Mnemonic	Instruction	Behavior
MRS	Move System register to general-purpose register	See B1.9.1 *MRS*
MSR (register)	Move general-purpose register to System register	See B1.9.2 *MSR (register)*
MSR (immediate)	Move immediate to PSTATE field	See B1.9.3 *MSR (immediate)*

Table B1.4: System instructions

Mnemonic	Instruction	Behavior
SYS	System instruction	See B1.9.4 *SYS and SYSL*

Mnemonic	Instruction	Behavior
SYSL	System instruction with result	See B1.9.4 *SYS and SYSL*
IC	Instruction cache maintenance	Transaction fails with ERR cause
DC except DC ZVA	Data cache maintenance	Transaction fails with ERR cause
DC ZVA	Data cache zero	Same as in Non-transactional state
AT	Address translation	Transaction fails with ERR cause
TLBI	TLB Invalidate	Transaction fails with ERR cause

Table B1.5: Hint instructions

Mnemonic	Instruction	Behavior
NOP	No operation	Same as in Non-transactional state
YIELD	Yield hint	Same as in Non-transactional state
WFE	Wait for event	See B1.9.5 *Wait for Event*
WFI	Wait for interrupt	Transaction fails with ERR cause
SEV	Send event	Same as in Non-transactional state
SEVL	Send event local	Same as in Non-transactional state
HINT	Unallocated hint	Same as in Non-transactional state

Table B1.6: Barrier and CLREX instructions

Mnemonic	Instruction	Behavior
CLREX	Clear exclusive monitor	Same as in Non-transactional state
DSB	Data synchronization barrier	Transaction fails with ERR cause
DMB	Data memory barrier	See B1.9.6 *DMB*
ESB	Error synchronization barrier	Transaction fails with ERR cause
ISB	Instruction synchronization barrier	See B1.9.7 *ISB*
PSB CSYNC	Profiling synchronization barrier	Same as in Non-transactional state
TSB CSYNC	Trace synchronization barrier	Same as in Non-transactional state

B1.9.1 MRS

R_{KTMD} Registers encoded with op0==0b10 are not accessible at any Exception level.

R_{ZTYF} Registers encoded with op0==0b11 and CRn==12, except ICC_HPPIR0_EL1, ICC_HPPIR1_EL1, ICC_RPR_EL1, are not accessible at any Exception level.

R_{VJST} If enhanced nested virtualization is enabled and the read of a permitted System register is transformed to a read from memory, then the generated read is considered transactional.

R_{MCFP} Attempting to read a register that is not accessible at the current Exception level fails the transaction with ERR cause without trapping.

R_{KPDH} If a read from memory generates any exception, the exception is suppressed and the transaction fails with ERR cause without trapping.

B1.9.2 `MSR` (register)

R_{CKFL} Registers FPCR, FPSR, NZCV, DAIF, ICC_PMR_EL1, and PMSWINC_EL0 are accessible at the same Exception levels as in Non-transactional state.

R_{STQJ} All other registers are not accessible at any Exception level.

R_{XZGW} Attempting to write a register that is not accessible at the current Exception level fails the transaction with ERR cause without trapping.

B1.9.3 `MSR` (immediate)

R_{MNFL} Only the instruction forms that select the MSR DAIFSet and MSR DAIFClr instructions are defined.

R_{SGNZ} All other encodings are reserved, and the corresponding instructions are UNDEFINED.

R_{DNMG} Attempting to execute an UNDEFINED instruction fails the transaction with ERR cause without trapping.

B1.9.4 `SYS` and `SYSL`

R_{LKNM} The accessibility of the instructions encoded with op0=`0b01` and CRn=`0b1x11` is IMPLEMENTATION DEFINED.

R_{GKLF} Attempting to execute an undefined instruction fails the transaction with ERR cause without trapping.

B1.9.5 Wait for Event

R_{BXPX} If executing a `WFE` instruction in Non-transactional state would trap to a higher Exception level, then the transaction fails with ERR cause without trapping.

Otherwise, the `WFE` instruction behaves the same as in Non-transactional state.

R_{GPGL} A transaction that has entered low-power state due to the execution of a `WFE` instruction is called a *waiting transaction*.

R_{FCPL} A PE that enters a low-power state continues to track and respond to transactional conflicts with memory accesses from other PEs.

R_{GFJL} It is IMPLEMENTATION DEFINED whether a waiting transaction that receives a *WFE wake-up event* resumes execution without failing. For more information, see [1] *Wait for Event mechanism and Send event*.

R_{KYSZ} A waiting transaction is permitted to fail for any IMPLEMENTATION DEFINED reason before a wake-up event is received.

I_{PMZF} Arm recommends that a waiting transaction fails on a transactional conflict with another PE, for performance reasons.

I_{FFTN} Arm recommends that waiting transactions do not fail upon receiving a wake-up event that is not an interrupt that must be taken, for performance reasons.

The following is a non-exhaustive list of wake-up events that could safely resume a transaction:

R_{ZVBJ} • The execution of an SEV instruction on any other PE in the multiprocessor system.

R_{ZRNN} • An event sent by the timer event stream for the PE. For more information, see [1] *Event streams*.

R_{FDMK} • An event caused by the clearing of the global monitor for the PE.

R_{LPBD} • A masked interrupt.

B1.9.6 DMB

R_{SHWX} Transactional accesses to Device or Normal Non-cacheable memory that appear before the DMB in program order are merged with transactional accesses to Device or Normal Non-cacheable memory of the same type (read or write) to the same Location that appear in program order after the DMB, if they are executed in the same transaction.

R_{XMCB} If transactional accesses, executing in the same transaction containing the DMB, access the same memory-mapped peripheral of arbitrary system-defined size, then it is not guaranteed that accesses in program order before the DMB that are accessing Device or Normal Non-cacheable memory will arrive at the peripheral before accesses in program order after the DMB that are accessing Device or Normal Non-cacheable memory.

B1.9.7 ISB

R_{NLMZ} Executing an ISB instruction in Transactional state is a *Context synchronization event*, with the same effects of a Context synchronization event in Non-transactional state except that unmasked interrupts that are pending at the time of the Context synchronization event are not required to be taken.

R_{YHLW} If halting is allowed, any Halting debug event that is pending before the ISB instruction is executed fails the transaction with DBG cause.

R_{JVGJ} It is IMPLEMENTATION DEFINED whether the transaction fails if there are pending unmasked interrupts when the ISB instruction is executed.

I_{NMWC} If the first instruction after exiting Transactional state generates a synchronous exception, then the architecture does not define whether the PE takes the interrupt or the synchronous exception first.

See also B1.8 *Interrupt masking*.

B1.9.8 First-fault and Non-fault load instructions

I_{HMFW} SVE provides a First-fault option for some SVE vector load instructions. For more information, see [1] *Glossary*.

R_{NMXZ} In Transactional state, SVE's First-fault option causes memory access faults to be suppressed without causing the transaction to fail if they do not occur as a result of the First active element of the vector.

Instead, the FFR is updated to indicate which of the active vector elements were not successfully loaded (see [1] *FFR, First Fault Register*).

I_{GHCR} SVE provides a Non-fault option for some SVE vector load instructions. For more information, see [1] *Glossary*.

R_{QVYD} In Transactional state, SVE's Non-fault option causes all memory access faults to be suppressed without causing the transaction to fail.

Instead, the FFR is updated to indicate which of the active Vector elements were not successfully loaded (see [1] *FFR, First Fault Register*).

B1.10 Reset

R_{KFBV} All the rules described in the [1] *Reset* section apply whether or not the PE is in Transactional state when a Cold or a Warm reset is asserted.

R_{ZGNK} If the PE resets to AArch64 state using either a Cold or a Warm reset, the PE resets to Non-transactional state.

B1.11 Identification mechanism

R_{XXMT} The implementation of TME is identified by ID_AA64ISAR0_EL1.TME.

I_{WFVR} Although TME defines no instruction enables and disables, or trap controls, Arm recommends the addition of an instruction disable control in ACTLR_ELx for the highest implemented Exception level which if set has the following effect:

- The bits in ID_AA64ISAR0_EL1.TME are RES0.
- The TME instructions are UNDEFINED at EL0 and above.

Chapter B2
Debug, PMU, and Trace

B2.1 Self-hosted debug

B2.1.1 Breakpoint Instruction exceptions

R_{BTRM} In Transactional state, executing a *breakpoint instruction* fails the transaction with a DBG cause, without raising a Breakpoint Instruction exception. For more information on breakpoint instructions, see [1] *Breakpoint Instruction exceptions*.

I_{FDMB} A transaction with a breakpoint instruction cannot make forward progress; it will always fail. The software is responsible for reading the failure information returned by TSTART and acting accordingly.

B2.1.2 Breakpoint exceptions

R_{HLHZ} In Transactional state, Breakpoint exceptions are suppressed and fail the transaction with a DBG cause. For more information on breakpoint exceptions, see [1] *Breakpoint exceptions*.

I_{VMWG} A hardware breakpoint will continuously fail a restarting transaction until either the breakpoint conditions are not met (*e.g.*, the transactional code follows a different execution path), or the breakpoint is disabled. It is the responsibility of the software to detect this situation and act accordingly.

B2.1.3 Watchpoint exceptions

R_{WSMX} In Transactional state, Watchpoint exceptions are suppressed and fail the transaction with a DBG cause. For more information on watchpoint exceptions, see [1] *Watchpoint exceptions*.

I_{QHCS} A hardware watchpoint will continuously fail a restarting transaction until either the watchpoint conditions are not met (*e.g.*, the transactional code accesses different Locations), or the watchpoint is disabled. It is the responsibility of the software to detect this situation and act accordingly.

B2.1.4 Software Step exceptions

R_{TCSR} In Non-transactional state, executing a `TSTART` instruction when software step is active-not-pending fails the transaction with DBG cause. For more information on active-not-pending, see [1] *Software Step exceptions*.

I_{VYYF} Enabling or disabling software step is not possible in Transactional state because attempting to update MDSCR_EL1.SS fails the transaction. For more information, see B1.9.2 *MSR (register)*.

R_{JNVJ} If PSTATE.D is cleared inside a transaction and MDSCR_EL1.SS is 1 when entering Transactional state, the transaction fails with DBG cause.

B2.2 External debug

For the definitions of the various Halting debug events, see [1] *Halting Debug Events*.

B2.2.1 Breakpoint and Watchpoint debug events

R_{CLHP} In Transactional state, a Breakpoint debug event or a Watchpoint debug event that would otherwise cause entry to Debug state, fails the transaction with DBG cause without entering Debug state.

For more information, see [1] *Breakpoint and Watchpoint debug events*.

B2.2.2 Halting Instruction debug event

R_{CVJX} If EDSCR.HDE == 0 or if halting is prohibited, then executing a HLT instruction in Transactional state fails the transaction with ERR cause.

R_{XHFX} If EDSCR.HDE == 1 and halting is allowed, then executing a HLT instruction in Transactional state fails the transaction with a DBG cause without entering Debug state.

For more information, see [1] *Halt Instruction debug event*.

B2.2.3 Halting Step debug events

R_{BWLB} In Non-transactional state, executing a `TSTART` instruction when *Halting step* is active-not-pending fails the transaction with DBG cause. For more information on *Halting step*, see [1] *Halting Step debug events*.

I_{GTHQ} Enabling or disabling Halting step is not possible in Transactional state because attempting to update EDECR.SS fails the transaction as described in B1.9.2 *MSR (register)*.

For more information, see [1] *Halting Step debug events*.

B2.2.4 External Debug Request debug event

R_{BWYJ} If halting is allowed, all of the following applies:

- *External Debug Request* debug events asserted in Transactional state are pended.
- Unmasked External Debug Request debug events are taken when the PE exits Transactional state.
- In the absence of a *Context synchronization event*, it is IMPLEMENTATION DEFINED if the delivery of an unmasked External Debug Request debug event fails the transaction, but the architecture requires that the External Debug Request debug event is taken in finite time as per [1] *Synchronization and External Debug Request debug events*.
- If the delivery of an unmasked External Debug Request debug event fails the transaction, the failure cause reported is DBG.

See also:

- *External Debug Request debug event* in the [1].
- B1.9.7 *ISB*.

B2.2.5 Reset Catch debug event

R_{BJTP} If halting is allowed, all of the following applies:

- *Reset Catch* debug events asserted in Transactional state are pended and are taken when the PE exits Transactional state.
- In the absence of a *Context synchronization event*, it is IMPLEMENTATION DEFINED if the delivery of a Reset Catch debug event fails the transaction, but the architecture requires that the Reset Catch debug event is taken in finite time as per [1] *Synchronization and Halting debug events*.
- If the delivery of a Reset Catch debug event fails the transaction, the failure cause reported is DBG.

See also:

- *Reset Catch debug events* in the [1].
- B1.9.7 *ISB*.

B2.2.6 Other Halting debug events

I_{WTFX} *Exception Catch* debug events cannot occur inside a transaction because an exception entry or exception return cannot occur inside a transaction. For more information on Exception Catch, see [1] *Exception Catch debug event*.

I_{JYVR} *OS Unlock Catch* debug events, and *Software Access* debug events cannot occur inside a transaction because they are generated by accesses to System registers that cannot occur inside a transaction.

See also:

- *OS Unlock Catch debug event* in the [1].
- *Software Access debug event* in the [1].

B2.2.7 Behavior in Debug state

R_{FKXF} The TCOMMIT instruction is unchanged in Debug state.

I_{FDRG} TCOMMIT follows the rules described in the *Any instruction that is* UNDEFINED *in Non-debug state* topic of the [1] since the PE cannot enter Transactional state in Debug state and TCOMMIT is UNDEFINED in Non-transactional state.

R_{RVKB} TCANCEL and TTEST are CONSTRAINED UNPREDICTABLE in Debug state.

I_{ZKFG} TCANCEL and TTEST follow the rules described in the [1] *All other instructions*.

R_{NQSW} TSTART is CONSTRAINED UNPREDICTABLE in Debug state.

R_{XYLB} TSTART behaves in one of the following ways:

- It is UNDEFINED.
- It executes as a NOP.
- It does not enter Transactional state and it returns an UNKNOWN value.

B2.2.8 The PC Sample-based Profiling Extension

R_{NPQB} All the rules described in [1] *The PC Sample-based Profiling Extension* chapter apply to a PE in Transactional state too.

R_{QCCR} Additionally, *Transactional Memory Extension* (TME) extends PPMPCSR to indicate if a sample references an instruction executed in Transactional state or Non-transactional state.

I_{SHHH} Like in Non-transaction state, only reference instructions that were committed for execution are sampled in Transactional state.

I_{VYCF} Samples can reference instructions from failed or canceled transactions.

B2.3 The Statistical Profiling Extension

B2.3.1 Memory accesses by profiling operations

R_{TYGM} The profiling operation executes independently of the instructions that are executed on the PE and acts as a separate memory observer from the PE in the system. For more information, see [1] *Synchronization and Statistical Profiling*.

R_{XXLF} If a profiling write operation overlaps with the read-set or write-set of a transaction, it is constraint UNPREDICTABLE whether:

- The write has the same effect on the transaction as a store by any other Observer to that address.
- The write has no effect on this transaction.

R_{ZVWK} A profiling operation executes independently of the instruction or instructions that are executed on the PE and acts as a separate memory observer from the PE in the system.

R_{TSLS} Writes to the Profiling Buffer generated by profiling operations in Transactional state are considered non-transactional and as such:

- They are not part of the transactional write set.
- They are observable even if the transaction fails or is canceled.

R_{GJXT} For a sampled operation, if the operation is executed in Transactional state then Events packet.E[16] (Transactional) is set to 1.

S_{PHDN} Software can use PMSEVFR_EL1[16] to filter recording of sampled operations based on the Transactional flag.

B2.3.2 Events packet payload

TME extends existing the SPE protocol with the following events packet payload:

R_{QPZV} **E[16], byte 2 bit [0]**

If TME is not implemented, this bit reads-as-zero. The possible values of this bit are:

0	Operation executed in Non-transactional state.
1	Operation executed in Transactional state.

B2.3.3 Profile Buffer management interrupts

I_{NTMY} See B1.8 *Interrupt masking* for the treatment of interrupts in Transactional state.

B2.4 The Embedded Trace Extension

For information, see [1] *The Embedded Trace Extension*.

B2.5 The Performance Monitors Extension

B2.5.1 Event filtering

I_{WKDW} TME extends the filtering capabilities of the PMU to enable filtering by Transactional state.

R_{NDRV} For each *Attributable* event, if the value of PMEVTYPER<n>_EL0.T is 1, then the event is counted only if the PE is in Transactional state.

Otherwise, for each *Unattributable* event, it is IMPLEMENTATION DEFINED whether the filtering applies.

R_{LFKP} TME adds new events that count transitions between Transactional and Non-transactional states. It is IMPLEMENTATION DEFINED if these events are considered to occur in Transactional or Non-transactional state. See the description of the individual events in Table B2.2 for more details.

I_{LNQD} For the definition of *Attributable* and *Unattributable*, see [1] *Attributability*.

B2.5.2 Accuracy of event filtering

R_{QJQP} TME does not require filtering by Transactional state to be accurate. For more information, see [1] *Accuracy of event filtering*.

I_{WXMC} For many events, during a transition between Transactional and Non-transactional states, events generated by instructions executed in one state can be counted in the other state.

R_{VTYX} It is not permitted for the following events to be counted in the wrong state:

- Any event classified as *Instruction architecturally executed.*
- Any event classified as *Instruction architecturally executed, Condition code check pass*.
- EXC_TAKEN, Exception taken.

I_{VMRV} For the definition of *Instruction architecturally executed*, and *Instruction architecturally executed, Condition code check pass*, see [1] *PMU events and event numbers*.

R_{FWHG} TME adds the following *required* events.

Table B2.2: TME related events

Number	Type	Event
0x4030	Architectural	B2.5.3 *TSTART_RETIRED*
0x4031	Architectural	B2.5.4 *TCOMMIT_RETIRED*
0x4032	Architectural	B2.5.5 *TME_TRANSACTION_FAILED*
0x4034	Architectural	B2.5.6 *TME_INST_RETIRED_COMMITTED*
0x4035	Microarchitectural	B2.5.7 *TME_CPU_CYCLES_COMMITTED*
0x4038	Microarchitectural	B2.5.8 *TME_FAILURE_CNCL*
0x403A	Microarchitectural	B2.5.9 *TME_FAILURE_ERR*
0x403B	Microarchitectural	B2.5.10 *TME_FAILURE_IMP*
0x403C	Microarchitectural	B2.5.11 *TME_FAILURE_MEM*

Number	Type	Event
0x4039	Microarchitectural	B2.5.12 *TME_FAILURE_NEST*
0x403D	Microarchitectural	B2.5.13 *TME_FAILURE_SIZE*
0x403E	Microarchitectural	B2.5.14 *TME_FAILURE_TLBI*
0x403F	Microarchitectural	B2.5.15 *TME_FAILURE_WSET*

B2.5.3 TSTART_RETIRED

R_{NRPB} The counter increments for every architecturally executed TSTART instruction that starts an outer transaction.

R_{HKGP} If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not TSTART_RETIRED increments the counter.

B2.5.4 TCOMMIT_RETIRED

R_{CKYT} The counter increments for every architecturally executed TCOMMIT instruction that commits an outer transaction.

R_{XCGL} If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not TCOMMIT_RETIRED increments the counter.

B2.5.5 TME_TRANSACTION_FAILED

R_{YVWS} The counter increments for every transaction that fails or is canceled.

R_{JKPJ} If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not TME_TRANSACTION_FAILED increments the counter.

B2.5.6 TME_INST_RETIRED_COMMITTED

R_{LPZF} The counter increments for every architecturally executed instruction in Transactional state if the currently executing transaction commits.

I_{WBGV} It is permissible for an implementation to limit the increment that the execution of a transaction can generate to the counter to a maximum value of 2^{32}-1.

I_{CJMQ} Two possible implementations of this functionality are:

- The implementation accumulates events to the counter directly. If the transaction fails, the counter is restored to the value it had when the transaction started.
- The implementation accumulates events without updating the counter. If the transaction commits, the counter is updated with the accumulated value.

B2.5.7 TME_CPU_CYCLES_COMMITTED

R_{RMSG} The counter increments on every cycle the PE is in Transactional state if the currently executing transaction commits.

R_{DSFD} All counters are subject to changes in clock frequency, including when a WFI or WFE instruction stops the clock. This means that it is CONSTRAINED UNPREDICTABLE whether or not TME_CPU_CYCLES_COMMITTED continues to increment when the clocks are stopped by WFI and WFE instructions.

R_{JBQT} In a multithreaded implementation, TME_CPU_CYCLES_COMMITTED counts each cycle for the processor for which this PE thread was active and could issue an instruction. For more information, see [1] *Cycle event counting on multithreaded implementations*.

I_{WMBD} It is permissible for an implementation to limit the increment that the execution of a transaction can generate to the counter to a maximum value of 2^{32}-1.

I_{CFNH} Two possible implementations of this functionality are:

- The implementation accumulates events to the counter directly. If the transaction fails, the counter is restored to the value it had when the transaction started.
- The implementation accumulates events without updating the counter. If the transaction commits, the counter is updated with the accumulated value.

B2.5.8 TME_FAILURE_CNCL

R_{BTWV} The counter increments for every transaction that fails with CNCL cause.

R_{LVHX} If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not TME_FAILURE_CNCL increments the counter.

B2.5.9 TME_FAILURE_ERR

R_{NJXX} The counter increments for every transaction that fails with ERR cause.

R_{XDTJ} If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not TME_FAILURE_ERR increments the counter.

B2.5.10 TME_FAILURE_IMP

R_{TCHY} The counter increments for every transaction that fails with IMP cause.

R_{SFBT} If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not TME_FAILURE_IMP increments the counter.

B2.5.11 TME_FAILURE_MEM

R_{FFTX} The counter increments for every transaction that fails with MEM cause.

R_{ZTDY} If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not TME_FAILURE_MEM increments the counter.

B2.5.12 TME_FAILURE_NEST

R_{LRJQ} The counter increments for every transaction that fails with NEST cause.

R_{QWVR} If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not TME_FAILURE_NEST increments the counter.

B2.5.13 TME_FAILURE_SIZE

R_{LDPC} The counter increments for every transaction that fails with SIZE cause.

R_{BZTQ} If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not TME_FAILURE_SIZE increments the counter.

B2.5.14 TME_FAILURE_TLBI

R_{MXRY} The counter increments for every transaction that fails with IMP cause due to the execution of a TLBI instruction by another PE.

R_{FFTW} If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not TME_FAILURE_TLBI increments the counter.

B2.5.15 TME_FAILURE_WSET

R_{BSJR} The counter increments for every transaction that fails with SIZE cause due to a memory access that causes an eviction of an entry from the transactional write set.

R_{QZHV} If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not TME_FAILURE_WSET increments the counter.

B2.5.16 Behavior on overflow

R_{RGWF} A Performance Monitors counter overflow while in Transactional state behaves the same as in Non-transactional state. For more information, see [1] *Behavior on overflow*.

R_{DGMD} A Performance Monitors counter that is configured to count the TME_INST_RETIRED_COMMITTED or the TME_CPU_CYCLES_COMMITTED events does not set the overflow status bit in PMOVSCLR if the currently executing transaction fails.

R_{PPFR} A Performance Monitors counter that is configured to count the TME_INST_RETIRED_COMMITTED or the TME_CPU_CYCLES_COMMITTED events does not generate an overflow interrupt request in Transactional state.

I_{KHXN} Two possible implementations of this functionality are:

- The implementation accumulates events to the counter directly and sets the overflow status bit when the counter overflows. If the system is programmed to generate an interrupt on overflow, the interrupt is not generated until the transaction commits. If the transaction fails, both the counter and the overflow status bit are restored to the value they had when the transaction started, and no interrupt is generated.
- The implementation accumulates events without updating the counter. If the transaction commits, the counter is updated with the accumulated value. If the counter update overflows the counter value, then the overflow status bit is set, and if the system is programmed to generate an interrupt on overflow, then an interrupt is generated.

Chapter B3
System registers

B3.1 General system control registers

B3.1.1 CTR_EL0

R_{LCCW} **ERG, bits [23:20]**

Exclusives reservation granule, and, if *Transactional Memory Extension* (TME) is implemented, transactional reservation granule. Log_2 of the number of words of the maximum size of the reservation granule for the Load-Exclusive and Store-Exclusive instructions, and, if TME is implemented, for detecting transactional conflicts.

A value of `0b0000` indicates that this register does not provide granule information and the architectural maximum of 512 words (2KB) must be assumed.

Value `0b0001` and values greater than `0b1001` are reserved.

B3.1.2 ID_AA64ISAR0_EL1

R_{VJLM} **TME, bits [27:24]**

Indicates whether TME instructions are implemented. Defined values are:

0000	No TME instructions are implemented.
0001	`TCANCEL`, `TCOMMIT`, `TSTART`, and `TTEST` instructions are implemented.

B3.1.3 TCR_EL1

R_{NJGX} **NFD1, bit [54]**

Present only if SVE or TME is implemented.

Non-fault translation table walk disable for stage 1 translations using TTBR1_EL1.

This bit controls whether to perform a stage 1 translation table walk in response to a non-fault access from EL0 for a virtual address that is translated using TTBR1_EL1.

If SVE is implemented, the affected access types include:

- All accesses due to an SVE non-fault contiguous load instruction.
- Accesses due to an SVE first-fault gather load that are not for the First active element. Accesses due to an SVE first-fault contiguous load instruction are not affected.
- Accesses due to prefetch instructions might be affected, but the effect is not architecturally visible.

If TME is implemented, the affected access types include all accesses generated by a load or store instruction in Transactional state.

Defined values are:

0	Does not disable stage 1 translation table walks using TTBR1_EL1.
1	A TLB miss on a virtual address that is translated using TTBR1_EL1 due to the specified access types causes the access to fail without taking an exception. No stage 1 translation table walk is performed.

If neither SVE nor TME is implemented, this field is RES0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

R_{XNRP} **NFD0, bit [53]**

Present only if SVE or TME is implemented.

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 access from EL0 for a virtual address that is translated using TTBR0_EL1.

If SVE is implemented, the affected access types include:

- All accesses due to an SVE non-fault contiguous load instruction.
- Accesses due to an SVE first-fault gather load that are not for the First active element. Accesses due to an SVE first-fault contiguous load instruction are not affected.
- Accesses due to prefetch instructions might be affected, but the effect is not architecturally visible.

If TME is implemented, the affected access types include all accesses generated by a load or store instruction in Transactional state.

Defined values are:

0	Does not disable stage 1 translation table walks using TTBR0_EL1.
1	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.

If neither SVE nor TME is implemented, this field is RES0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

B3.1.4 TCR_EL2

R_{QWCK} **NFD1, bit [54]**

Present only if SVE or TME is implemented.

Non-fault translation table walk disable for stage 1 translations using TTBR1_EL2.

This bit controls whether to perform a stage 1 translation table walk in response to a non-fault access from EL0 for a virtual address that is translated using TTBR1_EL2.

If SVE is implemented, the affected access types include:

- All accesses due to an SVE non-fault contiguous load instruction.
- Accesses due to an SVE first-fault gather load that are not for the First active element. Accesses due to an SVE first-fault contiguous load instruction are not affected.
- Accesses due to prefetch instructions might be affected, but the effect is not architecturally visible.

If TME is implemented, the affected access types include all accesses generated by a load or store instruction in Transactional state.

Defined values are:

0	Does not disable stage 1 translation table walks using TTBR1_EL2.
1	A TLB miss on a virtual address that is translated using TTBR1_EL2 due to the specified access types causes the access to fail without taking an exception. No stage 1 translation table walk is performed.

If neither SVE nor TME is implemented, this field is RES0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

R_{MWBR} **NFD0, bit [53]**

Present only if SVE or TME is implemented.

Non-fault translation table walk disable for stage 1 translations using TTBR0_EL2.

This bit controls whether to perform a stage 1 translation table walk in response to a non-fault access from EL0 for a virtual address that is translated using TTBR0_EL2.

If SVE is implemented, the affected access types include:

- All accesses due to an SVE non-fault contiguous load instruction.
- Accesses due to an SVE first-fault gather load that are not for the First active element. Accesses due to an SVE first-fault contiguous load instruction are not affected.
- Accesses due to prefetch instructions might be affected, but the effect is not architecturally visible.

If TME is implemented, the affected access types include all accesses generated by a load or store instruction in Transactional state.

Defined values are:

0	Does not disable stage 1 translation table walks using TTBR0_EL2.
1	A TLB miss on a virtual address that is translated using TTBR0_EL2 due to the specified access types causes the access to fail without taking an exception. No stage 1 translation table walk is performed.

If neither SVE nor TME is implemented, this field is RES0.

When this register has an architecturally-defined reset value, this field resets to a value that is architecturally UNKNOWN.

B3.1.5 ISS encoding for an exception from a TSTART instruction

R_{JKTZ} **Bits [24:10]** Reserved, RES0

Rd, Bits [9:5] The Rd value from the issued instruction, the general purpose register used for the destination.

Bits [4:0] Reserved, RES0

B3.1.6 SCTLR_EL1

R_{NBFV} **TMT0, bit [50]**

Forces a trivial implementation of TME at EL0.

The defined values are:

0b0	This control does not cause TSTART instructions to fail.
0b1	When the AArch64 TSTART instruction is executed at EL0, the transaction fails with TRIVIAL cause.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to a value that is architecturally UNKNOWN.

Otherwise:

Reserved, RES0.

R_{QHHT} **TMT, bit [51]**

Forces a trivial implementation of TME at EL1.

The defined values are:

0b0	This control does not cause TSTART instructions to fail.
0b1	When the AArch64 TSTART instruction is executed at EL1, the transaction fails with TRIVIAL cause.

In a system where the PE resets into EL1, this field resets to a value that is architecturally UNKNOWN.

Otherwise:

Reserved, RES0.

R_{YCNC} **TME0, bit [52]**

Enables the AArch64 TSTART instruction at EL0, otherwise traps to EL1.

The defined values are:

0b0	Any attempt at EL0 to execute the AArch64 TSTART instruction is trapped to EL1, (reported with ESR_ELx.EC value 0b011011), subject to the exception prioritization rules, unless HCR_EL2.TME or SCR_EL3.TME causes TSTART instructions to be UNDEFINED at EL0.
0b1	This control does not cause TSTART instructions to be trapped.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to a value that is architecturally UNKNOWN.

Otherwise:

Reserved, RES0.

R_{NQPY} **TME, bit [53]**

Enables the AArch64 TSTART instruction at EL1.

The defined values are:

0b0	Any attempt at EL1 to execute the AArch64 TSTART instruction is trapped to EL1, (reported with ESR_ELx.EC value 0b011011), subject to the exception prioritization rules, unless HCR_EL2.TME or SCR_EL3.TME causes TSTART to be UNDEFINED at EL1.
0b1	This control does not cause TSTART instructions to be trapped.

In a system where the PE resets into EL1, this field resets to a value that is architecturally UNKNOWN.

Otherwise:

Reserved, RES0.

B3.1.7 SCTLR_EL2

R_{HYJD} **TMT0, bit [50]**

When HCR_EL2.{E2H,TGE} is {1,1}, forces a trivial implementation of TME at EL0.

The defined values are:

0b0	This control does not cause TSTART instructions to fail.
0b1	When the AArch64 TSTART instruction is executed at EL0, the transaction fails with TRIVIAL cause.

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

Otherwise:

Reserved, RES0.

R_{TXJP} **TMT, bit [51]**

Forces a trivial implementation of TME at EL2.

The defined values are:

0	This control does not cause TSTART instructions to fail.
1	When the AArch64 TSTART instruction is executed at EL2, the transaction fails with TRIVIAL cause.

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

Otherwise:

Reserved, RES0.

R_{QZST} **TME0, bit [52]**

When HCR_EL2.{E2H,TGE} is {1,1}, enables the AArch64 TSTART instruction at EL0, otherwise traps to EL2.

The defined values are:

0b0	Any attempt at EL0 to execute the AArch64 TSTART instruction is trapped to EL2, (reported with ESR_ELx.EC value 0b011011), subject to the exception prioritization rules, unless HCR_EL2.TME or SCR_EL3.TME causes TSTART instructions to be UNDEFINED at EL0.
0b1	This control does not cause TSTART instructions to be trapped.

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

Otherwise:

Reserved, RES0.

R_{HHMY} **TME, bit [53]**

Enables the AArch64 TSTART instruction at EL2.

The defined values are:

0b0	Any attempt at EL2 to execute the AArch64 TSTART instruction is trapped to EL2, (reported with ESR_ELx.EC value 0b011011), subject to the exception prioritization rules, unless HCR_EL2.TME or SCR_EL3.TME causes TSTART to be UNDEFINED at EL2.
0b1	This control does not cause TSTART instructions to be trapped.

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

Otherwise:

Reserved, RES0.

B3.1.8 SCTLR_EL3

R_{YDGG} **TMT, bit [51]**

Forces a trivial implementation of TME at EL3.

The defined values are:

0	This control does not cause TSTART instructions to fail.
1	When the AArch64 TSTART instruction is executed at EL3, the transaction fails with TRIVIAL cause.

In a system where the PE resets into EL3, this field resets to a value that is architecturally UNKNOWN.

Otherwise:

Reserved, RES0.

R_{ZRLR} **TME, bit [53]**

Enables the AArch64 TSTART instruction at EL3.

The defined values are:

0b0	Any attempt at EL3 to execute the AArch64 TSTART instruction is trapped to EL3, (reported with ESR_ELx.EC value 0b011011), subject to the exception prioritization rules, unless HCR_EL2.TME or SCR_EL3.TME causes TSTART to be UNDEFINED at EL0, EL1 and EL2.
0b1	This control does not cause TSTART instructions to be trapped.

In a system where the PE resets into EL3, this field resets to a value that is architecturally UNKNOWN.

Otherwise:

Reserved, RES0.

B3.1.9 HCR_EL2

R_{WBJM} **TME, bit [39]**

Enables the AArch64 TSTART, TCOMMIT, TTEST and TCANCEL instructions at EL{0,1}.

The defined values are:

0b0	The AArch64 TSTART, TCOMMIT, TTEST and TCANCEL instructions are UNDEFINED at EL{0,1}, and EL1 reads from ID_AA64ISAR0_EL1.TME return 0, when EL2 is enabled in the current Security state.
0b1	This control does not cause these instructions to be UNDEFINED.

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

If EL2 is not implemented or is disabled in the current Security state, the system behaves as if this bit is 1.

Otherwise:

Reserved, RES0.

B3.1.10 SCR_EL3

R_{XYXB} **TME, bit [34]**

Enables the AArch64 TSTART, TCOMMIT, TTEST and TCANCEL instructions at EL{0,1,2}.

The defined values are:

0b0	The AArch64 TSTART, TCOMMIT, TTEST and TCANCEL instructions are UNDEFINED at EL{0,1,2}, and EL{1,2} reads from ID_AA64ISAR0_EL1.TME return 0.
0b1	This control does not cause these instructions to be UNDEFINED.

In a system where the PE resets into EL3, this field resets to an architecturally unknown value.

Otherwise:

Reserved, RES0.

B3.2 Performance Monitors registers

B3.2.1 PMEVTYPER<n>_EL0

R_{DQWK} **T, bit [23]**

Transactional state filtering bit. Controls counting in Transactional state. If TME is not implemented, this bit is RES0. The possible values of this bit are:

0	Count events in Non-transactional state and in Transactional state.
1	Count events in Transactional state only.

B3.2.2 PMCCFILTR_EL0

R_{KCZZ} **T, bit [23]**

Non-transactional state filtering bit. Controls counting in Non-transactional state. If TME or PMUv3 are not implemented, this bit is RES0. The possible values of this bit are:

0	Count cycles in Non-transactional state and in Transactional state.
1	Count cycles in Transactional state only.

This bit resets to an architecturally UNKNOWN value on a reset.

B3.2.3 PMSEVFR_EL1

R_{RHWX} **E, bit [16]**

Transactional. The possible values of this bit are:

0	Transactional event is ignored.
1	Do not record samples that have event 16 (Transactional) == 0.

This bit is ignored by the PE when PMSFCR_EL1.FE == 0.

This bit resets to an architecturally UNKNOWN value on a reset.

B3.3 Performance Monitors external registers

B3.3.1 PMPCSR

R_{RCJT} **T, bit [60]**

Transactional state of the sample. Indicates the Transactional state that is associated with the most recent PMPCSR sample or, when it is read as a single atomic 64-bit read, the current PMPCSR sample.

0	Sample is from Non-transactional state.
1	Sample is from Transactional state.

This field resets to a value that is architecturally UNKNOWN.

Chapter B4
Instructions

Transactional Memory Extension (TME) adds the following instructions.

B4.1 TCANCEL

R_{VPTY} The TCANCEL instruction exits Transactional state and discards all state modifications that are due to instructions that were executed transactionally.

R_{DSCF} Execution continues at the instruction that follows the TSTART instruction of the outer transaction.

R_{YFDC} The destination register of the TSTART instruction of the outer transaction is written with the immediate operand of TCANCEL.

```
TCANCEL #<imm>
```

B4.2 TCOMMIT

R_{VVNT} The TCOMMIT instruction commits the current transaction.

R_{YHKK} If the current transaction is an outer transaction, then Transactional state is exited, and all state modifications due to instructions that were executed transactionally are committed to the architectural state.

R_{XJVQ} TCOMMIT takes no inputs and returns no value.

R_{SJJW} Execution of TCOMMIT is UNDEFINED in Non-transactional state.

```
TCOMMIT
```

B4.3 TSTART

R_{WBJV} This instruction starts a new transaction.

R_{NXXP} If the transaction started successfully, the destination register is set to zero.

R_{VNLM} If the transaction failed or was canceled, then all state modifications that are due to instructions that were executed transactionally are discarded and the destination registers is written with a non-zero value that encodes the cause of the failure.

```
TSTART <Xd>
```

B4.4 TTEST

R_{FZLN} The TTEST instruction takes no inputs.

R_{VLYG} The TTEST instruction writes the depth of the transaction to the destination register, or the value 0 otherwise.

```
TTEST <Xd>
```

Chapter B5
Interaction with Memory Tagging Extension

This section describes the interaction of *Transactional Memory Extension* (TME) with the Memory Tagging Extension.

R_{SKRL} The MTE instructions for Tag generation, Tag setting and getting, are allowed within a transaction. This means in particular that the accesses to GCR_EL1 and RGSR_EL1 stemming from the MTE instructions are allowed within a transaction, but it is IMPLEMENTATION DEFINED whether they are checkpointed.

R_{MYGP} In the case of an asynchronous Tag Check Failure within a Transaction:

- Tag check failures configured to asynchronously accumulate failure status should not expect transaction failure with ERR cause.
- If the transaction succeeds then reading TFSR_ELx.TFy status determines if there are any errors.

Part C
Appendixes

Chapter C1
Transactional Memory Extension (TME) Litmus tests

This appendix is to help understand, via examples, how transactions extend the Armv8 memory model.

See also Section B1.4 *Memory model*, [1] *Definition of the Armv8 memory model*, and [1] *Barrier Litmus Tests*.

C1.1 Conventions

Many of the examples are written in a stylized extension to Arm assembler, to avoid confusing the examples with unnecessary code sequences, using the same conventions as [1] *Load-Acquire Exclusive, Store-Release Exclusive and barriers*. In addition, we define the following constructs.

The construct `TX{<code>}` describes the following sequence:

```
loop:
TSTART X12 ; attempt to start a new transaction
CBNZ X12, loop ; retry forever
<code>
TCOMMIT
```

Note: This construct is unsafe in the general case because a transaction is permitted to never commit and should be avoided. But, for the simple examples that are presented in this section it is expected that an implementation will be able to commit the transaction eventually.

C1.2 Transaction strong isolation

TME transactions are *strongly isolated*. Strongly isolated transactions require both *non-interference* and *containment* from other transactions as well as from non-transactional code executing concurrently.

C1.2.1 Containment

The containment property of transactions means that only the last write to a Location is observable outside of the transaction:

P1	P2
`LDR W5, [X1]`	`TX { STR W5, [X1] STR W6, [X1] }`

In this example, the result of `P1:W5==0x55` is not permissible.

C1.2.2 Non-interference

The non-interference property of transactions means that multiple reads to the same memory Location inside a transaction should return the same value:

P1	P2
`STR W5, [X1]`	`TX { LDR W5, [X1] LDR W6, [X1] }`

In this example, it is required for `P2:W5` and `P2:W6` to contain the same value.

The non-interference property of transactions also means that a read to Location following a write to the same Location inside a transaction should return the value of the write:

P1	P2
`STR W5, [X1]`	`TX { STR W6, [X1] LDR W5, [X1] }`

In this example, it is required that `P2:W5==0x66`.

C1.3 Transactions and barriers

The following sections show that most of the examples in [1] *Simple ordering and barrier cases* can be achieved using transactions without the need for additional barriers.

C1.3.1 Simple weakly consistent ordering

The *simple weakly consistent ordering* example in [1] *Simple ordering and barrier cases* can be solved by the use of transactions in various ways. In the following examples, the result of `P1:W6==0`, `P2:W5==0` is not permissible.

Memory accesses after the transaction cannot be observed by other observers before the transaction:

P1	P2
`TX { STR W5, [X1] } LDR W6, [X2]`	`TX { STR W6, [X2] } LDR W5, [X1]`

Memory accesses before the transaction cannot be observed by other observers after the transaction:

P1	P2
`STR W5, [X1] TX { LDR W6, [X2] }`	`STR W6, [X2] TX { LDR W5, [X1] }`

An empty transaction behaves like a barrier instruction:

P1	P2
`STR W5, [X1] TX {} LDR W6, [X2]`	`STR W6, [X2] TX {} LDR W5, [X1]`

C1.3.2 Message passing

The *weakly-ordered message passing* problem in [1] *Simple ordering and barrier cases* can be solved by the use of transactions in various ways. In the following examples, the result of `P2:W5==0` is not permissible.

Using a transaction when accessing the data:

P1	P2
`TX { STR W5, [X1] } STR W0, [X2]`	`WAIT([X2]==1) AND X12, X12, #0` `LDR W5, [X1, X12]`

An empty transaction behaves like a barrier instruction:

P1	P2
`STR W5, [X1] TX {} STR W0, [X2]`	`WAIT([X2]==1) AND X12, X12, #0` `LDR W5, [X1, X12]`

These approaches also work with multiple observers, viz, with extra observers running the same sequence as P2.

Chapter C2
Transactional Memory Extension (TME) Transactional Lock Elision

C2.1 Overview

Contended locks can lead to software scalability problems on multicore systems. Hardware Transactional Memory may improve the scalability of contended locks by implementing transactional lock elision.

With transactional lock elision, critical regions are converted into transactions and multiple threads can execute their critical regions in parallel as long as there are no transactional conflicts. When such conflicts occur, the implementation resolves them by failing transactions as necessary.

Failed transactions must re-execute the critical region for the application to progress, but since transactions are best effort, a fallback execution path is necessary. In the case of transactional lock elision, typically the fallback path acquires a lock and executes the critical region non-transactionally.

The most popular implementations of transactional lock elision use the same programming model as locks, so they can be applied to existing programs.

Arm notes however that not all locking libraries are equal with respect to lock elision. Certain existing libraries cannot be elided soundly (see below for an example) and will need reviewing or perhaps revisiting entirely (e.g. by adding extra barriers or using atomic operations) if they are intended to be used in this context. This is the case both for the surrounding locks on other threads, and the lock used as the fallback path.

C2.2 Conventions

Many of the examples are written in a stylized extension to Arm assembler, to avoid confusing the examples with unnecessary code sequences, using the same conventions as the *Conventions* topic in the [1] *Barrier Litmus Tests* chapter. In addition, we define the following constructs.

The construct `LOCK(Xx)` describes the following sequence from [1] *Acquiring a lock*:

```
PRFM PSTL1KEEP, [Xx] ; preload into cache in unique state
loop:
LDAXR W5, [Xx] ; read lock with acquire
CBNZ W5, loop ; check if 0
STXR W5, W0, [Xx] ; attempt to store new value
CBNZ W5, loop ; test if store succeeded and retry if not
```

The construct `UNLOCK(Xx)` describes the following sequence from [1] *Releasing a lock*:

```
STLR WZR, [Xx] ; clear the lock with release semantics
```

The construct `CHECK(Xx)` describes the following sequence:

```
LDR W5, [Xx] ; read lock
```

The construct `CHECK_ACQ(Xx)` describes the following sequence:

```
LDAR W5, [Xx] ; read lock with acquire
```

The construct `WAIT_ACQ(Xx==0)` describes the following sequence:

```
loop:
LDAR W5, [Xx] ; load acquire ensures it is ordered before subsequent loads/stores
CBNZ W5, loop
```

In the rest of this chapter, the `LOCK(Xx)` construct is used as one example of lock acquisition, the `UNLOCK(Xx)` construct is used as one example of lock release, the `CHECK(Xx)` and `CHECK_ACQ(Xx)` constructs are used as one example of reading the status of a lock, and the `WAIT_ACQ(Xx==0)` construct is used as one example of waiting until the lock is free. Unless otherwise stated, the examples where these constructs are used would work the same if these constructs were mapped to different locking primitives, such as but not limited to the ones presented in [1] *Ticket locks* and [1] *Use of Wait For Event (WFE) and Send Event (SEV) with locks*.

C2.3 Acquiring a lock

The recommended instruction sequence for acquiring a lock using transactional lock elision is as follows (where `W6` contains a retry count for the transaction):

```
loop:
TSTART X5 ; attempt to start a new transaction
CBNZ X5, fallback ; check if TSTART succeeded
CHECK_ACQ(X1) ; add the fallback lock to the transactional read set ; and set W5 to 0 if the fallback lock is free.
CBZ W5, enter ; if the fallback lock is free enter the critical region
TCANCEL #0xFFFF ; otherwise cancel the transaction with RTRY set to 1
fallback:
TBZ X5, #15, lock ; if RTRY is 0 take the fallback lock
SUB W6, W6, #1 ; decrement the retry count
CBZ W6, lock ; take the lock if 0
WAIT_ACQ(X1==0) ; wait until the lock is free
B loop ; retry the transaction
lock:
LOCK(X1) ; elision failed, acquire the fallback lock
DMB ISH ; block loads/stores from the critical region
enter:
```

C2.3.1 Checking the lock inside the transaction

When eliding a lock, it is required to check the status of the lock inside the transaction because the memory accesses of a thread that executes the critical region non-transactionally are not tracked by hardware. In the following example, mutual exclusion cannot be guaranteed:

P1	P2
`LOCK(X1) LDR W5, [X2]`	`TSTART X5 CBNZ X5, fallback`
`ADD W5, W5, #1 STR W5, [X2]`	`LDR W5, [X2] ADD W5, W5, #1`
`UNLOCK(X1)`	`STR W5, [X2] TCOMMIT`

To ensure mutual exclusion when the transaction by P2 commits after the load from the address in `X2` by P1 and before the store to the address in `X2` by P1, P2 must ensure that the lock variable is contained within the transactional read set, which occurs as a side effect of adding the `CHECK_ACQ(Xx)` construct inside the transaction.

To ensure mutual exclusion when the transaction by P2 starts after the `LOCK(Xx)` construct by P1 and the transaction in P2 commits before the store to the address in `X2` by P1, P2 must test the value of the lock returned by the `CHECK_ACQ(Xx)` construct and cancel the transaction as appropriate.

Using `CHECK_ACQ(Xx)` instead of `CHECK(Xx)` ensures that read from a critical region executing in a transaction do not take their values from writes from a mutually excluded critical region that acquires a lock, including when the acquisition of the lock generates a conflict that fails the transaction.

C2.3.2 Checking the lock at the fallback path

When eliding a lock, it is recommended to use the `WAIT_ACQ(Xx==0)` construct in the fallback handler to avoid the Lemming effect, in which one thread acquiring the lock causes all other concurrent threads to do so too because they retry too many times while the first thread holds the lock.

C2.3.3 Synchronization between transactions and the fallback path

When transactional lock elision fails, and the lock is acquired, it is required that loads and stores from the critical region are not observable before the lock is acquired.

In the following example, mutual exclusion cannot be guaranteed:

P1	P2
`LOCK(X1) LDR W5, [X2]`	`TSTART X5 CBNZ X5, fallback`
`ADD W5, W5, #1 STR W5, [X2]`	`CHECK_ACQ(X1) CBNZ W5, cancel`
`UNLOCK(X1)`	`LDR W5, [X2] ADD W5, W5, #1`
	`STR W5, [X2] TCOMMIT`

The following architecturally permissible ordering violates mutual exclusion:

- P1 performs the Load-Exclusive of the `LOCK(Xx)` construct.
- P1 performs the load of the shared variable from the critical region (reading 0).
- P2 enters the transaction, executes the critical region writing 1 to the shared variable, and commits the transaction (because the lock is not acquired yet).
- P1 performs the Store-Exclusive of the `LOCK(Xx)` construct. The Store-Exclusive does not fail because there are no intervening writes to the lock variable (P2 only reads the lock)
- P1 performs the store of the shared variable from the critical region (writing 1).

To ensure mutual exclusion when the fallback lock acquisition implementation permits reads or writes from the critical region to be observable before the lock variable is updated, a DMB is added before the first load or store of the critical region.

All the recommended locking acquisition sequences from [1] *Load-Acquire Exclusive, Store-Release Exclusive and barriers* that use Load-Exclusive and Store-Exclusive to acquire the lock are affected.

Lock implementations that use an atomic operation with Acquire or Acquire-Release semantics (such as `LDADDA`, `SWPA`, etc.) to update the lock variable are not affected.

C2.4 Releasing a lock

The recommended instruction sequence for releasing a lock using transactional lock elision is as follows:

```
CHECK(X1) ; set W5 to 0 if the fallback lock is free
CBNZ W5, unlock ; check if 0
TCOMMIT ; the lock was elided, exit the transaction
B exit
unlock:
UNLOCK(X1) ; elision failed, release the fallback lock
exit:
```

C2.4.1 Elision and nesting

When releasing a lock that has potentially been elided it is advisable to use the `CHECK(Xx)` construct to check if the lock is acquired instead of using `TTEST` to check if the PE is in Transactional state, because inside a nested transaction using `TTEST` is not sufficient to distinguish if the lock was elided or not.

Chapter C3
Transactional Memory Extension (TME) Implementation recommendations

C3.1 Permitted architectural difference between PEs

The architecture does not support implementations where the value of ID_AA64ISAR0_EL1.TME differs between PEs in a single system.

C3.2 Individual operation latency

In order to not affect single-thread performance when using transactional lock elision, Arm recommends that the latency of starting and committing a transaction is not higher than the latency of the illustrative code sequence for acquiring and releasing a spinlock.

In an application that successfully employs transactional lock elision, it is expected that most transactions will not fail, so it acceptable that failing or canceling a transaction is a slower operation than committing a transaction. Even so, in order to not affect single-thread performance, Arm recommends that the latency of failing or canceling a transaction is not unreasonably high compared to the latency of committing a transaction.

C3.3 Read and write set capacity

Arm recommends that, for adequate performance of applications written in Java and C/C++, hardware supports a read set size of at least 512 objects and a write set size of at least 300 objects – assuming average object size to be 128 bytes.

C3.4 State tracking

The properties of the transactional read set and the transactional write set imply that the implementation tracks the addresses of transactional reads and writes and buffers the data values of transactional writes throughout the execution of a transaction.

Arm expects a typical transaction to execute between a few tens to a few thousand instructions and to access up to several hundreds or even thousands of distinct transactional reservation granules.

Arm expects the transactional write set to contain a significantly smaller number of transactional reservation granules compared to the transactional read set of a typical transaction.

Arm expects the capacity of a typical Level 1 data or unified cache to be enough to hold the transactional write set, but not enough to hold the transactional read set in many cases.

Arm expects the associativity of a typical Level 1 data or unified cache to not be enough to hold the transactional write set or the transactional read set in many cases.

Arm considers a typical Level 1 data or unified cache to have a capacity between 32KB and 64KB with an associativity between 2 to 4.

Arm recommends that implementations take these expectations into consideration in order to avoid frequent transactional failures due to insufficient hardware resources.

For holding the transactional write set, Arm recommends the use of hardware structures in addition to the Level 1 data or unified cache that can provide the illusion of high associativity, such as a small fully associative cache.

For holding the transactional read set, Arm recommends the use of hardware structures in addition or instead of the Level 1 data or unified cache that are capable of holding the addresses of tens of thousands of transactional reservation granules, such as higher-level caches, Bloom filters, Signatures, or other similar structures.

C3.5 Transactional conflicts

Arm recommends that the hardware cache coherency facilities of the processor be used to detect transactional conflicts. This is also known as *eager conflict detection* because conflicts are detected when the read or write requests are generated. The alternative, *lazy conflict detection*, defers the detection of conflicts until the transaction attempts to commit.

Arm recommends that implementations do not generate a transactional conflict when a read generated by a PRFM instruction or by hardware prefetching accesses a Location within the transactional write set of a transaction.

Chapter C4
Stages of execution

This section shows the relationship between the *stages of execution*. The terms are defined in the Glossary.

C4.1 Stages of execution without *Transactional Memory Extension* (TME)

I_{KHCRN} Figure C4.1 shows the stages of execution in a PE that does not implement FEAT_TME.

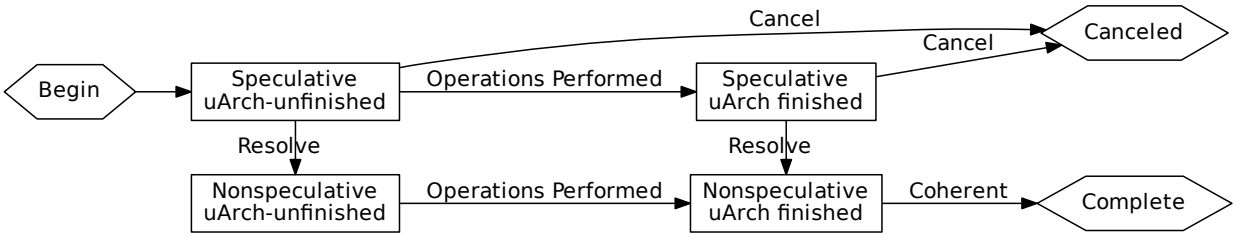

Figure C4.1: Stages of execution without TME

C4.2 Stages of execution with TME

I_{BBNYK} Figure C4.2 shows the stages of execution in a PE that does implement FEAT_TME.

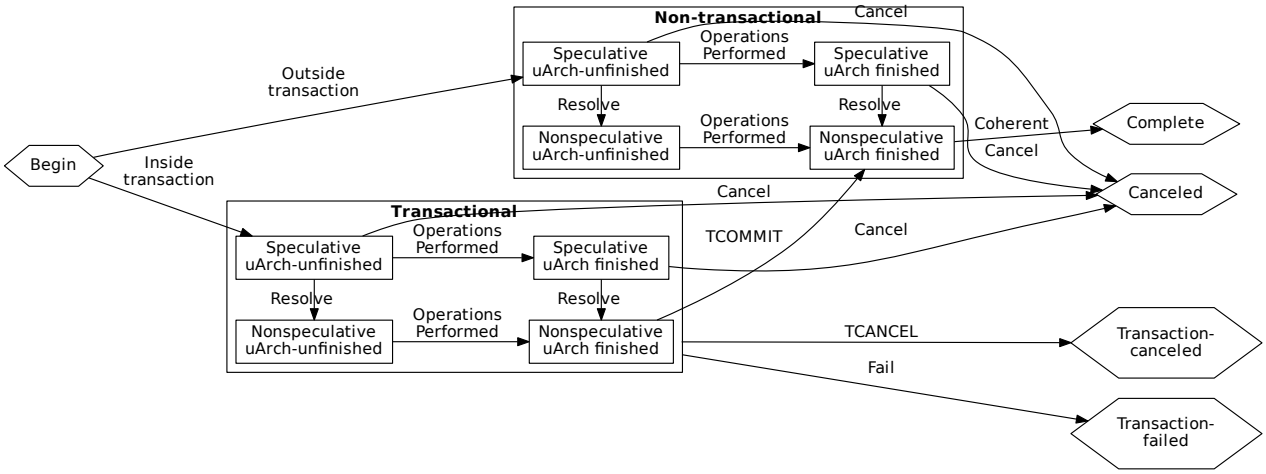

Figure C4.2: Stages of execution with TME

Part D
Glossary

Chapter D1
Glossary

Canceled

An operation on an incorrectly predicted execution path.

Complete

An operation that has finished all its operational pseudocode, and the results of any memory accesses, including translation table walks and updates, are coherent with other observers. For more information see the [1].

Microarchitecturally-finished

An operation that has finished all its operational pseudocode, although the results of any memory accesses, including translation table walks and updates, are not yet coherent with other observers.

Microarchitecturally-unfinished

An operation that has not completed all its operational pseudocode.

Nonspeculative

An operation on a confirmed execution path.

Nonspeculative Microarchitecturally-finished

An operation that has finished all its operational pseudocode, on a confirmed execution path, although the results of any memory accesses, including translation table walks and updates, are not yet coherent with other observers and the operation is not Complete.

Nonspeculative Microarchitecturally-unfinished

An operation that is in progress on a confirmed execution path.

Speculative

An operation on a predicted execution path. For more information see the [1].

Speculative Microarchitecturally-finished

An operation that has finished all its operational pseudocode, on a predicted execution path.

Speculative Microarchitecturally-unfinished

An operation that is in progress on a predicted execution path.

TME

Transactional Memory Extension

Transaction-canceled

An operation that was part of a transaction that was canceled by a `TCANCEL` instruction.

Transaction-failed

An operation that was part of a transaction that failed.

Transactional

An operation that is part of a transaction and the transaction has not yet succeeded, failed or been canceled. The operation can be any of:

- Speculative Microarchitecturally-unfinished.
- Speculative Microarchitecturally-finished.
- Nonspeculative Microarchitecturally-unfinished.
- Nonspeculative Microarchitecturally-finished.