



Arm Architectural Structured Trace Format

Revision: v0.11

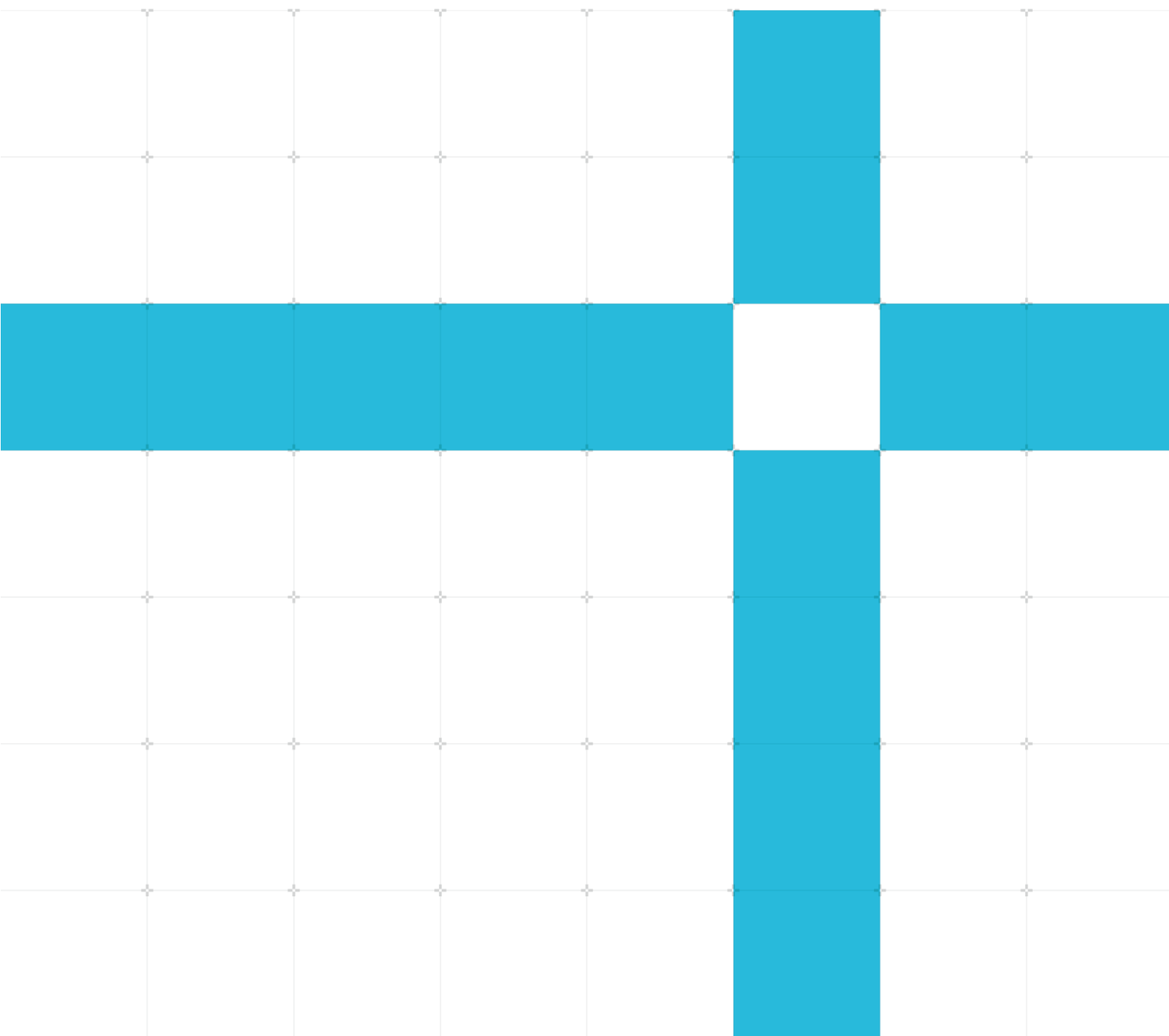
Specification

Confidential

Copyright © 2020-2023 Arm Limited (or its affiliates).
All rights reserved.

Issue 0.11r1

107637



Arm Architectural Structured Trace Format Specification

Copyright © 2020-2023 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
0.10r0	2020/04/08	Arm Internal	Initial 0.10 draft specification
0.10r1	2021/11/12	Arm Confidential	Clarified trapping memory accesses due to Data Aborts. Clarified Timestamps.
0.10r2	2022/06/30	Arm Confidential	Reviewed and reformatted for external publication. Clarified Timestamps around suspend/resume.
0.11r0	2023/08/08	Arm Confidential	Extended support to Armv9.3-A and SME2 architecture.
0.11r1	2023/11/06	Arm Confidential	Clarified SVE faulting memory operations, failing ST64BV0 behavior, improved text and fixed errata

Confidential Proprietary Notice

This document is **CONFIDENTIAL** and any use by you is subject to the terms of the agreement between you and Arm or the terms of the agreement between you and the party authorized by Arm to disclose this document to you.

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information: **(i)** for the purposes of determining whether implementations infringe any third party patents; **(ii)** for developing technology or products which avoid any of Arm's intellectual property; or **(iii)** as a reference for modifying existing patents or patent applications or creating any continuation, continuation in part, or extension of existing patents or patent applications; or **(iv)** for generating data for publication or disclosure to third parties, which compares the performance or functionality of the Arm technology described in this document with any other products created by you or a third party, without obtaining Arm's prior written consent.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR

CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm's trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>.

Copyright © 2020-2023 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.
110 Fulbourn Road, Cambridge, England CB1 9NJ.
(LES-PRE-20348)

Confidentiality Status

This document is Confidential. This document may only be used and distributed in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Product Status

The information in this document is for a product under development.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on Arm Architectural Structured Trace Format, create a ticket on <https://support.developer.arm.com>.

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

Inclusive language commitment

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

Contents

1	Introduction.....	6
1.1	Product revision status.....	6
1.2	Intended audience	6
1.3	Conventions.....	6
1.3.1	Glossary.....	6
1.3.2	Typographical conventions	7
1.4	Additional reading	8
2	Introduction to traces	9
2.1	Introduction to ASTF	9
2.2	Limitations.....	10
3	Trace levels.....	12
3.1	Basic.....	12
3.2	BasicPlus	13
3.3	Full.....	13
4	Trace format	14
4.1	Record types.....	14
4.2	Header and file format	14
4.2.1	Trace storage and compression	16
4.3	Sanitized virtual addresses.....	16
4.4	Context changes.....	17
4.4.1	PSTATE.....	18
4.4.2	Unsupported execution mode.....	18
4.5	Instruction execution	19
4.5.1	Branch instructions	20
4.5.2	Memory-accessing instructions.....	20
4.5.3	Trapping instructions.....	23
4.6	Extra EA.....	25
4.7	Traps.....	27
4.7.1	Synchronous exception without instruction	27
4.7.2	Synchronous exception or trap with instruction	28

4.7.3	Asynchronous trap, exception or interrupt	28
4.7.4	Exception return.....	28
4.7.5	PE reset.....	29
4.8	String, String continuation	30
4.9	Timestamp	31
5	Example trace	33
5.1	Detailed record contents	34
5.1.1	Record 1: Header	34
5.1.2	Record 2 and 3: String metadata	35
5.1.3	Record 4: Timestamp	35
5.1.4	Record 5: Context.....	36
5.1.5	Record 6: Instruction	36
5.1.6	Record 7: Instruction, branch not-taken.....	37
5.1.7	Record 8: Instruction, load-pair merged access	37
5.1.8	Record 9, 10, and 11: Instruction, memory set with tag.....	38
5.1.9	Record 12: Instruction, branch taken	39
5.1.10	Record 13 and 14: Instruction and Extra EA, load-pair fragmented	40
5.1.11	Record 15: Instruction, store-pair with Data Abort	41
5.1.12	Record 16: Trap, synchronous exception	41
5.1.13	Record 17: Timestamp	42
5.1.14	Record 18: Context switch to EL1.....	42
5.1.15	Record 19: Instruction enabling SSVE mode.....	43
5.1.16	Record 20: Context for new SSVE state.....	43
5.1.17	Record 21: Instruction, indexed SME access.....	44

1 Introduction

1.1 Product revision status

The x.yrz identifier indicates the revision status of the product described in this book, for example, 0.11r0, where:

x.y	identifies the major.minor version of the trace format layout, for example, 0.11
rz	Identifies the minor revision of the specification and rules for the trace format, for example r0.

1.2 Intended audience

This specification is written for hardware and software engineers who want to become familiar with ASTF in order to develop tools that can read or write architectural execution traces in this format.

1.3 Conventions







The following subsections describe conventions used in Arm documents.

1.3.1 Glossary

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

See the Arm Glossary for more information: <https://developer.arm.com/glossary>.

1.3.2 Typographical conventions

Convention	Use
<i>italic</i>	Citations.
bold	Interface elements, such as menu names. Signal names. Terms in descriptive lists, where appropriate.
monospace	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
monospace bold	Language keywords when used outside example code.
monospace <u>underline</u>	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
<and>	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: <pre>MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2></pre>
SMALL CAPITALS	Terms that have specific technical meanings as defined in the Arm® Glossary. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.
 Caution	Recommendations. Not following these recommendations might lead to system failure or damage.
 Warning	Requirements for the system. Not following these requirements might result in system failure or damage.
 Danger	Requirements for the system. Not following these requirements will result in system failure or damage.
 Note	An important piece of information that needs your attention.
 Tip	A useful tip that might make it easier, better, or faster to perform a task.
 Remember	A reminder of something important that relates to the information you are reading.

1.4 Additional reading

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

Table 1-1: Arm publications

Document name	Document ID	Licensee only
Arm Architecture Reference Manual, for A-profile architecture	DDI 0487J.a	No



Arm tests its PDFs only in Adobe Acrobat and Acrobat Reader. Arm cannot guarantee the quality of its documents when used with any other PDF reader.
Adobe PDF reader products can be downloaded at <http://www.adobe.com>.

2 Introduction to traces

Instruction traces provide a way to capture the dynamic execution of a workload, so that the trace can be analyzed and run on performance models or other stand-alone analysis tools. Such traces capture the architectural execution of the workload by recording the instruction PC, instruction encoding, and virtual addresses accessed, for each dynamic instance of an instruction as it was executed. A trace may also record additional information about the architectural state and events in the machine, for example traps or exceptions, register state, memory mappings, memory state, and I/O.

Traces are obtained from the execution of the workload of interest on the target platform of interest, which is assumed to be an Armv8-A or Armv9-A AArch64 platform in this document. Traces can be generated by running the workload on a (functional) simulator, (dynamic) binary instrumentation, or from other pre-silicon environments. Special tools will then intercept all (or some of) the instructions executed and their associated information, and generate an output file with the information required, following the defined trace format.

Such a trace file can then be used to drive a CPU performance model which has been enhanced to not only execute binary programs using the execute-driven simulation paradigm, but also to support trace-driven simulation. Additionally, simple analysis programs and stand-alone simulators can be written to consume trace files to further study the behavior and properties of the workload. For example, an analyzer could gather instruction frequencies, study ILP limits, correlate memory access patterns, or be used to explore branch predictor algorithms. Traces are a powerful method to capture regions of interest of large, complex workloads, that require a lot of effort to set up and run, and to create a representation with a high re-use value.

Traces can be obtained from multiple different sources that might capture different levels of detail. Also, they might be consumed by tools that are aimed at analyzing different details and features. Therefore, it is important that a trace format is designed to be able to encapsulate these different levels of representation. This was one of the design goals of the Architectural Structured Trace Format (ASTF).

2.1 Introduction to ASTF

The Architectural Structured Trace Format has been designed to be a modular trace format to capture workloads that run on the Armv8-A and Armv9-A AArch64 architectures. An ASTF trace consists of a stream of fixed size, typed, records, which capture different aspects of the architectural execution of the workload, such as the instruction that was retired, whether a trap was taken, and whether the execution context of the processor changed. Each trace represents the architectural flow of instructions as observed by a single processing element (PE). When tracing a workload on a multi-core or SMT system, each PE would produce an individual trace stream.

The approach of fixed size, typed records gives ASTF several interesting properties:

- It allows traces to represent different levels of detail.
- Typed records allow simulators, analyzers, or stand-alone models to easily filter relevant information from a trace, or to skip ahead through it.

- Fixed-size records with a binary layout provide additional encoding space for extensions and new record types for future architectural features. This provides a strong forward and backward compatibility guarantee.
- The fixed size organization is very compression-friendly.

The design of the ASTF records has been balanced between providing a trace format that is convenient to interpret without fully decoding every instruction, as well as one that can be stored efficiently. While (performance) models and analyzers operate on streams of raw ASTF records, the ASTF trace reading/writing library, `libastf`, can transparently compress and decompress stored traces.

This definition of ASTF has been designed to support all relevant features of only the 64-bit subset (AArch64) of the Armv9.3-A architecture, including FEAT_SME2 support.

2.2 Limitations

ASTF traces cannot be generated from hardware tracing features such as ETM, FEAT_ETE or Arm CoreSight IP. These tracing features cannot provide sufficient information about memory accesses to create an ASTF trace.

ASTF only supports the 64-bit A-class architecture and does not currently support 32-bit modes. No AArch32-specific architecture extensions are supported. It does not support R-class or M-class architectures.

ASTF does not currently support a complete set of virtualization features related to FEAT_VHE and further virtualization extensions, as it currently does not support different VMIDs. Running a host operating system in EL2 through the use of `HCR_EL2.E2H` is supported.

ASTF does not currently support the FEAT_RME Realm Management Extension.

ASTF does not currently support the FEAT_TME Transactional Memory Extension.

ASTF is designed to capture the architectural execution of common applications and is not intended to be architecturally complete, or to record debug or microarchitectural effects. Therefore, features that expose such effects cannot be represented in an ASTF trace, such as Debug, RAS, Performance Monitors and Statistical Profiling. A comprehensive list of such features, up to the Armv9.3 architecture, is shown in [Table 2](#).

The presence of any non-supported features in a platform that a trace is obtained from is not a problem in itself. If the workload being traced accesses, uses, or interacts with these features, the resulting representation in the trace is UNDEFINED, and not covered by this specification.

ASTF traces are not suitable for exploring design parameters that have architecture-visible effects, such as the (performance) effects of different SVE (or SME) vector lengths.

Table 2: List of architectural features not supported in ASTF related to Debug, RAS, Performance Monitors and Statistical Profiling

Feature	Description
FEAT_DoubleLock	Armv8.0 OS Double Lock
FEAT_PCSRv8	Armv8.0 PC Sample-based Profiling Extension

Feature	Description
FEAT_PMUv3	Armv8.0 Performance Monitors Extension version 3
FEAT_PMUv3_EXT	Armv8.0 External PMUv3
FEAT_ETMv4	Armv8.0 Embedded Trace Macrocell architecture
FEAT_TRC_EXT	Armv8.0 External access for trace registers
FEAT_TRC_SR	Armv8.0 System Register trace registers
FEAT_PMUv3p1	Armv8.1 Performance Monitors Extension version 3.1
FEAT_Debugv8p2	Armv8.2 Debug
FEAT_PCSRv8p2	Armv8.2 PC Sample-based Profiling
FEAT_RAS	Armv8.2 Reliability, Availability, Serviceability Extension
FEAT_SPE	Armv8.2 Statistical Profiling Extension
FEAT_SPEv1p1	Armv8.3 Statistical Profiling Extensions
FEAT_DoPD	Armv8.3 Debug over Powerdown
FEAT_Debugv8p4	Armv8.4 Debug Extensions
FEAT_TRF	Armv8.4 Self-hosted Trace Extensions
FEAT_PMUv3p4	Armv8.4 PMU Extensions v3.4
FEAT_RASv1p1	Armv8.4 RAS Extension v1.1
FEAT_DoubleFault	Armv8.4 Double Fault Extension
FEAT_AMU	Armv8.4 Activity Monitors Extension
FEAT_MPAM	Armv8.4 Memory Partitioning and Monitoring Extension
FEAT_PMUv3p5	Armv8.5 PMU Extensions v3.5
FEAT_AMUv1p1	Armv8.6 Activity Monitor Extensions v1.1
FEAT_MTPMU	Armv8.6 Multi-threaded PMU Extensions
FEAT_PMUv3p7	Armv8.7 PMU extensions
FEAT_SPEv1p2	Armv8.7 SPE features
FEAT_PMUv3p8	Armv8.8 PMU extensions
FEAT_PMUv3_TH	Armv8.8 PMU Event Counting Threshold
FEAT_SPEv1p3	Armv8.8 Statistical Profiling Extensions
FEAT_Debugv8p8	Armv8.8 Debug
FEAT_PMUv3_EXT64	Armv8.8 64-bit external PMU programmers' model extension
FEAT_ETE	Armv9.0 Embedded Trace Extension
FEAT_TRBE	Armv9.0 Trace Buffer Extension
FEAT_ETE1p1	Armv9.1 Embedded Trace Extension
FEAT_ETE1p2	Armv9.2 Embedded Trace Extension
FEAT_BRBE	Armv9.2 Branch Record Buffer Extension
FEAT_BRBE1p1	Armv9.3 Branch Record Buffer Extension

3 Trace levels

This section details the standardized trace levels that can be represented in ASTF traces. As the format is modular, many different levels of representation can exist with different trace features present. An ASTF trace has a set of feature flags recorded in the header which indicates which ASTF features and record types are present in this trace. ASTF supports three different levels of traces, each of which contains increasingly more detail and information about the traced execution:

1. Basic
2. BasicPlus
3. Full

3.1 Basic

Basic level traces are defined as the bare minimum that is required to support trace-driven CPU performance models. Basic traces only contain instructions executed in a single exception level, usually EL0, which corresponds to capturing instructions of user-space execution only.



Basic traces do not require execution to happen at EL0; A bare metal workload that does not switch exception levels and fully executes in EL3_s, can be represented in a Basic trace.

A Basic trace is required to use the following record types (see [Section 4.1](#)):

- Header
- Instruction
- Extra EA
- Context

The use of the following record types is optional:

- String, String continuation
- Timestamp (requires the `timestamps` feature flag to be set)

Basic traces must have the `single_el` feature flag set.

Basic traces do not need to guarantee full PC continuity of execution in EL0. However, they must provide a Context record when a PC discontinuity occurs, to indicate the point where EL0 execution resumes. This applies to all system calls and system call returns, as well as any synchronous or asynchronous traps, exceptions, or interrupts that change the execution to a different location than the PC that was interrupted, when returning to EL0. If Timestamp records are used, a timestamp should be provided before the Context record to indicate when the EL0 execution resumes, if it would be different from the last seen Timestamp.

Basic traces do not need to record instructions whose execution was not completed due to an exception.

3.2 BasicPlus

BasicPlus level traces extend the capability of Basic traces by supporting full trap information, as well as tracing the instruction flow through all exception levels. BasicPlus traces, in contrast to Basic traces, do provide full PC continuity by providing information on how the PC is redirected on trap or exception entry and return.

A BasicPlus trace is required to use the following record types (see [Section 4.1](#)). All except Trap are also present in Basic traces:

- Header
- Instruction
- Extra EA
- Context
- Trap

The use of the following record types is optional:

- String, String continuation
- Timestamp (requires the `timestamps` feature flag to be set)

BasicPlus traces must not have the `single_el` feature flag set.

BasicPlus traces should record the attempted execution of an instruction that trapped, such as memory instructions that page fault on their data access, if the trace source is able to provide this information.

3.3 Full

Out of the scope of this version of the format specification. Further support for recording memory mappings and register and memory values is still under design and review for ASTF.

4 Trace format

This section describes the format of an ASTF trace. A trace consists of a stream of records of various types. Each record occupies 32 bytes, where the first byte of the record identifies the record type. An example of a short trace is shown in [Section 5](#).

4.1 Record types

The record types defined by this version of ASTF are shown in [Table 3](#). The detailed layout of each record type is documented in the following sections. The `Id` is the hexadecimal value of the first byte of the record, corresponding to the `type` field that identifies the type, while the ASTF record type column gives the symbolic representation.

Table 3: List of ASTF record types

Id	ASTF record type	Name	Purpose
0x01	ASTF_REC_INSTR	Instruction	PC, opcode, effective address, and properties
0x05	ASTF_REC_TRAP	Trap	Trap, exception or interrupt taken on or after the last instruction, or exception return
0x07	ASTF_REC_EXTRA_EA	Extra EA	Additional effective address information for an instruction that has multiple accesses
0x08	ASTF_REC_CONTEXT	Context	Switch in execution state or context
0x41	ASTF_REC_FILE_HEADER	Header	Header record, value corresponds to ASCII Character 'A' to form magic "ASTF" bytes
0x43	ASTF_REC_STRING	String	Start of metadata string, and total size
0x44	ASTF_REC_STRING_CONT	String continuation	String continuation; next 31 characters of metadata string
0x45	ASTF_REC_TIMESTAMP	Timestamp	Timekeeping record

4.2 Header and file format

An ASTF trace consists of a stream of 32-byte records, the first of which is a Header type record which provides information on which features are present in this trace, the version of the trace format, as well as how the remainder of the stream has been encoded. This is illustrated in [Table 4](#).

ASTF_REC_FILE_HEADER	Byte offset = 0
Record 1	Byte offset = 32
Record 2	Byte offset = 64
Record 3	Byte offset = 96
...	...

Table 4: Record stream layout

Table 5: Layout of the Header record

Bits	Field name	Description
[7:0]	<code>type</code>	0x41: ASTF_FILE_HEADER encoding, equal to 'A' character
[31:8]	<code>magic_header</code>	Bytes set to characters 'S','T','F' to form "ASTF" header bytes
[39:32]	<code>version_major</code>	Major version number
[47:40]	<code>version_minor</code>	Minor version number
[63:48]	<code>reserved16</code>	Reserved
[127:64]	<code>format_cfg</code>	Format flags, individual flags below in gray
[64:64]	<code>compressed</code>	0b0: Trace is in uncompressed RAW format 0b1: Trace is in compressed format
[65:65]	<code>use_zlib</code>	0b0: Use LZMA compression (default) 0b1: Use Zlib compression instead of LZMA if the trace is compressed
[127:66]	<code>reserved62</code>	Reserved
[191:128]	<code>feature_cfg</code>	Feature flags, individual flags below in gray
[128:128]	<code>reginfo</code>	Defined for future use, must be 0
[129:129]	<code>meminfo</code>	Defined for future use, must be 0
[130:130]	<code>addrinfo_d</code>	Defined for future use, must be 0
[131:131]	<code>addrinfo_i</code>	Defined for future use, must be 0
[132:132]	<code>pseudo</code>	Defined for future use, must be 0
[133:133]	<code>single_el</code>	0b1: Trace only contains a single exception level (Basic trace)
[134:134]	<code>timestamps</code>	0b1: Time information is available in the trace using timestamp records
[191:135]	<code>reserved57</code>	Reserved
[255:192]	<code>reserved64</code>	Reserved

Each record and all fields within a record in an ASTF trace are encoded using little-endian data representation.

The first record in an ASTF trace must be a Header record, and no further Header records must appear in the trace. The Header record's `type` and `magic_header` fields must form the four ASCII characters "ASTF" to identify an ASTF trace.

A Header record must have the `version_major` and `version_minor` fields set to the corresponding ASTF version used to encode this trace.

A Header record has the `format_cfg` field which contains flags that control how the records after the Header record are encoded when the trace is stored, which can enable compression (see [Section 4.3](#)). When all bits in `format_cfg` are 0, the trace is stored as a RAW, uncompressed, record stream.

A Header record has the `feature_cfg` field which contains the feature flags that control which features are present in this trace.

The next section, [Section 4.3](#), describes the different compression options for the trace storage format. The rest of this document describes the organization of the uncompressed record stream. It is

recommended that the (de)compression and (de)serialization of the record stream are implemented by a separate trace I/O library, and that trace analyzers and performance models that consume traces use that library to receive an uncompressed stream. The `libastf` library is an implementation of such a trace I/O library.

When a trace I/O library is used, it is responsible for generating the Header record when writing a trace, and interpreting the Header record when reading a trace. The trace analyzer or performance model that consumes the trace shall receive the Header record in the stream from the trace I/O library, and can use it to determine the trace version and the features present in the trace. When a tool writes a trace using the trace I/O library, it should not write a Header record, as the trace I/O library is responsible for generating it.

4.2.1 Trace storage and compression

The ASTF storage format currently supports three methods to encode and store a trace. Future compression and storage methods may be developed, which will be handled transparently by the trace I/O library. The storage encoding is controlled by the `format_cfg` field in the Header record:

- If the `compression` flag in `format_cfg` is set to 0, the trace is stored in RAW, uncompressed, format. This means that the records are stored in the trace file without change, using little-endian data representation. The stream of records continues at an offset of 32 bytes into the file, after the Header record.
- If the `compression` flag in `format_cfg` is set to 1, and the `use_zlib` flag is set to 0, the trace is stored in LZMA format. This means that the records beyond the header record are compressed with LZMA compression. The `liblzma` library is used to read/write this from/into the trace file at an offset of 32 bytes, after the Header record.
- If the `compression` flag in `format_cfg` is set to 1, and the `use_zlib` flag is set to 1, the trace is stored in Zlib format. This means that the records beyond the header record are compressed with Zlib compression. The `libz` library is used to read/write this from/into the trace file at an offset of 32 bytes, after the Header record.

4.3 Sanitized virtual addresses

Any record containing a virtual address must contain the sanitized address, even when stage 1 translation is disabled. A sanitized address is stripped of any address tags or PAC field bits above the size of the Input Address (IA) of stage 1 translation. The size of the IA is determined by the effective value of `TCR_ELx.TnSZ`, therefore the sanitization applies to bits[63:64 - `TCR_ELx.TnSZ`] of the virtual address and is not influenced by the effect of the `TCR_ELx.TBI` fields (Top Byte Ignore). If the current translation regime supports a single VA range, `TCR_ELx.T0SZ` is used and all bits[63:64 - `TCR_ELx.T0SZ`] must be set to 0. If the translation regime supports two VA ranges, bit[55] of the VA controls `n` to select `T0SZ` or `T1SZ`, and all bits[63:64 - `TCR_ELx.TnSZ`] must be set equal to the value of bit[55] of the VA.

4.4 Context changes

Table 6: Layout of the Context record

Bits	Field name	Description
[7:0]	<code>type</code>	0x08: <code>ASTF_REC_CONTEXT</code> encoding
[8:8]	<code>pstate_sm</code>	If FEAT_SME, 0b1: Streaming SVE mode enabled/disabled
[9:9]	<code>pstate_dit</code>	If FEAT_DIT, 0b1: Data independent timing enabled/disabled
[10:10]	<code>pstate_ssbs</code>	If FEAT_SSBS, 0b1: Speculative store bypass allowed/disallowed
[15:11]	<code>reserved5</code>	Reserved
[31:16]	<code>reserved16</code>	Reserved
[35:32]	<code>evl</code>	If FEAT_SVE or FEAT_SME, effective vector length (equal to VL or SVL)
[47:36]	<code>reserved12</code>	Reserved
[48:48]	<code>unsupported</code>	0b1: PE is in an unsupported execution mode, for example AArch32
[50:49]	<code>mte_tcf</code>	If FEAT_MTE2, effective current Tag Check Fault mode
[55:51]	<code>reserved5</code>	Reserved
[57:56]	<code>el</code>	Exception Level
[59:58]	<code>reserved2</code>	Reserved
[60:60]	<code>secure</code>	0b0: Non-secure mode 0b1: Secure mode
[61:61]	<code>handler</code>	0b0: Non-thread handler mode, EL>0 uses <code>SP_EL0</code> 0b1: Thread handler mode: ELx uses <code>SP_ELx</code>
[62:62]	<code>pid_valid</code>	0b1: <code>pid</code> field holds valid data
[63:63]	<code>tid_valid</code>	0b1: <code>tid</code> field holds valid data
[127:64]	<code>pid</code>	Process ID
[191:128]	<code>tid</code>	Thread ID
[255:192]	<code>reserved68</code>	Reserved

A Context record is used to provide information on context and execution state changes. It must at least capture the exception level in the `el` field, indicate secure/non-secure state with the `secure` flag, and trap/handler state with the `handler` flag, based on `PSTATE.SP`.

The information in a Context record applies to the next Instruction record seen.

A Context record must be present before the first Instruction record is seen in a trace, to inform the reader which execution state the processing element is in.

The effective vector length of the current execution mode must be recorded in the `evl` field:

- If FEAT_SVE, or FEAT_SME and `PSTATE.SM=0`, then `evl` is set to the value of `ZCR_Elx.LEN` for the current EL.
- If FEAT_SME and `PSTATE.SM=1`, the value is determined by `SMCR_Elx.LEN`.
- In all other cases, `evl` must be set to 0.

If FEAT_MTE2 or FEAT_MTE3, the two bits indicating the effective current Tag Check Fault mode (depending on `SCTLR_E1x.TCF` or `SCTLR_E1x.TCF0` for ELO), must be recorded in the `mte_tcf` field. If FEAT_MTE2 is not present, `mte_tcf` should always be 0.

If the trace source has information available on which process ID and/or thread ID is executing in ELO, this information should be provided in the `pid` and/or `tid` fields of any Context record that indicates ELO execution, and set the appropriate `pid_valid` and `tid_valid` flags. A process ID is assumed to identify a software execution context in ELO within its own address space (ASID), while different thread IDs with the same process ID are expected to share the same address space.

A Context record that has `tid_valid` set must have `pid_valid` set and have the `pid` field populated with a process ID.

When an instruction updates any system register field that is tracked in a Context record, a Context record must be present before the next Instruction record if and only if the value of the field has changed.

4.4.1 PSTATE

A Context record must reflect the state of several additional PSTATE fields depending on the implementation of the relevant architecture extension in the platform the trace was generated from. If an extension is not implemented, the associated field must be 0.

- `pstate_sm` if FEAT_SME, reflects PSTATE.SM Streaming SVE mode enabled/disabled.
- `pstate_dit` if FEAT_DIT, reflects PSTATE.DIT Data independent timing enabled/disabled.
- `pstate_ssbs` if FEAT_SSBS, reflects PSTATE.SSBS Speculative store bypass allowed/disallowed.

4.4.2 Unsupported execution mode

If the PE enters an execution mode that is not supported by ASTF, this must be represented by a Context record with the `unsupported` flag set. The unsupported modes currently include AArch32 mode and Debug state. Any other fields of the Context record can be set normally as for any other Context record, if the information is known when entering the unsupported execution state. For example, it can be useful to know the PID and TID when execution drops down to AArch32 in ELO.

The use of the `unsupported` flag in the Context record is only allowed in BasicPlus traces. When a trace is recorded in the Basic level representation, a switch of a PE into and subsequent execution in an unsupported mode should be ignored and not recorded in the trace.

While a PE is in an unsupported mode, no records must be recorded, including any switches between unsupported modes, until the PE returns to a mode supported by ASTF. If the transition back to the supported mode was made by an event that can be represented by a Trap record, this should be recorded, followed by a Context record indicating that execution resumed in a supported mode with the `unsupported` flag not set.

When Timestamp records are enabled, a timestamp can only be recorded when entering and exiting the unsupported mode. No Timestamps must be recorded for any events during the execution in the unsupported mode that would have been recorded if the PE was in a supported mode.

4.5 Instruction execution

Table 7: Layout of the Instruction record

Bits	Field name	Description
[7:0]	type	0x01: ASTF_REC_INSTR encoding
[9:8]	xattr_mode	Extended attribute field mode: 0b00 or 0b11: Not in use, xattr field as reserved64 0b01: EA range-based access, xattr field contains ea_range 0b10: SME information, xattr field contains sme.index
[15:10]	reserved6	Reserved
[16:16]	reserved1	Reserved
[17:17]	ea_tagged	If FEAT_MTE, 0b1: This memory operation caused a tag memory access
[18:18]	trap	0b1: This instruction received a trap
[19:19]	is_branch	0b1: This instruction is a branch
[20:20]	is_read	0b1: This instruction reads data from memory
[21:21]	is_write	0b1: This instruction writes data into memory
[22:22]	ea_valid	Does this instruction have a valid effective address: <ul style="list-style-type: none"> ea_valid = 0b1 and is_branch = 0b0: This instruction is a memory operation, address encoded in ea field ea_valid = 0b1 and is_branch = 0b1: This instruction is a taken branch, target encoded in ea field
[23:23]	extra_eas	0b1: This instruction is an extended memory operation with one or more virtual addresses, or misaligned across multiple pages
[31:24]	ea_size	Size of the memory access minus one
[63:32]	iword	Instruction word, 32-bit Instruction opcode
[127:64]	pc	PC of the instruction
[191:128]	ea	Effective address of the instruction: <ul style="list-style-type: none"> is_branch = 0b0: Virtual address for a memory access is_branch = 0b1: Destination PC for a branch instruction
[255:192]	xattr	Extended attribute field

Table 8: Layout of the xattr field when xattr_mode=0b00 or 0b11

Bits	Field name	Description
[63:0]	reserved64	Reserved

Table 9: Layout of the xattr field when xattr_mode=0b01

Bits	Field name	Description
[63:0]	ea_range	Size in bytes of the associated memory range access

Table 10: Layout of the `xattr` field when `xattr_mode=0b10`

Bits	Field name	Description
[7:0]	<code>sme.index</code>	Index into SME ZA tile
[63:8]	<code>sme.reserved56</code>	Reserved

An Instruction record is used to provide information on the (architectural) execution of an instruction. An Instruction record must capture the 32-bit instruction word (or opcode) in the `word` field, and the 64-bit instruction address (program counter, field: `pc`) of the instruction.

If FEAT_SME, an indexing SME instruction is an instruction that computes a slice index or vector group within a ZA tile. An Instruction record representing such an indexing SME instruction must have the `xattr_mode` field set to `0b10` and the `sme.index` field in `xattr` must contain the effective computed index into the ZA tile. This index is either the slice index or the index for the first vector of a vector group. Other SME instructions that do not use an index must have `xattr_mode` set to `0b00` and must leave the `xattr` fields 0.

An Instruction record containing a virtual address, either in the `pc` or `ea` field, must contain the sanitized virtual address as defined in [Section 4.3](#).

4.5.1 Branch instructions

An Instruction record representing a branch instruction must have the `is_branch` flag set.

An Instruction record representing a taken branch must have the `ea_valid` flag set, and the virtual address of the branch target must be set as the effective address in the `ea` field.

An Instruction record representing a non-taken conditional branch must not have the `ea_valid` flag set, and the `ea` field must be 0.

Return from function call instructions such as `RET` are considered branch instructions, while return from exception instructions such as `ERET`, debug restore `DRPS`, and all exception generating instructions such as `SVC`, `HVC`, and `SMC` are not considered branch instructions for this purpose.

4.5.2 Memory-accessing instructions

An Instruction record representing a memory-accessing instruction must have the `ea_valid` flag set, and must record the virtual address of the first accessed memory address as the effective address in the `ea` field. The size of this access must be encoded in the `ea_size` field as size – 1 in bytes. If the size exceeds the capacity of the `ea_size` field, the access should be split into one access with `ea_size` set to 255 to represent the first 256 bytes of the access, followed by one or more Extra EA records to represent the remainder of the memory access.

An Instruction record representing a memory-accessing instruction that makes more than one memory access to a region of memory that is not contiguous with the first access, must have the `extra_eas` flag set. That Instruction record must be followed by one or more Extra EA records to represent the additional memory accesses.

An Instruction record representing a memory-accessing instruction that makes multiple accesses to a contiguous region (both in virtual and physical addresses), for example, an `LDP` or `STP` that does not

cross a page boundary, it is recommended to be encoded as an Instruction record with one merged memory access and a proportionally increased `ea_size`.

An Instruction record representing a memory-accessing instruction that performs a read-modify-write cycle, both the `is_read` and the `is_write` flags must be set.

An Instruction record representing a memory load instruction must have the `is_read` flag set if it read from memory.

An Instruction record representing a memory store instruction must have the `is_write` flag set if it wrote to memory.

An Instruction record representing a memory prefetch instruction must not have the `is_read` or `is_write` flags set.

An Instruction record representing a memory instruction that due to a predicate did not perform any memory accesses must not have `ea_valid` set, and must have `ea` and `ea_size` set to 0. It must not have `is_read` or `is_write` set.

An Instruction record representing an instruction that conditionally writes to memory, such as a CAS or STX, must have the `is_write` flag set only when the write was successfully performed. Similarly, for ST64BV/ST64BV0 instructions, the `is_write` flag is only set when the write was to a supported location and completed successfully.

An Instruction record representing a memory-accessing instruction that takes an exception before any architecturally visible memory accesses complete, must not have the `is_read` or `is_write` flag set.

If FEAT_MTE2, an Instruction record representing a memory-accessing instruction that performed a Tag Checked memory access, must have the `ea_tagged` flag set. For a Tag Unchecked access, or if FEAT_MTE2 is not present, it must not be set.

If FEAT_MTE, an Instruction record representing a tag-read or tag-write load/store instruction must have `ea_tagged` set if it accesses a tagged address and allocation tag access is enabled. The `ea_valid` field must be set and the `ea` field must contain the tag granule-aligned base address for which tags are loaded or stored. The appropriate `is_read` or `is_write` flag must be set and the `ea_size` field must contain the number of memory locations in bytes, minus one, affected by the loaded or stored tags.

If FEAT_MTE, an Instruction record representing a tag-read or tag-write load/store instruction must not have `ea_tagged` set if it accesses a non-tagged address or if allocation tag access is disabled. If this instruction also performed a memory store, for example STGP, STZG, DC GZVA, it must be populated as a regular store access. If this instruction does not perform a memory store, `ea_valid` and `is_read/is_write` must not be set and `ea` and `ea_size` must be 0.

An Instruction record representing a special operation targeting a memory address such as prefetch, cache or TLB maintenance, or address translation (PRFM/PRFUM, IC/DC, TLBI, AT), must have the `ea_valid` and `ea` fields populated and `ea_size` set to 0. It must not have the `is_read` or `is_write` flags set. PRFM/PRFUM, TLBI and AT always perform Tag Unchecked accesses and therefore must not have `ea_tagged` set.

If FEAT_SVE, an Instruction record representing any `PRF*` memory prefetch instruction from the SVE encodings group that performs one or more memory accesses based on a given element size must have the appropriate `ea_size` set. It can use Extra EA records if an access exceeds the encodable size or multiple non-contiguous prefetch accesses are made.

If FEAT_SVE, a vector load instruction with First-Faulting or Non-Faulting semantics must record any suppressed faulting memory accesses in addition to any successfully completed memory accesses. Each suppressed access must not have `is_read` or `is_write` set, and only have `ea_tagged` set if it was suppressed by a Synchronous Tag Check Fault, and `ea_size` set to the element size, minus one. For suppressed faults, the `trap` flag must not be set.

An Instruction record representing a special memory initialization operation such as `DC ZVA`, or if FEAT_MTE also `DC GVA` or `DC GZVA`, is treated as a memory store instruction rather than a special operation. If a store was performed, it must have the `is_write` flag set and `ea_size` corresponding to the size of the access as `size - 1` in bytes. If the access size exceeds 256 bytes, it must be split across one or more Extra EA records.

4.5.2.1 Memory range accesses

An instruction performing a range-based memory access is one that affects a number of memory locations that are determined by the value in a register, rather than being determined by decoding the instruction. Examples are range-based TLB invalidations from FEAT_TLBIRANGE and Memory copy/set operations defined by FEAT_MOPS. Memory-accessing instructions that are subject to the state of predicate registers are not considered range-based accesses as they have a known maximum size.

An Instruction record representing an instruction that successfully made a range-based access must have the base of the range set in the `ea` field and `ea_valid` must be set. The `xattr_mode` field must be set to `0b01` and the `ea_range` field in `xattr` encodes the total size of the access in bytes.

An Instruction record representing a range-based instruction must have `ea_size` set to 0.

An Instruction record representing a range-based instruction must have flag `ea_read` or `ea_write` set to indicate if the range is being read or written. A single range definition cannot have both `is_read` and `is_write` flags set. If the instruction performs both a read and a write these must be recorded as two separate range accesses by setting the `extra_eas` flag and the Instruction record must be followed by an Extra EA record for the additional range. In this case, the range that is being read from must be encoded before the range that is being written to.

If FEAT_MTE2 if a range consists of Tag Checked memory accesses, it must have `ea_tagged` set. If only a subset of the range contains Tag Checked accesses, these should be represented as separate ranges.

4.5.2.2 Memory Copy and Memory Set instructions (FEAT_MOPS)

The prologue, main, and epilogue instructions of a FEAT_MOPS operation may each make zero or more memory accesses based on the total size of the operation and the alignment and transaction sizes used by an implementation.

When a trace reader is interpreting a trace with FEAT_MOPS instructions, it is recommended that all three of prologue, main and epilogue and their associated range accesses are read and interpreted as a whole, as the implementation from which the trace was recorded may have used any of the architecturally allowed trade-offs in aligning and splitting accesses between these operations.

An Instruction record representing a FEAT_MOPS instruction that makes zero memory accesses and did not take an exception during execution must record one or two range accesses with size 0 and the appropriate base addresses that it would have operated on would it have accessed memory. It must have `is_read` or `is_write` flags set appropriately for each range and must not have `ea_tagged` set.

An Instruction record representing a FEAT_MOPS instruction which makes multiple memory transactions must be recorded as a single architectural instruction execution, with aggregated range accesses representing the ranges that were set or copied.

4.5.3 Trapping instructions

Trapping instructions are instructions that raised a synchronous exception during their execution.

4.5.3.1 Basic traces

An Instruction record representing a syscall instruction (SVC) generates an exception and must have the `trap` flag set. Before the next Instruction record is seen, a Context record must be present which indicates the execution state when it is resumed. It is not followed by a Trap record.

4.5.3.2 BasicPlus traces

An Instruction record representing an instruction that generated an exception when it was executed must have the `trap` flag set.

A branch instruction must not have `ea_valid` or `ea` set if the branch took a trap or exception. A branch target exception introduced by FEAT_BTI is not considered a branch that traps as the exception is taken on the next instruction at the branch destination.

A memory-accessing instruction must have `ea_valid` and `ea` populated if and only if a memory access was attempted and the taken exception was a Data Abort. If the attempted address is obtained from a `FAR_ELx` register, it must be sanitized as defined in [Section 4.3](#). The Instruction record may be followed by one or more Extra EA records if multiple memory accesses were attempted by the execution of the instruction. If it was a Synchronous Tag Check Fault, `ea_tagged` must be set.

If FEAT_SVE, a memory-accessing vector instruction that takes a Data Abort during execution must record any memory accesses associated with this instruction that became architecturally visible before the exception was taken. These must be followed by a final memory access that indicates the faulting address.

An Instruction record representing a FEAT_MOPS instruction that takes an exception during execution must have ranges associated for any architecturally-completed memory accesses and have the appropriate `is_read` or `is_write` flags set.

An Instruction record representing a FEAT_MOPS instruction that takes a Data Abort during execution must have ranges associated for any architecturally-completed memory accesses and have the appropriate `is_read` or `is_write` flags set. It should record a final (non-range) memory access that indicates the faulting address without `is_read` or `is_write` flags set. If the Data Abort was a Synchronous Tag Check Fault, `ea_tagged` must be set.

The Instruction record and any associated Extra EA records must be followed by a Trap record that records the detailed information about the exception, see [Section 4.7](#).

4.6 Extra EA

Table 11: Layout of the Extra EA record

Bits	Field name	Description
[7:0]	type	0x07: ASTF_REC_EXTRA_EA encoding
[15:8]	mem0_reserved8	Reserved
[20:16]	mem0_reserved5	Reserved
[21:21]	mem0_tagged	If FEAT_MTE, 0b1: This memory operation caused a tag memory access
[22:22]	mem0_is_read	0b1: Read transaction
[23:23]	mem0_is_write	0b1: Write transaction
[31:24]	mem0_size	Transaction size
[63:32]	mem0_reserved32	Reserved
[127:64]	mem0_addr	Effective address
[128:128]	mem1_valid	0b1: mem1 request fields are valid and encode an additional memory access
[129:129]	extra_eas	0b1: There are more Extra EA records with associated memory requests after the current one
[130:130]	mem1_addr_range	0b1: The mem1_addr field is valid and contains a range-access size for mem0_addr. All other mem1 fields are invalid.
[135:131]	reserved5	Reserved
[143:136]	reserved8	Reserved
[148:144]	reserved5	Reserved
[149:149]	mem1_tagged	If FEAT_MTE, 0b1: This memory operation caused a tag memory access
[150:150]	mem1_is_read	0b1: Read transaction
[151:151]	mem1_is_write	0b1: Write transaction
[159:152]	mem1_size	Transaction size
[191:160]	mem1_reserved32	Reserved
[255:192]	mem1_addr	Encodes an additional address or range, depending on mem1_addr_range: <ul style="list-style-type: none"> If mem1_addr_range = 0b0 and mem1_valid = 0b0: Reserved If mem1_addr_range = 0b0 and mem1_valid = 0b1: Effective address If mem1_addr_range = 0b1: Range size associated with mem0_addr

An Extra EA record represents one or more additional effective addresses, in other words, memory accesses, associated with the last seen Instruction record, which must have the `extra_eas` flag set.

An Extra EA record encodes the first additional memory access in the `mem0_*` fields. It uses the same conditions for setting the `tagged`, `is_read`, `is_write`, and `size` fields, as well as the conditions for merging accesses into one representation, as Memory-accessing instructions described in [Section 4.7](#). The `addr` field needs to be set with the virtual address that was accessed by this memory access.

An Extra EA record encodes the second additional memory access, if present, using the `mem1_*` fields and requires the `mem1_valid` flag to be set to indicate that the second access is present in the record.

An Extra EA record containing a virtual address in `mem0_addr`, and optionally `mem1_addr`, must contain the sanitized virtual address as defined in [Section 4.3](#).

An Extra EA record can encode a memory range access which is indicated by having `mem1_addr_range` set. The `mem0_addr` field must contain the base address, and `mem1_addr` must contain the range size. No other `mem1` fields can be set, `mem1_valid` must not be set. The field `mem0_size` must be 0 and fields `mem0_tagged`, `mem0_is_read`, and `mem0_is_write` are set similarly to regular memory accesses.

If more additional memory accesses or ranges need to be encoded, additional Extra EA records are used to encode all the memory accesses associated with the last Instruction record. Each Extra EA record but the last in the sequence has the `extra_eas` flag set to indicate that more Extra EA records follow this record, that belong to the same instruction.

Extra EA records must immediately follow the Instruction record they belong to. No Timestamp, Trap, or other records can interleave with this sequence. If the Instruction record that an Extra EA record belongs to has the `trap` flag set, the required Trap and/or Context records appear after the last Extra EA record associated with that instruction.

4.7 Traps



Trap records are only valid in BasicPlus traces.

Table 12: Layout of the Trap record

Bits	Field name	Description
[7:0]	type	0x05: ASTF_REC_TRAP encoding
[8:15]	reserved8	Reserved
[18:16]	trap_type	0x0: Synchronous exception 0x1: Serror 0x2: IRQ 0x3: FIQ 0x4: Exception return 0x5: PE reset 0x6-0x7: Reserved
[19:19]	reserved1a	Reserved
[21:20]	syscall_type	0b00: None 0b01: SVC 0b10: HVC 0b11: SMC
[23:22]	reserved2	Reserved
[24:24]	is_virtual	0b1: Virtualised vIRQ, vFIQ or vSError
[25:25]	reserved1b	Reserved
[31:26]	ec	Exception class reported in ESR_ElX
[47:32]	syscall_index	Syscall, Hypercall or Firmware call index
[63:48]	reserved16	Reserved
[127:64]	target_pc	PC to which the program flow is redirected when the trap is taken
[255:128]	reserved128	Reserved

A Trap record is used to signal one of the four scenarios outlined in the next sections, where execution is redirected due to a trap, exception or interrupt. The `target_pc` field must be set with the virtual address of the next instruction that execution was redirected to, and must contain the sanitized virtual address as defined in [Section 4.3](#).

A Trap record must appear before a Context or Instruction record associated with the start of the exception handler.

4.7.1 Synchronous exception without instruction

When a Trap record is used to represent a synchronous exception that is not associated with an instruction, the last seen Instruction record does not have the `trap` flag set and executed and

completed normally. An example of a synchronous exception without an associated instruction is an Instruction Abort, where the instruction failed to fetch.

A Trap record for a synchronous exception without an instruction must have the `trap_type` set to 0 to indicate a Synchronous Exception. The `ec` field must contain the exception class.

4.7.2 Synchronous exception or trap with instruction

When a Trap record is used to represent a synchronous exception associated with an instruction, the last seen Instruction record must have the `trap` flag set, which indicates that the instruction failed to complete execution. The Trap record must appear after the instruction and any associated Extra EA records.

A Trap record for a synchronous exception associated with an instruction must have the `trap_type` set to 0 to indicate a Synchronous Exception. The `ec` field must contain the exception class.

If the Trap record is used to represent a synchronous exception associated with a syscall type instruction such as `SVC`, `HVC` or `SMC`, this type is recorded in the `syscall_type` field, and the syscall index is decoded and stored in the `syscall_index` field. In all other cases these fields must be 0.

4.7.3 Asynchronous trap, exception or interrupt

When a Trap record is used to represent an asynchronous exception, it is not associated with an instruction. It must appear after the last successfully executed Instruction record and any associated Extra EA records.

A Trap record representing an asynchronous exception must have `trap_type` set to indicate the type of asynchronous exception:

- 0x1 for Serror
- 0x2 for IRQ
- 0x3 for FIQ

If it is a Serror, the `ec` field must be set accordingly with the exception class, and otherwise it should be 0.

A Trap record used to signal a virtualized vSError, vIRQ or vFIQ must have the `is_virtual` flag set.

4.7.4 Exception return

A Trap record is used to represent exception return, when the `trap_type` field is set to 0x4 (`ERET`). This Trap record must appear after the Instruction record representing the `ERET` instruction, which must not have the `trap` flag set.

A Trap record used to represent exception return has the `target_pc` field set to indicate the virtual instruction address that execution will be attempted to be resumed at. All other fields but `trap_type` and `target_pc` must be 0.

4.7.5 PE reset

A Trap record is used to represent a PE reset event with the `trap_type` field set to `0x5` (PE reset), even though this is not regarded as a trap or exception in the architecture. The `target_pc` field must be populated with the appropriate PC where execution starts, based on the relevant `RVBAR_E1x` value. All other fields but `trap_type` and `target_pc` must be 0.

4.8 String, String continuation

Table 13: Layout of the String record

Bits	Field name	Description
[7:0]	<code>type</code>	0x43: ASTF_REC_STRING encoding
[15:8]	<code>reserved8</code>	Reserved
[31:16]	<code>size</code>	Size in bytes
[255:32]	<code>string</code>	String bytes

Table 14: Layout of the String continuation record

Bits	Field name	Description
[7:0]	<code>type</code>	0x44: ASTF_REC_STRING_CONT encoding
[255:8]	<code>string</code>	String bytes

String records can be used to record metadata strings in a trace, such as information about the workload, license restrictions, or source of these traces. There is currently no standard for how this information should be represented. The maximum length of metadata is 65535 bytes and does not have to be null terminated. It is recommended that meta information is represented as a printable C-style string. However, this is not mandatory, and the metadata can contain zero or more null bytes.

A String record represents the start of metadata. The `size` field indicates the total length of a metadata string in bytes. The first 28 bytes of metadata are held in the `string` field. A String record representing a metadata string shorter than 28 bytes has the remaining bytes in the `string` field set to 0.

A String continuation record follows a String or other String continuation record, and represents the next, up to, 31 bytes of the metadata string in the `string` field. If not all 31 bytes of a String continuation record are required to complete the metadata string, the remaining bytes in the `string` field are set to 0.

String and String continuation record sequences belonging to a single metadata string must not be interleaved with other record types.

It is recommended that metadata strings are recorded before the first Context and Instruction record in a trace, unless they are used to annotate a specific point in the trace.

4.9 Timestamp



Note

The use of Timestamp records is optional, and if used, the `timestamps` flag in `feature_cfg` must have been set in the trace Header record.

Table 15: Layout of the Timestamp record

Bits	Field name	Description
[7:0]	<code>type</code>	0x45: ASTF_REC_TIMESTAMP encoding
[15:8]	<code>unit</code>	0x00: Invalid 0x01: Microseconds 0x02-0xff: Reserved
[31:16]	<code>reserved16</code>	Reserved
[63:32]	<code>reserved32</code>	Reserved
[127:64]	<code>value[0]</code>	First timestamp value, use subject to selected unit
[191:128]	<code>value[1]</code>	Second timestamp value, reserved
[255:192]	<code>reserved64</code>	Reserved

A Timestamp record is used to represent the progression of time of the trace source. Timestamp information can be used by trace readers to throttle trace replay, or to keep multiple traces loosely synchronized when they have been generated in parallel from more than one PE. This version of ASTF only supports timestamps based on a microsecond count.

A Timestamp record provides timing information for the Instruction records and other records that follow it.

A Timestamp record must have the `unit` field set to 1 to indicate the use of microseconds and the `value[0]` field must be set to an unsigned 64-bit count of microseconds.

A Timestamp record must be present at the start of a trace, before the first Context record and the first Instruction.

A Timestamp record must not interleave with earlier defined record sequences, such as String/String continuation or Instruction/Extra EA.

Timestamp records must have monotonically increasing values in a trace, and no two Timestamp records with the same value are allowed within a single trace.

It is implementation defined if Timestamp values represent absolute or relative time, for example the number of microseconds since the start of the trace. Traces generated in parallel from more than one PE should all observe the same time reference.

A Timestamp record is not allowed to contain the maximum value 0xffff_ffff_ffff_ffff in field `value[0]`, this value is reserved. An implementation generating a trace should take care to pick a time reference such that the maximum Timestamp value will not likely be reached or overflowed. If this cannot be avoided, an error should be thrown when the maximum Timestamp value has been reached.

When a PE suspends execution for enough time, two Timestamp records must be present in the trace to represent when the PE suspends and resumes. These Timestamp records must come after the Instruction record of a **WFX** instruction if its execution is the reason the PE suspends.

If the timestamp at which the PE suspends is the same as the previous Timestamp record seen in the trace, the nearest subsequent timestamp value must be recorded if that falls within the period of suspended execution. If it falls outside of this period, no Timestamp record is recorded when the PE suspends.

If the timestamp at which the PE resumes is the same as the previous Timestamp record seen in the trace, including the Timestamp recorded when the PE suspends, it is not recorded. This means that:

- If the PE suspends execution for at least the duration of two timestamp intervals, this is represented by two consecutive Timestamp records.
- Zero or one Timestamp records after a **WFX** instruction indicates that the PE either did not suspend at all, or only suspended for a short period of time.



In a BasicPlus level trace, two consecutive Timestamp records without any Instruction records in between must always mean that the PE suspended for that period.

It is recommended that Timestamp records are output to a trace periodically, for example every 5000 instructions or so, or at a granularity where at least one or more microseconds have passed. No periodic Timestamps should be output to the trace for a PE that is in suspended execution state. When traces are generated in parallel from more than one PE, the implementation should attempt to align the periodic Timestamp records across the PEs as closely as possible.

It is recommended to write a Timestamp before any Context record that indicates a context switch, if this Timestamp would be different from the last recorded Timestamp.

5 Example trace

Figure 1 shows a condensed textual representation of a short, 21 record, snippet of an ASTF trace that executes 11 instructions, using the BasicPlus level of representation with timestamps enabled. Each line represents a record in the trace. In this section we will look at the details of how each field in each record is filled in to represent this execution. Note that this is a contrived example and does not resemble a realistic execution or workload.

Figure 1: Example ASTF trace (textual representation)

```

1 fileheader: ASTF trace, version 0.11 compression mode : LZMA - BasicPlus level trace
2 string    : ..
3 string    : "Example ASTF tracefile for the specification"
4 timestamp : Value: 100
5 context   : CPU in EL0, non-secure, thread-mode PID: 29193 TID: 29197 SVE128, TC-sync, SSBS
6 instr     : el0t [0x000000000836e128] CMP      w3,w0
7 instr     : el0t [0x000000000836e12c] B.NE      {pc}-0x44; 0x836e0e8 NT
8 instr     : el0t [0x000000000836e130] LDP       x19,x20,[sp,#0x10] v[0x0000ffffe7624250]
9 instr     : el0t [0x000000000836e134] SETGP     [x19]!,x20!,x0 v[0x0000a4330b765010]t..[0x0000a4330b765020]
10 instr    : el0t [0x000000000836e138] SETGM     [x19]!,x20!,x0 v[0x0000a4330b765020]t..[0x0000a4330b765080]
11 instr    : el0t [0x000000000836e13c] SETGE     [x19]!,x20!,x0 v[0x0000a4330b765080]..[0x0000a4330b765080]
12 instr    : el0t [0x000000000836e140] BLR       x2 T v<0x0000ffff83ffb2f0>
13 instr    : el0t [0x0000ffff83ffb2f0] LDP      w11,w12,[x6,#0x4c] v[0x0000a4330b765ffc]t
14 extra_ea : v[0x0000a4330b766000]t
15 instr    : el0t [0x0000ffff83ffb2f4] STP      x11,x12,[x7],#0x10 v[0x0000ffff95ceb000]t tr
16 trap     : Synchronous Exception to <0xffff7dffe7fc400> : 100100 Data Abort from lower
17 timestamp : Value: 102
18 context   : CPU in EL1, non-secure, handler-mode SVE256, TC-asm
19 instr     : ellh [0xffff7dffe7fc400] SMSTART SM
20 context   : CPU in EL1, non-secure, handler-mode SSVE512, TC-asm
21 instr     : ellh [0xffff7dffe7fc404] LDR      ZA[w12,4],[x0,#4,MUL VL] i{5} v[0xffff8000053bd500]

```

The first column in Figure 1 shows the record number, then the record type, and then a summary of the data represented in the record. This example trace starts with a Header record, and then contains the following elements:

- A String and String continuation record to form the metadata string shown on line 3.
- Line 4 and 5 have a Timestamp and Context record, after which the execution of several instructions starts, which cover several scenarios:
 - ♦ A non-taken conditional branch on line 7.
 - ♦ A merged load-pair on line 8.
 - ♦ A sequence of three memory set (with tag) instructions on lines 9-11.
 - ♦ A taken non-conditional branch on line 12.
 - ♦ A fragmented load-pair access across a page boundary on line 13 and 14, using the Extra EA record.
- The store pair on line 15 takes a page fault, and the trap record identifying a synchronous Data Abort exception is on line 16.
- A new timestamp is written before the context switch, the context switches to EL1 execution on line 18, and the first instruction of the exception handler executes on line 19.
- On line 19 streaming SVE mode is enabled, which results in the associated context update on line 20, followed by an SME ZA Array vector load on line 21.

5.1 Detailed record contents

The following subsections and tables list the contents of the relevant fields in each of the 17 records from the example. Most of the fields that are left as 0 are omitted from the tables.

5.1.1 Record 1: Header

```
8 fileheader : ASTF trace, version 0.11 compression mode : LZMA - BasicPlus level trace
```

The header record indicates an 0.11 ASTF trace that has been compressed with LZMA compression, as it has the `compressed` flag set, and `use_zlib` is not set in the `format_cfg` flags. In the `feature_cfg` flags, `single_el` is not set because this is a BasicPlus level trace, so it contains instructions executing at multiple exception levels, and full trap information. In addition to this, this trace contains timestamp information, so the `timestamps` flag is set, indicating we should expect Timestamp records.

Table 16, Record 1: File/trace Header

Bits	Field name	Value and explanation
[7:0]	<code>type</code>	0x41: <code>ASTF_FILE_HEADER</code> , equal to 'A' character
[31:8]	<code>magic_header</code>	Bytes set to characters 'S','T','F' to form "ASTF" header bytes
[39:32]	<code>version_major</code>	0
[47:40]	<code>version_minor</code>	11
[127:64]	<code>format_cfg</code>	Format flags, individual flags below in grey
[64:64]	<code>compressed</code>	0b1: Trace is in compressed format
[65:65]	<code>use_zlib</code>	0b0: Use LZMA compression (default)
[191:128]	<code>feature_cfg</code>	Feature flags, individual flags below in grey
[133:133]	<code>single_el</code>	0b0: Trace contains multiple exception levels and uses trap records
[134:134]	<code>timestamps</code>	0b1: Time information is available in the trace using timestamp records

5.1.2 Record 2 and 3: String metadata

```
2 string : ..
3 string : "Example ASTF tracefile for the specification"
```

The header is followed by a String and a String continuation record to record the 44 characters long metadata string "Example ASTF tracefile for the specification". Record 2, the String record, records the size in the `size` field and contains the first 28 characters of the string (not null terminated) in the `string` field.

Table 17, Record 2: String metadata start

Bits	Field name	Value and explanation
[7:0]	<code>type</code>	0x43: ASTF_REC_STRING, start of metadata string
[31:16]	<code>size</code>	44 Bytes
[255:32]	<code>string</code>	"Example of ASTF tracefile for t"

Record 3, the String continuation record, can encode an additional 31 characters, but only 16 are used in the `string` field to complete the 44 characters metadata string. Note that the last String record does not need to be null terminated either, though in this case it is implicitly, as the remaining 15 bytes of the `string` field should be 0.

Table 18, Record 3: String metadata continued

Bits	Field name	Value and explanation
[7:0]	<code>type</code>	0x44: ASTF_REC_STRING_CONT, continued metadata string
[255:8]	<code>string</code>	"he specification"

5.1.3 Record 4: Timestamp

```
4 timestamp : Value: 100
```

The Timestamp record at the start of the trace gives information about the initial point in time. This record indicates a value of 100 microseconds in the `value[0]` field.

Table 19, Record 4: Initial Timestamp

Bits	Field name	Value and explanation
[7:0]	<code>type</code>	0x45: ASTF_REC_TIMESTAMP
[15:8]	<code>unit</code>	0x01: Microseconds
[127:64]	<code>value[0]</code>	100

5.1.4 Record 5: Context

```
5 context : CPU in EL0, non-secure, thread-mode PID: 29193 TID: 29197 SVE128, TC-sync, SSBS
```

Before the first Instruction record is seen, the Context record provides information about the execution state of the PE. If the Context record indicates EL0, and the trace source had this information available, the Context record also reports the process/thread id information of the process that is executing.

The trace starts in EL0, non-secure, handler mode so the `el`, `secure`, and `handler` fields are set accordingly. The PE is configured for 128-bit SVE, synchronous MTE Tag Check Faults, and Speculative Store Bypass Safe mode is enabled. As the PID of the executing process was known to be 29192 by the trace source, this is written to the `pid` field and `pid_valid` is set to indicate that the PID is present. Similarly, the TID has been set to 29197.

Table 20, Record 5: Initial Context

Bits	Field name	Value and explanation
[7:0]	<code>type</code>	0x08: ASTF_REC_CONTEXT
[10:10]	<code>pstate_ssbs</code>	0b1: SSBS mode is enabled
[35:32]	<code>evl</code>	0x0: Indicating a 128-bit SVE vector length
[50:49]	<code>mte_tcf</code>	0b01: Synchronous MTE Tag Check Fault mode on both loads and stores
[57:56]	<code>el</code>	0b00: Execution is in EL0
[60:60]	<code>secure</code>	0b0: Non-secure mode
[61:61]	<code>handler</code>	0b0: Non-thread handler mode, EL>0 uses SP_ELO
[62:62]	<code>pid_valid</code>	0b1: The <code>pid</code> field holds valid data
[63:63]	<code>tid_valid</code>	0b1: The <code>tid</code> field holds valid data
[127:64]	<code>pid</code>	29192: Process id of program running in EL0
[191:128]	<code>tid</code>	29197: Thread id of program running in EL0

5.1.5 Record 6: Instruction

```
6 instr : el0t [0x000000000836e128] CMP w3, w0
```

The first Instruction that was executed in the trace. The opcode is set in the `word` field and the PC is set in the `pc` field. As the instruction is neither a branch nor a memory access, the rest of the fields are set to 0.

Table 21, Record 6: First Instruction

Bits	Field name	Value and explanation
[7:0]	<code>type</code>	0x01: ASTF_REC_INSTR
[63:32]	<code>word</code>	0x6b00007f: Encoding for CMP w3, w0
[127:64]	<code>pc</code>	0x000000000836e128

5.1.6 Record 7: Instruction, branch not-taken

```
7 instr : e10t [0x000000000836e12c] B.NE {pc}-0x44; 0x836e0e8 NT
```

The second instruction is an example of the encoding of a non-taken branch. This is indicated by having the `is_branch` flag set, but not having `ea_valid` set. The instruction did not have a valid effective address, that is, it had no branch destination encoded in the record, which means that the execution followed the fall-through path and the next instruction is at PC+4.

Table 22, Record 7: First branch Instruction, not-taken

Bits	Field name	Value and explanation
[7:0]	type	0x01: ASTF_REC_INSTR
[19:19]	is_branch	0b1: This instruction is a branch
[22:22]	ea_valid	0b0: Branch is not-taken (as <code>is_branch</code> = 0b1)
[63:32]	iword	0x54fffd1: Encoding for B.NE {pc}-0x44
[127:64]	pc	0x000000000836e12c

5.1.7 Record 8: Instruction, load-pair merged access

```
8 instr : e10t [0x000000000836e130] LDP x19,x20,[sp,#0x10] v[0x0000ffffe7624250]
```

The third Instruction record shows a load-pair instruction. As this is a memory access, the `is_read` and `ea_valid` flags are set, and `is_branch` is not set. This load-pair does two 8-byte reads, but as the two addresses map to contiguous virtual and physical addresses, they are merged as a single 16-byte read in the trace for encoding efficiency. Therefore, the `ea_size` field is set to 15, and the `extra_eas` flag is not set. The `ea` field contains the lower address of the two accesses. The memory accesses were not Tag Checked accesses, and therefore did not cause a tag memory access.

Table 23, Record 8: First load-pair Instruction, merged access

Bits	Field name	Value and explanation
[7:0]	type	0x01: ASTF_REC_INSTR
[17:17]	ea_tagged	0b0: This instruction's memory access was not a Tag Checked access
[19:19]	is_branch	0b0: This instruction is not a branch
[20:20]	is_read	0b1: This instruction reads data from memory
[22:22]	ea_valid	0b1 and <code>is_branch</code> = 0b0: This instruction has a memory operation, address encoded in the <code>ea</code> field
[23:23]	extra_eas	0b0: This is not an extended memory operation with a second access
[31:24]	ea_size	15: This instruction accessed 15+1=16 bytes of memory
[63:32]	iword	0xa94153f3: Encoding for LDP x19,x20,[sp,#0x10]
[127:64]	pc	0x000000000836e130
[191:128]	ea	0x0000ffffe7624250: Virtual address accessed by instruction

5.1.8 Record 9, 10, and 11: Instruction, memory set with tag

```

9  instr : e10t [0x000000000836e134] SETGP [x19]!,x20!,x0 v[0x0000a4330b765010]t..[0x0000a4330b765020]
10 instr : e10t [0x000000000836e138] SETGM [x19]!,x20!,x0 v[0x0000a4330b765020]t..[0x0000a4330b765080]
11 instr : e10t [0x000000000836e13c] SETGE [x19]!,x20!,x0 v[0x0000a4330b765080]..[0x0000a4330b765080]

```

The fourth through sixth instructions are a sequence of three memory set instructions that also write to tag memory, and together write 112 bytes of memory. This example assumes a PE implementation that aligns the main memory set operation to 32-byte boundaries. Therefore, the prologue instruction performs a ranged write of the first 16 bytes to make the next part 32-byte aligned. The main instruction then writes the remaining 96 bytes, and the epilogue writes 0 bytes. Note that the visualization of records shows the start and limit (exclusive) addresses of a range rather than base and size.

Table 24, Record 9: Memory set prologue

Bits	Field name	Value and explanation
[7:0]	type	0x01: ASTF_REC_INSTR
[9:8]	xattr_mode	0b01: This instruction has an extended attribute with range information
[17:17]	ea_tagged	0b1: This instruction accessed tag memory
[21:21]	is_write	0b1: This instruction wrote data to memory
[22:22]	ea_valid	0b1 and xattr_mode = 0b01: This instruction is making a memory range access with the range base encoded in the ea field
[23:23]	extra_eas	0b0: This instruction is not followed by Extra EA records
[31:24]	ea_size	0: This instruction uses a range, therefore size must be 0
[63:32]	iword	0x1dc00693: Encoding for SETGP [x19]!,x20!,x0
[127:64]	pc	0x000000000836e134
[191:128]	ea	0x0000a4330b765010: Base of the memory range access
[255:192]	xattr.ea_range	0x0000000000000010: Range access of 16 bytes from the base

Table 25, Record 10: Memory set main

Bits	Field name	Value and explanation
[7:0]	type	0x01: ASTF_REC_INSTR
[9:8]	xattr_mode	0b01: This instruction has an extended attribute with range information
[17:17]	ea_tagged	0b1: This instruction accessed tag memory
[21:21]	is_write	0b1: This instruction wrote data to memory
[22:22]	ea_valid	0b1 and xattr_mode = 0b01: This instruction is making a memory range access with the range base encoded in the ea field
[63:32]	iword	0x1dc04693: Encoding for SETGM [x19]!,x20!,x0
[127:64]	pc	0x000000000836e138
[191:128]	ea	0x0000a4330b765020: Base of the memory range access
[255:192]	xattr.ea_range	0x0000000000000060: Range access of 96 bytes from the base

Each of the instructions has the `ea_valid` flag set and the base of the range set in the `ea` field. The `xattr_mode` is set to `0b01` to indicate a range-based access, and the range sizes of 16, 96 and 0 are set in the `xattr.ea_range` fields respectively. The first two operations have `ea_tagged` set as they update the tag memory, but the third memory set operation does not perform any memory write and therefore does not have `ea_tagged` set. All three instructions have `is_write` set to indicate they're part of the same set of writes and have `extra_eas` and `ea_size` both set to 0.

Table 26, Record 11: Memory set epilogue with zero access

Bits	Field name	Value and explanation
[7:0]	<code>type</code>	<code>0x01</code> : <code>ASTF_REC_INSTR</code>
[9:8]	<code>xattr_mode</code>	<code>0b01</code> : This instruction has an extended attribute with range information
[17:17]	<code>ea_tagged</code>	<code>0b0</code> : This instruction did not access tag memory
[21:21]	<code>is_write</code>	<code>0b1</code> : This instruction's range was part of the write set
[22:22]	<code>ea_valid</code>	<code>0b1</code> and <code>xattr_mode = 0b01</code> : This instruction is making a memory range access with the range base encoded in the <code>ea</code> field
[63:32]	<code>iword</code>	<code>0x1dc08693</code> : Encoding for <code>SETGE</code> <code>[x19]!, x20!, x0</code>
[127:64]	<code>pc</code>	<code>0x000000000836e13c</code>
[191:128]	<code>ea</code>	<code>0x0000a4330b765080</code> : Base of the memory range access
[255:192]	<code>xattr.ea_range</code>	<code>0x0000000000000000</code> : Range access of 0 bytes from the base

5.1.9 Record 12: Instruction, branch taken

```
12 instr      : el0t [0x000000000836e140] BLR      x2      T v<0x0000ffff83ffb2f0>
```

The seventh Instruction record, and the second branch, resulted in a taken branch. Therefore besides having the `is_branch` flag, it has the `ea_valid` flag set and the `ea` field contains the destination PC of the taken branch.

Table 27, Record 12: Address generation Instruction

Bits	Field name	Value and explanation
[7:0]	<code>type</code>	<code>0x01</code> : <code>ASTF_REC_INSTR</code>
[19:19]	<code>is_branch</code>	<code>0b1</code> : This instruction is a branch
[22:22]	<code>ea_valid</code>	<code>0b1</code> and <code>is_branch = 0b1</code> : Instruction is a taken branch, target encoded in the <code>ea</code> field
[63:32]	<code>iword</code>	<code>0xd63f0040</code> : Encoding for <code>BLR</code> <code>x2</code>
[127:64]	<code>pc</code>	<code>0x000000000836e140</code>
[191:128]	<code>ea</code>	<code>0x0000ffff83ffb2f0</code> : Destination PC for a branch instruction

5.1.10 Record 13 and 14: Instruction and Extra EA, load-pair fragmented

```

13 instr      : e10t [0x0000ffff83ffb2f4] LDP      w11,w12,[x6,#0x4c]      v[0x0000ffff83ff9ffc]t
14 extra_ea   : v[0x0000ffff83ffa000]t

```

The eighth Instruction record shows a load-pair instruction which does two 4-byte accesses but crosses a page boundary and does not map to contiguous physical addresses. Note that physical addresses are not shown or recorded in this version of the format. Both accesses are subject to a tag check.

The `is_read`, `ea_valid` and `ea` fields are set as expected, but the `ea_size` field is set to 3 to encode an access size of 4. The `extra_eas` flag is set, and the subsequent record 12, is of the Extra EA type. The `ea_tagged` flag is set in the instruction to indicate the first access was a Tag Checked access.

Table 28, Record 13: Second load-pair Instruction, fragmented access

Bits	Field name	Value and explanation
[7:0]	type	0x01: ASTF_REC_INSTR
[17:17]	ea_tagged	0b1: This instruction accessed tag memory as the memory access was Tag Checked
[20:20]	is_read	0b1: This instruction reads data from memory
[22:22]	ea_valid	0b1 and <code>is_branch</code> = 0b0: This instruction is a memory operation, address encoded in <code>ea</code> field
[23:23]	extra_eas	0b1: This is an extended memory operation with one or more virtual addresses, or misaligned across multiple pages
[31:24]	ea_size	3: This instruction accessed 4 bytes (3+1) at this address
[63:32]	iword	0x2949b0cb: Encoding for LDP w11,w12,[x6,#0x4c]
[127:64]	pc	0x0000ffff83ffb2f4
[191:128]	ea	0x0000ffff83ff9ffc: Virtual address for first memory access

The Extra EA record contains the second memory access associated with the load-pair instruction. The flag `mem1_valid` is not set which indicates one additional access is encoded in this record using only the `mem0` fields. The `mem0_is_read` flag is set, `mem0_tagged` is set as it was a Tag Checked access, and `mem0_size` is set to 3. The address of the second access is the EA of the load-pair plus 4, and is set in `mem0_addr`. As there are no further Extra EA records following this one to encode more accesses associated with this instruction, the `extra_eas` flag is not set.

Table 29, Record 14: Second memory access of fragmented load-pair Instruction

Bits	Field name	Value and explanation
[7:0]	type	0x07: ASTF_REC_EXTRA_EA
[21:21]	mem0_tagged	0b1: This memory access included a tag memory access, as it was Tag Checked
[22:22]	mem0_is_read	0b1: Read transaction
[31:24]	mem0_size	3: This instruction accessed 4 bytes (3+1) at this address
[127:64]	mem0_addr	0x0000ffff83ffa000: Virtual address for second memory access
[128:128]	mem1_valid	0b0: No additional memory request present in <code>mem1</code> fields
[129:129]	extra_eas	0b0: No further Extra EA records belonging to this sequence after this one

5.1.11 Record 15: Instruction, store-pair with Data Abort

```
15 instr      : e10t [0x0000ffff83ffb2f8] STP      x11,x12,[x7],#0x10    v[0x0000ffff95ceb000]t tr
```

The ninth Instruction record in the trace, in record 15, represents a store-pair instruction that takes a synchronous Tag Check Fault and triggers a Data Abort exception. As the exception is caused by this instruction, it has the `trap` flag set. As the store was not performed, it does not have the `is_write` flag set. It has the `ea_valid` flag and `ea` field set with the attempted virtual address the store attempted to access (if it was known), and `ea_tagged` is set to indicate that a Tag Checked access was performed.

Table 30, Record 15: Trapping store-pair Instruction with Data Abort

Bits	Field name	Value and explanation
[7:0]	<code>type</code>	0x01: ASTF_REC_INSTR
[17:17]	<code>ea_tagged</code>	0b1: This instruction accessed tag memory as the memory access was Tag Checked
[18:18]	<code>trap</code>	0b1: This instruction triggered a trap
[21:21]	<code>is_write</code>	0b0: This instruction did not write data to memory
[22:22]	<code>ea_valid</code>	0b1 and <code>is_branch</code> = 0b0: This instruction is a memory operation, address encoded in the <code>ea</code> field
[31:24]	<code>ea_size</code>	15: This instruction accessed 16 bytes (15+1) at this address
[63:32]	<code>iword</code>	0xa88130eb: Encoding for STP x11,x12,[x7],#0x10
[127:64]	<code>pc</code>	0x0000ffff83ffb2f8
[191:128]	<code>ea</code>	0x0000ffff95ceb000: the Virtual address accessed by instruction

5.1.12 Record 16: Trap, synchronous exception

```
16 trap      : Synchronous Exception to <0xffff7dffe7fc400> : 100100 Data Abort from lower
```

The Trap record after the faulting store-pair instruction gives information about the exception that was triggered, and the exception handler address to which the PC is redirected. The `trap_type` field is set to signal a Synchronous Exception, and the `ec` field contains the Exception Class indicating a Data Abort from lower with value 0x24. The `target_pc` field gives the PC of the next instruction, which is the first instruction of the exception handler.

Table 31, Record 16: Trap information of Data Abort

Bits	Field name	Value and explanation
[7:0]	<code>type</code>	0x05: ASTF_REC_TRAP
[18:16]	<code>trap_type</code>	0x0: Synchronous exception
[31:26]	<code>ec</code>	0x24: Data Abort from lower
[127:64]	<code>target_pc</code>	0xffff7dffe7fc400: Target PC redirected to for exception handler

5.1.13 Record 17: Timestamp

17 timestamp : Value: 102

Record 17 is a Timestamp record which contains a newly updated timestamp after the context switch, as 2 microseconds of time have passed. This is recorded by value 102 in the `value[0]` field.

Table 32, Record 17: Second Timestamp at context switch

Bits	Field name	Value and explanation
[7:0]	type	0x45: ASTF_REC_TIMESTAMP
[15:8]	unit	0x01: Microseconds
[127:64]	value[0]	102

5.1.14 Record 18: Context switch to EL1

18 context : CPU in EL1, non-secure, handler-mode SVE256, TC-asym

Record 18 is a Context record to signal the execution mode change when taking the exception. The exception is taken to EL1, handler mode, so the `el` field is set to 1, and the `handler` flag is set. The `pid_valid` and `tid_valid` flags are no longer set as there is no valid PID/TID information for what is running in EL1. EL0 was configured differently than EL1, therefore several other fields changed:

- The SVE vector length in EL1 is configured for 256 bits, so `evl` is set to 1
- SSBS mode is no longer enabled, so `pstate_ssbs` is not set
- The MTE Tag Check Fault mode is set to asynchronous, so `mte_tcf` is set to 0b11.

Table 33, Record 18: New Context of exception handler

Bits	Field name	Value and explanation
[7:0]	type	0x08: ASTF_REC_CONTEXT
[10:10]	pstate_ssbs	0b0: SSBS mode is disabled
[35:32]	evl	0b1: Indicating a 256-bit SVE vector length
[50:49]	mte_tcf	0b11: Asymmetric MTE Tag Check Fault mode, sync on loads and async on stores
[57:56]	el	0b01: Execution is in EL1
[60:60]	secure	0b0: Non-secure mode
[61:61]	handler	0b1: Thread handler mode: ELx uses SP_ELx
[62:62]	pid_valid	0b0: The pid field does not hold valid data
[63:63]	tid_valid	0b0: The tid field does not hold valid data

5.1.15 Record 19: Instruction enabling SSVE mode

```
19 instr      : e11h [0xffff7dffff7fc400] SMSTART SM
```

Record 19 contains the first instruction of the exception handler, which switches the PE into Streaming SVE mode. and it has the `iword` and `pc` fields set. As this changes the execution state, it is followed by a Context record to inform us of the new mode in Record 20.

Table 34, Record 19: First Instruction of exception handler

Bits	Field name	Value and explanation
[7:0]	type	0x01: ASTF_REC_INSTR
[63:32]	iword	0xd503437f: Encoding for SMSTART SM
[127:64]	pc	0xffff7dffff7fc400

5.1.16 Record 20: Context for new SSVE state

```
20 context    : CPU in EL1, non-secure, handler-mode SSVE512, TC-asym
```

Record 20 is a Context record to signal that the execution state switched from SVE to SSVE mode after the SMSTART instruction. In SSVE mode, the vector length changed to 512 bits.

Table 35, Record 20: New Context for Streaming SVE mode

Bits	Field name	Value and explanation
[7:0]	type	0x08: ASTF_REC_CONTEXT
[8:8]	pstate_sm	0b1: Streaming SVE mode is enabled
[35:32]	evl	0x3: Indicating a 512-bit SSVE vector length

5.1.17 Record 21: Instruction, indexed SME access

```
21 instr      : ellh [0xffff7dffe7fc404] LDR      ZA[w12,4], [x0,#4,MUL VL] i{5} v[0xffff8000053bd500]
```

Record 21 contains the second instruction of the exception handler, and the last record of this example sequence. This instruction loads 64 bytes from memory into an SME ZA array vector. It has `xattr_mode` set to `0x2` to indicate that the extended attribute contains SME information, and `xattr.sme.index` contains the index into the ZA array to identify the vector that the data was loaded into.

As the instruction performs a contiguous access equal to the current streaming vector length of 512 bits, `ea_size` is set to 63 to represent a 64-byte access. The `ea_valid` and `is_read` flags are set, and the accessed memory address is held in the `ea` field. The access was not a Tag Checked access, therefore `ea_tagged` is not set.

Table 36, Record 21: Last Instruction of the example

Bits	Field name	Value and explanation
[7:0]	<code>type</code>	0x01: <code>ASTF_REC_INSTR</code>
[9:8]	<code>xattr_mode</code>	0b10: This instruction has an extended attribute with SME information
[17:17]	<code>ea_tagged</code>	0b0: This instruction did not access tag memory
[20:20]	<code>is_read</code>	0b1: This instruction read data from memory
[22:22]	<code>ea_valid</code>	0b1: Instruction is a memory operation, address encoded in <code>ea</code> field
[31:24]	<code>ea_size</code>	63: This instruction accessed 64 (63+1) bytes of memory
[63:32]	<code>iword</code>	0xe1000004: Encoding for <code>LDR ZA[w12,4], [x0,#4,MUL VL]</code>
[127:64]	<code>pc</code>	0xffff7dffe7fc404
[191:128]	<code>ea</code>	0xffff8000053bd500: Virtual address accessed by instruction
[255:192]	<code>xattr.sme.index</code>	0x05: This instruction loaded data into the SME ZA array vector at index 5