RISC-V N-Trace (Nexus-based Trace)

Version 0.9.17, March 7, 2023: This document is in Development state. Assume it may change.

Table of Contents

Preamble

Document State

PDF generated on: 2023-03-07 15:59:52 UTC

2023/03/07: Version 0.9.17-Development

> See wiki.riscv.org/display/HOME/Specification+States for explanation of specification states.

RISC-V N-Trace (Nexus-based Trace) Specification

Document State

PDF generated on: 2023-03-07 15:59:52 UTC

2023/3/7: Version 0.9.17-Development

See wiki.riscv.org/display/HOME/Specification+States for explanation of specification states.

Chapter 1. Version and History

[TODO] This is a working area describing (agreed) changes which still need to be done.

- Clarify 'Implicit Return Mode' (swap reasons)
- Make sure all option bits are described in Trace Control.
- Describe differences between non-optimized/optimized BTM/HTM modes.

Table 1. History of Changes

Date	Version (Major. Minor. Edit)	Version description
TODO	1.0.0	**Ready for freeze (older history removed)**

Date	Version (Major. Minor. Edit)	Version description
2023/3/7	0.9.17	**After discussion during TG meeting (Jay’s notes)** Extended B-TYPE to distinguish interrupts and exceptions Added 'Virtual Addresses' chapter. Added 'Corner Cases and Sequences' Removed 'Alternative Messages' Clarified I-CNT overflow (with example). Added 'Custom Instructions' chapter. All TOC chapters have numbers. Used 'direct conditional branch' and 'unconditional jump' everywhere (to avoid confusion) - also added to 'Definitions and Terminology' Swapped RCODE=0/1 (big mistake). Expanded 'Definitions and Terminology'. Clarifying reason for two itype tables. Using MSEO[1:0] in a consistent way. Clarifying Nexus conformance levels, 1-bit variable-length. Added ECODE (as variable-length) instead of tricky PAD. Clarifying PROCESS (RTL-like bit encoding). Making HIST field in ProgramTraceCorrelation compulsory in HTM mode (Nino’s comment as well). Other smaller changes as addressed by Jay.

Date	Version (Major. Minor. Edit)	Version description
2023/2/16	0.9.16	**Pure formatting changes (pages/chapters all the same!)** Fixed links displayed as 'Section …'. Added missing '-' to some of B-TYPE/B-CNT/F-ADDR/U-ADDR/I-CNT field names. Removed versions from links to E-Trace and RISC-V Trace control. Some typos, missed fields, hex TCODE notes from email (by Nino) taken into account. Removed 'max' from list of fields (as Nino suggested). Adjusted width of some tables (so all names are fit).
2023/2/14	0.9.15	Added a/the/an articles (in many places) and some minor spelling/grammar fixes.
2023/2/14	0.9.14	Update (with PDF generated)
2023/1/31	0.9.13	Improved (not published)
2023/1/10	0.9.12	Initial version.

Trace/debug tool should read trTeImpl register (described in details in RISC-V Trace Control Interface specification) and extract trTeProtocolMajor and trTeProtocolMinor fields from it.

Chapter 2. Introduction to N-Trace

N-Trace specification provides specification of complete, end-to-end, trace system for RISC-V cores, harts and SoC/MCU designs. N-Trace standard is based on a well established Nexus IEEE 5001 trace standard.

This document is describing N-Trace Trace Encoder and Messaging Protocol version 1.0. It serves multiple audiences:

- N-Trace encoder logic/IP developers.
- Validation teams seeking validation of N-Trace trace implementation.
- Debug and trace tools (probes, decoders, analyzers) developers.

During development of this specification the following key design decisions were made:

- Trace ingress port (connection between hart and trace sub-system) is identical as in ratified E-Trace specification.
- N-Trace messages are kept compatible with the original Nexus specification
  - An appropriate subset applicable to RISC-V was selected.
  - Subset was limited to program trace only, but it will be followed by Nexus compliant data and bus trace.
  - Handful of Nexus-compatible extensions allowing better trace compression are defined.
- Trace control layer defined in Nexus specification was message based and it would be hard to adopt it. Instead, donated by SiFive, a proven working control layer specification was adopted and extended. It assured that in the moment of N-Trace ratification most trace tool vendors will be able to provide full support with minimal changes in trace control software.
  - This control specification was agreed to be shared with ratified E-Trace specification, so the RISC-V trace sub-system will be more unified and easier to understand and handle.
- Trace connectors defined by Nexus were debug oriented, so could not be easily used. Instead, industry standard MIPI-compliant connectors (MIPI20 and Mictor-38) which are supported by all debug and trace probes for a long time are used (with small, generic extensions).
  - These connectors are pure extensions of connectors defined in ratified RISC-V Debug Specification.

> This specification does NOT require developers (both IP developers and trace tool developers) to become familiar with any other documentation besides PDF files provided below. These PDF files are providing links to original PDF files (Nexus Specification, SiFive Control Layer Donation, MIPI Connectors White Paper) as references.

2.1. Related Specifications

This document provides reference to separated documents developed together as part of RISC-V N-Trace Specification:

- Specification of RISC-V Nexus Trace Messages - Defines RISC-V Nexus-based trace messages.
- Specification of RISC-V Trace Control Interface - Defines RISC-V trace control interface.
- Specification of RISC-V Trace Connectors - Defines RISC-V trace connectors (for external trace probes).

Document `Specification of RISC-V Trace Control Interface` is intended to be shared with ratified `Efficient Trace for RISC-V Specification (v2.0.0)` document.

> Currently the above links reference working versions of PDFs. Once ratified, these links will point to officially ratified versions.

> Currently each PDF has a different 'working version' (`v0.9.x`). Ratified set of documents (including this document) will all have identical version numbers (`v1.0.0`).

The following documents are referenced from this specification:

[TODO] - unify above links once settled.

- Efficient Trace for RISC-V Specification - it describes RISC-V Trace Ingress Port signals.
  - At the moment of this writing this is version 2.0 (ratified May 5-th 2022).
- RISC-V Trace Control Interface Specification - it defines a common trace control interface.
  - At the moment of this writing this is version 1.0 (ratified together with this document).

> Above links are pointing into github repositories, as there is no consistent storage or naming conventions for ratified RISC-V specifications.

2.2. Trace Encoder Interfaces

Diagram below shows only a single RISC-V hart. In a system with multiple cores/harts the **Trace Ingress Port**, **Trace Encoder Control** and **Trace Encoder** blocks should be replicated for each hart. The main **Trace Control Layer** controlling other (shared) components in the trace system is not replicated.

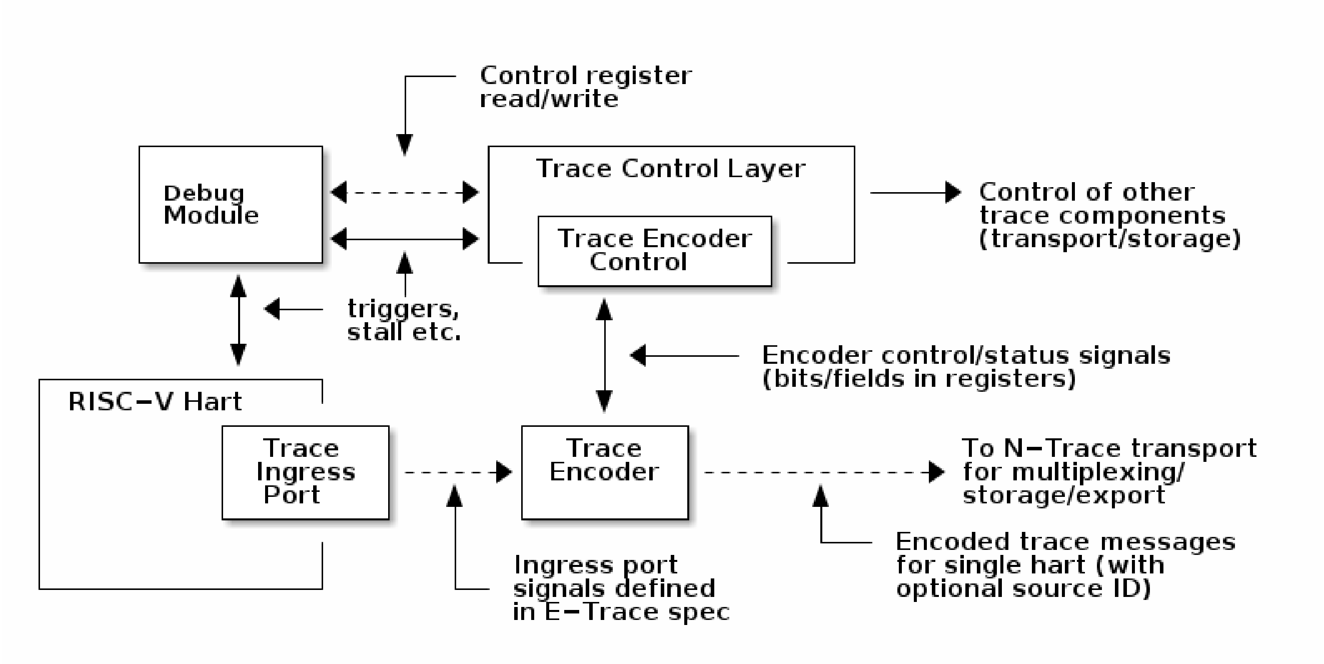

Figure 1. Trace Encoder Interfaces

2.3. Definitions and Terminology

Table 2. Terms Used In This Specification

Term	Definition
Message	N-Trace messages are sequences of bytes. First byte of every message includes the TCODE field, which defines the type of information carried in the message and its format. When messages are transmitted or stored a protocol, described in N-Trace Transmission Protocol chapter, defines the start and the end of each message.
Field	A field is a distinct piece of the information contained within a message, and messages may contain one or more fields (in addition to the first TCODE field). Fields can be either of fixed-length or variable-length. Several fields may be packet into single byte and single field may span across multiple bytes. Definitions of all fields can be found in Fields in Messages chapter.
Variable-length Field	Specifying that a field is variable-length (**Var** used as field size definition) means that the message must contain the field, but that the field's size may vary from a minimum of 1 bit. When messages are transferred or stored, variable-length fields must end on a byte boundary. If necessary, they must zero-fill bit positions beyond the highest order bit of the variable-length data. Because variable-length fields may be of different lengths in messages of the same type, when messages are transmitted or stored a protocol, described in N-Trace Transmission Protocol chapter, defines the end of each variable-length field.
Configurable Field	Configurable field (**Cfg** used as field size) means that existence and size of this field depends on some configuration setting. See N-Trace Specific Trace Controls chapter for details.

Term	Definition
N-Trace	Nexus Based Trace for RISC-V (as defined by this specification).
E-Trace	Efficient Trace for RISC-V (as defined by E-Trace Specification).
Unconditional Jump	On RISC-V ISA all jump instructions are always unconditional, but these two words are always used to avoid any confusions with the term 'branch' used by the Nexus standard.
Direct Conditional Branch	On RISC-V ISA all branch instructions are always direct and conditional (and also relative), but these three words are always used together to avoid confusions with the term 'branch' used by the Nexus standard.
[TODO]	Add more?

Chapter 3. Trace Ingress Port

N-Trace is using the same ingress port as specified in E-Trace Specification (chapter **4 Instruction Trace Interface**).

- As this specification does not define the data trace yet, sub-chapters **4.3 Data Trace Interface requirements** and **4.4 Data Trace Interface** are not applicable.
- It is an ambition to extract single, shared **RISC-V Trace Ingress Port** specifications (combining this chapter with relevant E-Trace chapter).

Table below provides a detailed mapping of encodings of instructions into **itype** signal - it should be used during development of ingress port logic inside of a hart. Please be aware that not only instructions, but also arguments matter (for example jalr rd,rs1 may generate 5 different, distinct **itype** values).

Table 3. Generating itype for different instructions

Instruction Retired	Condition/Notes	itype Value
Interrupted instruction	Any instruction	2 = Interrupt
Exception in instruction	Any instruction	1 = Exception
Conditional branch	Non-taken	4 = Non-taken branch
	Taken	5 = Taken branch
ebreak, ecall, c.ebreak	ecall is reported after retirement	1 = Exception
mret, sret, uret		3 = Exception or interrupt return
cm.jt	Defined by Zcmt extension	0 = No special type
non-jump		0 = No special type
Values of itype (4-bit) needed for Implicit Return Optimization		
jal rd	rd = `link`	9 = Inferable call
	rd != `link`	15 = Other inferable jump
jalr rd, rs1	rd = `link` and rs1 != `link`	8 = Uninferable call
	rd = `link` and rs1 = `link` and rd != rs1	12 = Coroutine swap
	rd = `link` and rs1 = `link` and rd = rs1	8 = Uninferable call
	rd != `link` and rs1 = `link`	13 = Return
	rd != `link` and rs1 != `link`	14 = Other uninferable jump
c.jal	Implicit x1	9 = Inferable call
c.jalr rs1	rs1 = x5	12 = Coroutine swap
	rs1 != x5	8 = Uninferable call
c.jr rs1	rs1 = `link`	13 = Return
	rs1 != `link`	14 = Other uninferable jump

Instruction Retired	Condition/Notes	itype Value
c.j	No registers, only offset	15 = Other inferable jump
cm.jalt	Defined by Zcmt extension	9 = Inferable call
cm.popret*	Defined by zcmp extension	13 = Return
Values of itype (3-bit) without Implicit Return Optimization		
jal rd		0 = No special type
jalr		6 = Uninferable jump
c.j or c.jal		0 = No special type
cm.jalt	Defined by Zcmt extension	0 = No special type
cm.popret*	Defined by Zcmp extension	6 = Uninferable jump

- Symbol `link` means register **x1** or **x5** as specified in **The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA** document.
- **itype** with codes 8..15 are only necessary when Implicit Return Optimization is implemented.
- Tail calls (defined as allowed **itype** values 10 and 11) in E-Trace Specification cannot be distinguished from normal direct/indirect unconditional jumps and as such are impossible to be generated by a hart (unless someone implements Custom Instructions).

Table below defines what N-Trace encoder should do after seeing different **itype** values on trace ingress port.

Table 4. Handling of different itype values

#	itype	Encoder Action	Stack Update
0	None below	Only update I-CNT field.	-
1	Exception	Update I-CNT field. Emit Indirect Branch message with B-TYPE=2 or 1. **IMPORTANT:** An address emitted is known at the next ingress port cycle.	-
2	Interrupt	Update I-CNT field. Emit Indirect Branch message with B-TYPE=3 or 1. **IMPORTANT:** An address emitted is known at the next ingress port cycle.	-

#	itype	Encoder Action	Stack Update
3	Exception or interrupt return	Update I-CNT field. Emit Indirect Branch message with B-TYPE=0. **IMPORTANT:** An address emitted is known at the next ingress port cycle.	-
4	Non-taken branch	For BTM mode: Only update I-CNT field. For HTM mode: Reset I-CNT field. Add 0 as LSB to HIST field. See HIST Field Overflows for handling of overflow.	-
5	Taken branch	For BTM mode: Update I-CNT field. Generate DirectBranch message. For HTM mode: Reset I-CNT field. Add 1 as LSB to HIST field. See HIST Field Overflows for handling of overflow.	-
6	Un-inferable jump if itype is 3-bits wide, reserved otherwise	Update I-CNT field. Emit Indirect Branch message with B-TYPE=0. **IMPORTANT:** An address emitted is known at the next ingress port cycle.	-
7	reserved	-	-
8	Un-inferable call	Same as for **itype**=**6** above.	Push
9	Inferrable call	Same as for **itype**=**0** above.	Push
10	Un-inferable tail-call	**NOT POSSIBLE** (see Custom Instructions)	Pop,Push
11	Inferrable tail-call	**NOT POSSIBLE** (see Custom Instructions)	Pop,Push
12	Co-routine swap	Same as for **itype**=**13** below.	Pop,Push
13	Return	If Pop returns the same address as PC at next ingress port cycle, then same as for **itype**=**0** above. Otherwise the same as for **itype**=**6** above.	Pop
14	Other un-inferable jump	Same as for **itype**=**6** above.	-
15	Other inferable jump	Same as for **itype**=**0** above.	-

As almost every ingress port cycle is updating I-CNT it may overflow. See I-CNT Field Overflows for more details.

> N-Trace encoder does not require **cause** and **tvar** ingress port signals (valid for exceptions and interrupts only) as these are not reported in N-Trace messages. N-Trace is only providing the address of an exception/interrupt handler.

Chapter 4. N-Trace Transmission Protocol

The Nexus standard defines a trace messaging protocol using a number of **MDO** (Message Data Out) signals and one or two flag signals known as **MSEO** (Message Start/End Out). A Nexus message is sent or stored in a record composed of **MDO** and **MSEO**.

N-Trace specification defines 6-bit **MDO** and 2-bit **MSEO** so both fit in a single byte.

- It allows easy storage in memory as well as sending using 1-bit/ 2-bit/ 4-bit/ 8-bit/ 16-bit parallel transport (which is supported by many existing trace probes and connectors).
- Decoding software may work on bytes and 32-bit/64-bit words and expect MSEO bits at two LSB bits of each byte.

N-Trace messages transmission protocol is a strict subset of Nexus trace messaging protocol.

Protocol Feature	Defined in Nexus IEEE 5001	N-Trace (strict subset of Nexus)
Number of **MSEO** bits	1 or 2	2
Number of **MDO** bits	At least 1	6
Total (**MDO**+**MSEO**) bits	At least 2	8 (one byte)
Order (transmitted or stored)	Vendor defined	**MSEO** before **MDO**, each LSB first
Max field size	Not specified	64 bits (some 32 bits or less)
Max message size	Not specified	38 bytes (worst sum of all fields)

Max message size (38 bytes) is calculated for IndirectBranchHistSync message which includes TCODE/ SRC/ SYNC/ B-TYPE(5 bytes total), I-CNT(30 bits, 5 bytes), F-ADDR(63 bits, 11 bytes), HIST(32 bits, 6 bytes) TSTAMP(64 bits, 11 bytes).

- Particular hardware may provide a smaller limit (usually I-CNT is smaller), but always must assure that internal FIFOs must be designed to hold at least two longest messages.
- Decoding software may avoid allocating dynamic memory, but every conforming decoder must survive any size of message as trace memory may be corrupted (trace with all 0-s may be considered as a very long variable-length field).

4.1. MSEO Sequences

MSEO[1:0] bits (on LSB part of each byte) are defined by the follow rules:

- The first byte of a message sends the LSBs of the message and is indicated by **MSEO[1:0]=00**.
- The last byte of a variable-length field is indicated by **MSEO[1:0]=01**.
  - A variable-length field in a message always ends on a byte boundary (zero extended as needed).
  - Bytes occupied by fixed-length fields and initial parts of longer variable-length fields are sent using **MSEO[1:0]=00**.

- The last byte of a message is indicated by **MSEO[1:0]=11**. **It also implies an end of the last (fixed-length or variable-lenght) field of a message.
- Idle bytes (between messages or used as padding) are indicated by **MSEO[1:0]=11** and **MDO[5:0]=111111** (entire byte is **0xFF**).
- Value of **MSEO[1:0]=10** is reserved for future extensions.

Table below provides possible sequences of **MSEO[1:0]** bits (in addition to above rules):

Table 5. Allowed MSEO Transitions

MSEO Function	Dual MSEO[1:0] Sequence
Start of message	11s-00
End of message	00 (or 01)-11-(more 11s)
End of variable-length field	00 (or 01)-01
Message transmission	00s
Idle (no message)	11s
Reserved	any-10

Original Nexus specification defines the MSEO protocol as follows:

- Two `1`-s followed by one `0` indicates the start of a message.
- `0` followed by two or more `1`-s indicates the end of a message.
- `0` followed by `1` followed by `0` indicates the end of a variable-length field.
- `0`-s at all other clocks during transmission of a message.
- `1`-s at all clocks during no message transmission (idle).

Dual MSEO protocol (utilized by this N-Trace specification) is a subset of this general (single and dual) MSEO protocol definition.

4.2. Unified N-Trace Message Structure

Each N-Trace message has identical structure (100% compatible with Nexus):

- Very first field is ALWAYS fixed-length **TCODE** (Transport Code) which defines the meaning and format of subsequent fields.
- In case of simultaneous tracing from more than one hart, the second field is ALWAYS fixed-length **SRC** (Message Source) field, which provides a unique ID of message source.
  - This field allows trace decoders to separate messages from different trace sources (Trace Encoders, harts) without knowing any details of each of the messages.
  - This method can be used to handle different (opaque) trace or debug or performance data using N-Trace transport/storage/export infrastructure.
- One or more (fixed-length or variable-length) payload fields. Sequence and selection of these fields depend on the value of **TCODE** field.

    ◦ In some rare cases one of preceding fields may de

- Very last field is (optional) variable-length **TSTAMP** (Timestamp) field.
    ◦ It may be possible to generate and analyze timestamps in a unified (simpler) way.

4.3. N-Trace Message Example

Table below shows one N-Trace message with several fields. It is an output from N-Trace dump tool (part of N-Trace reference C code) with an added **Explanation** column.

Table 6. MDO and MSEO Encoding Example

Byte	MDO [5:0]	MSEO [1:0]	Decoded (by reference tool)	Explanation
0xFF	111111	11	Idle	Most likely idle, but can also be the last byte of the previous message.
0x70	011100	00	TCODE[6] = 28 - IndirectBranchHist	First byte, all 6 MDO bits have TCODE.
Here we could have an SRC field (it would shift the start of B-TYPE).				
0xD0	110100	00	BTYPE[2] = 0x0	This is a 2-bit (fixed-length) field. As B-TYPE is a fixed-length field, four MSB bits are part of the next field (I-CNT).
0x1D	000111	01	ICNT[10] = 0x7D	This is a second byte of the 7-bit (0x7D) variable-length I-CNT field. Here three MSB bits are all 0-s to assure that the variable-length field uses all 6 MDO bits.
0x1D	000111	01	UADDR[6] = 0x7	This is a single byte variable-length U-ADDR field (with three MSB 0-s bits).
0xF8	111110	00		Normal transfer of new field (6 LSB bits).
0xFF	111111	11	HIST[12] = 0xFFE	Last byte of message. It implies the end of the 12-bit HIST field. In this field we do not have any extra 0-bits on MSB.
Here we could have TSTAMP field (previous MSEO should became 01, what means end of field, but not end of message)				
0xFF	111111	11	Idle	This is idle as this is the second byte with MSEO=11 (NOTE: Last byte of message is also 0xFF).

Chapter 5. N-Trace Specific Trace Controls

This chapter describes how some fields and bits from Trace Encoder control registers are influencing N-Trace messages being generated.

Table 7. Trace Parameters and Controls

Trace Control Field	Bits	How generated messages are affected
trTeProtocolMajor	4	Must be 1 to encode version 1.0 of N-Trace protocol. Value different than 1 is considered a non-compatible version and must be rejected.
trTeProtocolMinor	4	Must be 0 to encode version 1.0 of N-Trace protocol. Different values are considered as down-compatible extensions. Any non-compatible feature should be specifically enabled, so older tools should work with it.
trTeInstMode	3	N-Trace compliant trace encoder must support one or more of the following values: 3: BTM (Branch Trace Messaging) mode 4: Optimized BTM mode 6: HTM (History Branch Messaging) mode 7: Optimized HTM mode See Main N-Trace Trace Modes chapter for more explanations.
trTeInhibitSrc	1	If set to 1 SRC field will NOT be emitted (it is equivalent to set teTrSrcBits = 0).
trTeSrcBits	4	Number of bits of SRC field (in range 0..12). It must be identical for all enabled trace sources in the same trace stream.
trTeSrcID	12	Value of SRC field emitted by this trace encoder. It must be different for each enabled trace source in the same trace stream.
trTeInstEnRepeatedHistory	1	If this bit is set to 1 some sequences of conditional direct branches may be detected and more compressed trace will be generated. See Repeated History Optimization chapter for details.
trTeInstEnSequentialJump	1	If set to 1 encoder may detect indirect flow changes (JR/JALR) following instructions which set a register to a statically known value. See Sequential Jump Optimization chapter for details.

Trace Control Field	Bits	How generated messages are affected
trTeInstEnImplicitReturn	1	If set to 1 some returns from a function may not be reported as indirect flow changes but treated as implicit direct unconditional jumps. [TODO - change to two bits] See Implicit Return Optimization chapter for details.
trTeContext		TODO - describe
trTsEnable		TODO - describe
[TODO]		Add more

Above table does not provide names of trace control registers as names of bits/fields used in Trace Control Interface are unique.

Chapter 6. Main N-Trace Trace Modes

Nexus standard defined two main modes of tracing program flow:

- BTM (Branch Trace Messaging) - every taken direct conditional branch is generating at least two byte message, but repeated branches may be counted and reported as a single message with a count (instead of many identical messages).
- HTM (Branch History Messaging) - every direct conditional branch (taken or not-taken) adds a single bit to the history buffer. It is much more efficient.

Encoder must implement at least one of these modes, however it is unlikely both HTM and BTM modes will be available.

> The Nexus standard defines different conformance levels. These levels are not directly applicable to N-Trace as Nexus levels always include debug levels. Different N-Trace options are provided in N-Trace Specific Trace Controls chapter.

Chapter 7. N-Trace Messages (Overview)

Names `Indirect Branch …` used by Nexus standard may be confusing as RISC-V ISA only allows direct conditional (and always relative) branches. Also RISC-V ISA is differentiating jumps (unconditional flow changes) and branches (conditional flow changes), while in Nexus terminology any flow change (including exceptions/interrupts) are always named as branches.

7.1. Fields in Messages

Table below shows all types of messages. Single row shows all fields in particular message type. Many messages share fields and these fields are always present in the same order.

Attributes of fields is described as follows:

- **[n]** means **n**-bit (fixed-length) field
- **[Var]** means variable-length, at least 1-bit wide, field
- **[Cfg]** means configurable field (existence and size of this field depends on the encoder configuration option)

Table 8. Fields in Messages

Message ID/Field [size]	TCODE [6]	SRC [Cfg]	SYNC [4]	B-TYPE [2]	Other fields	I-CNT [Var]	x-ADDR [Var]	HIST [Var]
Ownership	2	Cfg			PROCESS **[Var]**			
DirectBranch	3	Cfg				Yes		
IndirectBranch	4	Cfg		Yes		Yes	U-ADDR	
Error	8	Cfg			ETYPE **[4]** + ECODE **[Var]**			
ProgTraceSync	9	Cfg	Yes			Yes	F-ADDR	
DirectBranchSync	11	Cfg	Yes			Yes	F-ADDR	
IndirectBranchSync	12	Cfg	Yes	Yes		Yes	F-ADDR	
ResourceFull	27	Cfg			RCODE **[4]** + RDATA **[Var]**			
IndirectBranchHist	28	Cfg		Yes		Yes	U-ADDR	Yes
IndirectBranchHistSync	29	Cfg	Yes	Yes		Yes	F-ADDR	Yes
RepeatBranch	30	Cfg			B-CNT **[Var]**			
ProgTraceCorrelation	33	Cfg			EVCODE **[4]** + CDF **[2]**	Yes		**Cfg**
Vendor Defined	56..62	Cfg	Vendor defined message (dedicated Nexus **TCODE** range)					

Message ID/Field [size]	TCODE [6]	SRC [Cfg]	SYNC [4]	B-TYPE [2]	Other fields	I-CNT [Var]	x-ADDR [Var]	HIST [Var]
Reserved	other	Cfg	Reserved for future extensions of N-Trace specification					

> Any message may include the optional TSTAMP **[Var,Cfg]** field as the very last field of a message (it is not shown in above table because of lack of space). It must be enabled by trTsEnable control bit. Timestamp field always starts at byte-boundary (as it is always preceded by variable-length field). See Timestamp Reporting chapter for more details.

Messages marked as **Reserved** or **Vendor Defined** should be ignored by decoders interested in program flow only. However decoders should provide an option to display/dump them and/or generate a warning as such a message may be seen when trace capture is corrupted. **Vendor Defined** messages can be used for prototyping, debugging, validation and maintenance purposes.

Reference code header github.com/riscv-non-isa/tg-nexus-trace/blob/master/refcode/c/NexRvMsg.h defines all messages in machine-readable format:

```
NEXM_BEG(IndirectBranchSync, 12),
  NEXM_FLD(SYNC, 4),
  NEXM_FLD(BTYPE, 2),
  NEXM_VAR(ICNT),
  NEXM_ADR(FADDR),
  NEXM_VAR(TSTAMP),
NEXM_END(),

NEXM_BEG(ResourceFull, 27),
  NEXM_FLD(RCODE, 4),
  NEXM_VAR(RDATA),
  NEXM_VAR(TSTAMP),
NEXM_END(),

NEXM_BEG(IndirectBranchHist, 28),
  NEXM_FLD(BTYPE, 2),
  NEXM_VAR(ICNT),
  NEXM_ADR(UADDR),
  NEXM_VAR(HIST),
  NEXM_VAR(TSTAMP),
NEXM_END(),
```

> Reference code is using plain C-style identifiers, so the field name as **B-TYPE** will become **BTYPE**.

7.2. Common Fields

Table below provides details for fields which are used in more than one message type. Fields which are present in only one message are described with each message.

Table 9. Details of Common Fields

Name	Bits	Description	Values/Notes
Fields used in many messages			
TCODE	6	Transfer Code	Message header that identifies the number and/or size of fields to be transferred, and how to interpret each of the fields following it. Table
SRC	**Cfg**	Source of Message Transmission	Width of SRC field is defined by trTeSrcBits control field and it may be enabled/disabled by trTeInhibitSrc control bit. This optional field is used to identify the source of the message transmission. In configurations that comprise only a single hart, this field need not be transmitted. For processors that comprise multiple harts, this field must be transmitted as part of the message to identify the source of the message transmission. Within a given device, the SRC field bit size should be the same size across all trace encoders associated with same trace stream.
SYNC	4	Reason for Synchronization	Encodings and details are provided in Synchronization Messages chapter. NOTE: The SYNC field is always sent together with the F-ADDR field, so decoding may start from this message.
B-TYPE	2	Branch Type	Reason for indirect flow change: 0: Standard: Indirect control flow change (jump, call or return). 1: Standard: Exception or interrupt (if the encoder is not capable of reporting 2 and 3). 2: **Extension:**: Exception 3: **Extension:**: Interrupt NOTE: Either 1-only or both 2 and 3 should be implemented and consistently reported. Extended values 2 and 3 allow trace tools to distinguish exceptions and interrupts easily.
I-CNT	**Var**	Instruction Count	As RISC-V allows variable-length instructions, this is a number of 16-bit half-instructions executed/retired since the I-CNT counter was transmitted or reset. See I-CNT Details chapter for more details.
F-ADDR	**Var**	Full Target Address	Full PC address (LSB bit, which is always 0 for RISC-V is skipped). See Address Compression chapter for more details. NOTE: The F-ADDR field is always sent together with the SYNC field.

Name	Bits	Description	Values/Notes
U-ADDR	**Var**	Unique part of Target Address	Unique part of PC address (XOR with recent x-ADDR drop). See Address Compression chapter for more details. The U-ADDR field is always sent together with the B-TYPE field.
HIST	**Var**	Direct Branch History map	MSB = 1 is 'stop-bit', LSB denotes the last direct conditional branch. See HIST Field Generation chapter for more details.
TSTAMP	**Var**	Timestamp (optional)	Either absolute or relative timestamp value. It must be enabled by trTsEnable control bit. See Timestamp Reporting chapter for more details.

Original Nexus specification does not define limits for variable-length fields, but N-Trace provides some limits. It will help to write efficient decoding software but is not limiting hardware in any way.

Table 10. Maximum Field Sizes

Field	Symbol	Bits	Description
SRC	NTRACE_MAX_SRC	12	Determined by size of Trace Control register field. Enough for 4096 (4K) trace sources.
I-CNT	NTRACE_MAX_ICNT	22	Usually a smaller value will be sufficient. MSB bit serves as overflow marker and I-CNT overflow must be generated when it is set.
F-ADDR, U-ADDR	NTRACE_MAX_ADDR	63	LSB bit is always 0 for RISC-V addresses so 63 bits only.
HIST	NTRACE_MAX_HIST	32	It includes stop-bit. This size is optimal for not wasting any bits in very often used ResourceFull messages.
TSTAMP	NTRACE_MAX_TSTAMP	64	It is certainly big enough. It corresponds to architecture defined timer and cycle count registers.

Chapter 8. N-Trace Messages (Details)

This chapter provides a detailed description of all N-Trace messages. Overview of all fields in all messages is provided in the Fields in Messages table above.

Common fields are described in the Common Fields chapter, but fields specific to particular message **TCODE** are explained here.

Size of field in **Bits** column may be one or more of the following values:

- **n (1..6)** - This is an **n**-bits wide, fixed-length field.
- **Var** - This is a variable-length, at least 1-bit wide field.
- **Cfg** - Size of this field depends on configuration setting (**Cfg** fields are always optional).
- **Opt** - This field is optional (depends on the value of one of the preceding fields).

Each message has its own table showing all fields in that message.

> Original Nexus specification is showing tables with **TCODE** (which is sent first) in the last row. This specification shows Fields in Messages in order of sending them (the first field sent is described first). This is consistent with storage, processing and text dump order.

8.1. Ownership Message

This message provides necessary context (privileged mode and Context ID assigned by operating system or hypervisor) allowing the decoder to associate program flow with different parts of code which belong to different programs. It must be explicitly enabled by the trTeContext control bit. It is reported in one of these three conditions:

- When an instruction which is changing privilege mode or **scontext/hcontext** CSR is executed.
- Immediately following any trace synchronization message (the one which includes the SYNC field).
    - If **hcontext** is implemented two messages must follow (first providing **hcontext** and second providing **scontext**). It is necessary so the decoder will be able to locate the code for a specific process.
- At entry and returns to/from exceptions and interrupts (as these are usually changing privilege modes).

Table 11. Ownership Message Fields

Bits	Name	Description
6	TCODE	Value=2(0x2). Standard Transfer Code (TCODE) field.
Cfg	SRC	Standard Message Source (SRC) field.
Var	PROCESS	This is a variable-length field, which encodes **V** and **PRV** privilege mode bits as well as **scontext/hcontext** CSR values. Details are provided below.

Bits	Name	Description
Var,Cfg	TSTAMP	Standard Timestamp (TSTAMP) field.

Explanations and Notes

Field PROCESS is encoded as 4 sub-fields (FORMAT, PRV, V, CONTEXT). Bit layout can be defined in RTL-like syntax as follows:

```
PROCESS[x+5:0] = {CONTEXT[x:0], V[0], PRV[1:0], FORMAT[1:0]}
```

Table 12. Encoding of PROCESS field (in LSB to MSB bit-order)

Reason	FORMAT[1:0]	PRV[1:0]	V[0]	CONTEXT[x:0]
V or PRV change	00	Yes	Yes	—
Reserved	01	—	—	—
Sync or **scontext** change	10	Yes	Yes	**scontext** value
Sync or **hcontext** change	11	Yes	Yes	**hcontext** value

Encodings of **V**/**PRV** follow ISA privilege mode encodings and are encoded as follows:

```
U-mode:     V=0, PRV[1:0]=00
S-mode:     V=0, PRV[1:0]=01
M-mode:     V=0, PRV[1:0]=11
VU-mode:    V=1, PRV[1:0]=00
VS-mode:    V=1, PRV[1:0]=01
```

All unused encodings are reserved.

Examples:

```
PROCESS=0x3B2 = 0b11101_1_00_10   => scontext=0x1D,V=1,PRV[1:0]=00  (VU-mode)
PROCESS=0xC         0b0_11_00     => V=0,PRV[1:0]=11                (M-mode)
```

8.2. DirectBranch Message

This message is generated when the taken direct conditional branch has retired. It is applicable to BTM mode only.

Table 13. Direct Branch Message Fields

Bits	Name	Description
6	TCODE	Value=3(0x3). Standard Transfer Code (TCODE) field.
Cfg	SRC	Standard Message Source (SRC) field.

Bits	Name	Description
Var	I-CNT	Standard Instruction Count (I-CNT) field.
Var,Cfg	TSTAMP	Standard Timestamp (TSTAMP) field.

Explanations and Notes

Last instruction in the code block (or blocks) with all inferable instructions (described by I-CNT) is a taken, direct conditional branch instruction. Next PC is determined by taking [+-]offset (from the opcode of that direct conditional branch instruction) and adding it to an address of direct conditional branch instruction.

> Non-taken direct conditional branches or direct unconditional jumps are NOT generating any trace but increase I-CNT (and direct unconditional jumps are changing PC to direct unconditional jump destination address), so PC of last instruction in code block[s] can be found.

8.3. IndirectBranch Message

This message is generated when an instruction causing indirect control flow change has retired. It is applicable to BTM mode only.

Table 14. Indirect Branch Message Fields

Bits	Name	Description
6	TCODE	Value=4(0x4). Standard Transfer Code (TCODE) field.
Cfg	SRC	Standard Message Source (SRC) field.
2	B-TYPE	Standard Branch Type (B-TYPE) field.
Var	I-CNT	Standard Instruction Count (I-CNT) field.
Var	U-ADDR	Standard Unique Address (U-ADDR) field.
Var,Cfg	TSTAMP	Standard Timestamp (TSTAMP) field.

Explanations and Notes

Last instruction in the code block (or blocks) (described by I-CNT) is an indirect control flow change (jump, call, return) instruction. Next PC is determined by the XOR of the U-ADDR field with the recent address being transmitted (either as F-ADDR or as U-ADDR). See Address Compression chapter for more details.

> Not-taken direct conditional branches or direct unconditional jumps are NOT generating any trace but increase I-CNT (and direct unconditional jumps are changing PC to direct unconditional jump destination address), so PC of last instruction in code block[s] can be found.

8.4. Error Message

Table 15. Error Message Fields

Bits	Name	Description
6	TCODE	Value=8(0x8). Standard Transfer Code (TCODE) field.
Cfg	SRC	Standard Message Source (SRC) field.
4	ETYPE	Error type. Subset of standard Nexus encoding: 0: Standard: Queue Overrun caused messages (one or more) to be lost. 1..7: Standard: Reserved. 8..15: Standard: Reserved for Vendor Defined Error(s).
Var	ECODE	Error code. Subset of standard Nexus encoding (set of bits) 0: Exact reason unknown/not-provided. xxxxxxx1: Standard: Reserved. xxxxxx1x: Standard: Reserved (for data trace in future). xxxxx1xx: Standard: Program Trace Message(s) lost. xxxx1xxx: Standard: Ownership Trace Message(s) lost. xxx1xxxx: Standard: Reserved xx1xxxxx: Standard: Reserved (for data trace in future). x1xxxxxx: Standard: Reserved 1xxxxxxx: Standard: Vendor Defined Message(s) lost. **IMPORTANT:** Implementation may always report this field as 0. It is important to have this field ALWAYS generated as it assures that the TSTAMP field will start at the byte boundary.
Var,Cfg	TSTAMP	Standard Timestamp (TSTAMP) field.

Explanations and Notes

Error Message must be sent immediately prior to a synchronization message as soon as space is available in the Trace Encoder output queue. It should be time-stamped at the moment when the trace messages got dropped.

This message **is required** as otherwise decoder (despite the fact that restart after FIFO overflow is signaled) would not be aware that trace was lost in case of the following sequence of events:

- Trace is turned off by trigger (or from any other reason).
- Message reporting 'trace off' event is lost (due to lack of space for it).
- Trace is never restarted.
- Trace is stopped (this will not generate any trace as trace is turned off)

8.5. ProgTraceSync Message

Table 16. Program Trace Synchronization Message Fields

Bits	Name	Description
6	TCODE	Value=9(0x9). Standard Transfer Code (TCODE) field.
Cfg	SRC	Standard Message Source (SRC) field.
4	SYNC	Standard Synchronization Reason (SYNC) field.
Var	I-CNT	Standard Instruction Count (I-CNT) field.
Var	F-ADDR	Standard Full Address (F-ADDR) field.
Var,Cfg	TSTAMP	Standard Timestamp (TSTAMP) field.

Explanations and Notes

This message is generated at start/restart of trace. I-CNT field must be 0 in such a case. However, for some values of SYNC (like `External Trace Trigger`), I-CNT field may not be 0 and may be used to identify the exact PC location when that particular trigger/event happened. Field F-ADDR provides a full PC address.

8.6. DirectBranchSync Message

Table 17. Direct Branch with Sync Message Fields

Bits	Name	Description
6	TCODE	Value=11(0xB). Standard Transfer Code (TCODE) field.
Cfg	SRC	Standard Message Source (SRC) field.
4	SYNC	Standard Synchronization Reason (SYNC) field.
Var	I-CNT	Standard Instruction Count (I-CNT) field.
Var	F-ADDR	Standard Full Address (F-ADDR) field.
Var,Cfg	TSTAMP	Standard Timestamp (TSTAMP) field.

Explanations and Notes

This message is generated in the same conditions as DirectBranch message, but additionally provides a reason for synchronization (SYNC field) and full PC (F-ADDR field).

8.7. IndirectBranchSync Message

Table 18. Indirect Branch with Sync Message Fields

Bits	Name	Description
6	TCODE	Value=12(0xC). Standard Transfer Code (TCODE) field.
Cfg	SRC	Standard Message Source (SRC) field.
4	SYNC	Standard Synchronization Reason (SYNC) field.
2	B-TYPE	Standard Branch Type (B-TYPE) field.
Var	I-CNT	Standard Instruction Count (I-CNT) field.
Var	F-ADDR	Standard Full Address (F-ADDR) field.
Var,Cfg	TSTAMP	Standard Timestamp (TSTAMP) field.

Explanations and Notes

Last instruction in the code block (described by I-CNT) is an indirect control flow change (jump, call, return) instruction. Next PC is provided as an F-ADDR field in this message.

Not-taken direct conditional branches or direct unconditional jumps are NOT generating any trace but increase I-CNT (and direct unconditional jumps are changing PC to direct unconditional jump destination address).

8.8. Resource Full Message

This message is emitted when the HIST mask or I-CNT counter has reached maximum value for particular encoder implementation.

Table 19. Resource Full Message Fields

Bits	Name	Description
6	TCODE	Value=27(0x1B). Standard: Transfer Code (TCODE) field.
Cfg	SRC	Standard Message Source (SRC) field.

Bits	Name	Description
4	RCODE	Standard Resource Code field (defines a meaning of RDATA fields). 0: Standard: I-CNT counter has overflowed and is reported in the RDATA[0] field. 1: Standard: HIST field has overflowed and is reported in the RDATA[0] field. **2: Extension:** HIST field has overflowed and is repeated. RDATA[0] field holds HIST value and RDATA[1] field holds HREPEAT (History Repeat) value. 3..7: Standard: Reserved for future encodings. 8..15: Standard: Reserved for vendor specific encodings.
Var	RDATA [0]	Standard: For RCODE=0, this is the I-CNT field. For RCODE=1 this is the HIST field (with MSB=1 being stop-bit). **Extension:** For RCODE=2 this is the HIST field (with MSB=1 being stop-bit).
Var,Opt	RDATA [1]	**Extension:** When RCODE=2 is reported this field includes HREPEAT (History Repeat) count.
Var,Cfg	TSTAMP	Standard Timestamp (TSTAMP) field.

Explanations and Notes

- I-CNT value (with RCODE=0) will be reported with the MSB bit in the NTRACE_MAX_ICNT-bit counter. It is just a simple counter, but when MSB bit is set a message with overflown I-CNT should be generated.
  - See I-CNT Field Overflows chapter for more details.
- Not repeated HIST field overflow (RCODE=1) will usually include the longest supported by a particular encoder HIST field.
  - However any number of HIST bits may be transmitted (from 2 to NTRACE_MAX_HIST bits).
  - See Repeated History Optimization chapter for more details.
- Both I-CNT and HIST cannot overflow at the same time as adding bit to HIST is (which may overflow HIST field) is resetting the I-CNT counter.

8.9. IndirectBranchHist Message

Table 20. Indirect Branch History Message Fields

Bits	Name	Description
6	TCODE	Value=28(0x1C). Standard Transfer Code (TCODE) field.
Cfg	SRC	Standard Message Source (SRC) field.

Bits	Name	Description
2	B-TYPE	Standard Branch Type (B-TYPE) field.
Var	I-CNT	Standard Instruction Count (I-CNT) field.
Var	U-ADDR	Standard Unique Address (U-ADDR) field.
Var	HIST	Standard Branch History (HIST) field.
Var,Cfg	TSTAMP	Standard Timestamp (TSTAMP) field.

Explanations and Notes

Last instruction in the code block (or blocks) (described by HIST and I-CNT fields) is an indirect control flow change (jump, call, return) instruction or this packet is generated when exception or interrupt is reported in the ingress port. See HIST Field Generation and I-CNT Details chapters for clarifications.

Next PC (after indirect unconditional jump or exception/interrupt handler) is determined by the XOR of the U-ADDR field with the recent address being transmitted (either as F-ADDR or as U-ADDR). See Address Compression chapter for more details.

8.10. IndirectBranchHistSync Message

Table 21. Indirect Branch History with Sync Message Fields

Bits	Name	Description
6	TCODE	Value=29(0x1D). Standard Transfer Code (TCODE) field.
Cfg	SRC	Standard Message Source (SRC) field.
4	SYNC	Standard Synchronization Reason (SYNC) field.
2	B-TYPE	Standard Branch Type (B-TYPE) field.
Var	I-CNT	Standard Instruction Count (I-CNT) field.
Var	F-ADDR	Standard Full Address (F-ADDR) field.
Var	HIST	Standard Branch History (HIST) field.
Var,Cfg	TSTAMP	Standard Timestamp (TSTAMP) field.

Explanations and Notes

Last instruction in the code block (or blocks) (described by HIST and I-CNT fields) is an indirect control flow change (jump, call, return) instruction or this packet is generated when exception or interrupt is reported in the ingress port. See HIST Field Generation and I-CNT Details chapters for clarifications.

Next PC (after indirect unconditional jump or exception/interrupt handler) is provided as an F-ADDR field. See Address Compression chapter for more details.

8.11. RepeatBranch Message

Table 22. Repeat Branch Message Fields

Bits	Name	Description
6	TCODE	Value=30(0x1E). Standard Transfer Code (TCODE) field.
Cfg	SRC	Standard Message Source (SRC) field.
Var	B-CNT	Standard Branch Count field. Number of times the previous branch message is repeated. Generated if I-CNT, HIST and target address is the same as in the previous branch message.
Var,Cfg	TSTAMP	Standard Timestamp (TSTAMP) field.

Explanations and Notes

This message is reported when an identical branch message is encountered (just to save trace bandwidth). Trace decoder should just repeat handling of previous branch message B-CNT times.

8.12. ProgTraceCorrelation Message

This message is emitted when the trace is disabled or stopped.

Table 23. Program Trace Correlation Message Fields

Bits	Name	Description
6	TCODE	Value=33(0x21). Standard Transfer Code (TCODE) field.
Cfg	SRC	Standard Message Source (SRC) field.
4	EVCODE	Reason to generate Program Correlation 0: Standard: Entry into Debug Mode. Required (do not send 4 instead!). 1: Standard: Entry into Low-power Mode. Optional. 2..3: Standard: Reserved for data trace. 4: Standard: Program Trace Disabled (hart is still running). Optional. 5..7: Standard: Reserved for future extensions of N-Trace specification. 8..15: Standard: Reserved for vendor specific encodings.

Bits	Name	Description
2	CDF	Define number of CDATA fields following it, 0: Standard: Only I-CNT field follows and there is no HIST field. 1: Standard: I-CNT field and single CDATA (HIST) field (for HTM trace). 2..3: Standard: Reserved for future extensions of N-Trace specification. **IMPORTANT:** IN BTM trace mode CDF must be 0. In HTM trace mode CDF must be 1 (even if HIST is empty=0x1).
Var	I-CNT	Standard Instruction Count (I-CNT) field.
Var,Cfg	HIST	Standard Branch History (HIST) field. **This field must be present in HTM mode** so decoder does not need to read CDF to determine it's existence.
Var,Cfg	TSTAMP	Standard Timestamp (TSTAMP) field.

Explanations and Notes

It provides a reason (in EVCODE field) plus I-CNT and HIST fields, which allows the decoder to determine the PC where an execution or the trace actually stopped.

Chapter 9. Field Encoding and Calculation Techniques

This chapter describes in detail how key fields (I-CNT, HIST, U-ADDR/F-ADDR and TSTAMP) are calculated and encoded.

9.1. Address Compression

Address transmissions are fully compliant with the Nexus specification.

- Only virtual (execution) addresses are reported. See Virtual Addresses chapter below for clarifications.
- Address fields are being sent beginning with bit 1 since all execution addresses are on 2-byte boundaries.
- Addresses sent in U-ADDR compressed form are computed based on a reference address sent by or computed from the most recent preceding message containing an address field.
- Starting with an F-ADDR, each U-ADDR modifies the reference address used for the next address.
- A U-ADDR is generated by XORing the full address with the reference address and sending the result starting with bit 1 and with high-order zeroes suppressed.
- The reverse process is used by software to recover the original full address.

Example:

Table 24. Address XOR Compression Example

Address	U-ADDR XOR calculations	F-ADDR/U-ADDR field sent	New REF Address
0x3FC04		F-ADDR=1_1111_1110_0000_0010=0x1FE02	0x3FC04
0x3F368	REF =0011_1111_1100_0000_0100 addr=0011_1111_0011_0110_1000 XOR =0000_0000_1111_0110_1100	U-ADDR=111_1011_0110=0x7B6	0x3F368
0x3E100	REF =0011_1111_0011_0110_1000 addr=0011_1110_0001_0000_0000 XOR =0000_0001_0010_0110_1000	U-ADDR=1001_0011_0100=0x934	0x3E100

9.2. Virtual Addresses

RISC-V ISA defines 3 different virtual memory addressing modes: Sv39, Sv48 and Sv57. In each of these modes the most significant bit (38, 47 or 56) is extended on all higher bits. It means that there is no need to send full 64-bit addresses and only report 39/48 or 57 bits of an address.

In case more than one mode is supported by a hart, the trace encoder address is a fixed width that depends on the maximum width of a virtual address implemented by particular hart. So if hart has have both Sv39 and Sv48, trace addresses will be 48 bits. An Sv39 virtual address requires that bits 38 through 63 are all the same (either all 1s or all 0s). So sign-extending from bit 47 (MSB bit in Sv48) will work for any kind of virtual address, including M-mode. A high address in Sv39 such as FFFF_FFC0_0000_0000 will be visible in the 47-bit F-ADDR field as 7FE0_0000_0000 (48-bit FFC0_0000_0000 address shifted 1 bit to the right to skip always-zero LSB address bit).

Decoders must be aware of maximum supported Sv39/Sv58/Sv59 virtual address mode by a hart to know which bit must be extended.

Table below shows how some boundary addresses for different virtual memory modes are reported in N-Trace packets:

Table 25. Sign Extension of Virtual Addresses

Full 64-bit Address	Meaning	Address Reported	F-Addr Sent
0000_0000_0000_0000	Min low (any mode)	0	0
0000_003F_FFFF_FFFE	Sv39 max low	3F_FFFF_FFFE	1F_FFFF_FFFF
FFFF_FFC0_0000_0000	Sv39 min high	40_0000_0000	20_0000_0000
FFFF_FFFF_FFFF_FFFE	Sv39 max high	7F_FFFF_FFFE	3F_FFFF_FFFF
0000_7FFF_FFFF_FFFE	Sv48 max low	7FFF_FFFF_FFFE	3FFF_FFFF_FFFF
FFFF_8000_0000_0000	Sv48 min high	8000_0000_0000	4000_0000_0000
FFFF_FFFF_FFFF_FFFE	Sv48 max high	FFFF_FFFF_FFFE	7FFF_FFFF_FFFF
00FF_FFFF_FFFF_FFFE	Sv57 max low	FF_FFFF_FFFF_FFFE	7F_FFFF_FFFF_FFFF
0100_0000_0000_0000	Sv57 min high	100_0000_0000_0000	80_0000_0000_0000
FFFF_FFFF_FFFF_FFFE	Sv57 max high	1FF_FFFF_FFFF_FFFE	FF_FFFF_FFFF_FFFF

F-ADDR Sent column is one bit shift right (as LSB bit is not encoded) of **Address Reported** column.

9.3. HIST Field Generation

When the encoder is operating in HTM mode direct conditional branches do NOT generate any messages. Instead each taken or not-taken direct conditional branch is adding a single bit as LSB bit of HIST field (simple left-shift register). If a direct conditional branch is taken, bit=1 is added at the LSB position. If a direct conditional branch is not-taken, bit=0 is added at the LSB position.

MSB value 1 in the HIST field is used as a stop-bit. It allows the HIST field to be transmitted as a variable-length field efficiently (as MSB=0 bits are not transmitted).

Examples:

```
Binary(MSB-LSB):   101=0x5  (two direct conditional branches, not-taken and taken)
Binary(MSB-LSB):  1111=0xF  (three direct conditional branches, all three taken)
Binary(MSB-LSB): 10000=0x10 (four direct conditional branches, all four not-taken)
Binary(MSB-LSB):     1=0x1  (no direct conditional branches at all)
```

The HIST field is reset (to 1, which is just a stop-bit with no bits encoding direct conditional branches) each time it is transmitted (including when any synchronization message is transmitted).

As LSB bit encodes the last direct conditional branch, decoders must interpret the HIST field starting from MSB bit (the one before stop-bit = 1). This is the bit which is describing the first encountered (taken or not-taken) direct conditional branch.

9.3.1. HIST Field Overflows

The HIST field is usually implemented as a shift register (initialized to 1 at reset). This register is shifted left and 0 or 1 is added to it. When the MSB bit of this register becomes 1, it means that the stop-bit reached the end of the HIST register and HIST field must be sent before next bit can be added.

If this is happening, a ResourceFull with the HIST field (RCODE=1 or 2) must be generated.

> Trace decoders do not have to be aware about the actual size of the HIST field implemented by the encoder, however in order to allow efficient implementation of trace encoders (and also allowing HIST pattern detection) N-Trace implementation limits HIST size to max 32-bits. Longer HIST fields would not provide much gain and are making HIST pattern detection more costly (in terms of hardware resources).

When a HIST buffer is identical in two or more consecutive ResourceFull messages, it can be detected and reported using the HIST + HREPEAT (History Repeat Counter) instead of many identical messages.

See Repeated History Optimization chapter for more details.

9.4. I-CNT Details

Field I-CNT (present in most messages) includes count of 16-bit instruction units reported as retired.

Here are key rules how encoder must calculate I-CNT field:

- Every retired instruction MUST increment I-CNT by 1 (for 16-bit instruction) or by 2 (for 32-bit instruction). Specifically:
  - If an instruction is changing the PC, that instruction itself MUST update the I-CNT.
  - An exception or interrupt before retirement of an instruction CANNOT update the I-CNT.
  - An exception or interrupt after retirement of an instruction MUST update the I-CNT.
  - In case of longer instructions (48-bit, 64-bit, ...) (future ISA standards or custom) I-CNT may

increment by 3 or more.

- Reset of I-CNT is described in the I-CNT Resets chapter below.

9.4.1. I-CNT Handling in BTM mode

As an illustration, let's consider the following piece of pseudo-code (… does not matter):

```
0x100:  c.add ...       ; Plain linear 16-bit instruction
0x102:  b... 0x200      ; Direct conditional branch (32-bit instruction)
0x106:  add ...         ; Plain linear 32-bit instruction
0x10A:  b... 0x300      ; Direct conditional branch (32-bit instruction)
0x10E:  c.add ...       ; Plain linear 16-bit instruction
0x110:  add ...         ; Plain linear 32-bit instruction
0x114:  c.ebreak        ; 16-bit breakpoint (to stop the code)
...
0x200:  c.add ...       ; Plain linear 16-bit instruction
0x202:  c.ebreak        ; 16-bit breakpoint (to stop the code)
...
0x300:  add ...         ; Plain linear 32-bit instruction
0x304:  c.ebreak        ; 16-bit breakpoint (to stop the code)
```

In the description below the range specified as <0x100..0x106) means that address 0x100 is included, but address 0x106 is NOT included.

Let's assume we start a trace from address 0x100 (ProgTraceSync with **I-CNT=0** and F-ADDR encoding address = 0x100 should be generated) and let's assume that we executed and collected a trace for above program (in BTM mode) three times:

- First time a direct conditional branch at address 0x102 is taken.
  - A DirectBranch message with **I-CNT=3** should be generated. It means, that a code block from <0x100..0x106) (as 6=2*3) was executed and a direct conditional branch at the end of this block was taken. Decoder will know PC=0x200 from an opcode of the direct conditional branch at an address 0x102.
  - Next message should be ProgTraceCorrelation with **I-CNT=1** describing range <0x200..0x202) till **c.ebreak** instruction
- Second time a direct conditional branch at address 0x102 is not-taken and a direct conditional branch at address 0x10A is taken.
  - A DirectBranch message with **I-CNT=7** should be generated. It means, that a code block from <0x100..0x10E) (as 0xE=2*7) was executed and a direct conditional branch at the end of this block was taken. Decoder will know PC=0x300 from an opcode of the direct conditional branch at an address 0x10A.
  - Next message should be ProgTraceCorrelation with **I-CNT=2** describing range <0x300..0x304) till **c.ebreak** instruction.
- The third time both direct conditional branches are not-taken.
  - In this case only ProgTraceCorrelation with **I-CNT=10** should be generated.It is describing a

range <0x100..0x114) till **c.ebreak** instructions.

> Decoder must look at each instruction in the code block to know its size. It cannot calculate **current PC+I-CNT*2** as it is UNKNOWN what is the size of the last instruction retired in that block - it may be (compressed) 16-bit or 32-bit (not-compressed) direct conditional branch. Without knowing an instruction size offset of that direct conditional branch cannot be determined.

Above we analyzed some I-CNT values. Let’s consider other I-CNT values.

- **I-CNT=1** is the correct value. The only valid reason to generate a message with I-CNT=1 would be an exception (or interrupt) AFTER an instruction at address 0x100. In this case an encoder should generate an IndirectBranch or IndirectBranchSync message with I-CNT=1, B-TYPE=1 (exception) and U-ADDR/F-ADDR field encoding an address of an exception/interrupt handler.
- **I-CNT=5** is also correct (which means that exception/interrupt happened before the retirement of an instruction at an address 0x10A).
- **I-CNT=0** is also possible. It should be generated when an interrupt was pending before we started the code (and trace) and instruction at address 0x100 was not executed/retired. Another reason for I-CNT=0 may be a case, where instruction at address 0x100 will generate page fault (prefetch abort) or is illegal.
- **I-CNT=4 or 6 or 9** are **INCORRECT values** as it would mean that only half of corresponding 32-bit instruction was executed.

> Decoders must report such incorrect I-CNT values and immediately abort decoding as it means that either an encoder is not conforming to this specification or a trace was captured incorrectly. Decoding may resume at the next synchronization message, but it is not mandatory for all decoders to do so.

9.4.2. I-CNT Handling in HTM mode

When the encoder is operating in HTM mode, I-CNT should be incremented at every retired instruction. However direct conditional branches (from code piece above …) will NOT generate any trace packets, but each of them will add a bit to the HIST field and also reset I-CNT.

Above code may generate messages with the following fields (exact types of messages depend on code not visible in that example):

- I-CNT=1, HIST=0b1_1… (MSB=1 is stop bit, bit pattern '1…' means that first direct conditional branch was taken). Encoder should continue from address 0x200 (as the first direct conditional branch encountered was reported as taken) and I-CNT=1 describes a code in <0x200..0x202> range.
- I-CNT=2, HIST=0b1_01… (MSB=1 is stop bit, bit pattern '01…' means that first direct conditional branch was not-taken and second direct conditional branch was taken). Encoder should continue from address 0x300 (as the second direct conditional branch encountered was reported as taken) and I-CNT=2 describes a code in <0x300..0x304> range.
- I-CNT=3, HIST-0b1_00… (MSB=1 is stop bit, bit pattern '00…' means that two direct conditional

branches were not-taken). Encoder should continue from address 0x10E and I-CNT=3 describes a code in <0x10E..0x114) range.

It is obviously visible that HTM mode provides much better trace compression as trace messages are not generated at every taken direct conditional branch.

9.4.3. I-CNT Resets

I-CNT is reset in one of these 3 situations:

- When a trace starts or is restarted (for any reason).
- After I-CNT field is sent in a message (all key messages).
- After a bit is added to HIST buffer (in HTM mode only).

Original Nexus specification does not reset I-CNT when HIST bit is added. During development of this standard 3-rd reset choice was considered as an optional feature but later it became mandatory for N-Trace.

Rationale

When an encoder is operating in HTM mode and the encoder will emit a HIST bit, it is really not necessary to know how many instructions were executed before or between (taken or not) direct conditional branch instructions as decoder (while processing HIST field) must analyze each instruction until direct conditional branch is encountered.

If we look at the above pseudo-code example, when the decoder knows HIST=0b100... pattern, it will analyze the code from instruction at address 0x100. It will continue forward until branch instruction is found. If branch instruction is found, it will either continue to the next PC (if branch was reported as not-taken) or calculate PC (from an opcode at current PC) and continue from branch destination address.

9.4.4. I-CNT Field Overflows

When I-CNT field overflows it may be reported in one of two ways:

- In BTM mode (or when the HIST buffer is empty) the ResourceFull message with RCODE=0 should be generated.
  - This message will be generated only when we have a long instruction block or when we have an infinite loop with unconditional direct jump[s].
- In HTM mode and when the HIST buffer is not empty, I-CNT overflow must be reported using a synchronization message with **SYNC=4 (Sequential Instruction Counter)**.
  - It is needed as otherwise the encoder would not be able to determine if I-CNT overflowed between conditional branches or after the last conditional branch.
  - First choice (**ResourceFull**) is optional - second choice (**SYNC=4**) can be always generated.

To illustrate **Sequential Instruction Counter** generation let's consider the following example code:

```
0x100:  c.add ...       ; Plain linear 16-bit instruction
0x102:  b... 0x200      ; Direct conditional branch (32-bit instruction)
0x106:  c.add ...       ; Plain linear 16-bit instruction
0x108:  add ...         ; Plain linear 32-bit instruction
0x10c:  add ...         ; Plain linear 32-bit instruction
0x110:  add ...         ; Plain linear 32-bit instruction
0x114:  add ...         ; Plain linear 32-bit instruction
0x118:  add ...         ; Plain linear 32-bit instruction
0x11C:  add ...         ; Plain linear 32-bit instruction
0x120:  c.ebreak        ; 16-bit breakpoint (to stop the code)
```

and let's assume (just for simplicity) that the I-CNT counter is 4-bit wide (MSB bit being an overflow flag) and that direct conditional branch at an address 0x102 is not-taken (so code will run from address 0x100 till breakpoint at address 0x120).

Trace of above code should generate 3 messages:

- ProgTraceSync (start of trace)
  - SYNC=3 (Exit from Debug Mode)
  - I-CNT=0 (nothing executed as we are stating)
  - F-ADDR=0x80 (encoding starting address 0x100)
- IndirectBranchHistSync (I-CNT overflown after processing address 0x114)
  - SYNC=4 (Sequential Instruction Counter)
  - **I-CNT=9** (see note below)
  - HIST=0x2 (one not taken conditional branch)
  - F-ADDR=0x8C (encoding address 0x118)
- ProgTraceCorrelation (from address 0x118 till end of trace at 0x120)
  - EVCODE=0 (Entry into Debug Mode)
  - CDF=1 (HIST field present after I-CNT)
  - **I-CNT=4** (see note below)
  - HIST=0x1 (no branches)

> - Overflown **I-CNT=9** decodes instructions from the one at addresses 0x106 to the one at address 0x114.
>   - Instructions at address 0x100 and 0x102 are not counted-up in I-CNT as adding of HIST bit resets the I-CNT.
> - **I-CNT=4** decodes instructions from addresses 0x118 and 0x11C. Debug Mode is entered before c.ebreak instruction (as it never retires), so c.ebreak is NOT included in I-CNT.

This method should be rather easy to implement as each encoder must implement 'periodic sync' (and may implement triggers as well). These will generate synchronization messages at any moment. The only difference between these would be different values of the SYNC field. It means a lot of already present (and required) logic can be reused.

9.5. Synchronization Messages

Synchronization messages provide SYNC code (described below) and full address (field F-ADDR) and are used to synchronize trace encoder as full PC is provided.

Table 26. SYNC Field Values

Value	Name	Encoder Reset	Required	Description
0	External Trace Trigger	**No**	No	This message serves as a marker (encoder state is not reset) of external trigger input. If trace is enabled by a trigger SYNC=5 should be used.
1	Exit from Reset	Yes	No	Core was reset without stopping (by watchdog for example). Address should be a reset vector, but HIST and I-CNT should provide the PC of the last instruction before reset.
2	Periodic Synchronization	Yes	Yes	Just periodic synchronization (to allow decoding the trace from the middle or when it was wrapped around).
3	Exit from Debug Mode	Yes	Yes	Very first synchronization message (unless trace starts disabled - see next chapter)
4	Sequential Instruction Counter	**No**	Yes/No	Generated when I-CNT overflows. See I-CNT Field Overflows chapter for details. Required for HTM mode.
5	Trace Enable	Yes	No	Generated after trace is re-enabled a gap caused by trace being disabled.
6	Trace Event	**No**	No	Serves as a marker (encoder state is not reset) when debug watchpoint with action=4 triggered .
7	Restart from FIFO overrun	Yes	Yes	First synchronization after a gap caused by lost trace
8	Reserved	Yes	-	
9	Exit from Power-down	Yes	No	When the hart is restarted after powered-down. Similar to SYNC=1 (Exit from Reset) described above.
10..13	Reserved	Yes	-	
14..15	Reserved	Yes	-	For vendor defined codes.

Decoders should report different synchronization codes (including reserved codes). Periodic synchronization may only be reported when desired by the user (for debugging?).

- All synchronization messages emit a full TSTAMP field (if enabled).
- Most synchronization messages fully reset the encoder state, so decoding can be started from this message.
    - When trigger is reported (either by debug watchpoint or external trigger) or I-CNT counter overflows, then decoder state is not reset, but still full address and absolute timestamp is reported.

9.6. Corner Cases and Sequences

Normal program flow generates a sequence of messages with I-CNT>0 (reporting at least 1 instruction retired), some HIST fields (to report conditional branches) and x-ADDR fields (to report non-inferable flow changes).

However, sometimes normal flow is interrupted (by exception or interrupt) or some other extra event (trigger/enable/disable) happens and sequence of messages or values of some fields may be a bit unusual. Table below is trying to explain some corner cases.

Table 27. Corner Cases

Sequence of events	Messages Generated
Back to back return	Second message should have I-CNT=1 or 2 (depending on the size of the second return instruction).
Other back to back jumps or branches	Same as above (depending on the size of a second instruction)
Back to back exceptions	Second message with B-TYPE=2 or 1 (Exception) and I-CNT=0 (nothing executed in between).
Exception at interrupt destination	Same as above.
Pending interrupt at start of hart	ProgTraceSync with SYNC=3 followed by message with B-TYPE=3 or 1 (Interrupt).
Exception at first instruction traced	ProgTraceSync with SYNC=3 followed by a message with B-TYPE=2 or 1 (Exception).
Trace starts disabled	ProgTraceCorrelation with EVCODE=4 (Trace Disabled). Once trace is enabled message with SYNC=5 (Trace Enable).
Hart stops with trace disabled	ProgTraceCorrelation with EVCODE=0 (Enter Debug mode) and I-CNT=0 (nothing executed).

9.7. Timestamp Reporting

Timestamp recording must be enabled by trTsEnable trace control bit.

If timestamp is enabled all synchronization messages include an absolute timestamp value with upper zeroes suppressed. Other message types with timestamp emit the timestamp relative to recently reported (absolute or relative timestamp).

> ℹ The TSTAMP field is a variable-length field and MSB bits=0 will not be transmitted. It will provide good compression for relative and absolute timestamps.

To reconstruct the full timestamp, software begins at a synchronization message and stores the TSTAMP value found there, zero-extended to the full timestamp width. Shortly after starting a trace session, even a 64-bit timestamp will typically require far less than 64 bits to transmit. Software extracts the compressed TSTAMP from each message thereafter and adds it with the previous decompressed timestamp to obtain the full timestamp value associated with this message.

The following rules must be observed:

- If timestamps are enabled, ALL synchronization messages (which include full address) must include absolute TSTAMP value.
    - Otherwise some sections of decoded trace would have a timestamp and some not and it would be hard for a programmer to comprehend such a trace.
- It is permitted that some non-synchronization messages are not reporting timestamp but debugger may not be able to provide profiling data.
- Absolute timestamp cannot exceed 64 bits (even with 1ps resolution, 64-bit counters will overflow in about 584 years).
    - Implementation may choose a smaller counter - trace tools may assume timestamp will not overflow in a single session, however it would not be very hard to add support for it.
- It is suggested that in multi-hart systems all Trace Encoders use a shared timestamp (for better code correlation), but it is not necessary.
- Timestamp at all cases, when an address is provided should be at a time when an event leading to that particular address being sent happened.
    - If the above is not possible, timestamps should be at least reported in a consistent way, so distance between distant events can be reliably calculated.
    - It is needed to assure that time reported at exceptions/interrupt handlers will be a moment when exception or interrupt was observed.

Chapter 10. Optional, Optimization Extension to Nexus Standard

N-Trace messages are defined as a strict subset of standard Nexus messages. However in order to provide better compression some optional extensions are defined and must be specifically enabled. Table Details_Control_Parameters describes all control bits to enable these optimizations.

10.1. Sequential Jump Optimization

This optimization must be enabled by trTeInstEnSequentialJump control bit.

By default, the target of an indirect unconditional jump is always considered an uninferable PC discontinuity. However, if the register that specifies the jump target was loaded with a constant then it can be considered inferable under some circumstances. The hart must identify unconditional jumps with sequentially inferable targets and provide this information separately to the encoder. The final decision as to whether to treat the unconditional jump as inferable or not must be made by the encoder. Both the constant load and the unconditional jump must be traced in order for the decoder to be able to infer the unconditional jump target.

Jump targets that are supplied via

- an **lui** or **c.lui** (a register which contains a constant), or
- an **auipc** (a register which contains a constant offset from the PC).

Such unconditional jump targets are classified as sequentially inferable if the pair of instructions are retired consecutively (i.e. the **auipc**, **lui** or **c.lui** immediately precedes the indirect unconditional jump).

> The restriction that the instructions must be retired consecutively is necessary in order to minimize the additional signals needed between the hart and the encoder, and should have a minimal impact on trace efficiency as it is anticipated that consecutive execution will be the norm.

10.2. Implicit Return Optimization

This optimization must be enabled by the trTeInstEnImplicitReturn control bit.

Although a function return is usually an indirect unconditional jump, well behaved programs return to the point in the program from which the function was called using a standard calling convention. For those programs, it is possible to determine the execution path without being explicitly notified of the destination address of the return. The implicit return mode can result in very significant improvements in trace encoder efficiency.

Returns can only be treated as inferable if the associated call has already been reported in an earlier packet. The encoder must ensure that this is the case. This can be accomplished by utilizing a counter to keep track of the number of nested calls being traced. The counter increments on calls and decrements on returns.

The counter will not over or underflow, and is reset to 0 whenever a synchronization packet is sent. Returns will be treated as inferable and will not generate a trace packet if the count is non-zero (i.e. the associated call was already reported in an earlier packet).

Such a scheme is low cost, and will work as long as programs are "well behaved". The encoder will not be able to check that the return address is actually that of the instruction following the associated call. As such, any program that modifies return addresses cannot be traced using this mode with this minimal implementation.

Alternatively, the encoder can maintain a stack of expected return addresses, and only treat a return as inferable if the actual return address matches the prediction. This is fully robust for all programs, but is more expensive to implement. In this case, if a return address does not match the prediction, it must be reported explicitly via a packet. This ensures that the decoder can determine which return is being reported.

As the third alternative call stack may not include all addresses, but only keep some LSB part of it and use them to compare if return is matching the call or not. Changes that program making incorrect return will return to address with the same LSB portion are very slim.

[TODO] Change to two bits (or do not allow 'counter-only')

> Decoder does not need to know what is actual depth of the call stack implemented by encoder but for efficiency reasons it should assume max depth. N-Trace implementation should never implement call stack deeper than 32 levels. Such deep calls will be most likely 'broken' by other events/messages (like periodic SYNC).

10.3. Repeated History Optimization

This optimization must be enabled by the trTeInstEnRepeatedHistory control bit.

When a simple loop is executed many times, it either has a direct conditional branch at the start of a loop (which must be 'taken' to terminate the loop) or has a direct conditional branch at the end of the loop (which must be 'taken' to repeat the loop). In the first case, the direct conditional branch is 'not-taken' most of the time and 'taken' once at the end. In the second case, the direct conditional branch is 'taken' most of the time, but 'not-taken' at the end of the loop.

Long loops in practical programs/functions (memcpy/strcpy/search ...) tend to execute many times and many times flow inside the loop is identical. Instead of sending the same history bits many times, repeated patterns can be detected and counted. This is a big saving! As an example, a memcpy of 4MB buffer using 32-bit transfers will execute at least 1M of direct conditional branches and 1M of history bits must be included in trace (it is a lot of trace).

Nexus standard defines Repeat Branch message. This message will provide a single B-CNT (Branch Count) field instead of generating many identical Direct Branch messages. But this message cannot be used in HTM mode as repeated messages (Direct Branch) do not include the HIST field.

In order to allow generation of repeated history of direct conditional branches in HTM mode an extra encoding for RCODE=2 in Resource Full message is added.

> It is allowed to generate any sequence of Resource Full messages as long as the logically concatenated sequence of (repeated or not …) HIST bits (excluding MSB stop-bit[s]) is the same.

Tracing of such simple, long loops would benefit from generating special messages/fields which provide counters of taken/not-taken direct conditional branches (in a way similar to Repeat Branch message)

But this approach will not work with more complex code with a conditional statement (or several of them) inside of a loop.

In such a case, it is desired to detect repeated sequences of taken/not-taken direct conditional branches and instead generate many messages with HIST fields, generate a message consisting of a HIST pattern and repeat count.

Let's assume that we have a loop, which generates a long sequence of repeated taken/not-taken, taken/not-taken direct conditional branches. Trace may generate Resource Full messages with the following HIST records:

```
Msg#1:
    TCODE=27 (ResourceFull)
    RCODE=1 (HIST record overflow is provided as RDATA)
    RDATA=0b1_01_0101_0101_0101_0101_0101_0101_0101 = 0x55555555
            (stop-bit + pattern 01 repeated 15 times)
Msg#2:
    TCODE=27 (ResourceFull)
    RCODE=1 (HIST record overflow is provided as RDATA)
    RDATA=0b1_01_0101_0101_0101_0101_0101_0101_0101 = 0x55555555
            (stop-bit + pattern 01 repeated 15 times)
...
Msg#10:
    TCODE=27 (ResourceFull)
    RCODE=1 (HIST record overflow is provided as RDATA)
    RDATA=0b1_01_0101_0101_0101_0101_0101_0101_0101 = 0x55555555
            (stop-bit + pattern 01 repeated 15 times)
```

Instead of generating many messages with identical HIST record, encoder can detect repeated pattern and generate the following single message:

```
Msg#1:
    TCODE=27 (ResourceFull)
    RCODE=2 (HIST record overflow is provided as RDATA and
            repeat count is provided as HREPEAT field)
    RDATA=0b1_01_0101_0101_0101_0101_0101_0101_0101 = 0x55555555
            (stop-bit + pattern 01 repeated 15 times)
```

```
    HREPEAT=10  (Repeat Count=10 instead 10 messages)
```

Above example shows a 2-bit pattern, but using the same technique it can be expanded to any size of pattern. Exact way to detect these patterns is not specified as it does not change encoding of messages. So, it is possible to generate the following, a bit smaller, message:

```
Msg#1:
    TCODE=27 (ResourceFull)
    RCODE=2 (HIST record overflow is provided as RDATA and
            repeat count is provided as HREPEAT field)
    RDATA=0b1_01 = 0x5 (stop-bit + single pattern 01)
    HREPEAT=150 (Repeat Count is bigger, but pattern is smaller)
```

> This type of compression (reporting shorter patterns and larger counts) may not be practical as it may save only a little. Trace is compressed a lot already and it really should not matter if we report 150 iterations of a loop in 6 or 7 bytes. Example above is provided to assure that trace encoders must handle this type of trace compression.

Chapter 11. Rules of Generating Messages

This chapter directly mentions 16-bit and 32-bit instructions (from the currently ratified instruction set), but it is applicable to any size being multiple of 16-bit (as main ISA defines).

Main Rules

1. Plain linear instructions and direct, PC relative, direct unconditional jumps generate no trace.
   - These are called inferable instructions, where the next PC can be certainly known from looking at binary code.
2. Only direct conditional branches, indirect flow transfer instructions and exceptions/interrupts generate trace.
   - These are called non-inferable instructions, where the next PC cannot be known by looking at binary code.

Detailed Rules

1. If tracing was disabled and is restarted, a ProgTraceSync message is generated.
   - This message includes the reason for a start (SYNC field) and full address (F-ADDR field).
2. Any retired instruction increments I-CNT field (+1 or +2).
3. The following types of instructions allow trace decoders to know the next PC (nothing else is done for them).
   - Plain linear instruction ⇒ PC is at the next instruction (+2 or +4).
   - Direct (inferable…) unconditional jump ⇒ PC is unconditional jump destination (known from PC and opcode as all unconditional jumps are PC relative).
   - Not taken direct conditional branch (in BTM mode) ⇒ PC is next instruction (+2 or +4).
4. Branch (conditional) instruction is handled as:
   - In BTM mode it generates a DirectBranch message (only if taken).
   - In HTM mode it appends a single bit (1=taken or 0=not-taken) into the branch history buffer (HIST field).
5. In case the trace is stopped or disabled, ProgTraceCorrelation message is generated.
   - It included reason (EVCODE field) and I-CNT and (optional) HIST field, so the last PC can be calculated.
6. In case the generated message includes I-CNT/HIST fields, the corresponding value is reset.
   - In case I-CNT overflows, ResourceFull message (with I-CNT before overflow) is generated and I-CNT is reset.
   - In case HIST overflows, ResourceFull message (with HIST before overflow) is generated and HIST is reset.

Extended Rules

These rules are augmenting the above rules if the corresponding configuration setting is set.

1. Call and return instructions maintain call stack and if return is matching a call, no trace is generated.
    - This is described in detail in Implicit Return Optimization chapter.
2. As RISC-V architecture is only supporting PC relative unconditional jumps/calls, indirect unconditional jumps/calls are used.
    - Such instruction sequences may be detected and in such a case no trace is generated.
3. I-CNT field is reset after every (taken or not-taken) direct conditional branch.
    - Number of instructions between two direct conditional branches does not matter.

11.1. Custom Instructions

Custom instructions (or any future ratified instructions) which are not changing PC flow do not require any special treatment. Trace decoders should only look at instructions which may change PC flow and treat everything else as plain linear instructions.

> Above rule allowed to not change trace decoders after V or Zb extensions were added (as all instructions are plain linear). From all extensions ratified in the last few years only Zcmp and Zcmt extensions include PC changing instructions.

Custom instruction which may change PC should be traced in one of the following ways:

- If the PC just advances to the next instruction, it should be traced as plain linear instruction. Decoder will just advance the PC.
- If the PC changes, it should be traced as an indirect jump, so the destination address will be reported (as F-ADDR or U-ADDR fields). Decoder will change PC to an address specified in this message.

Such an approach will NOT require changes in trace decoders. To illustrate this let's consider the following piece of code with custom instruction **XYZ**:

```
0x100:  add ...         ; Plain linear 32-bit instruction
0x104:  XYZ             ; Custom conditional branch to 0x200 (it does not matter if
direct or indirect ...)
0x108:  c.add ...       ; Plain linear 16-bit instruction
0x10A:  c.ebreak        ; 16-bit breakpoint (to stop the code)
...
0x200:  c.add ...       ; Plain linear 16-bit instruction
0x202:  c.ebreak        ; 16-bit breakpoint (to stop the code)
```

It can be traced as follows (exact type of messages do not matter):

- Single message (if branch was not taken)
    - **I-CNT=5** ⇒ Instruction XYZ did not change the flow and code in range <0x100..0x10A) got executed
- Two messages (if branch was taken)

- I-CNT=4, F-ADDR=0x100 (denote address 0x200) ⇒ Code in range <0x100..0x108) got executed and next PC after instruction XYZ is 0x200
- **I-CNT=1** ⇒ Code in range <0x200..0x202) got executed next

If custom instruction will generate some other trace (for example some new type of direct conditional branch which may add HIST bit), decoders must be extended to be aware about type of this custom instruction.

11.2. Pseudo-code of Simple N-Trace Encoder

Code below is a simplified part of actual C-code used by the reference encoder (in C). It defines two functions:

- NTraceEncoderInit(void) - initialize state of encoder
- NTraceEncoderHandleRetired(uint64_t `addr`, uint32_t `flags`) - handle single retired instruction
  - `addr` - address of retired instruction
  - `info` - information about instruction (type, size, taken/non-taken)

```
// Use N-Trace TCODE messages
#define NEXUS_TCODE_Ownership                   2
#define NEXUS_TCODE_DirectBranch                3
#define NEXUS_TCODE_IndirectBranch              4
#define NEXUS_TCODE_Error                       8
#define NEXUS_TCODE_ProgTraceSync               9
#define NEXUS_TCODE_DirectBranchSync            11
#define NEXUS_TCODE_IndirectBranchSync          12
#define NEXUS_TCODE_ResourceFull                27
#define NEXUS_TCODE_IndirectBranchHist          28
#define NEXUS_TCODE_IndirectBranchHistSync      29
#define NEXUS_TCODE_RepeatBranch                30
#define NEXUS_TCODE_ProgTraceCorrelation        33

// Functions/macros which encode bits in 'info' (example...)
#define INFO_LINEAR   0x1   // Linear (plain instruction or not-taken BRANCH)
#define INFO_4        0x2   // If not 4, it must be 2 on RISC-V
#define INFO_INDIRECT 0x8   // Possible for most types above
#define INFO_BRANCH   0x10  // Always direct on RISC-V (may have LINEAR too)

#define InfoIsBranchTaken(info) (!((info) & INFO_LINEAR))
#define InfoIsSize32(info)      ((info) & INFO_4)
#define InfoIsBranch(info)      ((info) & INFO_BRANCH)
#define InfoIsIndirect(info)    ((info) & INFO_INDIRECT)

// Function which emit N-Trace packets (all are empty here)
void EmitFix(int nbits, uint32_t value);    // Emit fixed-size field
void EmitVar(uint64_t value);               // Emit variable size field
void EmitEnd();                             // Terminate message
```

```
// Encoder configuration options
const bool      enco_opt_branch_history = true;     // Configuration option
const uint32_t  enco_opt_limICNT    = 0x10000;      // Limit of ICNT (max is 6+6+4
bits)
const uint32_t  enco_opt_limHIST    = 0x40000000;   // Limit of HIST (max is 5*6 bits)

// Encoder state variables
static uint32_t encoNextEmit = 0;   // TCODE to be emitted next time
static uint32_t encoICNT = 0;       // ICNT accumulated
static uint32_t encoHIST = 1;       // HIST accumulated (MSB is guardian bit)
static uint64_t encoADDR = 0;       // Last emitted address

void NTraceEncoderInit()
{
    encoADDR = 0;
    encoICNT = 0;   // Empty ICNT and HIST
    encoHIST = 1;

    encoNextEmit = NEXUS_TCODE_ProgTraceSync;
}

void NTraceEncoderHandleRetired(uint64_t addr, uint32_t info)
{
    // Optionally emit what was determined previously
    if (encoNextEmit != 0)
    {
        EmitFix(6, encoNextEmit);   // Emit TCODE (as determined)

        // Emit message fields (accordingly ...)
        if (encoNextEmit == NEXUS_TCODE_ProgTraceSync)
        {
            EmitFix(4, 1);          // Emit SYNC=1  (4-bit)
            EmitVar(encoICNT);      // Emit ICNT    (variable)
            EmitVar(addr >> 1);     // Emit FADDR   (variable)
        }
        else if (encoNextEmit == NEXUS_TCODE_IndirectBranchHist ||
                 encoNextEmit == NEXUS_TCODE_IndirectBranch)
        {
            EmitFix(2, 0);                      // Emit BTYPE=0 (2-bit)
            EmitVar(encoICNT);                  // Emit ICNT    (variable)
            EmitVar((encoADDR ^ addr) >> 1);    // Emit UADDR   (variable)

            if (encoNextEmit == NEXUS_TCODE_IndirectBranchHist)
            {
                EmitVar(encoHIST);              // Emit HIST    (variable)
            }
        }
        else if (encoNextEmit == NEXUS_TCODE_DirectBranch)
        {
            EmitVar(encoICNT);                  // Emit ICNT    (variable)
```

```
        }

        EmitEnd();  // It will mark last entry with MSEO=11 and flush it

        if (encoNextEmit != NEXUS_TCODE_DirectBranch)
        {
            encoADDR = addr;  // This is new address
        }
        encoNextEmit = 0;   // Only one time

        encoICNT = 0;       // Start from 'empty' ICNT and HIST
        encoHIST = 1;
    }

    // Update ICNT
    uint32_t prevICNT = encoICNT;   // In case ICNT will overflow now, we need to emit
previous value ...
    if (InfoIsSize32(info)) encoICNT += 2; else encoICNT += 1;

    // Determine type of packet (only if this is branch or indirect ...)
    if (InfoIsBranch(info))
    {
        if (enco_opt_branch_history)
        {
            // Update branch history buffer (add LSB bit)
            if (InfoIsBranchTaken(info))
                encoHIST = (encoHIST << 1) | 0; // Mark branch as taken
            else
                encoHIST = (encoHIST << 1) | 1; // Mark branch as not-taken
        }
        else
        {
            if (InfoIsBranchTaken(info))
                encoNextEmit = NEXUS_TCODE_DirectBranch;    // Emit destination
address (next retired)
            else
                ;   // Not taken branch is considered as linear instruction
        }
    }
    else
    if (InfoIsIndirect(info))
    {
        if (enco_opt_branch_history)
            encoNextEmit = NEXUS_TCODE_IndirectBranchHist;  // Emit destination
address (next retired)
        else
            encoNextEmit = NEXUS_TCODE_IndirectBranch;      // Emit destination
address (next retired)
    }

    // Optionally emit ICNT overflow
```

```
    if (encoICNT > enco_opt_limICNT) // Instruction count overflown ...
    {
        // Emit ResourceFull with ICNT before this instruction
        EmitFix(6, NEXUS_TCODE_ResourceFull);
        EmitFix(4, 0);                          // RCODE=0 (ICNT overflow)
        EmitVar(prevICNT);                      // RDATA=ICNT
        EmitEnd();  // It will mark last entry with MSEO=11 and flush it

        // Set ICNT for this instruction
        if (InfoIsSize32(info)) encoICNT = 2; else encoICNT = 1;
    }

    // Optionally emit HIST overflow
    if (encoHIST & enco_opt_limHIST) // Is HIST buffer overflown?
    {
        // Emit history BEFORE this instruction (remove LSB bit)
        EmitFix(6, NEXUS_TCODE_ResourceFull);
        EmitFix(4, 1);                          // RCODE=1 (HIST overflow)
        EmitVar(encoHIST >> 1);                 // RDATA=HIST
        EmitEnd();  // It will mark last entry with MSEO=11 and flush it

        // Keep single HIST for this branch (guardian | single LSB bit from encoHIST)
        encoHIST = (0x1 << 1) | (encoHIST & 0x1);
    }
}
```

Chapter 12. N-Trace Decoding Guidelines

To decode an N-Trace encoded stream of messages (as any other compressed trace) access to opcodes of instructions which were executed is necessary. This is usually done by providing an ELF file of a program being executed, but it can also be read-out from the target. Three types of information is needed:

1. Size of each instruction (16-bit or 32-bit).
2. Types of all instructions (as reported via 'itype' signal on trace ingress port).
3. For direct unconditional jumps and direct conditional branches an offset (to jump/branch destination) encoded in an opcode.

At the beginning of the trace 'full PC' (F-ADDR field) is reported. From that moment decoder must follow the code and update PC according to what is provided in messages.

> In order to provide partial decoding of big trace, 'full PC' is dropped periodically. Periodic 'full PC' drop is also needed to decode trace from small, wrapped around buffers.

12.1. Decoding Algorithm Principles

Algorithm to reconstruct complete PC flow from N-Trace messages is very simple:

- Handle HIST field (if available and not 0x1)
  - Analyze code from the current PC through inferable unconditional jumps (all types) and direct conditional branches (each direct conditional branch will 'consume' a single bit from the **HIST** field).
  - At the end (after the LSB bit from **HIST** is processed), the PC will be after the last direct conditional branch (either taken or not-taken).
- Handle I-CNT field (if available and not 0x0)
  - Analyze code from current PC through inferable unconditional jumps (all types) - each encountered direct conditional branch must be treated as not-taken
  - Each encountered instruction should subtract 1 or 2 from I-CNT (depending on a size of particular instruction)
  - It will reach either non-inferable unconditional jump or I-CNT will become 0 to denote that some other 'event' (like exception, interrupt, trace off, trigger etc.) happened.
- At the last step the F-ADDR or U-ADDR field (if available) should be applied. This will be the next PC where analysis of the next trace message should start.
  - This is either a destination address of indirect unconditional jump or an address of an exception/interrupt handler.

> - Phrase **inferable unconditional jumps (all types)** include indirect unconditional jumps, which may be inferable.

- Some messages may encode I-CNT and HIST fields under different names (RDATA/CDATA), but meaning and processing is the same.
- Extra fields like SYNC/B-TYPE only provide extra details, but are NOT essential for a decoder to reconstruct the PC flow.

12.2. Decoding trace from multiple harts

Decoder for specific a hart should only look for messages with SRC for that particular hart. Each encoder for each enabled hart (in same trace stream) must have same 'trTeSrcBits' and different 'trTeSrcID' fields set.

12.3. Decoding trace of complex systems (Linux etc.)

In case of complex systems, where code consists of several independently built programs and libraries, decoders must be aware of different program images (ELF files) at different locations. Ownership messages should provide enough context. Decoders must be also aware of assignment of **scontext/hcontext** values for programs and processes being traced.

Operating systems may decide to migrate single process to different cores/harts. It may also be the case, when different threads from the same process (sharing code …) will run in the same time on more than one core/hart.

> This is not specific to N-Trace - correct tracing of complex, multi-threaded/multi-core systems is not very easy.

12.4. Decoding self-modifying or JIT (Just In Time compiled) code

Trace encoder is just encoding a stream of instructions passed by ingress port from the hart running it, but decoder must be aware of types of all instructions being executed. In case of self modifying code (or JIT code), binary image (at moment of execution) must be available to decoder. How this can be done is not in the scope of this specification.

> This is not specific to N-Trace - every trace system which is compressing execution flow heavily may not handle this case well.

Chapter 13. Nexus Compliance

The Nexus standard provides a lot of flexibility and in general N-Trace can be considered also fully compatible. There are two incompatible, small changes:

- Field ECODE is variable-length field (to assure TSTAMP field is on byte boundary).
- Field I-CNT is additionally reset when HIST bit is addded.

Compatible extensions are described in a dedicated chapter (each of them must be directly enabled).

Chapter 14. Additional Material

14.1. Trace Bandwidth Considerations

- SRC field (if enabled) may change the otherwise optimal layout of Fields in Messages.

14.2. Validation Considerations

- Resource Full message with I-CNT overflow is rare and may not be experienced in normal code. Simplest way to generate is to have an infinite loop and (rare) interrupt handler.
  - This loop should increment a register or memory location - this value should correspond to total accumulated I-CNT.

14.3. Potential Future Enhancements

Table below is proposing some future enhancements for Nexus compatible (N-Trace) messages. These were discussed during the development of the N-Trace specification.

Table 28. Future Enhancements

Enhancement	Conformance	Notes
Instrumentation Data Trace	Nexus Compatible	Very likely (Nexus defines appropriate messages). It will require software to be instrumented by code sending data using trace infrastructure (Arm CoreSight ITM enabled many use-cases).
Selective Data Trace	Nexus Compatible	Very likely (Nexus defines appropriate messages). It will allow sending some data in response to triggers (from debug module or external).
Full Data Trace	Nexus Compatible	Likely (E-Trace supports it), but necessary bandwidth may be a problem.
Smaller field sizes	Nexus Extension	Unlikely (too much of a change). Some of the fields may be made shorter (as not all cases are needed), but it may not be justified.
System Bus Trace	Nexus Compatible	Likely (Nexus defines appropriate messages and there is a need for more than trace of harts).
Additional TCODE	Nexus Extension	Possible, but more real-life examples are needed to justify it.
Single MSEO bit	Nexus Compatible	Unlikely to be considered. It may provide (12.5% instead of 25% MSEO overhead), but it is more complex to handle by both encoder and decoders.
More MDO bits	Nexus Compatible	Very unlikely to be considered. In order to keep byte alignment, 14 or 22 or 30-bit MDO may be considered. Even 14-bit will cause a lot of 'wasted' bits.

Each of the above enhancements should be first prototyped and validated using reference C encoder/decoder.