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

Version 0.9.15, Feb 14, 2023: This document is in Development state. Assume it may change.

Table of Contents

Preamble

Document State

PDF generated on: 2023-02-15 04:01:20 UTC

2023/02/14: Version 0.9.15-Development

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

Chapter 1. Version and History

Table 1. History of Changes

Date	Major.Minor Version	Version description
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]) and extract `trTeProtocolMajor` and `trTeProtocolMinor` fields from it.

Chapter 2. Introduction

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

1. N-Trace encoder logic/IP developers.
2. Validation teams seeking validation of N-Trace trace implementation.
3. Debug and trace tools developers (including trace decoder developers).

This PDF is referencing the following related documents:

- Efficient Trace for RISC-V specification version 2.0 - it described RISC-V Trace Ingress Port signals.
- RISC-V Trace Control Interface specification version 1.0.0 - it defines a common trace control interface.

Above links are pointing into different github locations, as of today there is no consistent storage or naming conventions for different ratified RISC-V specifications.

[TODO] - This MUST be clarified before official ratification of this specification.

2.1. Trace Encoder Interfaces

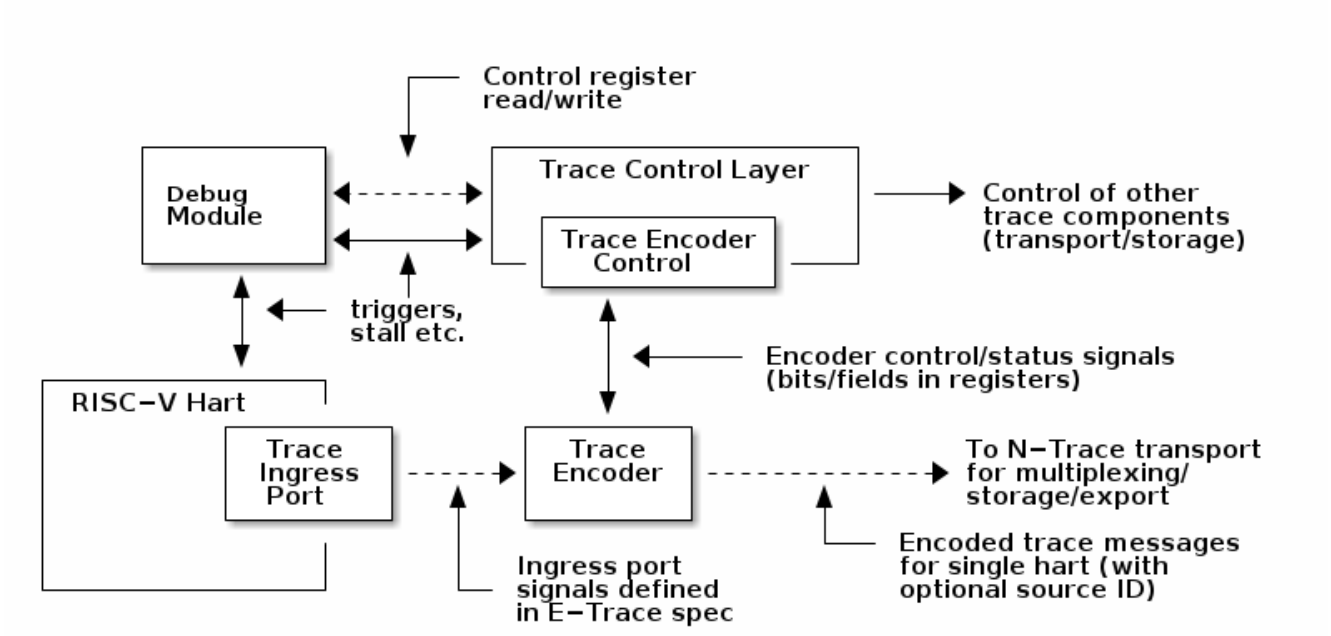

Figure 1. Trace Encoder Interfaces

Above diagram 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 are replicated. The main **Trace Control Layer** controlling other (shared)

> components in the trace system is not replicated.

Chapter 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 TCODE fields, 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 or variable size. Several fields may be packet into single byte and single field may span across multiple bytes. Definitions of all fields can be found in Section 6.1 chapter.
Variable Field	Specifying that a field is variable-size (**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-size fields must end on a byte boundary. If necessary, they must zero-fill bit positions beyond the highest order bit of the variable data. Because variable-size 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 field.
Configurable Field	Configurable field (**Cfg** used as field size) means that existence and size of this field depends on some configuration setting. See chapter N-Trace Specific Trace Controls for more details.
[TODO]	Add more?

- 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 will be consistent with storage order, processing and text dump order.

Chapter 4. Ingress Port

N-Trace encoder is using the same ingress port as defined in [E-Trace Specification] (chapter `4 Instruction Trace Interface`).

As this version of N-Trace encoder specification does not define data trace, [E-Trace Specification] chapter `4.3 Data Trace Interface requirements` and chapter `4.4 Data Trace Interface` are not applicable.

E-Trace Specification describes when each 'itype' (instruction type) is generated, but the table below provides a detailed map of an instruction encodings into 'itype'.

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 Section 9.2		
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
c.j	No registers, only offset	15 = Other inferable jump
cm.jalt	Defined by Zcmt extension	9 = Inferable call

Instruction Retired	Condition/Notes	itype Value
cm.popret*	Defined by zcmp extension	13 = Return
Values of itype (3-bit) without Section 9.2		
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 defined by jump types in ISA manual.
- **itype** with codes 8..15 are only necessary when Section 9.2 is implemented.
- Tail calls (defined as allowed **itype** values 10 and 11) in [E-Trace Specification]) cannot be distinguished from normal jumps and as such are impossible to be generated by a hart.

Table 4. Handling of different itype values

#	itype	Encoder Action	RAS Action
0	None below	Only update I-CNT field.	-
1	Exception	Update I-CNT field. Emit Indirect Branch message with B-TYPE=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=1. **IMPORTANT:** An address emitted is known at the next ingress port cycle.	-
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: Update I-CNT field. Add 0 as LSB to HIST field. If overflown emit ResourceFull with RCODE=0 or 2	-

#	itype	Encoder Action	RAS Action
5	Taken branch	For BTM mode: Update I-CNT field. Generate DirectBranch message. For HTM mode: Update I-CNT field. Add 1 as LSB to HIST field. If overflown emit ResourceFull with RCODE=0 or 2	-
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 NOTE above this table)	-
11	Inferrable tail-call	**NOT POSSIBLE** (see NOTE above this table)	-
12	Co-routine swap	Same as for **itype=13** below.	Pop,Push
13	Return	If Pop return same address as current PC, then same as for **itype=0** above. Otherwise 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 message is updating I-CNT it may overflow. In such a case emit ResourceFull with RCODE=1 field.

N-Trace encoder does 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 5. 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 slices composed of **MDO** and **MSEO**.

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 slice (**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)

- N-Trace specification defines 6-bit **MDO** and 2-bit **MSEO** so each slice fits 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.

- 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 size field).

5.1. MSEO Sequences

The first slice of a message sends the LSBs of the message and is indicated by **MSEO=00**.

A variable-length field in a message always ends on a slice boundary (zero extended as needed) and the last slice of a variable field is indicated by **MSEO=01**. Initial slices of longer variable-length fields are sent using **MSEO=00**.

The last slice of a message is indicated by **MSEO=11**. It also implies an end of the last field of message.

Value of **MSEO=10** is reserved for future extensions.

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 (defined in this N-Trace specification) is a subset of general (single and dual) MSEO protocol definition.

5.2. Unified N-Trace Message Structure

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

- Very first field is ALWAYS fixed size **TCODE** (Transport Code) which defines meaning and format of subsequent fields.
- In case of simultaneous tracing from more than one hart, second field is ALWAYS fixed size **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.
- Very last field is (optional) variable size **TSTAMP** (Timestamp) field.
  - It may be possible to generate and analyze timestamps in a unified (simpler) way.

5.3. 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 size) field. As B-TYPE is a fixed size 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 size I-CNT field. Here three MSB bits are all 0-s to assure that the variable size field uses all 6 MDO bits.
0x1D	000111	01	UADDR[6] = 0x7	This is a single byte variable size 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).

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 Nexus 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).
trTeSrcID	12	Value of SRC field emitted by this trace encoder.
trTeInstEnRepeatedHistory	1	If this bit is set to 1 some sequences of branches may be detected and more compressed trace will be generated. See Section 9.3 chapter for details.
trTeInstEnSequentialJump	1	If set to 1 encoder may detect indirect flow changes (JAR/JALR) following instructions which set a register to a statically known value. See Section 9.1 chapter for details.
trTeInstEnImplicitReturn	1	If set to 1 some returns from a function may not be reported as indirect flow changes but treated as implicit jumps. See Section 9.2 chapter for details.
trTeInstEnCountOptimize	1	When set, instruction count will be reset more often and smaller values will be sent and handled by hardware. See Section 8.3 chapter for details.

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

Nexus Trace Modes

Nexus standard defined two main modes of

- BTM (Branch Trace Messaging) - every taken branch is generating at least two byte message, but repeated branches may be counted and reported as count.
- HTM (Branch History Messaging) - every branch (taken or not-taken) adds a 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.

Chapter 6. Nexus Messages (Details)

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

6.1. 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.

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	Opt			PROCESS **[Var]**			
DirectBranch	3	Opt				Yes		
IndirectBranch	4	Opt		Yes		Yes	U-ADDR	
Error	8	Opt			ETYPE **[4]** + PAD **[Cfg]**			
ProgTraceSync	9	Opt	Yes			Yes	F-ADDR	
DirectBranchSync	11	Opt	Yes			Yes	F-ADDR	
IndirectBranchSync	12	Opt	Yes	Yes		Yes	F-ADDR	
ResourceFull	27	Opt			RCODE **[4]** + RDATA **[Var]**			
IndirectBranchHist	28	Opt		Yes		Yes	U-ADDR	Yes
IndirectBranchHistSync	29	Opt	Yes	Yes		Yes	F-ADDR	Yes
RepeatBranch	30	Opt			B-CNT **[Var]**			
ProgTraceCorrelation	33	Opt			EVCODE **[4]** + CDF **[2]**	Yes		Opt

1. Size of fields: **[n]** means **n**-bit (fixed-size) field, **[Var]** means variable size, always present field, **[Cfg]** means size which depends on the encoder configuration option.
2. Any message may include optional TSTAMP **[Var]** field as the very last field of a message.
    - Field PAD **[Cfg]** provides (optional) 0-bit padding to assure that TSTAMP field is starting at byte-boundary (size is specified as **[Cfg]** as its size depends on the size of SRC field).

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

- Reference code is using plain C-style identifiers for messages and message fields (Nexus-style field name **B-TYPE** will be used as **BTYPE** in reference C code).

Here is part of this header showing how above messages are defined:

```
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(),
```

6.2. 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, max	Desciption	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.

Name	Bits, max	Desciption	Values/Notes
SRC	**Cfg**, max=12	Source of Message Transmission	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 should be the same size across all trace encoders (associated).
SYNC	4	Reason for Synchronization	Fields values 0: Standard: External Trace Trigger 1: Standard: Exit from Reset 2: Standard: Periodic Synchronization 3: Standard: Exit from Debug Mode 4: Reserved 5: Standard: Trace Enable (first SYNC after gap or Error message) 6: Standard: Trace Event (watchpoint with action=4) 7: Standard: Restart from FIFO overrun 8: Reserved 9: Standard: Exit from Power-down 10-13: Reserved 14-15: Reserved for vendor defined codes The SYNC field is always sent together with F-ADDR field.
B-TYPE	2	Branch Type	Reason for indirect flow changes 0: Standard: Indirect control flow change (jump, call or return). 1: Standard: Exception or interrupt 2-3: Reserved

Name	Bits, max	Desciption	Values/Notes
I-CNT	**Var**, max=22	Instruction Count	As RISC-V allows variable size instructions, this is a number of 16-bit half-instructions executed/retired since the I-CNT counter was transmitted or reset (see Section 8.3 chapter).
F-ADDR	**Var**, max=63	Full Target Address	Full PC address (LSB bit, which is always 0 for RISC-V is skipped). The F-ADDR field is always sent together with SYNC field.
U-ADDR	**Var**, max=63	Unique part of Target Address	Unique part of PC address (XOR with recent xADDR drop). The U-ADDR field is always sent together with B-TYPE field.
HIST	**Var**, max=32	Direct Branch History map	MSB = 1 is 'stop-bit', LSB denotes the last branch. See Section 8.2 for more details.
TSTAMP	**Var**, max=64	Timestamp (optional)	See Timestamp Details for more details.

Original Nexus specification does not define limits for variable size 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 Size

Field	Symbol	Bits	Description
SRC	NTRACE_MAX_SRC	12	Determined by size of Trace Control register field. Enough for 4095 (4K-1) trace sources.
I-CNT	NTRACE_MAX_ICNT	22	Usually a smaller value will be sufficient.
x-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 Resource Full messages.
TSTAMP	NTRACE_MAX_TSTAMP	64	It is certainly big enough. It corresponds to architecture defined timer and cycle count registers.

Chapter 7. Message Details

This chapter provides a detailed description of all N-Trace messages. Each message has its own table showing all fields in that message. Fields are ALWAYS listed in order from first to last (and LSB to MSB if placed in the same byte). Common fields are described in Section 6.2 chapter, but fields specific to particular message TCODE are explained here.

Overview of all fields in all messages is provided in the Section 6.1 table above.

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

- **n (1..6)** - This is **n**-bits wide, fixed size field.
- **Var** - This is a variable size 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).

7.1. Ownership Message

This message provides necessary context (privileged mode and OS-assigned Context ID) allowing the decoder to associate program flow with different parts of code which belong to different programs. It is reported in one of these three conditions:

- When an instruction which is changing privilege mode is executed.
- Immediately following any trace synchronization message (the one which includes the SYNC field).
- 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. Standard Transfer Code (TCODE) field.
Cfg	SRC	Standard Message Source (SRC) field.
Var	PROCESS	This is variable size field, which encodes **V** and **PRV** privilege mode bits as well as **scontext/hcontext** values. Details are provided below.
Var,Cfg	TSTAMP	Standard Timestamp (TSTAMP) field.

Explanations and Notes

Field PROCESS is encoded as 4 sub-fields (FORMAT, PRV, V, CONTEXT):

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

Reason	FORMAT:2	PRV:2	V:1	Context:var
V or PRV change	00	Yes	Yes	—

Reason	FORMAT:2	PRV:2	V:1	Context:var
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=00
S-mode:     V=0, PRV=01
M-mode:     V=0, PRV=11
VU-mode:    V=1, PRV=00
VS-mode:    V=1, PRV=01
```

All unused encodings are reserved.

Examples:

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

7.2. DirectBranch Message

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

Table 13. Direct Branch Message Fields

Bits	Name	Description
6	TCODE	Value=3. Standard Transfer Code (TCODE) field.
Cfg	SRC	Standard Message Source (SRC) field.
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 direct taken branch instruction. Next PC is determined by taking [+-]offset (from the opcode of that branch instruction) and adding it to an address of branch instruction.

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

7.3. 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. Standard Transfer Code (TCODE) field.
Cfg	SRC	Standard Message Source (SRC) field.
2	BTYPE	Standard Instruction Count (BTYPE) 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 Section 8.1 chapter for more details.

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

7.4. Error Message

Table 15. Error Message Fields

Bits	Name	Description
6	TCODE	Value=8. Standard Transfer Code (TCODE) field.
Cfg	SRC	Standard Message Source (SRC) field.
4	ETYPE	Error type. Subset of standard Nexus encoding: 0: Queue Overrun caused messages (one or more) to be lost. 1..7: Reserved. 0x8..0xF: Reserved for Vendor Defined Error(s).

Bits	Name	Description
2,Cfg	PAD	Pad the ETYPE field with 0-s to end of byte, so TSTAMP field always begins on byte boundary. When the SRC field is not present PAD is a 2-bit field. Otherwise size is determined by (configurable) size of SRC field (and can be 0 if SRC is 2-bit field for example). This is the only place where padding like this is needed.
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)

7.5. ProgTraceSync Message

Table 16. Program Trace Synchronization Message Fields

Bits	Name	Description
6	TCODE	Value=9. 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.

7.6. DirectBranchSync Message

Table 17. Direct Branch with Sync Message Fields

Bits	Name	Description
6	TCODE	Value=11(0xC). 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).

7.7. 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.
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.

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

7.8. 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.
4	RCODE	Standard Resource Code field (defines a meaning of RDATA fields). 0: Standard: HIST field has overflowed and is reported in the RDATA[0] field. 1: Standard: I-CNT counter 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. 8-0xF: Standard: Reserved for vendor specific encodings.
Var	RDATA [0]	Standard: For RCODE=0 this is HIST field (with MSB=1 being stop-bit). For RCODE=1, this is the I-CNT field. **Extension:** For RCODE=2 this is HIST field (with MSB=1 being stop-bit). For RCODE=1, this is the I-CNT field.
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

- Reported I-CNT value (with RCODE=1) may be a bit larger than NTRACE_MAX_ICNT (it is because ingress port may provide several instructions retired in the same cycle). It means the encoder should have an I-CNT counter to be one bit bigger than NTRACE_MAX_ICNT and when the MSB bit is set, a message with RCODE=1 should be generated.
- Not repeated HIST field overflow (RCODE=0) 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).
- When both I-CNT and HIST are overflowing at the same time, the encoder may send HIST overflow and I-CNT overflow in any order and the decoder must handle this correctly.
- More details are provided in the [HIST Pattern Detection] chapter.

7.9. IndirectBranchHist Message

Table 20. Indirect Branch History Message Fields

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

Bits	Name	Description
Cfg	SRC	Standard Message Source (SRC) field.
2	B-TYPE	Standard Instruction Count (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 Section 8.2 and Section 8.3 chapters for clarifications.

Next PC (after indirect 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 Section 8.1 chapter for more details.

7.10. 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.
2	B-TYPE	Standard Instruction Count (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 Section 8.2 and Section 8.3 chapters for clarifications.

Next PC (after indirect jump or exception/interrupt handler) is provided as an F-ADDR field. See Section 8.1 chapter for more details.

7.11. 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	BCNT	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 BCNT times.

7.12. ProgTraceCorrelation Message

This message is emitted when trace is disabled.

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: Entry into debug mode 1: Entry into low-power mode 4: Program trace disabled
2	CDF	Define number of CDATA fields following it, 0: Only I-CNT field follows 1: HIST field follows (for HTM trace)
Var	I-CNT	Standard Instruction Count (ICNT) field.
Var,Opt	HIST	Standard Branch History (HIST) field.
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 the trace actually stopped.

Chapter 8. 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.

8.1. Address Compression

Address transmissions are fully compliant with the Nexus specification.

- 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

8.2. HIST Field Generation

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

Examples:

```
Binary: 101,    hex: 0x5  (two branches, first not taken, second taken)
Binary: 1111,   hex: 0xF  (three branches, all three taken)
Binary: 10000,  hex: 0x10 (four branches, all four not taken)
Binary: 1,      hex: 0x1  (no branches at all)
```

The HIST field is reset (to 1, which is just a stop-bit with no branches) each time it is transmitted (it includes every Sync message).

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) branch.

8.2.1. 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.

If this is happening, a ResourceFull with the HIST field 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 Section 9.3 chapter for more details.

8.3. 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 handle 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.

- If I-CNT is reported in a message it MUST be reset to 0.
  - I-CNT may be additionally reset after each conditional branch (in HTM mode) but it must be directly enabled (see below for more details).

8.3.1. I-CNT Handling in BTM mode

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

```
0x100:  ADD ...         ; Plain linear 16-bit instruction
0x102:  B... 0x200      ; Conditional branch (32-bit instruction)
0x106:  ADD ...         ; Plain linear 32-bit instruction
0x10A:  B... 0x300      ; Conditional branch (32-bit instruction)
0x10E:  ADD ...         ; Plain linear 16-bit instruction
0x110:  ADD ...         ; Plain linear 32-bit instruction
0x114   ...
```

Let's assume we start a trace from address 0x100 (ProgramTraceSync with I-CNT=0 and F-ADDR encoding address = 0x100 should be generated) and let's assume that we collect a trace for this program (in BTM mode) 3 times.

- First time a branch at address 0x102 is taken.
  - A Direct Branch 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 branch at the end of this block was taken. Decoder will know PC=0x200 from an opcode of the branch at an address 0x102.
- Second time a branch at address 0x102 is not taken and a branch at address 0x10A is taken.
  - A Direct Branch 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 branch at the end of this block was taken. Decoder will know PC=0x300 from an opcode of the branch at an address 0x10A.
- Third time both branches are not taken.
  - In this case we will see I-CNT > 7. It means that none of the branches were taken and the decoder should continue analysis of code from an address 0x10E.

Decoder must look at each instruction in code block to know it's size. It cannot calculate <current PC+I-CNT*2> as it is UNKNOWN what is the size of the last instruction being retired - it may be (compressed) 16-bit or 32-bit (not-compressed) branch.

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) BEFORE an instruction at address 0x102. In this case an encoder

should generate an **IndirectBranch** or **IndirectBranchSync** message with I-CNT=1, BTYPE=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 happened BEFORE an instruction at address 0x10A).
- I-CNT=0 is also possible. It should be generated when interrupt was pending before we started the core (and trace) and instruction at address 0x100 was not executed/retired. Another reason for ICNT=0 may be a case, where instruction at address 0x100 will generate page fault (prefetch abort) or is illegal.
- I-CNT=4 (and I-CNT=6) 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 SYNC message, but it is not mandatory for all decoders to do so.

8.3.2. I-CNT Handling in HTM mode

When the encoder is operating in HTM mode, these branches (from code piece above ...) by itself will NOT generate any trace packets, but each of them will add a bit to the HIST field. But still I-CNT should be incremented at every retired instruction.

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

- I-CNT >= 3, HIST=0b1_1... (MSB=1 is stop bit, bit pattern '1...' means that first branch was taken). Encoder should continue from address 0x200 (as first branch encountered was taken).
- I-CNT >= 7, HIST=0b1_01... (MSB=1 is stop bit, bit pattern '01...' means that first branch was not taken and second branch was taken). Encoder should continue from address 0x300 (as the second branch encountered was taken).
- I-CNT >= 7, HIST-0b1_00... (MSB=1 is stop bit, bit pattern '00...' means that two branches were not taken). Encoder should continue from address 0x10E.

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

8.3.3. Additional I-CNT resets

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) branch instructions.

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.

Number of instructions (value of I-CNT) only matters after the last branch (or before reaching the very first branch). If we reset I-CNT every time HIST bit (taken or not-taken is added), then reported I-CNT counters will be smaller. It is especially important when an [Inferable Return Optimization] is enabled as in such a case a lot of instructions may be encoded in a single message. Sending big I-CNT values would not provide any new information.

8.3.4. I-CNT Field Overflows

When I-CNT overflows, the Resource Full message with RCODE=1 should be generated.

Timestamp Details

If timestamp recording is enabled, Sync messages all 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).

The TSTAMP field is a variable size 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 Sync 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 XORs it with the previous decompressed timestamp to obtain the full timestamp value associated with this message. Example:

The following rules must be observed:

- If timestamps are enabled, ALL Sync 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-Sync messages are not reporting timestamp
- 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.

8.4. Alternative Messages

Nexus is permitting some messages to be replaced by other (equivalent or super-set) messages. Table below clarifies what N-Trace is allowing. This can be useful for smaller implementations (as less message types may be generated) but will not complicate the decoder.

Table 25. Alternative Messages

Original Message	Alternative Message	Explanation
ProgTraceSync (in BTM mode)	DirectBranchSync	It has identical fields.
ProgTraceSync (in HTM mode)	BranchHistorySync with HIST=1	It adds a HIST field.
[TODO]	[TODO]	There is more options

Chapter 9. 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 Table 7 describes all control bits to enable these optimizations.

9.1. Sequential Jump Optimization

This optimization must be enabled by [trTeInstEnSequentialJump] control bit.

By default, the target of an indirect 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 jumps with sequentially inferable targets and provide this information separately to the encoder. The final decision as to whether to treat the jump as inferable or not must be made by the encoder. Both the constant load and the jump must be traced in order for the decoder to be able to infer the 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 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 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.

9.2. Implicit Return Optimization

This optimization must be enabled by [trTeInstEnImplicitReturn] control bit.

Although a function return is usually an indirect 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] It would be wise if the decoder would be aware which mode is implemented by an encoder.

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).

9.3. Repeated History Optimization

This optimization must be enabled by [trTeInstEnRepeatedHistory] control bit.

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

Loops in a program (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 branches and trace of 1M of history bits (a lot of trace).

Nexus standard defines Repeat Branch message. This message will provide a single BCNT (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 branches in HTM mode an extra encoding for RCODE in Resource Full message is added.

- This feature must be specifically enabled by setting the

trTeInstEnBrachPrediction control bit. See N-Trace Specific Trace Controls chapter for details.

- 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) is the same.

Tracing of such simple, long loops would benefit from generating special messages/fields which provide counters of taken/non-taken 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 branches taken/not-taken and instead generate many HIST fields, generate a message consisting of a pattern and repeat count.

Let's assume that we have a loop, which generates a long sequence of repeated taken/non-taken branches. Trace may generate Resource Full messages with the following HIST records:

```
Msg#1:
    TCODE=27 (ResourceFull)
    RCODE=0 (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=0 (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=0 (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.

It is also 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 10. Rules of Generating Messages

Main Rules

1. Plain linear instructions and direct, PC relative jumps generate no trace.
    - These are called inferable instructions, where the next PC can be known from looking at binary code.
2. Only branches (conditional), 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 (FADDR 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…) jump ⇒ PC is jump destination (known from PC and opcode as all jumps are PC relative).
    - Not taken 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 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) 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 Section 9.2 chapter.
2. As RISC-V architecture is only supporting PC relative jumps/calls, indirect 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 branch.
    - Number of instructions between two branches does not matter.

10.1. 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 11. N-Trace Decoding Guidelines

To decode 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 ELF file of a program being executed, but it can be also 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 jumps and branches offset encoded in opcode.

At beginning of 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 buffer.

11.1. 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 current PC through inferable jumps (all types) and branches (each branch will 'consume' single bit from **HIST** field).
  - At the end (after the LSB bit from **HIST** is processed), the PC will be after the last branch (either taken or not taken).
- Handle I-CNT field (if available and not 0x0)
  - Analyze code from current PC through inferable jumps (all types) - each encountered branch must be treated as not-taken
  - It will reach either non-inferable jump or some other 'event' (like exception, interrupt, trace off, trigger etc.)
- At the last step apply F-ADDR or U-ADDR field value (if available). This will be the next PC where analysis of the next trace message should start.

- Phrase `inferable jumps (all types)` include indirect jumps, which are inferable.
- Some messages may encode ICNT 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.

11.2. Decoding trace from multiple harts

Decoder for specific a hart should only look for messages with SRC for that particular hart.

11.3. 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.

11.4. 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 12. Additional Material

12.1. Trace Bandwidth Considerations

- SRC field (if enabled) may change otherwise optimal layout of fields in messages.

12.2. Validation Considerations

- Resource Full message with ICNT 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 ICNT.

12.3. 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 26. 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.

Enhancement	Conformance	Notes
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.