Requirements List

of

Efficient Trace for RISC-V,

Version 2.0.3*

by

Mehmet Öner

Version 1.0

*`Efficient Trace for RISC-V, Version 2.0.3', Contributors Gajinder Panesar <gajinder.panesar@gmail.com>, Iain Robertson <iain.robertson@siemens.com>, RISC-V International, April 2024.
https://github.com/riscv-non-isa/riscv-trace-spec

About

The Document

In systems engineering approach, before doing anything with the design of the system under consideration, *requirements analysis* must be completed as one of the first tasks (if not the very first). Beginning with an itemized, atomic, classified and well defined list of requirements is essential. Because, following activities at various stages of development (like design coverage analysis, testing, verification, validation …) depend on the requirement specifications stated at the beginning.

RISC-V International provides the ISA (Instruction Set Architecture) and non-ISA requirement specifications for the RISC-V architecture (https://riscv.org/technical/specifications/). These documents in general are good written technical plain text documents. However, they lack some aspects of good requirement specification practices:

- Requirements are in free text form and not itemized: Itemized list of requirements enables requirement coverage in design, test, verification and validation phases.
- Text includes comments and information statements along with requirements: Statements must be clearly labeled and categorized.
- Some statements include more than one specifications: Each specification need to be isolated.
- Some specifications such as instruction definitions are distributed throughout the text: The distributed content need to be put together to have a complete the specification.

The aim of this document is providing an edited list of requirements for `Efficient Trace for RISC-V, Version 2.0.3', Contributors Gajinder Panesar <gajinder.panesar@gmail.com>, Iain Robertson <iain.robertson@siemens.com>, RISC-V International, April 2024. https://github.com/riscv-non-isa/riscv-trace-spec. This specification is licensed under the Creative Commons Attribution 4.0 International License (CC-BY 4.0). https://creativecommons.org/licenses/by/4.0/

It is licensed under the Creative Commons Attribution 4.0 International License (CC-BY 4.0). https://creativecommons.org/licenses/by/4.0/

In particular:

- Statements were itemized and given an ID number.
- Each itemized statement was referenced to the original document to provide traceability
- Itemized statements were categorized
- Complex statements were broken into simpler atomic requirement statements when needed.
- Distributed requirement information was put together to form complete specifications.

Special attention was given to preserve original statements, even when dividing complex statements into simpler atomic statements. But occasionally, some statements were re-written as to form a formal requirement statement.

The Editor (or the systems engineer)

After 34 years of my career, I retired from my regular job in 2023. Now, I do part time consulting services to interested parties, while I do work on projects that interest me more than a regular work.

In my career I dealt with very diverse fields of engineering: Academics, C/C++ desktop programming, embedded systems, analog circuit design, DSP algorithms, VLSI/FPGA design, underwater acoustics are to name few.

I've always enjoyed designing controllers and processors with generic HDL for VLSI or FPGA. So, as my personal project to work on, I decided to design RISC-V cores with different capabilities.

Having some defense sector background, I find systems engineering approach very useful. After reading RISC-V specification documents, I decided to take the initiative and edit the documents into customer requirements list format which is a tough, tedious and time consuming work.

In case someone else could find these documents useful, I share them on github:
https://github.com/vizionerco/RISC-V

Best regards,

Mehmet Öner, Ph.D.

www.linkedin.com/in/mehmet-oner-00453733/

Table of Contents

Definitions

Table 1: Requirement Types

TYPE	NAME	EXPLANATION
H	Heading	Headings in the original document. Headings are included to provide context to the subsequent requirements
I	Information	These are statements that explain some aspects of the subject, but actually do not specify any requirement. For these types, the statement(s) in the text is given as is.
C	Comment	These statements are original comment statements in the RISC_V documentation which are explained as: “Commentary on our design decisions is formatted as in this paragraph. This non-normative text can be skipped if the reader is only interested in the specification itself.” For these types, the statement(s) in the text is given as is.
R	Requirement	These are statements that specify a specific need to be fulfilled. For these types, the statement(s) in the text is given as is where possible. In some cases, information is collected from different tables and figures to form a complete specification. In some cases, context is added in parenthesis to make the requirement self explanatory. In some cases, the statements are broken into single statements requirement statements.
O	Optional Requirement	These are tatements that specify a property that is not mandatory to implement. However if it is chosen to fulfill, it should obey this requirement. Inclusion of the text is the same as R/Requirement type.
T	Tentative Requirement	These are requirements but are not frozen yet by the RISC-V committee. Inclusion of the text is the same as R/Requirement type.

Table 2: Abbreviations & Definitions

SHORT	MEANING
ATB	Arm trace bus
branch	an instruction which conditionally changes the execution flow
CSR	control/status register
decoder	a piece of software that takes the trace packets emitted by the encoder and reconstructs the execution flow of the code executed by the RISC-V hart
delta	a change in the program counter that is other than the difference between two instructions placed consecutively in memory
discontinuity	another name for ’delta’ (see above)
ELF	executable and linkable format
encoder	a piece of hardware that takes in instruction execution information from a RISC-V hart and transforms it into trace packets
EPC	exception program counter
exception	an unusual condition occurring at run time associated with an instruction in a RISC-V hart
hart	a RISC-V hardware thread
interrupt	an external asynchronous event that may cause a RISC-V hart to experience an unexpected transfer of control

SHORT	MEANING
ISA	instruction set architecture
jump	an instruction which unconditionally changes the execution flow
direct jump	an instruction which unconditionally changes the execution flow by changing the PC by a constant value
indirect jump	an instruction which unconditionally changes the execution flow by changing the PC to a computed value
inferable jump	a jump where the target address is supplied via a constant embedded within the jump opcode
uninferable jump	a jump which is not inferable (see above)
LSB	least significant bit
MSB	most significant bit
packet	the atomic unit of encoded trace information emitted by the encoder
PC	program counter
program counter	a register containing the address of the instruction being executed
retire	the final stage of executing an instruction, when the machine state is updated (sometimes referred to as ’commit’ or ’graduate’)
trap	the transfer of control to a trap handler caused by either an exception or an interrupt
updiscon	contraction of ’uninferable PC discontinuity’

CHAPTER 1 Introduction

ID	REFERENCE	TYPE	DEFINITION
RVET.1.1	1.0 (p.5)	H	Introduction
RVET.1.2	1.0 (p.5)	I	In complex systems understanding program behavior is not easy. Unsurprisingly in such systems, software sometimes does not behave as expected. This may be due to a number of factors, for example, interactions with other cores, software, peripherals, realtime events, poor implementations or some combination of all of the above.
RVET.1.3	1.0 (p.5)	I	It is not always possible to use a debugger to observe behavior of a running system as this is intrusive. Providing visibility of program execution is important. This needs to be done without swamping the system with vast amounts of data.
RVET.1.4	1.0 (p.5)	I	One method of achieving this is via a Processor Branch Trace.
RVET.1.5	1.0 (p.5)	I	This works by tracking execution from a known start address and sending messages about the address deltas taken by the program.
RVET.1.6	1.0 (p.5)	I	These deltas are typically introduced by jump, call, return and branch type instructions, although interrupts and exceptions are also types of deltas.
RVET.1.7	1.0 (p.5)	R	Conceptually, the system has one or more of the following fundamental components: • A core with an instruction trace interface that outputs all relevant information to allow the successful creation of a processor branch trace and more. This is a high bandwidth interface: in most implementations, it will supply a large amount of data (instruction address, instruction type, context information, …) for each core execution clock cycle; • A hardware encoder that connects to this instruction trace interface and compresses the information into lower bandwidth trace packets; • A transmission channel to transmit or a memory to store these trace packets; • A decoder, usually software on an external PC, that takes in the trace packets and, with knowledge of the program binary that's running on the originating hart, reconstructs the program flow. This decoding step can be done off-line or in real-time while the hart is executing.
RVET.1.8	1.0 (p.5)	I	In RISC-V, all instructions are executed unconditionally or at least their execution can be determined based on the program binary.
RVET.1.9	1.0 (p.5)	R	The instructions between the deltas can all be assumed to be executed sequentially. Because of this, there is no need to report sequential instructions in the trace, only whether the branches were taken or not and the address of taken indirect branches or jumps.
RVET.1.10	1.0 (p.5)	R	If the program counter is changed by an amount that cannot be determined from the execution binary, the trace decoder needs to be given the destination address (i.e. the address of the next valid instruction). Examples of this are indirect branches or jumps, where the next instruction address is determined by the contents of a register rather than a constant embedded in the program binary.
RVET.1.11	1.0 (p.5)	I	Interrupts generally occur asynchronously to the program's execution rather than intentionally as a result of a specific instruction or event.
RVET.1.12	1.0 (p.5)	I	Exceptions can be thought of in the same way, even though they can be typically linked back to a specific instruction address.

ID	REFERENCE	TYPE	DEFINITION
RVET.1.13	1.0 (p.5)	R	The decoder generally does not know where an interrupt occurs in the instruction sequence, so the trace encoder must report the address where normal program flow ceased, as well as give an indication of the asynchronous destination which may be as simple as reporting the exception type. When an interrupt or exception occurs, or the processor is halted, the final instruction retired beforehand must be included in the trace.
RVET.1.14	1.0 (p.5)	I	This document serves to specify the ingress port (the signals between the RISC-V core and the encoder), compressed branch trace algorithm and the packet format used to encapsulate the compressed branch trace information.
RVET.1.15	1.1 (p.6)	H	Terminology
RVET.1.16	1.1 (p.6)	I	The following terms have a specific meaning in this specification. (see Table 2: Abbreviations & Definitions)
RVET.1.17	1.12 (p.7)	H	Nomenclature
RVET.1.18	1.12 (p.7)	I	In the following sections items in **bold** are signals or fields within a packet.
RVET.1.19	1.12 (p.7)	I	Items in ***bold italics*** are mnemonics for instructions or CSRs defined in the RISC-V ISA
RVET.1.20	1.12 (p.7)	I	Items in *italics* with names ending *'_p'* refer to parameters either built into the hardware or configurable hardware values.

CHAPTER 2 Encoder Control

ID	REFERENCE	TYPE	DEFINITION
RVET.2.1	2.0 (p.9)	H	Encoder Control
RVET.2.2	2.0 (p.9)	I	The fields required to control a Trace Encoder are defined in the RISC-V Trace Control Interface Specification, which is intended to apply to any and all RISC-V trace encoders, regardless of encoding protocol.
RVET.2.3	2.0 (p.9)	I	This chapter details which of those fields apply to E-Trace.
RVET.2.4	2.0 (p.9)	I	To avoid replication, descriptions are not provided here; additional E-Trace specific context or clarification is provided only where required.
RVET.2.5	2.0 (p.9)	I	How fields are organized and accessed (e.g packet based or memory mapped) is outside the scope of this document.
RVET.2.6	2.0 (p.9)	I	If a memory mapped approach is adopted, this register map from the RISC-V Trace Control Interface Specification should be used.
RVET.2.7	2.0 (p.9)	C	Note: Upto and including the E-Trace v2.0.0 specification, which predated the creation of the RISC-V Trace Control Interface Specification, the full field definitions were included in this chapter. For versions later than this, the field definitions have simply moved from this specification to the RISC-V Trace Control Interface Specification, without any change to their meaning. However, in order to create a more widely applicable protocol agnostic specification it has been necessary to change the field names in the process.
RVET.2.8	2.0 (p.9)	I	The applicability of fields for E-trace is categorized as follows: • N: Not applicable • M: Mandatory • O: Optional • MD: Mandatory if data trace is supported • OD: Optional for data trace
RVET.2.9	2.1 (p.9)	H	Basic Control

ID	REFERENCE	TYPE	DEFINITION
RVET.2.10	2.1 (p.9) Table 1	R	The following fields control basic encoding behavior. Basic Control Field Applicability E-Trace Specific Details

Field	Applicability	E-Trace Specific Details
trTeActive	M	
trTeEnable	M	
trTeInstTracing	M	
trTeDataTracing	MD	
trTeInstTrigEnable	O	
trTeDataTrigEnable	OD	
trTeInstStallOrOverflow	O	
trTeDataStallOrOverflow	OD	
trTeInstStallEn	O	
trTeDataStallEn	OD	
trTeEmpty	O	Recommended if the trace datapath requires manual flushing when trace is disabled.
trTeDataDrop	OD	
trTeDataDropEn	OD	
trTeInhibitSrc	O	
trTeInstSyncMode	M	If hardcoded, must be to a non-zero value.
trTeInstSyncMax	M	May be hardcoded.
trTeFormat	M	Must be set to 0 (denoting E-Trace format).
trTeVerMajor	M	
trTeVerMinor	M	
trTeCompType	M	
trTeProtocolMajor	M	Must be 0 to indicate this version (2.0.x) of the E-Trace protocol.
trTeProtocolMinor	M	Must be 0.
trTeSrcID	O	
trTeSrcBits	O	

RVET.2.11	2.2 (p.10)	H	Optional Modes

ID	REFERENCE	TYPE	DEFINITION
RVET.2.12	2.2 (p.10) Table 2	R	See Section 3.2 for details of the modes covered in this section. Optional and run-time configurable modes

Field	Applicability	E-Trace Specific Details
trTeInstNoAddrDiff	O	
trTeInstNoTrapAddr	O	
trTeInstEnSequentialJump	O	
trTeInstEnImplicitReturn	O	
trTeInstEnBranchPrediction	O	
trTeInstJumpTargetCache	O	
trTeDataNoValue	OD	
trTeDataNoAddr	OD	
trTeDataAddrCompress	OD	
trTeContext	N	Hardcode to 0.
trTeInstMode	N	Hardcode to 7.
trTeInstImplicitReturnMode	N	Hardcode to 0.
trTeInstEnRepeatedHistory	N	Hardcode to 0.
trTeInstEnAllJumps	N	Hardcode to 0.
trTeInstExtendAddrMSB	N	Hardcode to 0.

RVET.2.13	2.3 (p.10)	H	Filtering
RVET.2.14	2.3 (p.10) Table 3	R	See Chapter 5 for details of the filtering capabilities covered in this section. Trace filtering selection

Field	Applicability	E-Trace Specific Details
trTeInstFilters	O	
trTeDataFilters	OD	
trTeFilter…	O	
trTeComp…	O	
trTeTrig…	N	Hardcode to 0.

CHAPTER 3 Branch Trace

ID	REFERENCE	TYPE	DEFINITION
RVET.3.1	3.0 (p.11)	H	Branch Trace
RVET.3.2	3.0 (p.11)	I	Instruction delta tracing, also known as branch tracing, works by tracking execution from a known start address by sending information about the deltas taken by the program.
RVET.3.3	3.0 (p.11)	I	Deltas are typically introduced by jump, call, return and branch type instructions, although interrupts and exceptions are also types of deltas.
RVET.3.4	3.0 (p.11)	I	Instruction delta tracing provides an efficient encoding of an instruction sequence by exploiting the deterministic way the processor behaves based on the program it is executing.
RVET.3.5	3.0 (p.11)	I	The approach relies on an offline copy of the program binary being available to the decoder, so it is generally unsuitable for either dynamic (self-modifying) programs or those where access to the program binary is prohibited.
RVET.3.6	3.0 (p.11)	I	While the program binary is sufficient, access to the assembly or higher-level source code will improve the ability of the decoder to present the decoded trace in the debugger by annotating the traced instructions with source code line numbers and labels, variable names etc.
RVET.3.7	3.0 (p.11)	I	This approach can be extended to cope with small sections of deterministically dynamic code by arranging for the decoder to request instruction memory from the target.
RVET.3.8	3.0 (p.11)	I	Memory lookups generally lead to a prohibitive reduction in performance, although they are suitable for examining modest jump tables, such as the exception/interrupt vector pointers of an operating system which may be adjusted at boot up and when services are registered.
RVET.3.9	3.0 (p.11)	I	Both static and dynamically linked programs can be traced using this approach.
RVET.3.10	3.0 (p.11)	I	Statically linked programs are straightforward as they generally operate in a known address space, often mapping directly to physical memory.
RVET.3.11	3.0 (p.11)	I	Dynamically linked programs require the debugger to keep track of memory allocation operations using either trace or stop-mode debugging.
RVET.3.12	3.1 (p.11)	H	Instruction delta trace concepts
RVET.3.13	3.1.1 (p.11)	H	Sequential instructions
RVET.3.14	3.1.1 (p.11)	R	For instruction set architectures such as RISC-V where all instructions are executed unconditionally or at least their execution can be determined based on the program binary, the instructions between the deltas are assumed to be executed sequentially.
RVET.3.15	3.1.1 (p.11)	R	Consequently, there is no need to report them in the trace.
RVET.3.16	3.1.1 (p.11)	R	The trace only needs to contain whether branches were taken or not, the addresses of taken indirect jumps, or other program counter discontinuities.
RVET.3.17	3.1.2 (p.11)	H	Uninferable PC discontinuities

ID	REFERENCE	TYPE	DEFINITION
RVET.3.18	3.1.2 (p.11)	I	An uninferable program counter discontinuity is a program counter change that can not be inferred from the program binary alone.
RVET.3.19	3.1.2 (p.11)	R	For these cases, the instruction delta trace must include a destination address: the address of the next valid instruction.
RVET.3.20	3.1.2 (p.11)	I	Indirect jumps are an example of this, where the next instruction address is determined by the contents of a register rather than a constant embedded in the program binary.
RVET.3.21	3.1.2 (p.11)	R	In this case, the address of the instruction following the jump (also known as the jump target) must be traced.
RVET.3.22	3.1.2 (p.11)	I	Interrupts and exceptions are another form of uninferable PC discontinuity; these are discussed in detail below.
RVET.3.23	3.1.3 (p.12)	H	Branches
RVET.3.24	3.1.3 (p.12)	I	A branch is an instruction where a jump is conditional on the value of a register or a flag.
RVET.3.25	3.1.3 (p.12)	R	For a decoder to able to follow program flow, the trace must include whether a branch was taken or not.
RVET.3.26	3.1.3 (p.12)	R	For a direct branch, where the destination address is encoded in the program binary (either as a constant, or as a constant offset from the program counter), no further information is required.
RVET.3.27	3.1.3 (p.12)	I	Direct branches are the only type of branch that is supported by the RISC-V ISA.
RVET.3.28	3.1.4 (p.12)	H	Interrupts and exceptions
RVET.3.29	3.1.4 (p.12)	I	Interrupts are a different type of delta that generally occur asynchronously to the program's execution rather than intentionally as a result of a specific instruction or event.
RVET.3.30	3.1.4 (p.12)	I	Exceptions can be thought of in the same way, even though they can be typically linked back to a specific instruction address.
RVET.3.31	3.1.4 (p.12)	R	The decoder generally does not know where an interrupt occured in the instruction sequence, so the trace must report the address where normal program flow ceased, as well as give an indication of the asynchronous destination which may be as simple as reporting the exception type.
RVET.3.32	3.1.4 (p.12)	R	When an interrupt or exception occurs, the final instruction retired beforehand must be traced.
RVET.3.33	3.1.4 (p.12)	R	Following this the next valid instruction address (the first of the trap handler) must be traced.
RVET.3.34	3.1.4 (p.12)	C	Note: not all exceptions and interrupts cause traps (see Section 1.1 for definitions). Most notably, floating point exceptions and disabled interrupts do not trap. If an exception or interrupt doesn't trap, the program counter does not change. So, there is no need to trace all exceptions/interrupts, just traps.
RVET.3.35	3.1.4 (p.12)	R	In this document, interrupts and exceptions are only traced when they cause traps to be taken.
RVET.3.36	3.1.5 (p.12)	H	Synchronization
RVET.3.37	3.1.5 (p.12)	I	In order to make the trace robust there must be regular synchronization points within the trace.
RVET.3.38	3.1.5 (p.12)	R	Synchronization is accomplished by sending a full valued instruction address (and potentially a context identifier).

ID	REFERENCE	TYPE	DEFINITION
RVET.3.39	3.1.5 (p.12)	O	The decoder and debugger may also benefit from sending the reason for synchronizing.
RVET.3.40	3.1.5 (p.12)	I	The frequency of synchronization is a trade-off between robustness and trace bandwidth.
RVET.3.41	3.1.5 (p.12)	R	The instruction trace encoder needs to synchronise fully: • For the first instruction traced after reset or resume from halt; • Any time that an instruction is traced and the previous instruction was not traced; • If the instruction is the first of an interrupt service routine or exception handler; • After a prolonged period of time.
RVET.3.42	3.1.6 (p.12)	H	End of trace
RVET.3.43	3.1.6 (p.12)	R	If tracing stops for any reason, the address of the final traced instruction must be output.
RVET.3.44	3.1.6 (p.12, p.13)	I	Some examples of why tracing may stop are: • The hart may be halted (entered debug mode); • The hart may be reset; • Encoding may be stopped (for example via a Trace-off trigger - see Section 4.2.4); • The matching criteria for any filtering capabilities implemented by the encoder may no longer be met; • The encoder may be disabled.
RVET.3.45	3.2 (p.13)	H	Optional and run-time configurable modes
RVET.3.46	3.2 (p.13)	O	An instruction trace encoder may support multiple tracing modes.
RVET.3.47	3.2 (p.13)	R	To ensure that the decoder treats the incoming packets correctly, it needs to be informed of the current active configuration.
RVET.3.48	3.2 (p.13)	R	The configuration is reported by a packet that is issued by the encoder whenever the encoder configuration is changed.
RVET.3.49	3.2 (p.13)	R	Here are common examples of such modes: • delta address mode: program counter discontinuities are encoded as differences instead of absolute address values. • full address mode: program counter discontinuities are encoded as absolute address values. • implicit exception mode: the destination address of an exception (i.e. the address of the exception trap) is assumed to be known by the decoder, and thus not encoded in the trace. • Sequentially inferable jump mode: The target of an indirect jump can be inferred by considering the combined effect of two instructions. • implicit return mode: the destination address of function call returns is derived from a call stack, and thus not encoded in the trace. • branch prediction mode: branches that are predicted correctly by an encoder branch predictor (and an identical copy in the decoder) are not encoded as taken/non-taken, but as a more efficient branch count number. • Jump target cache mode: Rather than reporting the address of an uninferable jump target, efficiency can be improved by caching recent jump targets, and reporting the cache entry index instead.
RVET.3.50	3.2 (p.13)	I	Modes may have associated parameters; see Table 40 for further details
RVET.3.51	3.2 (p.13)	R	All modes are optional apart from delta address mode, which must be supported.

ID	REFERENCE	TYPE	DEFINITION
RVET.3.52	3.2.1 (p.13)	H	Delta address mode
RVET.3.53	3.2.1 (p.13)	R	Related parameters: None
RVET.3.54	3.2.1 (p.13)	R	In delta address mode, addresses are encoded as the difference between the actual address of the current instruction and the actual address of the instruction reported in the previous packet that contained an address.
RVET.3.55	3.2.1 (p.13)	I	This differential encoding requires fewer bits than the full address, and thus results in more efficient trace compression.
RVET.3.56	3.2.2 (p.13)	H	Full address mode
RVET.3.57	3.2.2 (p.13)	R	Related parameters: None
RVET.3.58	3.2.2 (p.13)	R	In full address mode, all addresses in the trace are encoded as absolute addresses instead of in differential form.
RVET.3.59	3.2.2 (p.13)	I	This kind of encoding is always less efficient, but it can be a useful debugging aid for software decoder developers.
RVET.3.60	3.2.3 (p.14)	H	Implicit exception mode
RVET.3.61	3.2.3 (p.14)	R	Related parameters: None
RVET.3.62	3.2.3 (p.14)	I	The RISC-V Privileged ISA specification stores exception handler base addresses in the ***utvec/stvec/vstvec/mtvec*** CSR registers.
RVET.3.63	3.2.3 (p.14)	I	In some RISC-V implementations, the lower address bits are stored in the ***ucause/scause/vscause/mcause*** CSR registers.
RVET.3.64	3.2.3 (p.14)	R	By default, both the ****tvec*** and ****cause*** values are reported when an exception or interrupt occurs.
RVET.3.65	3.2.3 (p.14)	R	The implicit exception mode omits ****tvec*** (the trap handler address), from the trace and thus improves efficiency.
RVET.3.66	3.2.3 (p.14)	I	This mode can only be used if the decoder can infer the address of the trap handler from just the exception cause.
RVET.3.67	3.2.4 (p.14)	H	Sequentially inferable jump mode
RVET.3.68	3.2.4 (p.14)	R	Related parameters: *sijump_p*.
RVET.3.69	3.2.4 (p.14)	I	By default, the target of an indirect jump is always considered an uninferable PC discontinuity.
RVET.3.70	3.2.4 (p.14)	I	However, if the register that specifies the jump target was loaded with a constant then it can be considered inferable under some circumstances.
RVET.3.71	3.2.4 (p.14)	R	The hart must identify jumps with sequentially inferable targets and provide this information separately to the encoder.
RVET.3.72	3.2.4 (p.14)	R	The final decision as to whether to treat the jump as inferable or not must be made by the encoder.
RVET.3.73	3.2.4 (p.14)	R	Both the constant load and the jump must be traced in order for the decoder to be able to infer the jump target.
RVET.3.74	3.2.4 (p.14)	I	See Section 4.1.1 for details of what constitutes a sequentially inferable jump.
RVET.3.75	3.2.5 (p.14)	H	Implicit return mode
RVET.3.76	3.2.5 (p.14)	R	Related parameters: *call_counter_size_p*, *return_stack_size_p*, *itype_width_p*

ID	REFERENCE	TYPE	DEFINITION
RVET.3.77	3.2.5 (p.14)	I	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.
RVET.3.78	3.2.5 (p.14)	I	For those programs, it is possible to determine the execution path without being explicitly notified of the destination address of the return.
RVET.3.79	3.2.5 (p.14)	I	The implicit return mode can result in very significant improvements in trace encoder efficiency.
RVET.3.80	3.2.5 (p.14)	R	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.
RVET.3.81	3.2.5 (p.14)	R	This can be accomplished by utilizing a counter to keep track of the number of nested calls being traced. The counter increments on calls (but not tail calls), and decrements on returns (see Section 4.1.1 for definitions). 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).
RVET.3.82	3.2.5 (p.14)	I	Such a scheme is low cost, and will work as long as programs are "well behaved".
RVET.3.83	3.2.5 (p.14)	R	The encoder does not check that the return address is actually that of the instruction following the associated call.
RVET.3.84	3.2.5 (p.14)	I	As such, any program that modifies return addresses cannot be traced using this mode with this minimal implementation.
RVET.3.85	3.2.5 (p.14)	O	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.
RVET.3.86	3.2.5 (p.14, p.15)	I	This is fully robust for all programs, but is more expensive to implement.
RVET.3.87	3.2.5 (p.15)	R	In this case, if a return address does not match the prediction, it must be reported explicitly via a packet, along with the number of return addresses currently on the stack.
RVET.3.88	3.2.5 (p.15)	I	This ensures that the decoder can determine which return is being reported.
RVET.3.89	3.2.6 (p.15)	H	Branch prediction mode
RVET.3.90	3.2.6 (p.15)	R	Related parameters: *bpred_size_p*.
RVET.3.91	3.2.6 (p.15)	R	Without branch prediction, the outcome of each executed branch is stored in a branch map: a bit vector in which the taken/non-taken status of each branch is stored in chronological order.
RVET.3.92	3.2.6 (p.15)	I	While this encoding is efficient, at 1 bit per branch, there are some cases where this can still result in a relatively large volume of trace packets. For example: • Executing tight loops of code containing no uninferable jumps. Each iteration of the loop will add a bit to the branch map; • Sitting in an idle loop waiting for an interrupt. This produces large amounts of trace when nothing of any interest is actually happening! • Breakpoints, which in some implementations also spin in an idle loop.
RVET.3.93	3.2.6 (p.15)	I	A significant coding efficiency can be obtained by the addition of a branch predictor in the encoder.

ID	REFERENCE	TYPE	DEFINITION
RVET.3.94	3.2.6 (p.15)	R	To keep the encoder and decoder synchronized, a predictor with identical behavior will need to be implemented in the decoder software.
RVET.3.95	3.2.6 (p.15)	R	The predictor shall comprise a lookup table of $2^{bpred\_size\_p}$ entries.
RVET.3.96	3.2.6 (p.15)	R	Each entry is indexed by bits *bpred_size_p*:1 of the instruction address (or *bpred_size_p*+1:2 if compressed instructions aren't supported), and each contains a 2-bit prediction state: • 00: predict not taken, transition to 01 if prediction fails; • 01: predict not taken, transition to 00 if prediction succeeds, else 11; • 11: predict taken, transition to 10 if prediction fails; • 10: predict taken, transition to 11 if prediction succeeds, else 00.
RVET.3.97	3.2.6 (p.15)	R	The MSB represents the predicted outcome, the LSB the most recent actual outcome.
RVET.3.98	3.2.6 (p.15)	R	The prediction must fail twice for the predicted value to change.
RVET.3.99	3.2.6 (p.15)	R	The lookup table entries are initialized to 01 when a synchronization packet is sent.
RVET.3.100	3.2.6 (p.15)	O	Other predictors, such as the gShare predictor (see Hennessy & Patterson), should be considered.
RVET.3.101	3.2.6 (p.15)	I	Some further experimentation is needed to determine the benefits of different lookup table sizes and predictor algorithms.
RVET.3.102	3.2.7 (p.15)	H	Jump target cache mode
RVET.3.103	3.2.7 (p.15)	R	Related parameters: *cache_size_p*.
RVET.3.104	3.2.7 (p.15)	R	By default, the target address of an uninferable jump is output in the trace, usually in differential form.
RVET.3.105	3.2.7 (p.15)	I	If the same function is called repeatedly, (for example, in a loop), the same address will be output repeatedly.
RVET.3.106	3.2.7 (p.15)	I	An efficiency gain can be obtained by the addition of a jump target cache to the encoder.
RVET.3.107	3.2.7 (p.15, p.16)	R	To keep the encoder and decoder synchronized, a cache with identical behavior will need to be implemented in the decoder software.
RVET.3.108	3.2.7 (p.16)	I	Even a small cache can provide significant improvement.
RVET.3.109	3.2.7 (p.16)	R	The cache shall comprise $2^{cache\_size\_p}$ entries, each of which can contain an instruction address.
RVET.3.110	3.2.7 (p.16)	R	It will be direct mapped, with each entry indexed by bits cache_size_p:1 of the instruction address (or *cache_size_p*+1:2 if compressed instructions aren't supported).
RVET.3.111	3.2.7 (p.16)	R	Each uninferable jump target is first compared with the entry at its index in the cache.
RVET.3.112	3.2.7 (p.16)	R	If it is found in the cache, the index number is traced rather than the target address.
RVET.3.113	3.2.7 (p.16)	R	If it is not found in the cache, the entry at that index is replaced with the current instruction address.
RVET.3.114	3.2.7 (p.16)	R	The cache entries are all invalidated when a synchronization packet is sent.

CHAPTER 4 Hart to encoder interface

ID	REFERENCE	TYPE	DEFINITION
RVET.4.1	4.0 (p.17)	H	Hart to encoder interface
RVET.4.2	4.1 (p.17)	H	Instruction Trace Interface requirements
RVET.4.3	4.1 (p.17)	I	This section describes in general terms the information which must be passed from the RISC-V hart to the trace encoder for the purposes of Instruction Trace, and distinguishes between what is mandatory, and what is optional.
RVET.4.4	4.1 (p.17)	R	The following information is mandatory: • The number of instructions that are being retired; • Whether there has been an exception or interrupt, and if so the cause (from the ***ucause***/***scause***/***vscause***/***mcause*** CSR) and trap value (from the ***utval***/***stval***/***vstval***/***mtval*** CSR). The register set to output should be the set that is updated as a result of the exception (i.e. the set associated with the privilege level immediately following the exception); • The current privilege level of the RISC-V hart; • The *instruction_type* of retired instructions for: ◦ Jumps with a target that cannot be inferred from the source code; ◦ Taken and nontaken branches; ◦ Return from exception or interrupt (*ret instructions). • The *instruction_address* for: ◦ Jumps with a target that cannot be inferred from the source code; ◦ The instruction retired immediately after a jump with a target that cannot be inferred from the source code (also referred to as the target or destination of the jump); ◦ Taken and nontaken branches; ◦ The last instruction retired before an exception or interrupt; ◦ The first instruction retired following an exception or interrupt; ◦ The last instruction retired before a privilege change; ◦ The first instruction retired following a privilege change.

ID	REFERENCE	TYPE	DEFINITION
RVET.4.5	4.1 (p.17, p.18)	R	The following information is optional: • Context or Time information: ◦ The context and/or hart ID and/or time; ◦ The type of action to take when context or time data changes. • The *instruction_type* of instructions for: ◦ Calls with a target that cannot be inferred from the source code; ◦ Calls with a target that can be inferred from the source code; ◦ Jumps with a target that cannot be inferred from the source code; ◦ Jumps with a target that can be inferred from the source code; ◦ Returns with a target that cannot be inferred from the source code; ◦ Returns with a target that can be inferred from the source code; ◦ Co-routine swap; ◦ Other jumps which don't fit any of the above classifications with a target that cannot be inferred from the source code; ◦ Other jumps which don't fit any of the above classifications with a target that can be inferred from the source code. • If context or time is supported then the instruction_address for: ◦ The last instruction retired before a context or a time change; ◦ The first instruction retired following a context or time change. • Whether jump targets are sequentially inferable or not.
RVET.4.6	4.1 (p.18)	R	The mandatory information is the bare-minimum required to implement the branch trace algorithm outlined in Chapter 9.
RVET.4.7	4.1 (p.18)	O	The optional information facilitates alternative or improved trace algorithms: • Implicit return mode (see Section 3.2.5) requires the encoder to keep track of the number of nested function calls, and to do this it must be aware of all calls and returns regardless of whether the target can be inferred or not; • A simpler algorithm useful for basic code profiling would only report function calls and returns, again regardless of whether the target can be inferred or not; • Branch prediction techniques can be used to further improve the encoder efficiency, particularly for loops (see Section 3.2.6). This requires the encoder to be aware of the address of all branches, whether they are taken or not. • Uninferable jumps can be treated as inferable (which don't need to be reported in the trace output) if both the jump and the preceding instruction which loads the target into a register have been traced.
RVET.4.8	4.1.1 (p.18)	H	Jump classification and target inference
RVET.4.9	4.1.1 (p.18)	I	Jumps are classified as *inferable*, or *uninferable*.
RVET.4.10	4.1.1 (p.18)	R	An *inferable* jump has a target which can be deduced from the binary executable or representation thereof (e.g. ELF). For the purposes of this specification, the following strict definition applies: If the target of a jump is supplied via a constant embedded within the jump opcode, it is classified as *inferable*.
RVET.4.11	4.1.1 (p.18)	R	Jumps which are not *inferable* are by definition *uninferable*.

ID	REFERENCE	TYPE	DEFINITION
RVET.4.12	4.1.1 (p.18)	R	However, there are some jump targets which can still be deduced from the binary executable by considering pairs of instructions even though by the above definition they are classified as uninferable. Specifically, 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).
RVET.4.13	4.1.1 (p.18)	R	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).
RVET.4.14	4.1.1 (p.18, p.19)	R	Note: the restriction that the instructions are retired consecutively is necessary in order to minimize the additional signalling 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.
RVET.4.15	4.1.1 (p.19)	O	Support for sequentially inferable jumps is optional.
RVET.4.16	4.1.1 (p.19)	O	Jumps may optionally be further classified according to the recommended calling convention: • *Calls*: ◦ ***jal*** x1; ◦ ***jal*** x5; ◦ ***jalr*** x1, rs where rs != x5; ◦ ***jalr*** x5, rs where rs != x1; ◦ ***c.jalr*** rs1 where rs1 != x5; ◦ ***c.jal***. • *Jumps*: ◦ ***jal*** x0; ◦ ***c.j***; ◦ ***jalr*** x0, rs where rs != x1 and rs != x5; ◦ ***c.jr*** rs1 where rs1 != x1 and rs1 != x5. • *Returns*: ◦ ***jalr*** rd, rs where (rs == x1 or rs == x5) and rd != x1 and rd != x5; ◦ ***c.jr*** rs1 where rs1 == x1 or rs1 == x5. • *Co-routine swap*: ◦ ***jalr*** x1, x5; ◦ ***jalr*** x5, x1; ◦ ***c.jalr*** x5. • *Other*: ◦ ***jal*** rd where rd != x0 and rd != x1 and rd != x5; ◦ ***jalr*** rd, rs where rs != x1 and rs != x5 and rd != x0 and rd != x1 and rd != x5.
RVET.4.17	4.1.2 (p.19)	H	Relationship between RISC-V core and the encoder
RVET.4.18	4.1.2 (p.19)	R	The encoder is intended to encode the instructions executed on a single hart.
RVET.4.19	4.1.2 (p.19)	R	It is however commonplace for a RISC-V core to contain multiple harts. This can be supported by the core in several different ways: • Implement a separate instance of the interface per hart. Each instance can be connected to a separate encoder instance, allowing all harts to be traced concurrently. Alternatively, external muxing may be used in conjunction with a single encoder in order to trace one particular hart at a time; • Implement a singe interface for the core, with muxing inside the core to select which hart to connect to the interface.

ID	REFERENCE	TYPE	DEFINITION
RVET.4.20	4.1.2 (p.19, 9.20)	C	(Whilst it is technically feasible to use a single encoder with multiple harts operating in a fine-grained multi-threaded configuration, the frequent context changes that would occur as a result of thread-switching would result in extremely poor encoding efficiency, and so this configuration is not recommended.)
RVET.4.21	4.2 (p.20)	H	Instruction Trace Interface
RVET.4.22	4.2 (p.20)	I	This section describes the interface between a RISC-V hart and the trace encoder that conveys the information described in the section Section 4.1.
RVET.4.23	4.2 (p.20)	I	Signals are assigned to one of the following groups: • M: Mandatory. The interface must include an instance of this signal. • O: Optional. The interface may include an instance of this signal. • MR: Mandatory, may be replicated. For harts that can retire a maximum of N "special" instructions per clock cycle, the interface must include N instances of this signal. • OR: Optional, may be replicated. For harts that can retire a maximum of N "special" per clock cycle, the interface must include zero or N instances of this signal. • BR: Block, may be replicated. Mandatory for harts that can retire multiple instructions in a block. Replication as per OR. If omitted, the interface must include SR group signals instead. • SR: Single, may be replicated. Mandatory for harts that can only retire one instruction in a block. Replication as per OR (see Section 4.2.2). If omitted, the interface must include BR group signals instead.
RVET.4.24	4.2 (p.20)	I	"Special" instructions are those that require **itype** to be non-zero.

ID	REFERENCE	TYPE	DEFINITION
RVET.4.25	4.2 (p.20) Table 4	R	Instruction interface signals

Signal	Group	Function
itype[*itype_width_p*-1:0]	MR	Termination type of the instruction block. Encoding given in Table 7 (see Section 4.1.1 for definitions of codes 6 - 15).
cause[*ecause_width_p*-1:0]	M	Exception or interrupt cause (***ucause***/***scause***/ ***vscause***/***mcause***). Ignored unless **itype**=1 or 2.
tval[*iaddress_width_p*-1:0]	M	The associated trap value, e.g. the faulting virtual address for address exceptions, as would be written to the ***utval***/***stval***/***vstval***/***mtval*** CSR. Future optional extensions may define tval to provide ancillary information in cases where it currently supplies zero. Ignored unless **itype**=1.
priv[*privilege_width_p*-1:0]	M	Privilege level for all instructions retired on this cycle. Encoding given in Table 8. Codes 4-7 optional.
iaddr[*iaddress_width_p*-1:0]	MR	The address of the 1st instruction retired in this block. Invalid if iretire=0 unless itype=1, in which case it indicates the address of the instruction which incurred the exception.
context[*context_width_p*-1:0]	O	Context for all instructions retired on this cycle.
time[*time_width_p*-1:0]	O	Time generated by the core.
ctype[*ctype_width_p*-1:0]	O	Reporting behavior for context. Encoding given in Table #tab:context-type. Codes 2-3 optional.
sijump	OR	If **itype** indicates that this block ends with an uninferable discontinuity, setting this signal to 1 indicates that it is sequentially inferable and may be treated as inferable by the encoder if the preceding ***auipc***, ***lui*** or ***c.lui*** has been traced. Ignored for **itype** codes other than 6, 8, 10, 12 or 14.

RVET.4.26	4.2 (p.20) Table 5	R	Instruction interface signals - multiple retirement per block

Signal	Group	Function
iretire[*iretire_width_p*-1:0]	BR	Number of halfwords represented by instructions retired in this block.
ilastsize[*ilastsize_width_p*-1:0]	BR	The size of the last retired instruction is $2^{\text{ilastsize}}$ half-words.

RVET.4.27	4.2 (p.20)	R	Table 4 and Table 5 list the signals in the interface designed to efficiently support retirement of multiple instructions per cycle.
RVET.4.28	4.2 (p.20)	I	The following discussion describes the multiple-retirement behavior.
RVET.4.29	4.2 (p.20)	I	However, for harts that can only retire one instruction at a time, the signalling can be simplified, and this is discussed subsequently in Section 4.2.1.

ID **REFERENCE** **TYPE** **DEFINITION**

RVET.4.30 4.2 (p.21) Table 6 R Instruction interface signals - single retirement per block

Signal	Group	Function
iretire[0:0]	SR	Number of instructions retired in this block (0 or 1).
ilastsize[*ilastsize_width_p*-1:0]	SR	The size of the retired instruction is 2ilastsize half-words.

RVET.4.31 4.2 (p.21) Table 7 R Instruction Type (**itype**) encoding

Value	Description
0	Final instruction in the block is none of the other named **itype** codes
1	Exception. An exception that traps occurred following the final retired instruction in the block
2	Interrupt. An interrupt that traps occurred following the final retired instruction in the block
3	Exception or interrupt return
4	Nontaken branch
5	Taken branch
6	Uninferable jump if *itype_width_p* is 3, reserved otherwise
7	reserved
8	Uninferable call
9	Inferrable call
10	Uninferable jump
11	Inferrable jump
12	Co-routine swap
13	Return
14	Other uninferable jump
15	Other inferable jump

RVET.4.32 4.2 (p.21) R The information presented in a block represents a contiguous block of instructions starting at **iaddr**, all of which retired in the same cycle.

RVET.4.33 4.2 (p.21) R Note if **itype** is 1 or 2 (indicating an exception or an interrupt), the number of instructions retired may be zero.

RVET.4.34 4.2 (p.21) R **cause** and **tval** are only defined if **itype** is 1 or 2.

RVET.4.35 4.2 (p.21) R If **iretire**=0 and **itype**=0, the values of all other signals are undefined.

RVET.4.36 4.2 (p.21) Table 8 R Privilege level (**priv**) encoding

Value	Description
0	U
1	S/HS
2	reserved
3	M
4	D (debug mode)
5	VU
6	VS
7	reserved

RVET.4.37 4.2 (p.21) R **iretire** contains the number of (16-bit) half-words represented by instructions retired in this block, and **ilastsize** the size of the last instruction.

ID	REFERENCE	TYPE	DEFINITION
RVET.4.38	4.2 (p.21)	R	Half-words rather than instruction count enables the encoder to easily compute the address of the last instruction in the block without having access to the size of every instruction in the block.
RVET.4.39	4.2 (p.21)	R	**itype** can be 3 or 4 bits wide.
RVET.4.40	4.2 (p.21)	R	If *itype_width_p* is 3, a single code (6) is used to indicate all uninferable jumps.
RVET.4.41	4.2 (p.21)	I	This is simpler to implement, but precludes use of the implicit return mode (see Section 3.2.5), which requires jump types to be fully classified.
RVET.4.42	4.2 (p.21)	I	Whilst **iaddr** is typically a virtual address, it does not affect the encoder's behavior if it is a physical address.
RVET.4.43	4.2 (p.22)	R	For harts that can retire a maximum of N non-zero **itype** values per clock cycle, the signal groups MR, OR and either BR or SR must be replicated N times.
RVET.4.44	4.2 (p.22)	I	Typically N is determined by the maximum number of branches that can be retired per clock cycle.
RVET.4.45	4.2 (p.22)	R	Signal group 0 represents information about the oldest instruction block, and group N-1 represents the newest instruction block.
RVET.4.46	4.2 (p.22)	R	The interface supports no more than one privilege change, context change, exception or interrupt per cycle and so signals in groups M and O are not replicated.
RVET.4.47	4.2 (p.22)	R	Furthermore, **itype** can only take the value 1 or 2 in one of the signal groups, and this must be the newest valid group (i.e. **iretire** and **itype** must be zero for higher numbered groups).
RVET.4.48	4.2 (p.22)	R	If fewer than N groups are required in a cycle, then lower numbered groups must be used first.
RVET.4.49	4.2 (p.22)	R	For example, if there is one branch, use only group 0, if there are two branches, instructions up to the 1st branch must be reported in group 0 and instructions up to the 2nd branch must be reported in group 1 and so on.
RVET.4.50	4.2 (p.22)	R	**sijump** is optional and may be omitted if the hart does not implement the logic to detect sequentially inferable jumps.
RVET.4.51	4.2 (p.22)	R	If the encoder offers an **sijump** input it must also provide a parameter to indicate whether the input is connected to a hart that implements this capability, or tied off.
RVET.4.52	4.2 (p.22)	R	This is to ensure the decoder can be made aware of the hart's capability.
RVET.4.53	4.2 (p.22)	R	Enabling sequentially inferable jump mode in the encoder and decoder when the hart does not support it will prevent correct reconstruction by the decoder.
RVET.4.54	4.2 (p.22)	R	The context and/or the time field can be used to convey any additional information to the decoder.

ID	REFERENCE	TYPE	DEFINITION
RVET.4.55	4.2 (p.22)	I	For example: • The address space and virtual machine IDs (ASID and VMID respectively). Where present it is recommended these values be wired to bits [15:0] and [29:16]; • The software thread ID; • The process ID from an operating system; • It could be used to convey the values of CSRs to the decoder by setting context to the CSR number and value when a CSR is written; • In cases where a single encoder is being shared amongst multiple harts (see Section 4.1.2), it could also be used to indicate the hart ID, in cases where the hart ID can be changed dynamically. • Time from within the hart
RVET.4.56	4.2 (p.22) Table 9	R	Table 9 specifies the actions for the various **ctype** values. Context type **ctype** values and corresponding actions

Type	Value	Actions
Unreported	0	No action (don't report context).
Report context imprecisely	1	An example would be a SW thread or operating system process change. Report the new context value at the earliest convenient opportunity. It is reported without any address information, and the assumption is that the precise point of context change can be deduced from the source code (e.g. a CSR write).
Report context precisely	2	Report the address of the 1st instruction retired in this block, and the new context. If there were unreported branches beforehand, these need to be reported first. Treated the same as a privilege change.
Report context as an	3	An example would be a change of hart. asynchronous discontinuity

RVET.4.57	4.2 (p.22)	I	A typical behavior would be for this signal to remain zero except on the 1st retirement after a context change or when a time value should be reported.
RVET.4.58	4.2 (p.22)	R	*ctype_width_p* may be 1 or 2.
RVET.4.59	4.2 (p.22)	R	The reduced width option only provides support for reporting context changes imprecisely.
RVET.4.60	4.2.1 (p.22)	H	Simplifications for single-retirement
RVET.4.61	4.2.1 (p.22)	R	For harts that can only retire one instruction at a time, the interface can be simplified to the signals listed in Table 4 and Table 6.
RVET.4.62	4.2.1 (p.22, p.23)	R	The simplifications can be summarized as follows: • **iretire** simply indicates whether an instruction retired or not;
RVET.4.63	4.2.1 (p.23)	C	Note: **ilastsize** is still needed in order to determine the address of the next instruction, as this is the predicted return address for implicit return mode (see Section 3.2.5).
RVET.4.64	4.2.1 (p.23)	R	The parameter *retires_p* which indicates to the encoder the maximum number of instructions that can be retired per cycle can be used by an encoder capable of supporting single or multiple retirement to select the appropriate interpretation of **iretire**.
RVET.4.65	4.2.2 (p.23)	H	Alternative multiple-retirement interface configurations
RVET.4.66	4.2.2 (p.23)	R	For a hart that can retire multiple instructions per cycle, but no more than one branch, the preferred solution is to use one instance of signals from groups BR, MR and OR.

ID	REFERENCE	TYPE	DEFINITION
RVET.4.67	4.2.2 (p.23)	R	However, if the hart can retire N branches in a cycle, N instances of signals from groups MR, OR and either SR or BR must be used (each instance can be either a single instruction or a block).
RVET.4.68	4.2.2 (p.23)	R	If the hart can retire N instructions per cycle, but only one branch, it is allowed (though not recommended) to provide explicit details of every instruction retired by using N instances of signals from groups SR, MR and OR.
RVET.4.69	4.2.3 (p.23)	H	Optional sideband signals
RVET.4.70	4.2.3 (p.23)	O	Optional sideband signals may be included to provide additional functionality, as described in Table 10 and Table 11.
RVET.4.71	4.2.3 (p.23)	R	Note, any user defined information that needs to be output by the encoder will need to be applied via the context input.
RVET.4.72	4.2.3 (p.23) Table 10	R	Optional sideband encoder input signals

Signal	Group	Function
impdef[*impdef_width_p*-1:0]	O	Implementation defined sideband signals. A typical use for these would be for filtering (see Chapter 5).
trigger[2+:0]	1:0]: O [2+]: OR	A pulse on bit 0 will cause the encoder to start tracing, and continue until further notice, subject to other filtering criteria also being met. A pulse on bit 1 will cause the encoder to stop tracing until further notice. See Section 4.2.4).
halted	O	Hart is halted. Upon assertion, the encoder will output a packet to report the address of the last instruction retired before halting, followed by a support packet to indicate that tracing has stopped. Upon deassertion, the encoder will start tracing again, commencing with a synchronization packet. Note: If this signal is not provided, it is strongly recommended that Debug mode can be signalled via a 3-bit privilege signal. This will allow tracing in Debug mode to be controlled via the optional filtering capabilities.
reset	O	Hart is in reset. Provided the encoder is in a different reset domain to the hart, this allows the encoder to indicate that tracing has ended on entry to reset, and restarted on exit. Behavior is as described above for halt.

RVET.4.73	4.2.3 (p.23) Table 11	R	Optional sideband encoder output signals

Signal	Group	Function
stall	O	Stall request to hart. Some applications may require lossless trace, which can be achieved by using this signal to stall the hart if the trace encoder is unable to output a trace packet (for example due to back-pressure from the packet transport infrastructure).

RVET.4.74	4.2.4 (p.23)	H	Using trigger outputs from the Debug Module
RVET.4.75	4.2.4 (p.23)	R	The debug module of the RISC-V hart may have a trigger unit.

ID	REFERENCE	TYPE	DEFINITION
RVET.4.76	4.2.4 (p.23)	R	This defines a match control register (***mcontrol***) containing a 4-bit action field, and reserves codes 2 - 5 of this field for trace use.
RVET.4.77	4.2.4 (p.23)	R	These action codes are hereby defined as shown in table Table 12.
RVET.4.78	4.2.4 (p.23)	R	If implemented, each action must generate a pulse on an output from the hart, on the same cycle as the instruction which caused the trigger is retired.
RVET.4.79	4.2.4 (p.24) Table 12	R	Debug Module trigger support (***mcontrol*** action)

Value	Description
2	*Trace-on*. This should be connected to **trigger[0]** if the encoder provides it.
3	*Trace-off*. This should be connected to **trigger[1]** if the encoder provides it.
4	*Trace-notify*. This should be connected to **trigger[**1 + ***blocks***:2**]** if the encoder provides it. This will cause the encoder to output a packet containing the address of the last instruction in the block if it is enabled. One bit per block.

RVET.4.80	4.2.4 (p.24)	R	Trace-on and Trace-off actions provide a means for the hart to control when tracing starts and stops.
RVET.4.81	4.2.4 (p.24)	R	It is recommended that tracing starts from the oldest instruction retired in the cycle that Trace-on is asserted, and stops following the newest instruction retired in the cycle that Trace-off is asserted (subject to any optional filtering).
RVET.4.82	4.2.4 (p.24)	R	Trace-notify provides means to ensure that a specified instruction is explicitly reported (subject to any optional filtering). This capability is sometimes known as a watchpoint.
RVET.4.83	4.2.5 (p.24)	H	Example retirement sequences
RVET.4.84	4.2.5 (p.24) Table 13	I	Example 1 : 9 Instructions retired over four cycles, 2 branches

Retired	Instruction Trace Block
1000: ***divuw*** 1004: ***add*** 1008: ***or*** 100C: ***c.jalr***	**iretire**=7, **iaddr**=0x1000, **itype**=8
0940: ***addi*** 0944: ***c.beq***	**iretire**=3, **iaddr**=0x0940, **itype**=4
0946: ***c.bnez***	**iretire**=1, **iaddr**=0x0946, **itype**=5
0988: ***lbu*** 098C: ***csrrw***	**iretire**=4, **iaddr**=0x0988, **itype**=0

RVET.4.85	4.3 (p.24)	H	Data Trace Interface requirements
RVET.4.86	4.3 (p.24)	I	This section describes in general terms the information which must be passed from the RISC-V hart to the trace encoder for the purposes of Data Trace, and distinguishes between what is mandatory, and what is optional.
RVET.4.87	4.3 (p.24)	R	If Data Trace is not needed in a system then there is no requirement for the RISC-V hart to supply any of the signals in Section 4.4.
RVET.4.88	4.3 (p.24)	R	Data trace supports up to four data access types: load, store, atomic and CSR. Support for both atomic and CSR accesses are independently optional.

ID	REFERENCE	TYPE	DEFINITION
RVET.4.89	4.3 (p.24)	R	The signalling protocol can take one of two forms, depending on the needs of the RISC-V hart: *unified* or *split*.
RVET.4.90	4.3 (p.24)	I	Unified is the simplest form, suitable for simpler, in-order harts.
RVET.4.91	4.3 (p.24)	R	In this form, all information about a data access is signalled by the RISC-V hart in the same cycle that the associated data access instruction is reported on the instruction trace interface.
RVET.4.92	4.3 (p.24)	R	For harts with out of order or speculative execution capabilities, many loads may be in progress simultaneously, and this approach is not practical as it would require the hart to maintain a large amount of state relating to all the in-progress loads.
RVET.4.93	4.3 (p.24, p.25)	R	For this reason, the interface also supports splitting loads into two parts: • The request phase provides all the information about the load that originates from the hart (address, size, etc.) when the instruction retires; • The response phase provides the data and response status when it has been returned to the hart from the memory system.
RVET.4.94	4.3 (p.25)	R	The two parts of a split load are associated by use of a transaction ID.
RVET.4.95	4.3 (p.25)	R	The Zc (code-size reduction) extension introduced push and pop instructions (*cm.push*, *cm.pop*, *cm.popret* and *cm.popretz*) that each result in multiple loads or stores.
RVET.4.96	4.3 (p.25)	R	To allow the resulting loads or stores to be associated with the correct instruction, these multi-memory-access instructions (and any other future instructions with similar characteristics) must be reported on the instruction trace interface multiple times (once for each individual load or store) using **itype** 0 except for the final load or store, which must retire using the natural **itype** for the instruction (for example, a *cm.popret* instruction must use **itype** 13 for the final load to signal the return).
RVET.4.97	4.3 (p.25)	R	The instruction address reported will be the same for each occurrence.
RVET.4.98	4.3 (p.25)	I	The following illustrations show the retirement sequences when a single *cm.push* or *cm.popret* is used to push or pop 4 registers from the stack.
RVET.4.99	4.3 (p.25)	I	They assume a RISC-V to encoder interface that can report a block of 1 or more retired instructions and one load or store per cycle.
RVET.4.100	4.3 (p.25)	I	Each comprises 4 elements, and shows the instruction information reported for each load and store.
RVET.4.101	4.3 (p.25)	R	As detailed in section #sec:InstructionTraceInterface[1.2], this takes the form of the address of an instruction, the length of the block (1 for a single instruction) and the type of the final instruction in the block.
RVET.4.102	4.3 (p.25)	R	In each element, 'Block' indicates a block of 1 or more instructions (i.e. could also be a single instruction), whereas 'Single' indicates a single instruction (i.e. a block with a length of 1).
RVET.4.103	4.3 (p.25)	R	A *cm.push* is equivalent to 4 store instructions: 1. Block - last instruction is *cm.pus*h, **itype** 0 (data trace interface reports 1st store); 2. Single - *cm.push*, **itype** 0 (data trace interface reports 2nd store); 3. Single - *cm.push*, **itype** 0 (data trace interface reports 3rd store); 4. Block - 1st instruction is *cm.push*, **itype** dependent on last instruction in block (data trace interface reports 4th store);

ID	REFERENCE	TYPE	DEFINITION
RVET.4.104	4.3 (p.25)	R	A *cm.popret* is equivalent to 4 loads and a return: 1. Block - last instruction is *cm.popret*, **itype** 0 (data trace interface reports 1st load); 2. Single - *cm.popret*, **itype** 0 (data trace interface reports 2nd load); 3. Single - *cm.popret*, **itype** 0 (data trace interface reports 3rd load); 4. Single - *cm.popret*, **itype** 13 (data trace interface reports 4th load);
RVET.4.105	4.3 (p.25)	R	If an exception occurs part way through the sequence of loads or stores initiated by such an instruction, and the instruction is re-executed after the exception handler has been serviced, the load or store sequence must recommence from the beginning.
RVET.4.106	4.3 (p.25)	C	This is required for data trace only. If data trace is not implemented, the push or pop may instead be reported just once in the normal way when all associated loads or stores complete successfully.
RVET.4.107	4.4 (p.26)	H	Data Trace Interface
RVET.4.108	4.4 (p.26)	I	This section describes the interface between a RISC-V hart and the trace encoder that conveys the information described in the Section 4.3.
RVET.4.109	4.4 (p.26)	R	Signals are assigned to one of the following groups: • M: Mandatory. The interface must include an instance of this signal; • U: Unified. Mandatory for unified signalling; • S: Split. Mandatory for split load signalling; • O: Optional. The interface may include an instance of this signal.
RVET.4.110	4.4 (p.26)	R	All signals in M, U and O groups are only valid when **dretire** is high.
RVET.4.111	4.4 (p.26) Table 14	R	Signals in the S group are valid as indicated in the table below. Data interface signals

Signal	Group	Function
dretire	M	Data access retired (when high)
dtype[*dtype_width_p*-1:0]	M	Data access type. Encoding given in Table 15
daddr[*daddress_width_p*-1:0]	M	The data access address
dsize[*dsize_width_p*-1:0]	M	The data access size is 2dsize bytes
data[*data_width*_p-1:0]	U	The data
iaddr_lsbs[*iaddr_lsbs_width_p*-1:0]	O	LSBs of the data access instruction address. Required if retires_p > 1
dblock[*dblock_width_p*-1:0]	O	Instruction block in which the data access instruction is retired. Required if there are replicated instruction block signals
lrid[*lrid_width_p*-1:0]	S	Load request ID. Valid when **dretire** is high
lresp[*lresp_width_p*-1:0]	S	Load response:: None: reserved: Okay. Load successful; ldata valid: Error. Load failed; **ldata** not valid
lid[*lrid_width_p*-1:0]	S	Split Load ID. Valid when **lresp** is non-zero
sdata[*sdata_width_p*-1:0]	S	Store data. Valid when **dretire** is high
ldata[*ldata_width_p*-1:0]	S	Load data. Valid when **lresp** is non-zero

RVET.4.112	4.4 (p.26)	R	For harts that can retire a maximum of M data accesses per cycle, the implemented signal groups must be replicated M times.

ID	REFERENCE	TYPE	DEFINITION
RVET.4.113	4.4 (p.26)	R	If fewer than M groups are required in a cycle, then lower numbered groups must be used first.
RVET.4.114	4.4 (p.26)	I	For example, if there is one data access, use only group 0.
RVET.4.115	4.4 (p.26) Table 15	R	Data access type (dtype) encoding

Value	Description
0	Load
1	Store
2	reserved
3	reserved
4	CSR read-write
5	CSR read-set
6	CSR read-clear
7	reserved
8	Atomic swap
9	Atomic add
10	Atomic AND
11	Atomic OR
12	Atomic XOR
13	Atomic max
14	Atomic min
15	Conditional store failure

RVET.4.116	4.4 (p.27)	R	The maximum value of *dtype_width_p* is 4.
RVET.4.117	4.4 (p.27)	O	However, if only loads and stores are supported, *dtype_width_p* can be 1.
RVET.4.118	4.4 (p.27)	O	If CSRs are supported but atomics are not, *dtype_width_p* can be 3.
RVET.4.119	4.4 (p.27)	R	Atomic and CSR accesses have either both load and store data, or store data and an operand.
RVET.4.120	4.4 (p.27)	R	For CSRs and unified atomics, both values are reported via data, with the store data in the LSBs and the load data or operand in the MSBs.
RVET.4.121	4.4 (p.27)	R	*lrid_width_p* is determined by the maximum number of loads that can be in progress simultaneously, such that at any one time there can be no more than one load in progress with a given ID.
RVET.4.122	4.4 (p.27)	R	**iaddr_lsbs** and **dblock** are provided to support filtering of which data accesses to trace based on their instruction address.
RVET.4.123	4.4 (p.27)	I	This is best illustrated by considering the following instruction sequence: 1. load 2. <some non data access instruction> 3. load 4. <some non data access instruction> 5. <some non data access instruction>

ID	REFERENCE	TYPE	DEFINITION
RVET.4.124	4.4 (p.27)	I	Suppose the hart is capable of retiring up to 4 instructions in a cycle, via a single block. Instruction trace is enabled throughout, but the requirement is to collect data trace for the 1st load (instruction 1), and filtering is configured to match the address of this instruction only. However, information about instruction addresses is passed to the encoder at the block level, and the block boundaries are invisible to the decoder. For instruction trace, all instructions in a block are traced if any of the instructions in that block match the filtering criteria. That is fine for instruction trace - the address of the 1st and last traced instruction are output explicitly. There will be some fuzziness about precisely what those addresses will be depending on where the block boundaries fall, but this is not a concern as everything is always self-consistent.
RVET.4.125	4.4 (p.27)	I	However, that is not the case for data trace. Consider two scenarios: • Case 1: 1st block contains instructions 1, 2, 3; second block contains 4, 5 • Case 2: 1st block contains instructions 1, 2; second block contains 3, 4, 5
RVET.4.126	4.4 (p.27)	I	Given that **iretire** is non-zero in the same cycle that the data access retires, the encoder knows the address of the 1st and last instructions in a block, but does not know precisely where in the block the data access is. In both cases, the first block matches the filtering criteria (it contains the address of instruction 1), and the second block does not. But if the encoder traced all the data accesses in the matching block, then in case 1 it would trace both instructions 1 and 3, whereas in the second case it would trace only instruction 1. The decoder has no visibility of the block boundaries so cannot account for this. It is expecting only instruction 1 to be traced, and so may misinterpret instruction 3. If this code is in a loop for example, it will assume that the 2nd traced load is in fact instruction 1 from the next loop iteration, rather than instruction 3 from this iteration.
RVET.4.127	4.4 (p.27)	R	Providing the LSBs of the data access instruction address allows the decoder to determine precisely whether the data access should be traced or not, and removes the dependency on the block sizes and boundaries.
RVET.4.128	4.4 (p.27)	R	The number of bits required is one more bit than the number required to index within the block because blocks can start on any half-word boundary.
RVET.4.129	4.4 (p.27)	R	For harts that replicate the block signals to allow multiple blocks to retire per cycle it is also necessary to indicate which block each data access is associated with, so the encoder knows which block address to combine with the LSBs in order to construct the actual data access instruction address. 1 bit for 2 blocks per cycle, 2 bits for 4, and so on.

CHAPTER 5 Filtering

ID	REFERENCE	TYPE	DEFINITION
RVET.5.1	5.0 (p.29)	H	Filtering
RVET.5.2	5.0 (p.29)	I	The contents of this chapter are informative only.
RVET.5.3	5.0 (p.29)	I	Filtering provides a mechanism to control whether the encoder should produce trace. For example, it may be desirable to trace: • When the instruction address is within a particular range; • Starting from one instruction address and continuing until a second instruction address; • For one or more specified privilege levels; • For a particular context or range of contexts; • Exception and/or interrupt handlers for specified exception causes or with particular tval values; • Based on values applied to the impdef or trigger signals; • For a fixed period of time • etc.
RVET.5.4	5.0 (p.29)	I	How this is accomplished is implementation specific.
RVET.5.5	5.0 (p.29)	O	One suggested implementation partitions the architecture into filters and comparators in order to provide maximum flexibility at low cost. The number of filters and comparators is system dependent.
RVET.5.6	5.0 (p.29)	I	Each comparator unit is actually a pair of comparators (Primary and Secondary, or P, S) allowing a bounded range to be matched with a single unit if required, and offers: • input selected from iaddress, context and tval (and daddress if data trace is supported); • A range of arithmetic options (<, >, =, !=, etc) independently selectable for each comparator; • Secondary match value may be used as a mask for the primary comparator; • The two comparators can be combined in several ways: P, P&&S, ! (P&&S), latch (set on P clear on S); • Each comparator can also be used to explcitly report a particular instruction address (i.e. generate a watchpoint).
RVET.5.7	5.0 (p.29)	I	Each filter can specify filtering against instruction and optionally data trace inputs from the HART, and offers: • Require up to 3 run-time selectable comparator units to match; • Multiple choice selection for priv and cause inputs (and dtype if data trace is supported); • Masked matching for interrupt and impdef inputs.
RVET.5.8	5.0 (p.29)	I	Allowing for up to 3 comparators allows for simultaneous matching on Address, Trap value and context (unlikely, but should not be architecturally precluded).
RVET.5.9	5.0 (p.29)	I	The filtering configuration fields are detailed in Chapter 2. These support the architecture described above, though will also support simpler implementations, for example where the comparator function is more tightly coupled with each filter, or where filtering is provided on only some inputs (such as just instruction address).

CHAPTER 6 Timestamping

ID	REFERENCE	TYPE	DEFINITION
RVET.6.1	6.0 (p.31)	H	Timestamping
RVET.6.2	6.0 (p.31)	I	The support for Timestamps is optional and so the contents of this chapter are informative only.
RVET.6.3	6.0 (p.31)	I	In many systems it is desirable to periodically insert a timestamp packet into the trace stream, effectively marking that point in the stream with a time value.
RVET.6.4	6.0 (p.31)	I	This can be used to judge "time" between various point in the trace stream and, more notably, to be able to correlate trace streams from different harts (i.e. this point in hart A's stream occurred at roughly the same time as that point in hart B's trace stream). The former helps one to judge performance of sections of code execution (to the granularity of timestamp insertion). The latter helps debugging multi-hart MP problems.
RVET.6.5	6.0 (p.31)	I	An implementation may have the following: • A timestamp is (up to) a 64-bit time value. • Configurable options for generating timestamp values such as a hart's 'time' values or 'cycle' values. • Options could may also include things like taking 'time' values with the low 4 or 8 bits dropped off which would create a coarser granularity time values • Timestamp generation may be enabled or disabled. If enabled, a timestamp packet would be generated periodically which may be based on configurable interval or rate, e.g. once every 2^n items where 'n' and 'items' are configurable among some limited set of choices. The choices could be: ◦ Time ◦ Time scaled down. An implementation specific scaled or divided down derivative of time. This may be useful in providing a smaller coarser granularity values ◦ Time Interpolated up. An implementation specific interpolated up derivative of time. This may be useful in providing higher resolution time values ◦ Cycle ◦ Implementation specific • A timestamp packet may also be generated in conjunction with a sync packet • Timestamp packets are highly compressible and variable in size depending on the number of low bits of the current value that have changed wrt the last emitted timestamp value. If timestamp packets are emitted rarely (but not as rare as sync packets), then they will tend to be, say, 2-4 bytes in size (still much less than the full up to 64-bit size). If timestamp packets are emitted somewhat frequently, then they will tend to be 1-2 bytes in size. If timestamp packets are emitted very frequently, then they will tend to be <1 byte in size. Timestamp values associated with sync packets would always be the full implemented size.

CHAPTER 7 Instruction Trace Encoder Output Packets

ID	REFERENCE	TYPE	DEFINITION
RVET.7.1	7.0 (p.33)	H	Instruction Trace Encoder Output Packets
RVET.7.2	7.0 (p.33)	I	The bulk of this section describes the payload of packets output from the Instruction Trace Encoder.
RVET.7.3	7.0 (p.33)	I	The infrastructure used to transport these packets is outside the scope of this document, and as such the manner in which packets are encapsulated for transport is not specified.
RVET.7.4	7.0 (p.33)	R	However, the following information must be provided to the encapsulator: • The packet type; • The packet length, in bytes; • The packet payload.
RVET.7.5	7.0 (p.33)	I	Two example transport schemes are the Siemens Messaging Infrastructure, and the Arm Trace Bus.
RVET.7.6	7.0 (p.33) Figure 1	I	The figure below shows the encapsulation used for the Siemens infrastructure: Example encapsulated packet format  • The header byte contains a 5-bit field specifying the payload length in bytes, a 2-bit field indicating the "flow" (destination routing indicator), and a bit to indicate whether an optional 16-bit timestamp is present; • The index field indicates the source of the packet. The number of bits is system dependent, And the initial value emitted by the trace encoder is zero (it gets adjusted as it propagates through the infrastructure); • An optional 2-byte timestamp; • The packet payload.
RVET.7.7	7.0 (p.33)	I	Alternatively, for ATB, the source of the packet is indicated by the ATID bus field, and there is no equivalent of "flow", so an example encapsulation might be: • A 5-bit field specifying the payload length in bytes • A bit to indicate whether an optional 16-bit timestamp is present; • An optional 2-byte timestamp; • The packet payload.
RVET.7.8	7.0 (p.33)	I	It may be desirable for packets to start aligned to an ATB word, in which the ATBYTES bus field in the last beat of a packet can be used to indicate the number of valid bytes.
RVET.7.9	7.0 (p.33)	I	The remainder of this section describes the contents of the payload portion which should be independent of the infrastructure.
RVET.7.10	7.0 (p.33)	R	In each table, the fields are listed in transmission order: first field in the table is transmitted first, and multi-bit fields are transmitted LSB first.
RVET.7.11	7.0 (p.33)	R	This packet payload format is used to output encoded instruction trace.
RVET.7.12	7.0 (p.33)	R	Three different formats are used according to the needs of the encoding algorithm. The following tables show the format of the payload - i.e. excluding any encapsulation.

ID	REFERENCE	TYPE	DEFINITION
RVET.7.13	7.0 (p.33)	O	In order to achieve best performance, actual packet lengths may be adjusted using 'sign based compression'.
RVET.7.14	7.0 (p.34)	R	At the very minimum this should be applied to the address field of format 1 and 2 packets, but ideally will be applied to the whole packet, regardless of format.
RVET.7.15	7.0 (p.34)	R	This technique eliminates identical bits from the most significant end of the packet, and adjusts the length of the packet accordingly.
RVET.7.16	7.0 (p.34)	R	A decoder receiving this shortened packet can reconstruct the original full-length packet by sign-extending from the most significant received bit.
RVET.7.17	7.0 (p.34)	R	Where the payload length given in the following tables, or after applying sign-based compression, is not a multiple of whole bytes in length, the payload must be sign-extended to the nearest byte boundary.
RVET.7.18	7.0 (p.34)	I	Whilst offering maximum encoding efficiency, variable length packets can present some challenges, specifically in terms of identifying where the boundaries between packets occur either when packed packets are written to memory, or when packets are streamed offchip via a communications channel.
RVET.7.19	7.0 (p.34)	O	Two potential solutions to this are as follows: • If the maximum packet payload length is 2N-1 (for example, if N is 5, then the maximum length is 31 bytes), and the minimum packet payload length is 1, then a sequence of at least 2N zero bytes cannot occur within a packet payload, and therefore the first non-zero byte seen after a sequence of at least 2N zero bytes must be the first byte of a packet. This approach can be used for alignment in either memory or a data stream; • An alternative approach suitable for packets written to memory is to divide memory into blocks of M bytes (e.g. 1kbyte blocks), and write packets to memory such that the first byte in every block is always the first byte of a packet. This means packets cannot span block boundaries, and so zero bytes must be used to pad between the end of the last message in a block and the block boundary.
RVET.7.20	7.1 (p.34)	H	Format 3 packets
RVET.7.21	7.1 (p.34)	R	Format 3 packets are used for synchronization, traps, reporting context and supporting information.
RVET.7.22	7.1 (p.34)	R	There are 4 sub-formats.
RVET.7.23	7.1 (p.34)	I	Throughout this document, the term "synchronization packet" is used. This refers specifically to format 3, subformat 0 and subformat 1 packets.
RVET.7.24	7.2 (p.34)	H	Format 3 subformat 0 - Synchronisation
RVET.7.25	7.2 (p.34)	R	This packet contains all the information the decoder needs to fully identify an instruction.
RVET.7.26	7.2 (p.34)	R	It is sent for the first traced instruction (unless that instruction also happens to be the first in a trap handler), and when resynchronization has been scheduled by expiry of the resynchronisation timer.

ID	REFERENCE	TYPE	DEFINITION
RVET.7.27	7.2 (p.34) Table16	R	Packet format 3, subformat 0

Field Name	Bits	Description
format	2	11 (sync): synchronisation
subformat	2	00 (start): Start of tracing, or resync
branch	1	Set to 0 if the address points to a branch instruction, and the branch was taken. Set to 1 if the instruction is not a branch or if the branch is not taken.
privilege	*privilege_width_p*	The privilege level of the reported instruction
time	*time_width_p* or 0 if *notime_p* is 1	The time value.
context	*context_width_p*, or 0 if *nocontext_p* is 1	The instruction context.
address	*iaddress_width_p* - *iaddress_lsb_p*	Full instruction address. Address alignment is determined by *iaddress_lsb_p* Address must be left shifted in order to recreate original byte address.

RVET.7.28	7.2.1 (p.35)	H	Format 3 **branch** field
RVET.7.29	7.2.1 (p.35)	R	This bit indicates the taken/not taken status in the case where the reported address points to a branch instruction.
RVET.7.30	7.2.1 (p.35)	C	Overall efficiency would be slightly improved if this bit was removed, and the branch status was instead "carried over" and reported in the next *te_inst* packet. This was considered, but there are several pathological cases where this approach fails. Consider for example the situation where the first traced instruction is a branch, and this is then followed immediately by an exception. This results in format 3 packets being generated on two consecutive instructions. The second packet does not contain a branch map, so there is no way to report the branch status of the 1st branch, apart from by inserting a format 1 packet in between. There are two issues with this: • It would require the generation of 2 packets on the same cycle, which adds significant additional complexity to the encoder; • It would complicate the algorithm shown in Figure 2.
RVET.7.31	7.3 (p.35)	H	Format 3 subformat 1 - Trap
RVET.7.32	7.3 (p.35)	I	This packet also contains all the information the decoder needs to fully identify an instruction.
RVET.7.33	7.3 (p.35)	R	It is sent following an exception or interrupt, and includes the cause, the 'trap value' (for exceptions), and the address of the trap handler, or of the exception itself - Section 7.3.1.
RVET.7.34	7.3 (p.35)	R	If the implicit exception mode is enabled (see Section 3.2.3), the trap handler address is omitted if thaddr is 1.

ID	REFERENCE	TYPE	DEFINITION
RVET.7.35	7.3 (p.35) Table 17	R	Packet format 3, subformat 1

Field Name	Bits	Description
format	2	11 (sync): synchronisation
subformat	2	01 (trap): Exception or interrupt cause and trap handler address.
branch	1	Set to 0 if the address points to a branch instruction, and the branch was taken. Set to 1 if the instruction is not a branch or if the branch is not taken.
privilege	*privilege_width_p*	The privilege level of the reported instruction
time	*time_width_p* or 0 if *notime_p* is 1	The time value.
context	*context_width_p*, or 0 if *nocontext_p* is 1	The instruction context.
ecause	cause_width_p	Exception or interrupt cause.
interrupt	1	Interrupt.
thaddr	1	When set to 1, address points to the trap handler address. When set to 0, address points to the EPC for an exception at the target of an updiscon, and is undefined for other exceptions and interrupts.
address	*iaddress_width_p* - *iaddress_lsb_p*	Full instruction address. Address alignment is determined by *iaddress_lsb_p* Address must be left shifted in order to recreate original byte address.
tval	*iaddress_width_p*	Value from appropriate utval/stval/vstval/mtval CSR. Field omitted for interrupts

RVET.7.36	7.3.1 (p.35)	H	Format 3 **thaddr** and **address** fields
RVET.7.37	7.3.1 (p.35)	R	If an exception occurs at the target of an uninferable PC discontinuity, the value of the EPC cannot be infered from the program binary, and so address contains the EPC and **thaddr** is set to 0.
RVET.7.38	7.3.1 (p.35)	I	In this case, the trap handler address will be reported via a subsequent format 3, subformat 0 packet.
RVET.7.39	7.3.1 (p.35)	R	Usually when an exception or interrupt occurs, the cause is reported along with the 1st address of the trap handler, when that instruction retires.
RVET.7.40	7.3.1 (p.35)	R	In this case, **thaddr** is 1.
RVET.7.41	7.3.1 (p.35)	R	However, if a second interrupt or exception occurs immediately, details of this must still be reported, even though the 1st instruction of the handler hasn’t retired.
RVET.7.42	7.3.1 (p.35)	R	In this situation, **thaddr** is 0, and address is undefined (unless it contains the EPC as outlined in the previous paragraph).
RVET.7.43	7.3.1 (p.36)	C	(The reason for not reporting the EPC for all exceptions when thaddr is 0 is that it may be at either the address of the next instruction or current instruction depending on the exception cause, which can be inferred by the decoder without adding complexity to the encoder.)
RVET.7.44	7.3.2 (p.36)	H	Format 3 **tval** field
RVET.7.45	7.3.2 (p.36)	R	This field reports the "trap value" from the appropriate **utval**/**stval**/**vstval**/**mtval** CSR, the meaning of which is dependent on the nature of the exception.

ID	REFERENCE	TYPE	DEFINITION
RVET.7.46	7.3.2 (p.36)	R	It is omitted from the packet for interrupts.
RVET.7.47	7.4 (p.36)	H	Format 3 subformat 2 - Context
RVET.7.48	7.4 (p.36)	R	This packet contains only the context and/or the timestamp, and is output when the context value changes and can be reported imprecisely (see Table 9).
RVET.7.49	7.4 (p.36) Table 18	R	Packet format 3, subformat 2

Field Name	Bits	Description
format	2	11 (sync): synchronisation
subformat	2	10 (context): Context change
privilege	*privilege_width_p*	The privilege level of the new context.
time	*time_width_p* or 0 if *notime_p* is 1	The time value.
context	*context_width_p*, or 0 if *nocontext_p* is 1	The instruction context.

RVET.7.50	7.5 (p.36)	H	Format 3 subformat 3 - Support
RVET.7.51	7.5 (p.36)	I	This packet provides supporting information to aid the decoder.
RVET.7.52	7.5 (p.36)	R	It is issued when • Trace is enabled or disabled; • The operating mode changes; • One or more trace packets cannot be sent (for example, due back-pressure from the packet transport infrastructure).
RVET.7.53	7.5 (p.36)	R	The options field is a placeholder that must be replaced by an implementation specific set of individual bits - one for each of the optional modes supported by the encoder.

ID	REFERENCE	TYPE	DEFINITION
RVET.7.54	7.5 (p.36) Table 19	R	Packet format 3, subformat 3

Field Name	Bits	Description
format	2	11 (sync): synchronisation
subformat	2	11 (support): Supporting information for the decoder
ienable	*1*	Indicates if the instruction trace encoder is enabled
encoder_mode	N	Identifies trace algorithm Details and number of bits implementation dependent. Currently Branch trace is the only mode defined, indicated by the value 0.
qual_status	2	Indicates qualification status (no_change): No change to filter qualification (ended_rep): Qualification ended, preceding te_inst sent explicitly to indicate last qualification instruction (trace_lost): One or more instruction trace packets lost. (ended_ntr): Qualification
ioptions	N	Values of all instruction trace run-time configuration bits Number of bits and definitions implementation dependent. Examples might be - 'sequentially inferred jumps' Don't report the targets of sequentially inferable jumps - 'implicit return' Don't report function return addresses - 'implicit exception' Exclude address from format 3, sub-format 1 te_inst packets if trap vector can be determined from ecause - 'branch prediction' Branch predictor enabled - 'jump target cache' Jump target cache enabled - 'full address' Always output full addresses (SW debug option)
denable	1	Indicates if the data trace is enabled (if supported)
dloss	1	One of more data trace packets lost (if supported)
doptions	M	Values of all data trace run-time configuration bits Number of bits and definitions implementation dependent. Examples might be – 'no data' Exclude data (just report addresses) - 'no addr' Exclude address (just report data)

RVET.7.55	7.5.1 (p.37)	H	Format 3 subformat 3 **qual_status** field
RVET.7.56	7.5.1 (p.37)	R	When tracing ends, the encoder reports the address of the last traced instruction, and follows this with a format 3, subformat 3 (supporting information) packet.
RVET.7.57	7.5.1 (p.37)	R	Two codes are provided for indicating that tracing has ended: **ended_rep** and **ended_ntr**.
RVET.7.58	7.5.1 (p.37)	I	This relates to exactly the same ambiguous case described in detail in Section 7.6.2, and in principle, the mechanism described in that section can be used to disambiguate when the last traced instruction is at looplabel.
RVET.7.59	7.5.1 (p.37)	I	However, that mechanism relies on knowing when creating the format 1/2 packet, that a format 3 packet will be generated from the next instruction. This is possible because the encoding algorithm uses a 3-stage pipe with access to the previous, current and next instructions.
RVET.7.60	7.5.1 (p.37)	I	However, decoding that the next instruction is a privilege change or exception is straightforward, but determining whether the next instruction meets the filtering criteria is much more involved, and this information won't typically be available, at least not without adding an additional pipeline stage, which is expensive.

ID	REFERENCE	TYPE	DEFINITION
RVET.7.61	7.5.1 (p.37)	R	This means a different mechanism is required, and that is provided by having two codes to indicate that tracing has ended: • **ended_rep** indicates that the preceding packet would not have been issued if tracing hadn't ended, which means that tracing stopped after executing looplabel in the 1st loop iteration; • **ended_ntr** indicates that the preceding packet would have been issued anyway because of an uninferable PC discontinuity, which means that tracing stopped after executing looplabel in the 2nd loop iteration;
RVET.7.62	7.5.1 (p.37)	R	If the encoder implementation does have early access to the filtering results, and the designer chooses to use the **updiscon** bit when the last qualified instruction is also the instruction following an uninferable PC discontinuity, loss of qualification should always be indicated using **ended_rep**.
RVET.7.63	7.6 (p.37)	H	Format 2 packets
RVET.7.64	7.6 (p.37)	R	This packet contains only an instruction address, and is used when the address of an instruction must be reported, and there is no unreported branch information.
RVET.7.65	7.6 (p.37)	R	The address is in differential format unless full address mode is enabled (see Section 3.2.2).
RVET.7.66			

ID	REFERENCE	TYPE	DEFINITION
RVET.7.67	7.6 (p.37) Table 20	R	Packet format 2

Field Name	Bits	Description
format	2	10 (addr-only): differential address and no branch information
address	*iaddress_width_p* - *iaddress_lsb_p*	Differential instruction address.
notify	1	If the value of this bit is different from the MSB of **address**, it indicates that this packet is reporting an instruction that is not the target of an uninferable discontinuity because a notification was requested via **trigger[2]** (see Section 4.2.4).
updiscon	1	If the value of this bit is different from **notify**, it indicates that this packet is reporting the instruction following an uninferable discontinuity and is also the instruction before an exception, privilege change or resync (i.e. it will be followed immediately by a format 3 *te_inst*).
irreport	1	If the value of this bit is different from **updiscon**, it indicates that this packet is reporting an instruction that is either: following a return because its address differs from the predicted return address at the top of the implicit_return return address stack, or the last retired before an exception, interrupt, privilege change or resync because it is necessary to report the current address stack depth or nested call count.
irdepth	*return_stack_size_p* + (*return_stack_size_p* > 0 ? 1 : 0) + *call_counter_size_p*	If the value of **irreport** is different from **updiscon**, this field indicates the number of entries on the return address stack (i.e. the entry number of the return that failed) or nested call count. If **irreport** is the same value as **updiscon**, all bits in this field will also be the same value as **updiscon**.

RVET.7.68	7.6.1 (p.38)	H	Format 2 **notify** field
RVET.7.69	7.6.1 (p.38)	I	This bit is encoded so that most of the time it will take the same value as the MSB of the address field, and will therefore compress away, having no impact on the encoding efficiency.
RVET.7.70	7.6.1 (p.38)	R	It is required in order to cover the case where an address is reported as a result of a notification request, signalled by setting the **trigger[2]** input to 1.
RVET.7.71	7.6.2 (p.38)	H	Format 2 **notify** and **updiscon** fields
RVET.7.72	7.6.2 (p.38)	R	These bits are encoded so that most of the time they will compress away, having no impact on efficiency, by taking on the same value as the preceding bit in the packet (notify is normally the same value as the MSB of the address field, and **updiscon** is normally the same value as **notify**).
RVET.7.73	7.6.2 (p.38)	R	They are required in order to cover a pathological case where otherwise the decoding software would not be able to reconstruct the program execution unambiguously.

ID	REFERENCE	TYPE	DEFINITION

RVET.7.74	7.6.2 (p.38)	I	Consider the following code fragment:
			`looplabel -4: *_opcode A_*`
			`looplabel   : *_opcode B_*`
			`looplabel +4: *_opcode C_*`
			`:`
			`looplabel +N *_JALR_*  # Jump to looplabel`

RVET.7.75	7.6.2 (p.38)	I	This is a loop with an indirect jump back to the next iteration. This is an uninferable discontinuity, and will be reported via a format 1 or 2 packet. Note however that the initial entry into the loop is fallthrough from the instruction at looplabel - 4, and will not be reported explicitly. This means that when reconstructing the execution path of the program, the looplabel address is encountered twice. On first glance, it appears that the decoder can determine when it reaches the loop label for the 1st time that this is not the end of execution, because the preceding instruction was not one that can cause an uninferable discontinuity. It can therefore continue reconstructing the execution path until it reaches the **JALR**, from where it can deduce that opcode B at looplabel is the final retired instruction.
RVET.7.76	7.6.2 (p.38)	I	However, there are circumstances where this approach does not work. For example, consider the case where there is an exception at looplabel + 4. In this case, the decoder cannot tell whether this occurred during the 1st or 2nd loop iterations, without additional information from the encoder. This is the purpose of the updiscon field.
RVET.7.77	7.6.2 (p.38)	I	In more detail: There are four scenarios to consider: 1. Code executes through to the end of the 1st loop iteration, and the encoder reports looplabel using format 1/2 following the **JALR**, then carries on executing the 2nd pass of the loop. In this case **updiscon** == **notify**. The next packet will be a format 1/2; 2. Code executes through to the end of the 1st loop iteration and jumps back to looplabel, but there is then an exception, privilege change or resync in the second iteration at looplabel + 4. In this case, the encoder reports looplabel using format 1/2 following the **JALR**, with **updiscon** == !**notify**, and the next packet is a format 3; 3. An exception occurs immediately after the 1st execution of looplabel. In this case, the encoder reports looplabel using format 0/1/2 with **updiscon** == **notify**, and the next packet is a format 3; 4. The hart requests the encoder to notify retirement of the instruction at looplabel. In this case, the encoder reports the 1st execution of looplabel with **notify** == !**address**[MSB], and subsequent executions with **notify** == **address**[MSB] (because they would have been reported anyway as a result of the **JALR**).
RVET.7.78	7.6.2 (p.38, p.39)	I	Looking at this from the perspective of the decoder, the decoder receives a format 1/2 reporting the address of the 1st instruction in the loop (looplabel). It follows the execution path from the last reported address, until it reaches looplabel.

ID	REFERENCE	TYPE	DEFINITION
RVET.7.79	7.6.2 (p.39)	I	Because looplabel is not preceded by an uninferable discontinuity, it must take the value of **notify** and **updiscon** into consideration, and may need to wait for the next packet in order to determine whether it has reached the final retired instruction: • If **updiscon** == !**notify**, this indicates case 2. The decoder must continue until it encounters looplabel a 2nd time; • If **updiscon** == **notify**, the decoder cannot yet distinguish cases 1 and 3, and must wait for the next packet. ◦ If the next packet is a format 3, this is case 3. The decoder has already reached the correct instruction; ◦ If the next packet is a format 1/2, this is case 1. The decoder must continue until it encounters looplabel a 2nd time. • If **notify** == !**address**[MSB], this indicates case 4, 1st iteration. The decoder has reached the correct instruction.
RVET.7.80	7.6.2 (p.39)	I	This example uses an exception at looplabel + 4, but anything that could cause a format 3 for looplabel + 4 would result in the same behavior: a privilege change, or the expiry of the resync timer. It could also occur if looplabel was the last traced instruction (because tracing was disabled for some reason). See Section 7.5.1 for further discussion of this point.
RVET.7.81	7.6.2 (p.39)	C	Correct decoder behavior could have been achieved by implementing the **notify** bit only, setting it to the inverse of **address**[MSB] whenever an address is reported and it is not the instruction following an uninferable discontinuity. However, this would have been much less efficient, as this would have required **notify** to be different from **address**[MSB] the majority of the time when outputting a format 1/2 before an exception, interrupt or resync (as the probability of this instruction being the target of an uninferable jump is low). Using 2 separate bits results in superior compression.
RVET.7.82	7.6.3 (p.39)	H	Format 2 **irreport** and **irdepth**
RVET.7.83	7.6.3 (p.39)	I	These bits are encoded so that most of the time they will take the same value as the **updiscon** field, and will therefore compress away, having no impact on the encoding efficiency.
RVET.7.84	7.6.3 (p.39)	R	If implicit_return mode is enabled, the encoder keeps track of the number of traced nested calls, either as a simple count (*call_counter_size_p non-zero*) or a stack of predicted return addresses (*return_stack_size_p* nonzero).
RVET.7.85	7.6.3 (p.39)	R	Where a stack of predicted return addresses is implemented, the predicted return addresses are compared with the actual return addresses, and a *te_inst* packet will be generated with **irreport** set to the opposite value to **updiscon** if a misprediction occurs.
RVET.7.86	7.6.3 (p.39)	R	In some cases it is also necessary to report the current stack depth or call count if the packet is reporting the last instruction before an exception, interrupt, privilege change or resync.

ID	REFERENCE	TYPE	DEFINITION
RVET.7.87	7.6.3 (p.39, p.40)	R	There are two cases of concern: • If the reported address is the instruction following a return, and it is not mis-predicted, the encoder must report the current stack depth or call count if it is non-zero. Without this, the decoder would attempt to follow the execution path until it encountered the reported address from the outermost nested call; • If the reported address is not the instruction following a return, the encoder must report the current stack depth or call count unless: ◦ There have been no returns since the last call (in which case the decoder will correctly stop in the innermost call), or ◦ There has been at least one branch since the last return (in which case the decoder will correctly stop in the call where there are no unprocessed branches).
RVET.7.88	7.6.3 (p.40)	R	Without this, the decoder would follow the execution path until it encountered the reported address, and in most cases this would be the correct point.
RVET.7.89	7.6.3 (p.40)	R	However, this cannot be guaranteed for recursive functions, as the reported address will occur multiple times in the execution path.
RVET.7.90	7.7 (p.40)	H	Format 1 packets
RVET.7.91	7.7 (p.40)	R	This packet includes branch information, and is used when either the branch information must be reported (for example because the branch map is full), or when the address of an instruction must be reported, and there has been at least one branch since the previous packet.
RVET.7.92	7.7 (p.40)	R	If included, the address is in differential format unless full address mode is enabled (see Section 3.2.2).

ID	REFERENCE	TYPE	DEFINITION
RVET.7.93	7.7 (p.40) Table 21	R	Packet format 1 - address, branch map

Field Name	Bits	Description
format	2	01 (diff-delta): includes branch information and may include differential address
branches	*5*	Number of valid bits **branch_map**. The number of bits of **branch_map** is determined as follows: : (cannot occur for this format) : 1 bit -3: 3 bits -7: 7 bits -15: 15 bits -31: 31 bits For example if branches = 12, **branch_map** is 15 bits long, and the 12 LSBs are valid.
branch_map	Determined by **branches** field.	An array of bits indicating whether branches are taken or not. Bit 0 represents the oldest branch instruction executed. For each bit: : branch taken : branch not taken
address	*iaddress_width_p* - *iaddress_lsb_p*	Differential instruction address.
notify	1	If the value of this bit is different from the MSB of address, it indicates that this packet is reporting an instruction that is not the target of an uninferable discontinuity because a notification was requested via **trigger[2]** (see Section 4.2.4).
updiscon	1	If the value of this bit is different from the MSB of notify, it indicates that this packet is reporting the instruction following an uninferable discontinuity and is also the instruction before an exception, privilege change or resync (i.e. it will be followed immediately by a format 3 te_inst).
irreport	1	If the value of this bit is different from **updiscon**, it indicates that this packet is reporting an instruction that is either: following a return because its address differs from the predicted return address at the top of the implicit_return return address stack, or the last retired before an exception, interrupt, privilege change or resync because it is necessary to report the current address stack depth or nested call count.
irdepth	*return_stack_size_p* + (*return_stack_size_p* > 0 ? 1 : 0) + *call_counter_size_p*	If the value of **irreport** is different from **updiscon**, this field indicates the number of entries on the return address stack (i.e. the entry number of the return that failed) or nested call count. If **irreport** is the same value as **updiscon**, all bits in this field will also be the same value as updiscon.

ID	REFERENCE	TYPE	DEFINITION
RVET.7.94	7.7 (p.40) Table 22	R	Packet format 1 - no address, branch map

Field Name	Bits	Description
format	2	01 (diff-delta): includes branch information and may include differential address
branches	*5*	Number of valid bits in **branch_map**. The length of **branch_map** is determined as follows: : 31 bits, no **address** in packet -31: (cannot occur for this format)
branch_map	31	An array of bits indicating whether branches are taken or not. Bit 0 represents the oldest branch instruction executed. For each bit: :branch taken : branch not taken

RVET.7.95	7.7.1 (p.40)	H	Format 1 **updiscon** field
RVET.7.96	7.7.1 (p.40)	I	See Section 7.6.2.
RVET.7.97	7.7.2 (p.41)	H	Format 1 **branch_map** field
RVET.7.98	7.7.2 (p.41)	R	When the branch map becomes full it must be reported, but in most cases there is no need to report an address.
RVET.7.99	7.7.2 (p.41)	R	This is indicated by setting branches to 0.
RVET.7.100	7.7.2 (p.41)	R	The exception to this is when the instruction immediately prior to the final branch causes an uninferable discontinuity, in which case branches is set to 31.
RVET.7.101	7.7.2 (p.41)	R	The choice of sizes (1, 3, 7, 15, 31) is designed to minimize efficiency loss.
RVET.7.102	7.7.2 (p.41)	R	On average there will be some 'wasted' bits because the number of branches to report is less than the selected size of the branch_map field.
RVET.7.103	7.7.2 (p.41)	R	Using a tapered set of sizes means that the number of wasted bits will on average be less for shorter packets.
RVET.7.104	7.7.2 (p.41)	R	If the number of branches between updiscons is randomly distributed then the probability of generating packets with large branch counts will be lower, in which case increased waste for longer packets will have less overall impact.
RVET.7.105	7.7.2 (p.41)	R	Furthermore, the rate at which packets are generated can be higher for lower branch counts, and so reducing waste for this case will improve overall bandwidth at times where it is most important.
RVET.7.106	7.7.3 (p.41)	H	Format 1 **irreport** and **irdepth** fields
RVET.7.107	7.7.3 (p.41)	R	See Section 7.6.3.
RVET.7.108	7.8 (p.41)	H	Format 0 packets
RVET.7.109	7.8 (p.41)	R	This format is intended for optional efficiency extensions. Currently two extensions are defined, for reporting counts of correctly predicted branches, and for reporting the jump target cache index.
RVET.7.110	7.8 (p.41)	R	If branch prediction is supported and is enabled, then there is a choice of whether to output a full branch map (via format 1), or a count of correctly predicted branches.
RVET.7.111	7.8 (p.41)	R	The count format is used if the number of correctly predicted branches is at least 31. If there are 31 unreported branches (i.e. the branch map is full), but not all of them were predicted correctly, then the branch map will be output.

ID	REFERENCE	TYPE	DEFINITION
RVET.7.112	7.8 (p.41)	R	A branch count will be output under the following conditions: • A branch is mis-predicted. The count value will be the number of correctly predicted branches, minus 31. No address information is provided - it is implicitly that of the branch which failed prediction; • An updiscon, interrupt or exception requires the encoder to output an address. In this case the encoder will output the branch count (number of correctly predicted branches, minus 31); • The branch count reaches its maximum value. Strictly speaking an address isn't required for this case, but is included to avoid having to distinguish the packet format from the case above. It will occur so rarely that the bandwidth impact can be ignored.
RVET.7.113	7.8 (p.41)	R	If a jump target cache is supported and enabled, and the address to report following an updiscon is in the cache then the encoder can output the cache index using format 0, subformat 1.
RVET.7.114	7.8 (p.41)	R	However, the encoder may still choose to output the differential address using format 1 or 2 if the resulting packet is shorter. This may occur if the differential address is zero, or very small.
RVET.7.115	7.8 (p.41) Table 23	R	Packet format 0, subformat 0 - no address, branch count

Field Name	Bits	Description
format	2	00 (opt-ext): formats for optional efficiency extensions
subformat	*See Section 7.8.1*	0 (correctly predicted branches)
branch_count	32	Count of the number of correctly predicted branches, minus 31.
branch_fmt	2	00 (no-addr): Packet does not contain an **address**, and the branch following the last correct prediction failed. -11: (cannot occur for this format)

ID	REFERENCE	TYPE	DEFINITION
RVET.7.116	7.8 (p.42) Table 24	R	Packet format 0, subformat 0 - address, branch count

Field Name	Bits	Description
format	2	00 (opt-ext): formats for optional efficiency extensions
subformat	*See Section 7.8.1*	0 (correctly predicted branches)
branch_count	32	Count of the number of correctly predicted branches, minus 31.
branch_fmt	2	10 (addr): Packet contains an **address**. If this points to a branch instruction, then the branch was predicted correctly. (addr-fail): Packet contains an address that points to a branch which failed the prediction. ,01: (cannot occur for this format)
address	*iaddress_width_p* - *iaddress_lsb_p*	Differential instruction address.
notify	1	If the value of this bit is different from the MSB of address, it indicates that this packet is reporting an instruction that is not the target of an uninferable discontinuity because a notification was requested via **trigger[2]** (see Section 4.2.4).
updiscon	1	If the value of this bit is different from **notify**, it indicates that this packet is reporting the instruction following an uninferable discontinuity and is also the instruction before an exception, privilege change or resync (i.e. it will be followed immediately by a format 3 te_inst).
irreport	1	If the value of this bit is different from **updiscon**, it indicates that this packet is reporting an instruction that is either: following a return because its address differs from the predicted return address at the top of the implicit_return return address stack, or the last retired before an exception, interrupt, privilege change or resync because it is necessary to report the current address stack depth or nested call count
irdepth	*return_stack_size_p* + (*return_stack_size_p* > 0 ? 1 : 0) + *call_counter_size_p*	If the value of **irreport** is different from **updiscon**, this field indicates the number of entries on the return address stack (i.e. the entry number of the return that failed) or nested call count. If **irreport** is the same value as **updiscon**, all bits in this field will also be the same value as **updiscon**.

ID	REFERENCE	TYPE	DEFINITION
RVET.7.117	7.8 (p.42) Table 25	R	Packet format 0, subformat 1 - jump target index, branch map

Field Name	Bits	Description
format	2	00 (opt-ext): formats for optional efficiency extensions
subformat	*See Section 7.8.1*	1 (jump target cache)
index	*cache_size_p*	Jump target cache index of entry containing target address.
branches	5	Number of valid bits in **branch_map**. The length of **branch_map** is determined as follows: : (cannot occur for this format) : 1 bit -3: 3 bits -7: 7 bits -15: 15 bits -31: 31 bits For example if branches = 12, **branch_map** is 15 bits long, and the 12 LSBs are valid.
branch_map	Determined by **branches** field.	An array of bits indicating whether branches are taken or not. Bit 0 represents the oldest branch instruction executed. For each bit: : branch taken : branch not taken
irreport	1	If the value of this bit is different from branch_map[MSB], it indicates that this packet is reporting an instruction that is either: following a return because its address differs from the predicted return address at the top of the implicit_return return address stack, or the last retired before an exception, interrupt, privilege change or resync because it is necessary to report the current address stack depth or nested call count.
irdepth	*return_stack_size_p* + (*return_stack_size_p* > 0 ? 1 : 0) + *call_counter_size_p*	If the value of **irreport** is different from **branch_map[MSB]**, this field indicates the number of entries on the return address stack (i.e. the entry number of the return that failed) or nested call count. If **irreport** is the same value as **branch_map[MSB]**, all bits in this field will also be the same value as **branch_map[MSB]**.

ID	REFERENCE	TYPE	DEFINITION
RVET.7.118	7.8 (p.42) Table 26	R	Packet format 0, subformat 1 - jump target index, no branch map

Field Name	Bits	Description
format	2	00 (opt-ext): formats for optional efficiency extensions
subformat	*See Section 7.8.1*	1 (jump target cache)
index	*cache_size_p*	Jump target cache index of entry containing target address.
branches	5	Number of valid bits in **branch_map**. The length of branch_map is determined as follows: : no **branch_map** in packet -31: (cannot occur for this format)
irreport	1	If the value of this bit is different from **branches[MSB]**, it indicates that this packet is reporting an instruction that is either: following a return because its address differs from the predicted return address at the top of the implicit_return return address stack, or the last retired before an exception, interrupt, privilege change or resync because it is necessary to report the current address stack depth or nested call count.
irdepth	*return_stack_size_p* + (*return_stack_size_p* > 0 ? 1 : 0) + *call_counter_size_p*	If the value of **irreport** is different from **branches[MSB]**, this field indicates the number of entries on the return address stack (i.e. the entry number of the return that failed) or nested call count. If **irreport** is the same value as **branches[MSB]**, all bits in this field will also be the same value as **branches[MSB]**.

RVET.7.119	7.8.1 (p.43)	H	Format 0 subformat field
RVET.7.120	7.8.1 (p.43)	R	The width of this field depends on the number of optional formats supported.
RVET.7.121	7.8.1 (p.43)	R	Currently, two optional formats are defined (correctly predicted branches and jump target cache).
RVET.7.122	7.8.1 (p.43)	R	The width is specified by the *f0s_width* discovery field (see Section 10.1).
RVET.7.123	7.8.1 (p.43)	R	If multiple optional formats are supported, the field width must be non-zero.
RVET.7.124	7.8.1 (p.43)	R	However, if only one optional format is supported, the field can be omitted, and the value of the field inferred from the options field in the support packet (see Section 7.5).
RVET.7.125	7.8.1 (p.43)	R	This provision allows additional formats to be added in future without reducing the efficiency of the existing formats.
RVET.7.126	7.8.2 (p.43)	H	Format 0 **branch_fmt** field
RVET.7.127	7.8.2 (p.43)	R	This is encoded so that when no address is required it will be zero, allowing the upper bits of the **branch_count** field to be compressed away.
RVET.7.128	7.8.2 (p.43)	R	When a branch count is reported without an address it is because a branch has failed the prediction.

ID	REFERENCE	TYPE	DEFINITION
RVET.7.129	7.8.2 (p.43)	R	However, when an address is reported along with a branch count, it will be because the packet was initiated by an uninferable discontinuity, an exception, or because a branch has been encountered that increments **branch_count** to `0xffff_ffff`.
RVET.7.130	7.8.2 (p.43)	R	For the latter case, the reported address will always be for a branch, and in the former cases it may be.
RVET.7.131	7.8.2 (p.43)	R	If it is a branch, it is necessary to be explicit about whether or not the prediction was met or not.
RVET.7.132	7.8.2 (p.43)	R	If it is met, then the reported address is that of the last correctly predicted branch.
RVET.7.133	7.8.3 (p.43)	H	Format 0 **irreport** and **irdepth** fields
RVET.7.134	7.8.3 (p.43)	R	These bits are encoded so that most of the time they will take the same value as the immediately preceding bit (**updiscon**, **branch_map[MSB]** or **branches[MSB]** depending on the specific packet format).
RVET.7.135	7.8.3 (p.43)	I	Purpose and behavior is as described in Section 7.6.3.
RVET.7.136	7.8.3 (p.43)	R	For the jump target cache (subformat 1), they are included to allow return addresses that fail the implicit return prediction but which reside in the jump target cache to be reported using this format.
RVET.7.137	7.8.3 (p.43)	R	An implementation could omit these if all implicit return failures are reported using format 1.

CHAPTER 8 Data Trace Encoder Output Packets

ID	REFERENCE	TYPE	DEFINITION
RVET.8.1	8.0 (p.45)	H	Data Trace Encoder Output Packets
RVET.8.2	8.0 (p.45)	R	Data trace packets must be differentiated from instruction trace packets, and the means by which this is accomplished is dependent on the trace transport infrastructure.
RVET.8.3	8.0 (p.45)	I	Several possibilities exist: One option is for instruction and data trace to be issued using different IDs (for example, if using ATB transport, different ATID values).
RVET.8.4	8.0 (p.45)	I	Alternatively, an additional field as part of the packet encapsulation can be used (Siemens uses a 2-bit **msg_type** field to differentiate different trace types from the same source).
RVET.8.5	8.0 (p.45)	R	By default, all data trace packets include both address and data.
RVET.8.6	8.0 (p.45)	R	However, provision is made for runtime configuration options to exclude either the address or the data, in order to minimize trace bandwidth.
RVET.8.7	8.0 (p.45)	I	For example, if filtering has been configured to only trace from a specific data access address there is no need to report the address in the trace.
RVET.8.8	8.0 (p.45)	R	Alternatively, the user may want to know which locations are accessed but not care about the data value. Information about whether address or data are omitted is not encoded in the packets themselves as it does not change dynamically, and to do so would reduce encoding efficiency.
RVET.8.9	8.0 (p.45)	R	The run-time configuration should be reported in the Format 3, subformat 3 support packet (see Section 7.5). The following sections include examples for all three cases.
RVET.8.10	8.0 (p.45)	R	As outlined in Section 4.3, two different signaling protocols between the RISC-V hart and the encoder are supported: *unified* and *split*.
RVET.8.11	8.0 (p.45)	R	Accordingly, both *unified* and *split* trace packets are defined.
RVET.8.12	8.0 (p.45)	C	In the following tables, "clog2" is an abbreviation for "ceiling of log2".
RVET.8.13	8.1 (p.45)	H	Load and Store
RVET.8.14	8.1.1 (p.45)	H	format field
RVET.8.15	8.1.1 (p.45)	R	Types of data trace packets are differentiated by the format field.
RVET.8.16	8.1.1 (p.45)	R	This field is 2 bits wide if only unified loads and stores are supported, or 3 bits otherwise.
RVET.8.17	8.1.1 (p.45)	R	Unified loads and split load request phase share the same code because the encoder will support one or the other, indicated by a discoverable parameter.
RVET.8.18	8.1.1 (p.45)	R	Data accesses aligned to their size (e.g. 32-bit loads aligned to 32-bit word boundaries) are expected to be commonplace, and in such cases, encoding efficiency can be improved by not reporting the redundant LSBs of the address.

ID	REFERENCE	TYPE	DEFINITION
RVET.8.19	8.1.1 (p.45) Table 27	R	Packet format for Unified load or store, with address and data

Field Name	Bits	Description
format	2 or 3	Transaction type: 000: Unified load or split load address, aligned 001: Unified load or split load address, unaligned 010: Store, aligned address 011: Store, unaligned address (other codes select other packet formats)
size	*Max(1,clog2(clog2(data_width_p/8 + 1)))*	Transfer size is $2^{\textbf{size}}$ bytes
diff	*2*	00: Full address and data (sync) 01: Differential address, XOR-compressed data 10: Differential address, full data 11: Differentail address, differential data
data_len	**size**	Number of bytes of data is **data_len** + 1
data	8 * (**data_len** + 1)	Data
address	*daddress_width_p*	Byte address if format is unaligned, otherwise shift left by **size** to recover byte address

RVET.8.20	8.1.1 (p.46) Table 28	R	Packet format for Unified load or store, with address only

Field Name	Bits	Description
format	2 or 3	Transaction type 000: Unified load or split load address, aligned 001: Unified load or split load address, unaligned 010: Store, aligned address 011: Store, unaligned address (other codes select other packet formats)
size	*max(1,clog2(clog2(data_width_p/8 + 1)))*	Transfer size is $2^{\textbf{size}}$ bytes
diff	*1*	0: Full address (sync) 1: Differential address
address	*daddress_width_p*	Byte address if format is unaligned, otherwise shift left by **size** to recover byte address

ID	REFERENCE	TYPE	DEFINITION
RVET.8.21	8.1.1 (p.46) Table 29	R	Packet format for Unified load or store, with data only

Field Name	Bits	Description
format	2 or 3	Transaction type 000: Unified load or split load address, aligned 001: Unified load or split load address, unaligned 010: Store, aligned address 011: Store, unaligned address (other codes select other packet formats)
size	*max(1,clog2(clog2(data_width_p/8 + 1)))*	Transfer size is 2^{size} bytes
diff	*1 or 2*	00: Full data (sync) 01: Compressed data (XOR if 2 bits) 10: reserved 11 : Differential data
data	*data_width_p*	Data

ID	REFERENCE	TYPE	DEFINITION
RVET.8.22	8.1.1 (p.46) Table 30	R	Packet format for Split load - Address only

Field Name	Bits	Description
format	3	Transaction type 000: Unified load or split load address, aligned 001: Unified load or split load address, unaligned (other codes select other packet formats)
size	*max(1,clog2(clog2(data_width_p/8 + 1)))*	Transfer size is 2^{size} bytes
lrid	*lrid_width_p*	Load request ID
diff	*1*	0: Full address (sync) 1: Differential address
address	*daddress_width_p*	Byte address if format is unaligned, otherwise shift left by **size** to recover byte address

ID	REFERENCE	TYPE	DEFINITION
RVET.8.23	8.1.1 (p.46) Table 31	R	Packet format for Split load - Data only

Field Name	Bits	Description
format	3	Transaction type 100: split load data (other codes select other packet formats)
size	*max(1,clog2(clog2(data_width_p/8 + 1)))*	Transfer size is 2^{size} bytes
lrid	*lrid_width_p*	Load request ID
resp	*2*	00: Error (no data) 01: XOR-compressed data 10: Full data 11: Differential data
data	*data_width_p*	Data

ID	REFERENCE	TYPE	DEFINITION
RVET.8.24	8.1.2 (p.47)	H	size field
RVET.8.25	8.1.2 (p.47)	R	The width of this field is 2 bits if max size is 64-bits (*data_width_p* < 128), 3 bits if wider.

ID	REFERENCE	TYPE	DEFINITION
RVET.8.26	8.1.3 (p.47)	H	diff field
RVET.8.27	8.1.3 (p.47)	I	Unlike instruction trace, compression options for data trace are somewhat limited.
RVET.8.28	8.1.3 (p.47)	R	Following a synchronization instruction trace packet, the first data trace packet for a given access size must include the full (unencoded) data access address.
RVET.8.29	8.1.3 (p.47)	R	Thereafter, the address may be reported differentially (i.e. address of this data access, minus the address of the previous data access of the same size).
RVET.8.30	8.1.3 (p.47)	R	Similarly, following a synchronization instruction trace packet, the first data trace packet for a given access size must include the full (unencoded) data value.
RVET.8.31	8.1.3 (p.47)	R	Beyond this, data may be encoded or unencoded depending on whichever results in the most efficient representation.
RVET.8.32	8.1.3 (p.47)	O	Implementors may chose to offer one of XOR or differential compression, or both. XOR compression will be simpler to implement, and avoids the need for performing subtraction of large values.
RVET.8.33	8.1.3 (p.47)	R	If only one data compression type is offered, the **diff** field can be 1 bit wide rather than 2 for Table 29.
RVET.8.34	8.1.4 (p.47)	H	data_len field
RVET.8.35	8.1.4 (p.47)	R	However the data is compressed, upper bytes that are all the same value do not need to be included in the packet; the decoder can recreate the full-width value by sign extending from the most significant received bit.
RVET.8.36	8.1.4 (p.47)	R	In cases where data is not the final field in the packet, the width of data is indicated by this field.
RVET.8.37	8.2 (p.47)	H	Atomic
RVET.8.38	8.2.1 (p.47)	H	size field
RVET.8.39	8.2.1 (p.47)	R	Strictly, **size** could be just one bit as atomics are currently either 32 or 64 bits.
RVET.8.40	8.2.1 (p.47)	R	Defining as per regular loads and stores provisions for future extensions (proprietary or otherwise) that support smaller atomics.

ID	REFERENCE	TYPE	DEFINITION
RVET.8.41	8.2.1 (p.47) Table 32	R	Packet format for Unified atomic with address and data

Field Name	Bits	Description
format	3	Transaction type 110: Unified atomic or split atomic address (other codes other packet formats)
subtype	3	Atomic sub-type 000: Swap 001: ADD 010: AND 011: OR 100: XOR 101: MAX 110: MIN 111: reserved
size	*max(1,clog2(clog2(data_width_p/8 + 1)))*	Transfer size is $2^{\textbf{size}}$ bytes
diff	*2*	00: Full address and data (sync) 01: Differential address, XOR-compressed data 10: Differential address, full data 11: Differential address, differential data
op_len	**size**	Number of bytes of operand is **op_len** + 1
operand	8 * (**op_len** + 1)	Operand. Value from rs2 before operator applied
data_len	**size**	Number of bytes of data is **data_len** + 1
data	8 * (**data_len** + 1)	Data
address	*daddress_width_p*	Address, aligned and encoded as per size

RVET.8.42	8.2.1 (p.48) Table 33	R	Packet format for Unified atomic with address only

Field Name	Bits	Description
format	3	Transaction type 110: Unified atomic or split atomic address (other codes other packet formats)
subtype	3	Atomic sub-type 000: Swap 001: ADD 010: AND 011: OR 100: XOR 101: MAX 110: MIN 111: conditional store failure
size	*max(1,clog2(clog2(data_width_p/8 + 1)))*	Transfer size is $2^{\textbf{size}}$ bytes
address	*daddress_width_p*	Address, aligned and encoded as per size

ID	REFERENCE	TYPE	DEFINITION
RVET.8.43	8.2.1 (p.47) Table 34	R	Packet format for Unified atomic with data only

Field Name	Bits	Description
format	3	Transaction type 110: Unified atomic or split atomic address (other codes other packet formats)
subtype	3	Atomic sub-type 000: Swap 001: ADD 010: AND 011: OR 100: XOR 101: MAX 110: MIN 111: reserved
size	*max(1,clog2(clog2(data_width_p/8 + 1)))*	Transfer size is 2^{size} bytes
diff	*1 or 2*	00: Full data (sync) 01: Compressed data (XOR if 2 bits) 10: reserved 11: Differential data
op_len	**size**	Number of bytes of operand is **op_len** + 1
operand	8 * (**op_len** + 1)	Operand. Value from rs2 before operator applied
data	*data_width_p*	Data

RVET.8.44	8.2.2 (p.48)	H	diff field
RVET.8.45	8.2.2 (p.48)	I	See Section 8.1.3.
RVET.8.46	8.2.3 (p.48)	H	operand field
RVET.8.47	8.2.3 (p.48)	R	The operand value for the atomic operation.
RVET.8.48	8.2.3 (p.48)	R	Uncompressed, although upper bytes that are all the same value do not need to be included in the packet; the decoder can recreate the full-width value by sign extending from the most significant received bit; see Section 8.2.4.
RVET.8.49	8.2.4 (p.48)	H	data_len and op_len fields
RVET.8.50	8.2.4 (p.48)	R	Width of data and *operand fields respectively. See Section 8.1.4.

ID	REFERENCE	TYPE	DEFINITION
RVET.8.51	8.2.4 (p.49) Table 35	R	Packet format for Split atomic with operand only

Field Name	Bits	Description
format	3	Transaction type 110: Unified atomic or split atomic address (other codes other packet formats)
subtype	3	Atomic sub-type 000: Swap 001: ADD 010: AND 011: OR 100: XOR 101: MAX 110: MIN 111: reserved
size	*max(1,clog2(clog2(data_width_p/8 + 1)))*	Transfer size is $2^{\textbf{size}}$ bytes
lrid	*lrid_width_p*	Load request ID
diff	*1 or 2*	00: Full address and data (sync) 01: Differential address, XOR-compressed data 10: Differential address, full data 11: Differential address, differential data
op_len	**size**	Number of bytes of operand is **op_len** + 1
operand	8 * (**op_len** + 1)	Operand. Value from rs2 before operator applied
address	*daddress_width_p*	Address, aligned and encoded as per size

RVET.8.52	8.2.4 (p.49) Table 36	R	Packet format for Split atomic load data only

Field Name	Bits	Description
format	3	Transaction type 110: Split atomic data other codes other packet formats
lrid	*lrid_width_p*	Load request ID
resp	*2*	00: Error (no data) 01: XOR-compressed data 10: full data 11: differential data
data_len	**size**	Number of bytes of operand is *data_len* + 1. Not included if resp indicates an error (sign-extend **resp** MSB)
data	8 * (**data_len** + 1)	Data. Not included if resp indicates an error (sign-extend **resp** MSB)

RVET.8.53	8.3 (p.49)	H	CSR

ID	REFERENCE	TYPE	DEFINITION
RVET.8.54	8.3 (p.49) Table 37	R	Packet format for Unified CSR, with address, data and operand

Field Name	Bits	Description
format	3	Transaction type 101: CSR (other codes other packet formats)
subtype	2	CSR sub-type 00: RW 01: RS 10: RC 11: reserved
diff	*1 or 2*	00: Full data (sync) 01: Compressed data (XOR if 2 bits) 10: reserved 11 : Differential data
data_len	*2 or 3*	Number of bytes of data is **data_len** + 1
data	8 * (**data_len** + 1)	Data
addr_msbs	**6**	Address[11:6]
op_len	**2 or 3**	Number of bytes of operand is **op_len** + 1
operand	8 * (**op_len** + 1)	Operand. Value from rs1 before operator applied
addr_lsbs	6	Address[5:0]

RVET.8.55	8.3.1 (p.50)	H	Diff field
RVET.8.56	8.3.1 (p.50)	I	See Section 8.1.3.
RVET.8.57	8.3.2 (p.50)	H	Operand field
RVET.8.58	8.3.2 (p.50)	I	See Section 8.2.3.
RVET.8.59	8.3.3 (p.50)	H	data_len and op_len fields
RVET.8.60	8.3.3 (p.50)	R	2 bits wide if hart has 32-bit CSRs, 3 bits if 64-bit. Width of data and operand fields respectively.
RVET.8.61	8.3.3 (p.50)	I	See Section 8.1.4.
RVET.8.62	8.3.4 (p.50)	H	addr fields
RVET.8.63	8.3.4 (p.50)	R	The address is split into two parts, with the 6 LSBs output last as these are more likely to compress away.

ID	REFERENCE	TYPE	DEFINITION
RVET.8.64	8.3.4 (p.50) Table 38	R	Packet format for Unified CSR, with address and read-only data (as determined by addr[11:10] = 11)

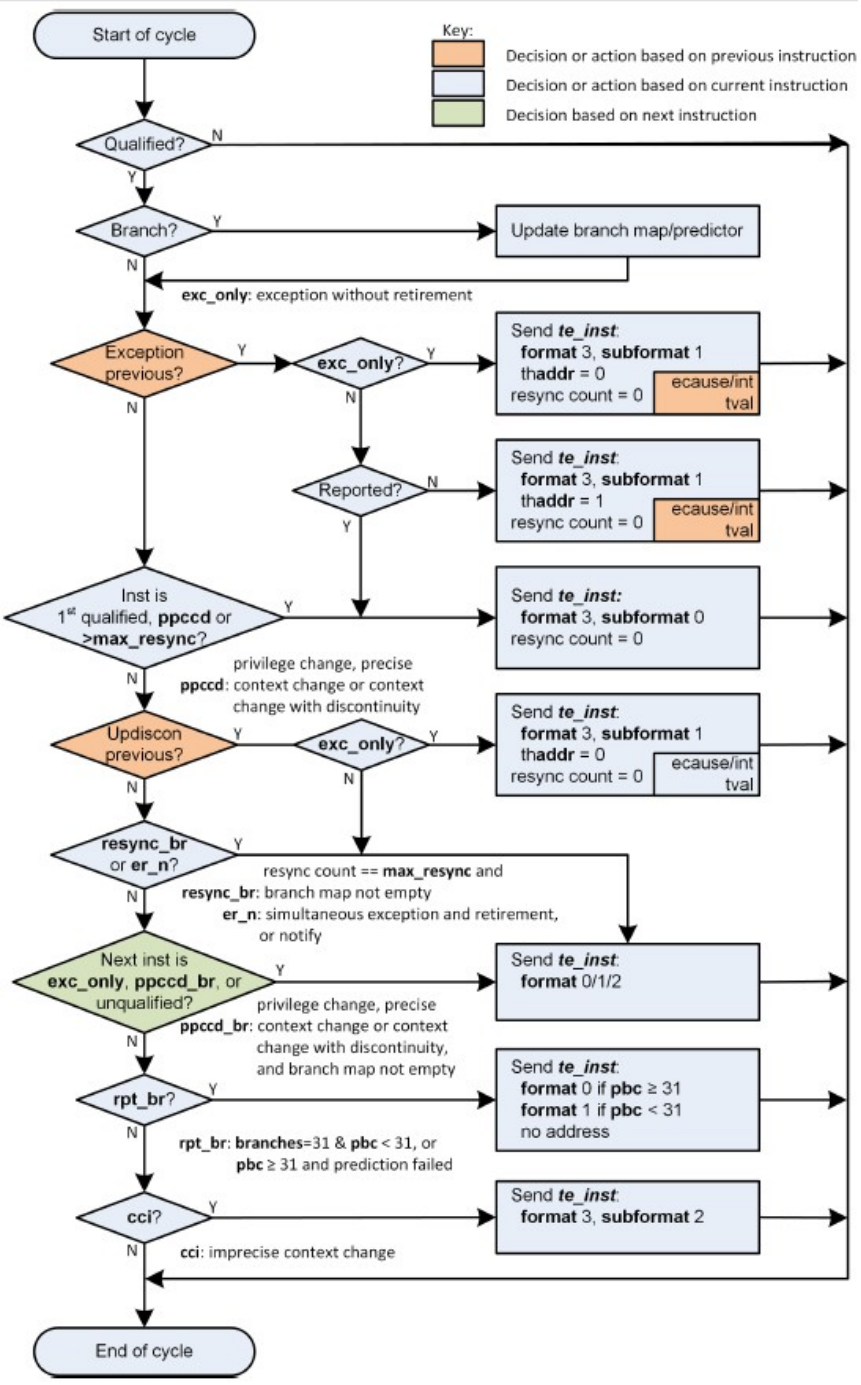
Field Name	Bits	Description
format	3	Transaction type 101: CSR (other codes other packet formats)
subtype	2	CSR sub-type 00: RW 01: RS 10: RC 11: reserved
diff	*1 or 2*	00: Full data (sync) 01: Compressed data (XOR if 2 bits) 10: reserved 11 : Differential data
data_len	*2 or 3*	Number of bytes of data is **data_len** + 1
data	8 * (**data_len** + 1)	Data
addr_msbs	**6**	Address[11:6]
addr_lsbs	6	Address[5:0]

RVET.8.65	8.3.4 (p.50) Table 39	R	Packet format for Unified CSR, with address only

Field Name	Bits	Description
format	3	Transaction type 101: CSR (other codes other packet formats)
subtype	3	CSR sub-type 00: RW 01: RS 10: RC 11: reserved
diff	*0 or 1*	0: Full address 1: Differential address
addr_msbs	**6**	Address[11:6]
addr_lsbs	6	Address[5:0]

CHAPTER 9 Reference Compressed Branch Trace Algorithm

ID	REFERENCE	TYPE	DEFINITION
RVET.9.1	9.0 (p.51)	H	Reference Compressed Branch Trace Algorithm
RVET.9.2	9.0 (p.51)	I	The contents of this chapter are informative only.
RVET.9.3	9.0 (p.51, p.53) Figure 2	I	A reference algorithm for compressed branch trace is given in the figure below. Instruction delta trace algorithm



ID	REFERENCE	TYPE	DEFINITION
RVET.9.4	9.0 (p.51)	I	In the diagram, the following terms are used: • *te_inst*. The name of the packet type emitted by the encoder (see Chapter 7); • *inst*. Abbreviation for 'instruction'; • *exception. Exception or interrupt signalled*; • *updiscon*. Uninferable PC discontinuity. This identifies an instruction that causes the program counter to be changed by an amount that cannot be predicted from the source code alone (**itype** values 8, 10, 12 or 14); • *Qualified*? An instruction that meets the filtering criteria is qualified, and will be traced; • *Branch*? Is the instruction a branch or not (**itype** values 4 or 5); • *branch map*. A vector where each bit represents the outcome of a branch. A 0 indicates the branch was taken, a 1 indicates that it was not; • *ppccd*. Privilege has changed, or context has changed and needs to be reported precisely or treated as an uninferable PC discontinuity (see Table 9); • *ppccd_br*. As above, but branch map not empty; • *er_n*. Instruction retirement and exception signalled on the same cycle, or Trace notify trigger (see Table 12); • *exc_only*. Exception or interrupt signalled without simultaneous retirement; • *cci*. context change that can be reported imprecisely (see Table 9); • *rpt_br*. Report branches due to full branch map or misprediction; • *branches*. The number of branches encountered but not yet reported to the decoder; • *pbc*. Correctly predicted branches count (always zero if branch predictor disabled or not present); • *Reported*? "Exception previous" reported with thaddr = 0 on the cycle it occured because it was preceded by an updiscon or immediately followed by another exception; • *resync count*. A counter used to keep track of when it is necessary to send a synchronization packet (see Section 9.2); • *max_resync*. The resync counter value that schedules a synchronization packet (see Section 9.2); • *resync_br*. The resync counter has reached the maximum value and there are entries in the branch map that have not yet been output (see Section 9.2).
RVET.9.5	9.0 (p.51)	I	Figure 2 shows instruction by instruction behavior, as would be seen in a single-retirement system only.
RVET.9.6	9.0 (p.51)	I	Whilst the core to encoder interface allows the RISC-V hart to provide information on multiple retiring instructions simultaneously, the resultant packet sequence generated by the encoder must be the same as if retiring one instruction at a time.
RVET.9.7	9.0 (p.51)	I	A 3-stage pipeline within the encoder is assumed, such that the encoder has visibility of the current, previous and next instructions.
RVET.9.8	9.0 (p.52)	I	All packets are generated using information relating to the current instruction. The orange diamonds indicate decisions based on the previous instruction, the green diamond indicates a decision based on the next instruction, and all other diamonds are based on the current instruction.
RVET.9.9	9.0 (p.52)	I	Additionally, the encoder can generate one further packet type, not shown on the diagram for clarity.

ID	REFERENCE	TYPE	DEFINITION
RVET.9.10	9.0 (p.52)	I	The support packet (format 3, subformat 3 - see Section 7.5) is sent when: • The encoder is enabled or disabled, or its configuration is changed, to inform the decoder of the operating mode of the encoder; • After the final qualified instruction has been traced, to inform the decoder that tracing has stopped; • If trace packets are lost (for example if the buffer into which packets are being written fills up), in this situation, the 1st packet loaded into the buffer when space next becomes available must be a support packet. Following this, tracing will resume with a sync packet.
RVET.9.11	9.0 (p.52)	I	Note: if the halted or reset sideband signals are asserted (see Table 10) the encoder will behave as if it has received an unqualified instruction (output te_inst reporting the address of the previous instruction, followed by te_support);
RVET.9.12	9.1 (p.53)	H	Format selection
RVET.9.13	9.1 (p.53)	I	In all cases but two, the packet format is determined only by a 'yes' outcome from the associated decision.
RVET.9.14	9.1 (p.54)	I	When reporting branch information on its own (without an address), the choice between format 1 and format 0, subformat 0 depends on the number of correctly predicted branches (this will be 0 if the predictor is not supported, or is disabled).
RVET.9.15	9.1 (p.54)	I	No packets are generated until there are at least 31 branches to report. Format 1 is used if the outcome of at least one of those 31 branches was not predicted correctly.
RVET.9.16	9.1 (p.54)	I	If all were predicted correctly, nothing is output at this time, and the encoder continues to count correctly predicted branch outcomes. As soon as one of the branch outcomes is not correctly predicted, the encoder will output a format 0, subformat 0 packet. See also Section 7.8.
RVET.9.17	9.1 (p.54)	I	The choice between formats for the "format 0/1/2" case in the middle of the diagram also needs further explanation. • If the number of correctly predicted branches is 31 or more, then format 0, subformat 0 is always used; • Else, if the jump target cache is supported and enabled, and the address being reported is in the cache, then normally format 0, subformat 1 will be used, reporting the cache index associated with the address. This will include branch information if there are any branches to report. However, the encoder may chose to output the equivalent format 1 or 2 packet (containing the differential address, with or without branch information) if that will result in a shorter packet (see Section 7.8); • Else, if there are branches to report, format 1 is used, otherwise format 2.
RVET.9.18	9.1 (p.54)	I	Packet formats 0, 1 and 2 are organized so that the address is usually the final field.
RVET.9.19	9.1 (p.54)	I	Minimizing the number of bits required to represent the address reduces the total packet size and significantly improves efficiency. See Chapter 7.
RVET.9.20	9.2 (p.54)	H	Resynchronisation

ID	REFERENCE	TYPE	DEFINITION
RVET.9.21	9.2 (p.54)	I	Per Section 3.1.5, a format 3 synchronisation packet must be output after "a prolonged period of time". The exact mechanism for determining this is not specified, but options might be to count the number of *te_inst* packets emitted, or the number of clock cycles elapsed, since the previous synchronization message was sent.
RVET.9.22	9.2 (p.54)	I	When the resync is required, the primary objective is to output a format 3 packet, so that the decoder can start tracing from that point without needing any of the history.
RVET.9.23	9.2 (p.54)	I	However, if the decoder is already synced, then it is also required that it can continue to follow the execution path up to and through the format 3 packet seamlessly.
RVET.9.24	9.2 (p.54)	I	As such, before outputting a format 3 packet, it is necessary to output a format 1 packet for the preceding instruction if there are any unreported branches (because format 3 does not contain a branch map).
RVET.9.25	9.2 (p.54)	I	The format 3 will be sent if the resync timer has been exceeded. On the cycle before this (when the resync timer value has been exactly reached), a format 1 will be generated if the branch map is not empty.
RVET.9.26	9.3 (p.54)	H	Multiple retirement considerations
RVET.9.27	9.3 (p.54)	I	As noted earlier in this section, for a single-retirement system the reference algorithm is applied to each retired instruction.
RVET.9.28	9.3 (p.54)	I	When instructions are retired in blocks, only the first and last instruction in a block need be considered, as all those in between are "uninteresting", and will have no effect on the encoder’s state (their route through Figure 2 does not pass through any of the rectangular boxes).
RVET.9.29	9.3 (p.54, p.55)	I	In most cases, either the first or last instruction of a block (but not both) is interesting, meaning that the encoder does not need to generate more than one packet from a block.
RVET.9.30	9.3 (p.55)	I	However, there are a few cases where this is not true, and it is possible that the encoder will need to generate two packets from the same block.
RVET.9.31	9.3 (p.55)	I	For example, the first instruction in a block must generate a packet if it is the first traced instruction. However, if the block also indicates an exception or interrupt (itype= 1 or 2), then the last instruction in the block must also generate a packet.
RVET.9.32	9.3 (p.55)	I	As generating multiple packets per cycle would significatly complicate the encoder, and as situations such as this will only occur infrequently, some elastic buffering in the encoder is the preferred approach. This will allow subsequent blocks to be queued whilst the encoder generates two successive packets from a block. The encoder can drain the elastic buffer any time there is a cycle when the hart doesn’t report anything, or if there is a block with itype = 0 (which is uninteresting to the encoder).
RVET.9.33	9.3 (p.55)	I	There are pathological cases where consecutive blocks could require packets to be generated from both first and last instructions, but elastic buffering is only required if the blocks are also input on consecutive cycles.

ID	REFERENCE	TYPE	DEFINITION
RVET.9.34	9.3 (p.55)	I	In practice there are very few cases where this can occur. The worst so far identified case is a variation on the example above, where the exception is an ecall, and that in turn encounters some other form of exception or interrupt in the first few instructions of the trap handler: • Block 1: itype = 1 (ecall), iretires > 1. Generate packet from first instruction (first traced), and last instruction (last before ecall); • Block 2: itype = 1 or 2 (some other exception or interrupt), iretires > 0. Generate packet from first instruction (ecall trap handler), and last instruction (last before other exception or interrupt); • Block 3: Generate packet from first instruction (other exception or interrupt trap handler)
RVET.9.35	9.3 (p.55)	I	Because the ecall is known to the hart’s fetch unit and can be predicted, it may be possible for block 2 to occur the cycle after block 1. However, it is reasonable to assume that the other exception or interrupt will not be predictable, and as a result there will be several cycles between blocks 2 and 3, which will allow the encoder to 'catch up'.
RVET.9.36	9.3 (p.55)	I	It is recommended that encoders implement sufficient elastic buffering to handle this case, and if for some reason the elastic buffer overflows, it should issue a support packet indicating trace lost.

CHAPTER 10 Parameters and Discovery

ID	REFERENCE	TYPE	DEFINITION
RVET.10.1	10.0 (p.57)	H	Parameters and Discovery
RVET.10.2	10.0 (p.57)	I	This document defines a number of parameters for describing aspects of the encoder such as the widths of buses, the presence or absence of optional features and the size of resources, as listed in Table 40 and Table 41.
RVET.10.3	10.0 (p.57)	I	Depending on the implementation, some parameters may be inherently fixed whilst others may be passed in to the design by some means.
RVET.10.4	10.0 (p.57) Table 40	R	Parameters to the encoder - instruction trace

Parameter Name	Range	Description
arch_p		The architecture specification version with which the encoder is compliant (0 for initial version).
blocks_p		Number of times **iretire**, **itype** etc. are replicated
bpred_size_p		Number of entries in the branch predictor is $2^{bpred\_size\_p}$. Minimum number of entries is 2, so a value of 0 indicates that there is no branch predictor implemented.
cache_size_p		Number of entries in the jump target cache is $2^{cache\_size\_p}$. Minimum number of entries is 2, so a value of 0 indicates that there is no jump target cache implemented.
call_counter_size_p		Number of bits in the nested call counter is $2^{call\_counter\_size\_p}$. Minimum number of entries is 2, so a value of 0 indicates that there is no implicit return call counter implemented.
ctype_width_p		Width of the **ctype** bus
context_width_p		Width of **context** bus
time_width_p		Width of **time** bus
ecause_width_p		Width of **exception cause** bus
ecause_choice_p		Number of bits of exception cause to match using multiple choice
f0s_width_p		Width of the **subformat** field in format 0 te_inst packets (see Section 7.8.1).
filter_context_p	0 or 1	Filtering on context supported when 1
filter_time_p	0 or 1	Filtering on time supported when 1
filter_excint_p		Filtering on exception cause or interrupt supported when non_zero. Number of nested exceptions supported is $2^{filter\_excint\_p}$
filter_privilege_p	0 or 1	Filtering on privilege supported when 1
filter_tval_p	0 or 1	Filtering on trap value supported when 1 (provided *filter_excint_p* is non-zero)
iaddress_lsb_p		LSB of instruction **address** bus to trace. 1 is compressed instructions are supported, 2 otherwise
iaddress_width_p		Width of instruction **address** bus. This is the same as DXLEN

ID	REFERENCE	TYPE	DEFINITION

iretire_width_p		Width of the **iretire** bus
ilastsize_width_p		Width of the **ilastsize** bus
itype_width_p		Width of the **itype** bus
nocontext_p	0 or 1	Exclude context from te_inst packets if 1
notime_p	0 or 1	Exclude time from te_inst packets if 1
privilege_width_p		Width of **privilege** bus
retires_p		Maximum number of instructions that can be retired per block
return_stack_size_p		Number of entries in the return address stack is $2^{return\_stack\_size\_p}$. Minimum number of entries is 2, so a value of 0 indicates that there is no implicit return stack implemented.
sijump_p	0 or 1	**sijump** is used to identify sequentially inferable jumps
impdef_width_p		Width of **implementation-defined input** bus

RVET.10.5 10.0 (p.57) Table 41 — R — Parameters to the encoder - data trace

Parameter Name	Range	Description
daddress_width_p		Width of the **daddress** bus
dblock_width_p		Width of the **dblock** bus
data_width_p		Width of the **data** bus
dsize_width_p		Width of the **dsize** bus
dtype_width_p		Width of the **dtype** bus
iaddr_lsbs_width_p		Width of the **iaddr_lsbs** bus
lrid_width_p		Width of the **lrid** bus
lresp_width_p		Width of the **lresp** bus
ldata_width_p		Width of the **ldata** bus
sdata_width_p		Width of the **sdata** bus

RVET.10.6 10.1 (p.58) — H — Discovery of encoder parameters

RVET.10.7 10.1 (p.58) — R — To operate correctly, the decoder must be able to determine some of the encoder's parameters at runtime, in the form of discoverable attributes.

RVET.10.8 10.1 (p.58) — R — These parameters must be discoverable by the decoder, or else be fixed at the default value (in other words, if an encoder does not make a particular parameter discoverable, it must implement only the default value of that parameter, which the decoder will also use).

RVET.10.9 10.1 (p.58) — R — To access the discoverable attributes, some external entity, for example a debugger or a supervisory hart, must request it from the encoder.

RVET.10.10 10.1 (p.58) — R — The encoder will provide the discovery information in one or more different formats.

RVET.10.11 10.1 (p.58) — R — The preferred format is a packet which is sent over the trace infrastructure.

RVET.10.12 10.1 (p.58) — R — Another format would be allowing the external entity to read the values from some register or memory mapped space maintained by the encoder.

RVET.10.13 10.1 (p.58) — I — Section 10.2 gives an example of how this may be accomplished.

ID	REFERENCE	TYPE	DEFINITION
RVET.10.14	10.1 (p.58) Table 42	R	Table 42 lists the required discoverable attributes for instruction trace. Required instruction trace attributes

Name	Default	Parameter mapping
arch	0	*arch_p*
bpred_size	0	*bpred_size_p*
cache_size	*0*	*cache_size_p*
call_counter_size	*0*	*call_counter_size_p*
context_width	0	*context_width_p - 1*
time_width	**0**	*time_width_P - 1*
ecause_width	3	*ecause_width_P - 1*
f0s_width	0	*f0s_width_p*
iaddress_lsb	0	*iaddress_lsb_p - 1*
iaddress_width	31	*iaddress_width_p - 1*
nocontext	1	*nocontext*
notime	1	*notime*
privilege_width	1	*privilege_width_p - 1*
return_stack_size	0	*return_stack_size_p*
sijump	0	*sijump_p*

RVET.10.15	10.1 (p.58)	R	For ease of use it is further recommended that all of the encoder's parameters be mapped to discoverable attributes, even if not directly required by the decoder.
RVET.10.16	10.1 (p.58)	R	In particular, attributes related to filtering capabilities. Table 43 lists the attributes associated with the filtering recommendations discussed in Chapter 5, Table 44 lists attributes related to other instruction trace parameters mentioned in this document, and Table 45 lists attributes related to data trace.
RVET.10.17	10.1 (p.58) Table 43	R	Optional filtering attributes

Name	Default	Parameter mapping
comparators	0	*comparators_p - 1*
filters	0	*filters_p - 1*
ecause_choice	*5*	*ecause_choice_p*
filter_context	*1*	*filter_context_p*
filter_time	1	*filter_time_p*
filter_excint	**1**	*filter_excint_p*
filter_privilege	1	*filter_privilege_p*
filter_tval	1	*filter_tval_p*

RVET.10.18	10.1 (p.59) Table 44	R	Other recommended attributes

Name	Default	Description
ctype_width	0	*ctype_width_p - 1*
ilastsize_width	0	*ilastsize_width_p - 1*
itype_width	*3*	*itype_width_p - 1*
iretire_width	*1*	*iretire_width_p - 1*
retires	0	*retires_p - 1*
impdef_width	**0**	*impdef_width_p - 1*

ID	REFERENCE	TYPE	DEFINITION
RVET.10.19	10.1 (p.59) Table 45	R	Data trace attributes

Name	Default	Description
daddress_width	31	*daddress_width_p - 1*
dblock_width	0	*dblock_width_p - 1*
data_width	*31*	*data_width_p - 1*
dsize_width	*2*	*dsize_width_p - 1*
dtype_width	0	*dtype_width_p - 1*
iaddr_lsbs_width	**0**	*iaddr_lsbs_width_p - 1*
lrid_width	0	*lrid_width_p - 1*
lresp_width	0	*lresp_width_p - 1*
ldata_width	31	*ldata_width_p - 1*
sdata_width	31	*sdata_width_p - 1*

RVET.10.20	10.2 (p.59)	H	Example ipxact description
RVET.10.21	10.2 (p.59)	I	This section provides an example of discovery information represented in the ipxact form.
RVET.10.22	10.2 (p.59-p.62)	I	See original text pages 59 to 62.

CHAPTER 11 Decoder

ID	REFERENCE	TYPE	DEFINITION
RVET.11.1	11.0 (p.63)	H	Decoder
RVET.11.2	11.0 (p.63)	I	This decoder implementation assumes there is no branch predictor or return address stack (*return_stack_size_p* and *bpred_size_p* both zero).
RVET.11.3	11.0 (p.63)	I	Reference Python implementations of both the encoder and decoder can be found at github.com/riscv-non-isa/riscv-trace-spec.
RVET.11.4	11.1 (p.63)	H	Decoder pseudo code
RVET.11.5	11.1 (p.63-p.68)	I	See original text pages 63 to 68

CHAPTER 12 Example code and packets

ID	REFERENCE	TYPE	DEFINITION
RVET.12.1	12.0 (p.69)	H	Example code and packets
RVET.12.2	12.0 (p.69)	I	In the following examples *ret* is referred to as uninferable, this is only true if implicit-return mode is off

RVET.12.3	12.0 (p.69)	I	1. Call to debug_printf(), from 80001a84, in main():
			`00000000800019e8 <main>:`
			`........: ...`
			`80001a80: f6d42423 {sw a3,-152(s0)}`
			`80001a84: ef4ff0ef {jal x1,80001178} <debug_printf>`
			
			PC: 80001a84 →80001178
			The target of the *jal* is inferable, thus NO te_inst packet is sent.
			
			`0000000080001178 <debug_printf>:`
			`80001178: 7139 {addi sp,sp,-64}`
			`8000117a: ...`

RVET.12.4	12.0 (p.69)	I	2. Return from debug_printf():
			`80001186: ...`
			`80001188: 6121 {addi sp,sp,64}`
			`8000118a: 8082 {ret}`
			
			PC: 8000118a →80001a88
			The target of the ret is uninferable, thus a *te_inst* packet IS sent: te_inst[format=2 (ADDR_ONLY): address=0x80001a88, updiscon=0]
			
			`80001a88: 00000597 {auipc a1,0x0}}`
			`80001a8c: 65058593 {addi a1,a1,1616}}`
			`# 800020d8 <main+0x6f0>`

ID	REFERENCE	TYPE	DEFINITION

RVET.12.5	12.0 (p.69, p.70)	I	3. exiting from Func_2(), with a final taken branch, followed by a *ret*
			`00000000800010b6 <Func_2>:`
			`........: ....`
			`800010da: 4781 {li a5,0}`
			`800010dc: 00a05863 {blez a0,800010ec} <Func_2+0x36>`
			
			PC: 800010dc →800010ec, add branch TAKEN to branch_map, but no packet sent yet.
			branches = 0; branch_map = 0;
			branch_map = 0 <<branches++;
			
			`800010ec: 60e2 {ld ra,24(sp)}`
			`800010ee: 6442 {ld s0,16(sp)}`
			`800010f0: 64a2 {ld s1,8(sp)}`
			`800010f2: 853e {mv a0,a5}`
			`800010f4: 6105 {addi sp,sp,32}`
			`800010f6: 8082 {ret}`
			
			PC: 800010f6 →80001b8a
			
			The target of the *ret* is uninferable, thus a *te_inst* packet is sent, with ONE branch in the branch_map
			te_inst[format=1 (DIFF_DELTA): branches=1, branch_map=0x0, address=0x80001b8a (Δ=0xab0) updiscon=0]
			
			`00000000800019e8 <main>:`
			`........: ....`
			`80001b8a: f4442603 {lw a2,-188(s0)}`
			`80001b8e: ....`

ID	REFERENCE	TYPE	DEFINITION
RVET.12.6	12.0 (p.70)	I	4. 3 branches, then a function return back to Proc_1()

```
0000000080001100 <Proc_6>:
........: ....
80001112: c080 {sw s0,0(s1)}
80001114: 4785 {li a5,1}
80001116: 02f40463 {beq s0,a5,8000113e <Proc_6+0x3e>}
```

PC: 80001116 →8000111a, add branch NOT taken to branch_map, but no packet sent yet. branches = 0; branch_map = 0; branch_map = 1 <<branches++;

```
8000111a: c81d {beqz s0,80001150 <Proc_6+0x50>}
```

PC: 8000111a →8000111c, add branch NOT taken to branch_map, but no packet sent yet. branch_map = 1 <<branches++;

```
8000111c: 4709 {li a4,2}
8000111e: 04e40063 {beq s0,a4,8000115e <Proc_6+0x5e>}
```

PC: 8000111e →8000115e, add branch TAKEN to branch_map, but no packet sent yet. branch_map = 0 <<branches++;

```
8000115e: 60e2 {ld ra,24(sp)}
80001160: 6442 {ld s0,16(sp)}
80001162: c09c {sw a5,0(s1)}
80001164: 64a2 {ld s1,8(sp)}
80001166: 6105 {addi sp,sp,32}
80001168: 8082 {ret}

00000000800011d6 <Proc\_1>:
........: ....
80001258: 00093783 {ld a5,0(s2)}
8000125c: ....
```

PC: 80001168 →80001258
The target of the *ret* is uninferable, thus a *te_inst* packet is sent, with THREE branches in the branch_map
te_inst[format=1 (DIFF_DELTA): branches=3, branch_map=0x3, address=0x80001258 (Δ=0x148), updiscon=0]

ID	REFERENCE	TYPE	DEFINITION
RVET.12.7	12.0 (p.70, p.71)	I	5. A complex example with 2 branches, 2 jal, and a ret

```
00000000800011d6 <Proc\_1>:
........: ....
8000121c: 441c {lw a5,8(s0)}
8000121e: c795 {beqz a5,8000124a} <Proc_1+0x74>
```

PC: 8000121e →8000124a, add branch TAKEN to branch_map, but no packet sent yet.
branches = 0; branch_map = 0;
branch_map = 0 <<branches++;

```
8000124a: 44c8 {lw a0,12(s1)}
8000124c: 4799 {li a5,6}
8000124e: 00c40593 {addi a1,s0,12}
80001252: c81c {sw a5,16(s0)}
80001254: eadff0ef {jal x1,80001100} <Proc_6>
```

PC: 80001254 →80001100
The target of the **jal** is inferable, thus no **te_inst** packet needs be sent.

```
0000000080001100 <Proc_6>:
80001100: 1101 {addi sp,sp,-32}
80001102: e822 {sd s0,16(sp)}
80001104: e426 {sd s1,8(sp)}
80001106: ec06 {sd ra,24(sp)}
80001108: 842a {mv s0,a0}
8000110a: 84ae {mv s1,a1}
8000110c: fedff0ef {jal x1,800010f8} <Func_3>
```

PC: 8000110c →800010f8
The target of the **jal** is inferable, thus no *te_inst* packet needs to be sent.

```
00000000800010f8 <Func_3>:
800010f8: 1579 {addi a0,a0,-2}
800010fa: 00153513 {seqz a0,a0}
800010fe: 8082 {ret}
```

PC: 800010fe →80001110
The target of the ret is uninferable, thus a te_inst packet will be sent shortly.

```
0000000080001100 <Proc_6>:
........: ....
80001110: c115 {beqz a0,80001134} <Proc_6+0x34>
80001112: ....
```

PC: 80001110 →80001112, add branch NOT TAKEN to branch_map.
branch_map = 1 <<branches++;
te_inst[format=1 (DIFF_DELTA): branches=2, branch_map=0x2, address=0x80001110 (Δ=0xfffffffffffffef4), updiscon=1]

CHAPTER 13 Code fragment and transport

ID	REFERENCE	TYPE	DEFINITION
RVET.13.1	13.0 (p.73)	H	Code fragment and transport
RVET.13.2	13.0 (p.73)	I	This section shows fragments of code, and associated data from one of the architectural tests in the repository.
RVET.13.3	13.0 (p.73)	I	For the individual fragments the ingress signals are shown and the corresponding packets generated.
RVET.13.4	13.0 (p.73)	I	It further shows how the packets are transported via on-chip transport fabric.
RVET.13.5	13.0 (p.73)	I	The fragments shown below are extracted from the test whilst it is being executed.
RVET.13.6	13.0 (p.73)	I	In order to give some context to the fragment of interest, code prior to and after the fragment is also given.
RVET.13.7	13.1 (p.73)	H	Illegal Opcode test
RVET.13.8	13.1 (p.73)	I	In this example the test executes an illegal opcode (at line labelled 14) and traps. We show the output from the patched spike execution in line 30. The input signals to the encoder are shown in lines labelled 38-46. The HART will have set the signals shown in line 42 when the illegal instruction is executed and as can be seen it is not retired. Lines labelled 53, 56 and 59 show the packets output from the encoder for this fragment.
RVET.13.9	13.1 (p.73)	I	See pages 73 to 74
RVET.13.10	13.2 (p.75)	H	Timer Long Loop
RVET.13.11	13.2 (p.75)	I	See pages 75 to 76
RVET.13.12	13.3 (p.76)	H	Startup xrle
RVET.13.13	13.2 (p.76)	I	See pages 76 to 78

CHAPTER 14 Future Directions

ID	REFERENCE	TYPE	DEFINITION
RVET.14.1	14.0 (p.79)	H	Future Directions
RVET.14.2	14.0 (p.79)	I	This chapter captures ideas and enhancements that may be useful for to consider in future versions of the E-Trace specification.
RVET.14.3	14.1 (p.79)	H	Vector
RVET.14.4	14.1 (p.79)	I	Now that the vector extension has been ratified it would be interesting to look at extending E-Trace to support instruction and data trace for vector operations.
RVET.14.5	14.2 (p.79)	H	Inter-instruction cycle counts
RVET.14.6	14.2 (p.79)	I	In this mode the encoder will trace where the hart is stalling by reporting the number of cycles between successive instruction retirements.