

Efficient Trace for RISC-V

Authors: Gajinder Panesar and Iain Robertson

Version 2.1, 2026-10-20: This document is in the Ratified state.

Table of Contents

Preamble 1
Copyright and license information 3
1. Introduction 5
1.1. Terminology 6
1.2. Nomenclature 7
2. Encoder Control 9
2.1. Basic Control 9
2.2. Optional Modes 10
2.3. Filtering 10
3. Branch Trace 13
3.1. Instruction delta trace concepts 13
3.1.1. Sequential instructions 13
3.1.2. Uninferable PC discontinuities 13
3.1.3. Branches 13
3.1.4. Interrupts and exceptions 14
3.1.5. Synchronization 14
3.1.6. End of trace 14
3.2. Optional and run-time configurable modes 15
3.2.1. Delta address mode 15
3.2.2. Full address mode 15
3.2.3. Implicit exception mode 15
3.2.4. Sequentially inferable jump mode 16
3.2.5. Implicit return mode 16
3.2.6. Branch prediction mode 16
3.2.7. Jump target cache mode 17
4. Hart to encoder interface 19
5. Filtering 21
6. Timestamping 23
7. Instruction Trace Encoder Output Packets 25
7.1. Format 3 packets 26
7.2. Format 3 subformat 0 - Synchronisation 26
7.2.1. Format 3 **branch** field 26
7.3. Format 3 subformat 1 - Trap 26
7.3.1. Format 3 **thaddr**, **address** and **privilege** fields 27
7.3.2. Format 3 **tval** field 27
7.4. Format 3 subformat 2 - Context 28
7.5. Format 3 subformat 3 - Support 28
7.5.1. Format 3 subformat 3 **qual_status** field 29
7.6. Format 2 packets 29
7.6.1. Format 2 **notify** field 30
7.6.2. Format 2 **notify** and **updiscon** fields 30
7.6.3. Format 2 **irreport** and **irdepth** 32
7.7. Format 1 packets 32

7.7.1. Format 1 **updiscon** field 34
7.7.2. Format 1 **branch_map** field 34
7.7.3. Format 1 **irreport** and **irdepth** fields 34
7.8. Format 0 packets 34
7.8.1. Format 0 subformat field 37
7.8.2. Format 0 **branch_fmt** field 37
7.8.3. Format 0 **irreport** and **irdepth** fields 37
8. Data Trace Encoder Output Packets **39**
8.1. Load and Store 39
8.1.1. format field 39
8.1.2. size field 42
8.1.3. diff field 42
8.1.4. data_len field 42
8.2. Atomic 42
8.2.1. size field 42
8.2.2. diff field 44
8.2.3. operand field 44
8.2.4. data_len and op_len fields 44
8.3. CSR 46
8.3.1. diff field 46
8.3.2. operand field 47
8.3.3. data_len and op_len fields 47
8.3.4. addr fields 47
9. Reference Compressed Branch Trace Algorithm **49**
9.1. Format selection 51
9.2. Resynchronisation 52
9.3. Multiple retirement considerations 52
9.4. Discovery of encoder parameters 55
9.5. Example ipxact description 56
10. Decoder **63**
10.1. Decoder pseudo code 63
11. Example code and packets **77**
12. Code fragment and transport **81**
12.1. Illegal Opcode test 81
12.1.1. Code fragment 81
12.1.2. Packet data 83
12.1.3. Siemens transport 83
12.1.4. ATB transport 83
12.2. Timer Long Loop 84
12.2.1. Code fragment 84
12.2.2. Packet data 86
12.2.3. Siemens transport 86
12.2.4. ATB transport 86
12.3. Startup xrle 87
12.3.1. Code fragment 87

12.3.2. Packet data ... 88
12.3.3. Siemens transport ... 88
12.3.4. ATB transport ... 89
13. Standard Support Packet v1.0 extension to Efficient Trace for RISC-V v2.0 ... **91**
13.1. Standard Support Packet format ... 91
14. Implicit Return v1.0 extension to Efficient Trace for RISC-V v2.0 ... **93**
14.1. Background ... 93
14.2. Revised packet formats ... 93
14.2.1. Format 2 **irreport** and **irets** ... 94
14.2.2. Format 1 **irreport** and **irets** fields ... 95
14.2.3. Format 0 **irreport** and **irets** fields ... 97
15. Data Trace of Multiple-Memory Access and Compare and Swap v1.0 extension to Efficient Trace for RISC-V v2.0 ... **99**
15.1. Revised packet formats ... 99
15.1.1. format field ... 99

Preamble



This document is in the Ratified state.

No changes are allowed. Any desired or needed changes can be the subject of a follow-on new extension. Ratified extensions are never revised.

Copyright and license information

This specification is licensed under the Creative Commons Attribution 4.0 International License (CC-BY 4.0). The full license text is available at creativecommons.org/licenses/by/4.0/.

Copyright 2025-2026 by RISC-V International.

Preface to Version 2.1 2026-10-20

- Chapter 4 contents replaced with a reference to the RISC-V Hart to Trace Interface (RHTI) Specification, V1.0.
  - This introduces a small normative change related to multi-memory access instructions; see Chapter 4 for details.
- Added Chapter 13, which describes the Standard Support Packet extension to Version 2.0.
- Added Chapter 14, which describes the Implicit Return extension to Version 2.0.
- Added Chapter 15, which describes the Data Trace of Multiple-Memory Access and Compare and Swap extension to Version 2.0.
- Updated the decoder pseudo-code in Chapter 10 to include support for Implicit Return, Branch Predictor and Jump Target Cache optional modes.
- Corrections to Chapter 2: **trTeInstMode** hardcoded to 6 (not 7 as previously stated), **trTeContext** and **trTeInstImplicitReturnMode** changed to optional rather than not-applicable.
- Fixed error in **branch_map** descriptions in tables in Section 7.7 introduced during earlier translation from LaTeX to asciidoc: values for branch taken or not taken were missing.
- Added informative recommendation regarding the use of partial timestamps in packet encapsulation to Chapter 6.
- Remove historical "Future directions" chapter.
- Minor clarifications, formatting and typo fixes.

Preface to Version 20250616

Clarifications only - no changes to normative behaviour.

- Updated Chapter 7 and Chapter 12 to reference the Unformatted Trace & Diagnostic Data Packet Encapsulation for RISC-V Specification;
- Removed ambiguity between 'last' and 'final'. Last was previously used to mean both the instruction before the current one, and the final instruction traced;
- Clarified behaviour when trace-on and trace-off triggers occur both occur in the same cycle (now in RHTI Specification);
- Clarified that full synchronization also takes place on a privilege change (see Section 3.1.5 and Section 7.3.1);
- Reworded jump classifications to align with terminology used in other specifications (now in RHTI Specification);
- Clarified when to issue sync-support packet when trace is enabled (see Section 7.5);
- Clarified that branch_map can be set to 31 any time an address needs to be reported and the branch map is full (see Section 7.7);
- Updated reference encoding algorithm to generalize resync behaviour and add missing trap conditions (see Chapter 9);

- Clarified that implicit returns are not stored in the jump target cache (see Section 3.2.7);
- Updated decoder to remove ECALL, EBREAK and C.EBREAK from is_uninferrable_jump function. Including them is harmless but unnecessary, as these instrucitons don't retire (see Chapter 10);
- Corrected description of why the EPC is not always reported (see Section 7.3.1).
- Minor clarifications, formatting and typo fixes.

Preface to Version 20240419

Formatting and typo fixes.

Preface to Version 20240305

First version in AsciiDoc format.

Preface to Version 20231215

Clarifications only - no changes to normative behaviour.

- Control field definitions removed from section 2, which now references the RISC-V Trace Control Interface Specification, V1.0
- Added detail on handling of multi-load/store instructions for data trace (now in RHTI Specification).
- Removed references to tail-calls in jump classifications (now in RHTI Specification).
- Corrected typos where `lrid` was inadvertently refered to by an earlier name (`index`) in Section 8.1.
- Corrected reference decoder in Chapter 10 to cover a corner-case related to trap returns.

Preface to Version 2.0

Ratified version of the Efficient Trace for RISC-V specification.

Chapter 1. Introduction

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.

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.

One method of achieving this is via a Processor Branch Trace.

This works by tracking execution from a known start address and sending messages about the address deltas taken by the program. These deltas are typically introduced by jump, call, return and branch type instructions, although interrupts and exceptions are also types of deltas.

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.

In RISC-V, all instructions are executed unconditionally or at least their execution can be determined based on the program binary. 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. 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.

Interrupts generally occur asynchronously to the program's execution rather than intentionally as a result of a specific instruction or event. Exceptions can be thought of in the same way, even though they can be typically linked back to a specific instruction address. 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.

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.

1.1. Terminology

The following terms have a specific meaning in this specification.

- **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)
- **E-Trace**: Abbreviation of *Efficient Trace for RISC-V*.
- **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
- **final instruction**: the instruction retired immediately before tracing stops.
- **last instruction**: in a block of sequentially retired instructions, the instruction at the end of the sequence
- **interrupt**: an external asynchronous event that may cause a RISC-V hart to experience an unexpected transfer of control
- **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'

1.2. Nomenclature

In the following sections items in **bold** are signals or fields within a packet.

Items in ***bold italics*** are mnemonics for instructions or CSRs defined in the RISC-V ISA

Items in *italics* with names ending *'_p'* refer to parameters either built into the hardware or configurable hardware values.

Chapter 2. Encoder Control

The fields required to control a Trace Encoder are defined in the RISC-V Trace Control Interface Specification V1.0, which is intended to apply to any and all RISC-V trace encoders, regardless of encoding protocol. This chapter details which of those fields apply to E-Trace. To avoid replication, descriptions are not provided here; additional E-Trace specific context or clarification is provided only where required.

How fields are organized and accessed (e.g packet based or memory mapped) is outside the scope of this document. If a memory mapped approach is adopted, the 'Trace Component Register Map' from the RISC-V Trace Control Interface Specification should be used.

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.

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

2.1. Basic Control

The following fields control basic encoding behavior.

Table 1. Basic Control

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	

Field	Applicability	E-Trace Specific Details
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	

2.2. Optional Modes

See Section 3.2 for details of the modes covered in this section.

Table 2. 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	O	
trTeInstMode	N	Hardcode to 6.
trTeInstImplicitReturnMode	0	
trTeInstEnRepeatedHistory	N	Hardcode to 0.
trTeInstEnAllJumps	N	Hardcode to 0.
trTeInstExtendAddrMSB	N	Hardcode to 0.

2.3. Filtering

See Chapter 5 for details of the filtering capabilities covered in this section.

Table 3. 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

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. Deltas are typically introduced by jump, call, return and branch type instructions, although interrupts and exceptions are also types of deltas.

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.

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.

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.

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. 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. Both static and dynamically linked programs can be traced using this approach. Statically linked programs are straightforward as they generally operate in a known address space, often mapping directly to physical memory. Dynamically linked programs require the debugger to keep track of memory allocation operations using either trace or stop-mode debugging.

3.1. Instruction delta trace concepts

3.1.1. Sequential instructions

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. Consequently, there is no need to report them in the trace. The trace only needs to contain whether branches were taken or not, the addresses of taken indirect jumps, or other program counter discontinuities.

3.1.2. Uninferable PC discontinuities

An uninferable program counter discontinuity is a program counter change that can not be inferred from the program binary alone. For these cases, the instruction delta trace must include a destination address: the address of the next valid instruction.

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. In this case, the address of the instruction following the jump (also known as the jump target) must be traced.

Interrupts and exceptions are another form of uninferable PC discontinuity; these are discussed in detail below.

3.1.3. Branches

A branch is an instruction where a jump is conditional on the value of a register or a flag. For a decoder to

able to follow program flow, the trace must include whether a branch was taken or not.

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. Direct branches are the only type of branch that is supported by the RISC-V ISA.

3.1.4. Interrupts and exceptions

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. Exceptions can be thought of in the same way, even though they can be typically linked back to a specific instruction address.

The decoder generally does not know where an interrupt occurred 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. When an interrupt or exception occurs, the final instruction retired beforehand must be traced. Following this the next valid instruction address (the first of the trap handler) must be traced.

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. In this document, interrupts and exceptions are only traced when they cause traps to be taken.

3.1.5. Synchronization

In order to make the trace robust there must be regular synchronization points within the trace. Synchronization is accomplished by sending a full valued instruction address (and potentially a context identifier). The decoder and debugger may also benefit from sending the reason for synchronizing. The frequency of synchronization is a trade-off between robustness and trace bandwidth.

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;
- If the privilege level changes;
- After a prolonged period of time.

3.1.6. End of trace

If tracing stops for any reason, the address of the final traced instruction must be output.

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 "Using trigger outputs from the Debug Module" section of RHTI Specification);
- The matching criteria for any filtering capabilities implemented by the encoder may no longer be met;
- The encoder may be disabled.

3.2. Optional and run-time configurable modes

An instruction trace encoder may support multiple tracing modes. To ensure that the decoder treats the incoming packets correctly, it needs to be informed of the current active configuration. The configuration is reported by a packet that is issued by the encoder whenever the encoder configuration is changed.

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.

Modes may have associated parameters; see Table 29 for further details.

All modes are optional apart from delta address mode, which must be supported.

3.2.1. Delta address mode

Related parameters: None

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. This differential encoding requires fewer bits than the full address, and thus results in more efficient trace compression.

3.2.2. Full address mode

Related parameters: None

In full address mode, all addresses in the trace are encoded as absolute addresses instead of in differential form. This kind of encoding is always less efficient, but it can be a useful debugging aid for software decoder developers.

3.2.3. Implicit exception mode

Related parameters: None

The RISC-V Privileged ISA specification stores exception handler base addresses in the ***stvec/vstvec/mtvec*** CSR registers. In some RISC-V implementations, the lower address bits are stored in the ***scause/vscause/mcause*** CSR registers.

By default, both the ***tvec*** and ***cause*** values are reported when an exception or interrupt occurs.

The implicit exception mode omits ***tvec*** (the trap handler address), from the trace and thus improves efficiency.

This mode can only be used if the decoder can infer the address of the trap handler from just the exception cause.

3.2.4. Sequentially inferable jump mode

Related parameters: *sijump_p*.

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

3.2.5. Implicit return mode

Related parameters: *call_counter_size_p*, *return_stack_size_p*, *itype_width_p*.

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

Returns can only be treated as inferable if the associated call has already been reported in an earlier packet. The encoder must ensure that this is the case. This can be accomplished by utilizing a counter to keep track of the number of nested calls being traced. The counter increments on calls (but not tail calls), and decrements on returns (see "Jump classification and target inference" section of RHTI Specification 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).

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

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

3.2.6. Branch prediction mode

Related parameters: *bpred_size_p*.

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.

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.

A significant coding efficiency can be obtained by the addition of a branch predictor in the encoder. To keep the encoder and decoder synchronized, a predictor with identical behavior will need to be implemented in the decoder software.

The predictor shall comprise a lookup table of $2^{bpred_size_p}$ entries. 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.

The MSB represents the predicted outcome, the LSB the most recent actual outcome. The prediction must fail twice for the predicted value to change.

The lookup table entries are initialized to 01 when a synchronization packet is sent.

3.2.7. Jump target cache mode

Related parameters: *cache_size_p*.

By default, the target address of an uninferable jump is output in the trace, usually in differential form. If the same function is called repeatedly, (for example, in a loop), the same address will be output repeatedly.

An efficiency gain can be obtained by the addition of a jump target cache to the encoder. To keep the encoder and decoder synchronized, a cache with identical behavior will need to be implemented in the decoder software. Even a small cache can provide significant improvement.

The cache shall comprise $2^{cache_size_p}$ entries, each of which can contain an instruction address. The addresses stored in the cache are the targets of uninferable jumps. 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).

Each uninferable jump target is first compared with the entry in the cache at the index derived from the jump target address. If it is found in the cache, the index number is traced rather than the target address. If it is not found in the cache, the entry at that index is replaced with the current instruction address.

Note that if implicit return mode is enabled, most function return targets are inferable and are not output in the trace. Any such inferrable return target must not be stored in the cache. This effectively puts the implicit return evaluation in series before the jump target cache, and this may present timing challenges. This can be mitigated by invalidating the cache entry that corresponds to a mis-predicted return rather

than updating the entry with the actual return address. Mis-predicted returns are rare so the performance impact will be negligible.

The cache entries are all invalidated when a synchronization packet is sent.

Chapter 4. Hart to encoder interface

This interface is now defined in the RISC-V Hart to Trace Interface (RHTI) Specification, V1.0. Originally developed as part of this specification, it has been separated into a standalone specification as it is not inherently E-Trace specific, and intended for use with any instruction and data trace encoding protocol.

This standalone specification is identical to the former contents of this chapter, with one small exception. In order to make the RHTI fully protocol agnostic, a normative change has been introduced related to how multi-memory access instructions are handled by harts providing support for data trace. See the **Multi-memory access Instructions** section of the RHTI Specification.

Chapter 5. Filtering

The contents of this chapter are informative only.

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.

How this is accomplished is implementation specific.

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.

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).

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.

Allowing for up to 3 comparators allows for simultaneous matching on Address, Trap value and context (unlikely, but should not be architecturally precluded).

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

The support for Timestamps is optional and so the contents of this chapter are informative only.

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.

This can be used to judge "time" between various points 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.

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 graularity 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.

Where packets are encapsulated according to the Unformatted Trace & Diagnostic Data Packet Encapsulation for RISC-V Specification, this includes provision for an optional N-byte timestamp within the encapsulation itself. This can be used to include a timestamp value with any of the trace packets defined in this specification. It is not usually desirable to include full timestamps (e.g. up to 64-bits) in each packet, as this adds a significant overhead, and the difference in timestamp values between successive packets is usually quite small. Using smaller, partial encapsulation timestamps (e.g. 16 bits) is typical.

With partial encapsulation timestamps, software can reconstruct the full timestamp value of any packet, provided that it initially receives a full timestamp (either in a sync packet, or possibly a dedicated timestamp packet), and then the interval between successive packets carrying only a partial encapsulation timestamp is less than 2^{N*8} ticks of the timestamp clock. The lower 2^{N*8} bits of the reconstructed timestamp are taken directly from the encapsulation timestamp. If the encapsulation timestamp value is smaller than the previous one, software can deduce that time has moved forward by an additional 2^{N*8} ticks, and add this

to the reconstructed timestamp.

However, this reconstruction process fails if the interval between packets exceeds 2^{N*8}. To prevent this, if the interval between packets does exceed 2^{N*8}, the first packet sent subsequently must contain a full timestamp. This is typically a sync packet, but could be a dedicated timestamp packet. Requesting the insertion of packets containing full timestamps should be automatically managed by the packet encapsulation process.

Chapter 7. Instruction Trace Encoder Output Packets

The bulk of this section describes the payload of packets output from the Instruction Trace Encoder. 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 mandated, but the Unformatted Trace & Diagnostic Data Packet Encapsulation for RISC-V Specification is recommended. Chapter 3 of that document specifically addresses the encapsulation requirements for E-Trace.

Any encapsulation must include the following information:

- The packet type;
- The packet length, in bytes;
- The packet payload.

Two example transport schemes are the Siemens Messaging Infrastructure, and the Arm AMBA Trace Bus. The Unformatted Trace & Diagnostic Data Packet Encapsulation for RISC-V Specification is well suited to either of these.

The remainder of this section describes the contents of the payload portion which should be independent of the infrastructure. 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.

This packet payload format is used to output encoded instruction trace. 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.

In order to achieve best performance, actual packet lengths may be adjusted using 'sign based compression'. 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. This technique eliminates identical bits from the most significant end of the packet, and adjusts the length of the packet accordingly. A decoder receiving this shortened packet can reconstruct the original full-length packet by sign-extending from the most significant received bit.

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.

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. Two potential solutions to this are as follows:

- If the maximum packet payload length is 2^N-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 2^N zero bytes cannot occur within a packet payload, and therefore the first non-zero byte seen after a sequence of at least 2^N 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.

7.1. Format 3 packets

Format 3 packets are used for synchronization, traps, reporting context and supporting information. There are 4 sub-formats.

Throughout this document, the term "synchronization packet" is used. This refers specifically to format 3, subformat 0 and subformat 1 packets.

7.2. Format 3 subformat 0 - Synchronisation

This packet contains all the information the decoder needs to fully identify an instruction. 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.

Table 4. 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 by *iaddress_lsb_p* in order to recreate original byte address.

7.2.1. Format 3 branch field

This bit indicates the taken/not taken status in the case where the reported address points to a branch instruction. 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 1.

7.3. Format 3 subformat 1 - Trap

This packet also contains all the information the decoder needs to fully identify an instruction. 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.

If the implicit exception mode is enabled (see Section 3.2.3), the trap handler address is omitted if **thaddr** is 1.

Table 5. 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	*ecause_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 by _iaddress-lsb_p_ in order to recreate original byte address.
tval	*iaddress_width_p*	Value from appropriate **utval/stval/vstval/mtval** CSR. Field omitted for interrupts

7.3.1. Format 3 thaddr, address and privilege fields

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. In this case, the trap handler address will be reported via a subsequent format 3, subformat 0 packet. An exception occuring on the 1st traced instruction is treated in the same way.

Usually when an exception or interrupt occurs, the cause is reported along with the 1st address of the trap handler, when that instruction retires. In this case, **thaddr** is 1. 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. In this situation, **thaddr** is 0, and **address** is undefined (unless it contains the EPC as outlined in the previous paragraph).

Note: The reason for not reporting the EPC for all exceptions when **thaddr** is 0 is that it can be inferred by the decoder using the exception cause.

Where an interrupt or exception causes a privilege change, this change comes into force from the start of the trap handler. As such, the privilege value reported when **thaddr** is 0 will be the privilege level prior to any change.

7.3.2. Format 3 tval field

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. It is omitted from the packet for interrupts.

7.4. Format 3 subformat 2 - Context

This packet contains only the context and/or the timestamp, and is output when the context value changes and can be reported imprecisely (see **ctype** definition in RHTI Specification).

Table 6. 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.

7.5. Format 3 subformat 3 - Support

This packet provides supporting information to aid the decoder. It must be issued:

- When trace is enabled, and before the first sync packet, in order to ensure the decoder is aware of how the encoder is configured. This could be as late as when **trTeInstTracing** becomes 1, but it is recommended this be sent as soon as **trTeEnable** changes from 0 to 1. This reduces the likelihood of having to generate two packets (support and sync-start) at the point tracing actually starts;
- When tracing ceases for any reason (**trTeEnable** or **trTeInstTracing** set to 0, trace-off trigger, halt, reset, loss of filter qualification, etc.), in order to inform the decoder that the preceding packet reported the address of the final traced instruction;
- If one or more trace packets cannot be sent (for example, due back-pressure from the packet transport infrastructure);
- If the operating mode of the encoder changes such that the the information output in a previous support packet no longer applies (i.e. if **encoder_mode**, **ioptions**, **doptions** or **denable** change).

The **ioptions** and **doptions** fields are placeholders that must be replaced by an implementation specific set of individual bits - one for each of the optional modes supported by the encoder.

Table 7. 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	*E*	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	00 (no_change): No change to filter qualification 01 (ended_rep): Qualification ended, preceding **te_inst** sent explicitly to indicate final qualification instruction 10 (trace_lost): One or more instruction trace packets lost. 11 (ended_ntr): Qualification ended, preceding **te_inst** would have been sent anyway due to an updiscon, even if it wasn't the final qualified instruction

Field name	Bits	Description
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)



The above packet is deprecated and has been superseded by the equivalent packet defined in the Standard Support Packet extension (see Chapter 13).

7.5.1. Format 3 subformat 3 qual_status field

When tracing ends, the encoder reports the address of the final traced instruction, and follows this with a format 3, subformat 3 (supporting information) packet. Two codes are provided for indicating that tracing has ended: **ended_rep** and **ended_ntr**. 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 final traced instruction is at looplabel. 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. 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. 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;

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**.

7.6. Format 2 packets

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. The address is in differential format unless full address mode is enabled (see Section 3.2.2).

Table 8. 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]**.
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**.



The above packet is deprecated and has been superseded by the equivalent packet defined in the Implicit Return extension (see Chapter 14). This replaces **irdepth** *with* **irets**.

7.6.1. Format 2 notify field

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. 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.

7.6.2. Format 2 notify and updiscon fields

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**). 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. Consider the following code fragment:

```
looplabel -4: *_opcode A_*
looplabel   : *_opcode B_*
looplabel +4: *_opcode C_*
:
looplabel +N *_JALR_* # Jump to looplabel
```

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 fall-through 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. 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. 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*).

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 previous reported address, until it reaches looplabel. 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 most recently 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.

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 final traced instruction (because tracing was disabled for some reason). See Section 7.5.1 for further discussion of this point.



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.

7.6.3. Format 2 irreport and irdepth



irdepth *has been superseded by* **irets** *defined in the Implicit Return extension (see Chapter 14).*

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. 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* non-zero).

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.

In some cases it is also necessary to report the current stack depth or call count if the packet is reporting the instruction immediately before an exception, interrupt, privilege change or resync. 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 previous call (in which case the decoder will correctly stop in the innermost call), or
  - There has been at least one branch since the previous return (in which case the decoder will correctly stop in the call where there are no unprocessed branches).

    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. However, this cannot be guaranteed for recursive functions, as the reported address will occur multiple times in the execution path.

7.7. Format 1 packets

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. If included, the address is in differential format unless full address mode is enabled (see Section 3.2.2).

Table 9. Packet format 1 - address, branch map

Field name	Bits	Description
format	2	01 (diff-delta): includes branch information and may include differential address

Field name	Bits	Description
branches	5	Number of valid bits **branch_map**. The number of bits of **branch_map** is determined as follows: 0: (cannot occur for this format) 1: 1 bit 2-3: 3 bits 4-7: 7 bits 8-15: 15 bits 16-31: 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: 0 : branch taken 1 : 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]**.
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**.



The above packet is deprecated and has been superseded by the equivalent packet defined in the Implicit Return extension (see Chapter 14). This replaces **irdepth** *with* **irets**.

Table 10. 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: 0: 31 bits, no **address** in packet 1-31: (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: 0 : branch taken 1 : branch not taken

7.7.1. Format 1 updiscon field

See Section 7.6.2.

7.7.2. Format 1 branch_map field

When the branch map becomes full it must be reported, but in most cases there is no need to report an address. This is indicated by setting **branches** to 0. If the address does need to be reported for any reason (for example because the instruction immediately prior to the final branch causes an uninferable discontinuity) **branches** is set to 31.

The choice of sizes (1, 3, 7, 15, 31) is designed to minimize efficiency loss. 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. Using a tapered set of sizes means that the number of wasted bits will on average be less for shorter packets. 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. 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.

7.7.3. Format 1 irreport and irdepth fields

See Section 7.6.3.

7.8. Format 0 packets

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.

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. 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. 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 (0xffff ffff). 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. Note: the decoder is reliant on the maximum value in order to identify this special case, so the 32-bit counter must be implemented in full.

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. 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.

Table 11. 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 previous correct prediction failed. 01 - 11: (cannot occur for this format)

Table 12. 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. 11 (addr-fail): Packet contains an **address** that points to a branch which failed the prediction. 00, 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]**.
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**.



The above packet is deprecated and has been superseded by the equivalent packet defined in the Implicit Return extension (see Chapter 14). This replaces **irdepth** *with* **irets**.

Table 13. 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.

Field name	Bits	Description
branches	5	Number of valid bits in **branch_map**. The length of **branch_map** is determined as follows:+ 0: (cannot occur for this format) 1: 1 bit 2-3: 3 bits 4-7: 7 bits 8-15: 15 bits 16-31: 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]**.



The above packet is deprecated and has been superseded by the equivalent packet defined in the Implicit Return extension (see Chapter 14). This replaces **irdepth** *with* **irets**.

Table 14. 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]**.



The above packet is deprecated and has been superseded by the equivalent packet defined in the Implicit Return extension (see Chapter 14). This replaces **irdepth** *with* **irets**.

7.8.1. Format 0 subformat field

The width of this field depends on the number of optional formats supported. Currently, two optional formats are defined (correctly predicted branches and jump target cache). The width is specified by the *f0s_width* discovery field (see Section 9.4). If multiple optional formats are supported, the field width must be non-zero. 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. This provision allows additional formats to be added in future without reducing the efficiency of the existing formats.

7.8.2. Format 0 branch_fmt field

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.

When a branch count is reported without an address it is because a branch has failed the prediction. 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. For the latter case, the reported address will always be for a branch, and in the former cases it may be. If it is a branch, it is necessary to be explicit about whether or not the prediction was met or not. If it is met, then the reported address is that of the last correctly predicted branch.

7.8.3. Format 0 irreport and irdepth fields



irdepth *has been superseded by* **irets** *defined in the Implicit Return extension (see Chapter 14).*

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). Purpose and behavior is as described in Section 7.6.3.

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. An implementation could omit these if all implicit return failures are reported using format 1.

Chapter 8. Data Trace Encoder Output Packets

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. 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). 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).

By default, all data trace packets include both address and data. However, provision is made for run-time configuration options to exclude either the address or the data, in order to minimize trace bandwidth. 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. 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. 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.

As outlined in the "Data Trace Interface requirements" section of the RHTI Specification, two different signaling protocols between the RISC-V hart and the encoder are supported: *unified* and *split*. Accordingly, both unified and split trace packets are defined. Where the *split* signalling is supported, data accesses with a load data component normally result in two packets, one each for the request and data phases. However, if address or data are omitted as described above, such that all the required information is from a single phase, then the appropriate unified packet format will be used. For example, if the data value is omitted then the *Unified load or store with address only* packet format will be used even if *split* signalling is used. Similarly, if the address is omitted then *Unified load or store with data only* is used.

In the following tables, "clog2" is an abbreviation for "ceiling of log2".

The packets described in this section do not support data trace for harts that support the Zcmp, Vector or Zacas extensions, and are deprecated in favour of the packets defined in the MMA/CAS extension (see Chapter 15)

8.1. Load and Store

8.1.1. format field

Types of data trace packets are differentiated by the **format** field. This field is 2 bits wide if only unified loads and stores are supported, or 3 bits otherwise.

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.

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.

Table 15. 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^{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



The above packet is deprecated and has been superseded by the equivalent packet defined in the MMA/CAS extension (see Chapter 15)

The following packet format is used when the encoder is configured to omit data values from data trace packets (trTeDataControl.trTeDataNoValue is 1), even with *split* siganalling.

Table 16. 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^{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



The above packet is deprecated and has been superseded by the equivalent packet defined in the MMA/CAS extension (see Chapter 15)

The following packet format is used when the encoder is configured to omit address values from data trace packets (trTeDataControl.trTeDataNoAddr is 1), even with *split* siganalling.

Table 17. 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



The above packet is deprecated and has been superseded by the equivalent packet defined in the MMA/CAS extension (see Chapter 15)

The following packet is used for the request phase of a split load.

Table 18. Packet format for Split load - request phase

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



The above packet is deprecated and has been superseded by the equivalent packet defined in the MMA/CAS extension (see Chapter 15)

The following packet is used for the data phase of a split load and split atomic.

Table 19. Packet format for Split load - data phase

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

Field name	Bits	Description
data	*data_width_p*	Data

8.1.2. size field

The width of this field is 2 bits if max size is 64-bits (*data_width_p* < 128), 3 bits if wider.

8.1.3. diff field

Unlike instruction trace, compression options for data trace are somewhat limited. Following a synchronization instruction trace packet, the first data trace packet for a given access size must include the full (unencoded) data access address. 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).

Similarly, following a synchronization instruction trace packet, the first data trace packet for a given access size must include the full (unencoded) data value. Beyond this, data may be encoded or unencoded depending on whichever results in the most efficient represenation. 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. Encoded values are determined on a per access size basis. For example, when using XOR compression, the address of this data access, XOR the address of the previous data access of the same size.

If only one data compression type is offered, the **diff** field can be 1 bit wide rather than 2 for Table 17.

8.1.4. data_len field

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. In cases where **data** is not the final field in the packet, the width of **data** is indicated by this field.

8.2. Atomic

8.2.1. size field

Strictly, **size** could be just one bit as atomics are currently either 32 or 64 bits. Defining as per regular loads and stores provisions for future extensions (proprietary or otherwise) that support smaller atomics.

Table 20. 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 select other packet formats)

Field name	Bits	Description
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	2	00: Full address and data/operand (sync) 01: Differential address, XOR-compressed data/operand 10: Differential address, full data/operand 11: Differential address, differential data/operand
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



The above packet is deprecated and has been superseded by the equivalent packet defined in the MMA/CAS extension (see Chapter 15)

The following packet format is used when the encoder is configured to omit data values from data trace packets (trTeDataControl.trTeDataNoValue is 1), even with *split* siganalling.

Table 21. 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 select 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^{size} bytes
diff	1	0: Full address 1: Differential address
address	*daddress_width_p*	Address, aligned and encoded as per size



The above packet is deprecated and has been superseded by the equivalent packet defined in the MMA/CAS extension (see Chapter 15)

The following packet format is used when the encoder is configured to omit address values from data trace packets (trTeDataControl.trTeDataNoAddr is 1), and supports *unified* signalling.

Table 22. Packet format for Unified atomic with data and operand only

Field name	Bits	Description
format	3	Transaction type 110: Unified atomic or split atomic address (other codes select 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/operand (sync) 01: Compressed data/operand (XOR if 2 bits) 10: reserved 11: Differential data/operand
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



The above packet is deprecated and has been superseded by the equivalent packet defined in the MMA/CAS extension (see Chapter 15)

8.2.2. diff field

See Section 8.1.3.

8.2.3. operand field

The operand value for the atomic operation. 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.

8.2.4. data_len and op_len fields

Width of **data** and ***operand** fields respectively. See **Section 8.1.4.**

The following packet is used for the request phase of a split atomic:

Table 23. Packet format for Split atomic - request phase

Field name	Bits	Description
format	3	Transaction type 110: Unified atomic or split atomic address (other codes select other packet formats)

Field name	Bits	Description
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
lrid	*lrid_width_p*	Load request ID
diff	1 or 2	00: Full address and operand (sync) 01: Differential address, XOR-compressed operand 10: Differential address, full operand 11: Differential address, differential operand
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



The above packet is deprecated and has been superseded by the equivalent packet defined in the MMA/CAS extension (see Chapter 15)

The following packet format is used when the encoder is configured to omit address values from data trace packets (trTeDataControl.trTeDataNoAddr is 1).

Table 24. Packet format for Split atomic without address - request phase

Field name	Bits	Description
format	3	Transaction type 110: Unified atomic or split atomic address (other codes select 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
lrid	*lrid_width_p*	Load request ID
diff	1 or 2	00: Full address and operand (sync) 01: Differential address, XOR-compressed operand 10: Differential address, full operand 11: Differential address, differential operand
operand	*data_width_p*	Operand. Value from rs2 before operator applied



The above packet is deprecated and has been superseded by the equivalent packet defined in the MMA/CAS extension (see Chapter 15)

Table 25. Packet format for Split atomic load - data phase

Field name	Bits	Description
format	3	Transaction type 111: Split atomic data (other codes select 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)



The above packet is deprecated and has been superseded by the equivalent packet defined in the MMA/CAS extension (see Chapter 15)

8.3. CSR

Table 26. Packet format for Unified CSR, with address, data and operand

Field name	Bits	Description
format	3	Transaction type 101: CSR (other codes select 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/operand (XOR if 2 bits) 10: reserved 11 : Differential data/operand
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]



The above packet is deprecated and has been superseded by the equivalent packet defined in the MMA/CAS extension (see Chapter 15)

8.3.1. diff field

See Section 8.1.3.

8.3.2. operand field

See Section 8.2.3.

8.3.3. data_len and op_len fields

2 bits wide if hart has 32-bit CSRs, 3 bits if 64-bit. Width of **data** and **operand** fields respectively. See Section 8.1.4.

8.3.4. addr fields

The address is split into two parts, with the 6 LSBs output last as these are more likely to compress away.

Table 27. 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 select 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]



The above packet is deprecated and has been superseded by the equivalent packet defined in the MMA/CAS extension (see Chapter 15)

The following packet format is used when the encoder is configured to omit data values from data trace packets (trTeDataControl.trTeDataNoValue is 1).

Table 28. Packet format for Unified CSR, with address only

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	0: Full address 1: Differential address

Field name	Bits	Description
addr_msbs	6	Address[11:6]
addr_lsbs	6	Address[5:0]



The above packet is deprecated and has been superseded by the equivalent packet defined in the MMA/CAS extension (see Chapter 15)

Chapter 9. Reference Compressed Branch Trace Algorithm

The contents of this chapter are informative only.

A reference algorithm for compressed branch trace is given in Figure 1. 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';
- *trap. 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 **ctype** definition in RHTI Specification);
- *ppccd_br.* As above, but branch map not empty;
- *ntf.* Trace notify trigger (see **trigger** definition in RHTI Specification);
- *cci.* context change that can be reported imprecisely (see **ctype** definition in RHTI Specification);
- *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);
- *trep* Previous trap already reported with **thaddr** = 0 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);
- *resync2_br.* The resync FSM is in state 2 and there are entries in the branch map that have not yet been output (see Section 9.2).
- *resync3.* The resync FSM is in state 3 (see Section 9.2);

Figure 1 shows instruction by instruction behavior, as would be seen in a single-retirement system only. 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. Note that even with a single-retirement system it is possible to retire an instruction and report a trap simultaneously (**itype** = 1 or 2 and **iretire** = 1). In this case the flow diagram must be traversed twice, first for the retired instruction, and then for the trap.

A 3-stage pipeline within the encoder is assumed, such that the encoder has visibility of the current, previous and next instructions. 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.

Additionally, the encoder can generate one further packet type, not shown on the diagram for clarity. 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.

Note: if the **halted** or **reset** sideband signals are asserted (see "Optional sideband signals" section of RHTI Specification) 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*);

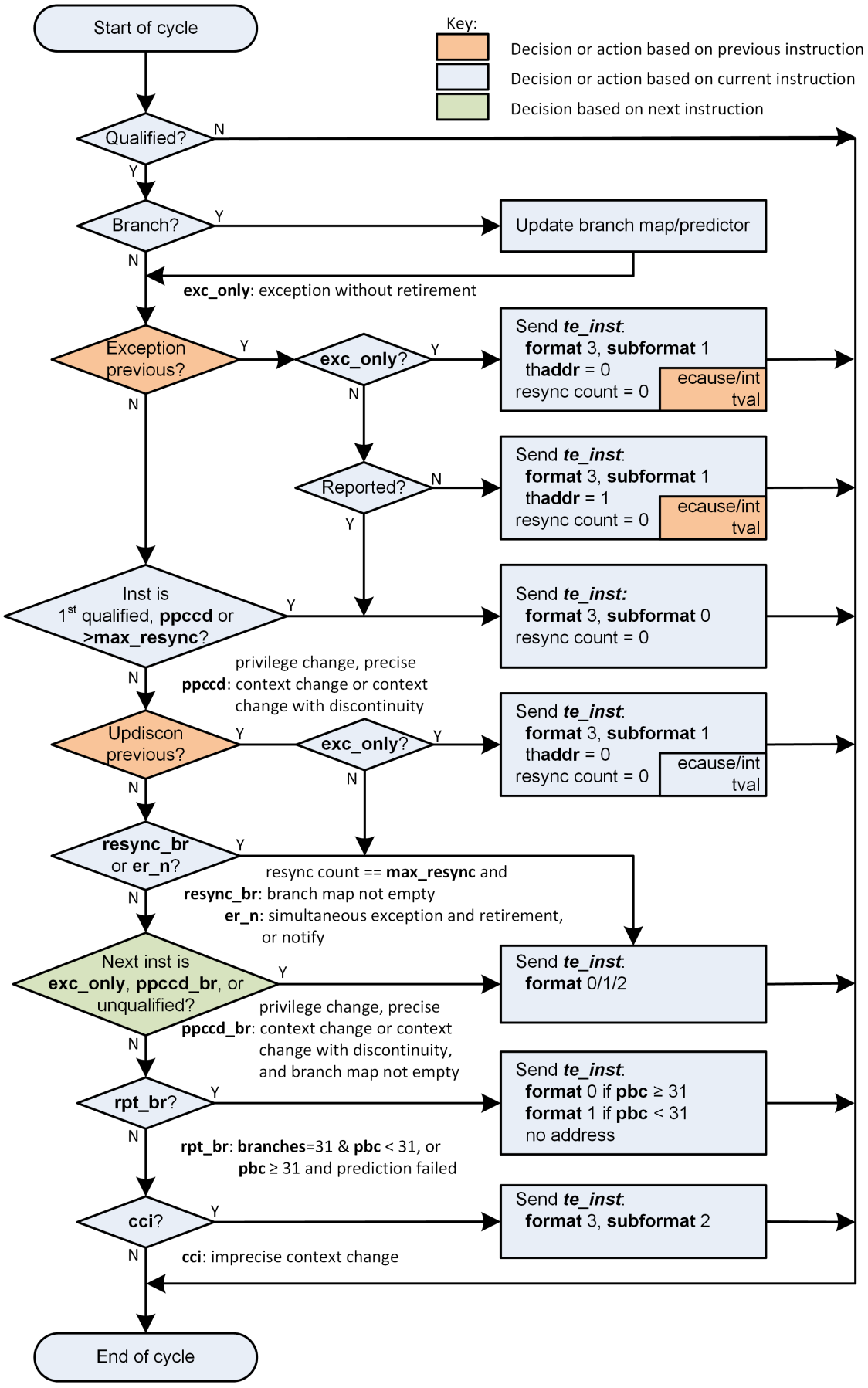


Figure 1. Instruction delta trace algorithm

9.1. Format selection

In all cases but two, the packet format is determined only by a 'yes' outcome from the associated decision.

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). 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. 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.

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.

Packet formats 0, 1 and 2 are organized so that the address is usually the final field. Minimizing the number of bits required to represent the address reduces the total packet size and significantly improves efficiency. See Chapter 7.

9.2. Resynchronisation

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.

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. 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. As such, before outputting a format 3 packet, it is necessary to output a format 0/1 packet for the preceding instruction if there are any unreported branches (because format 3 does not contain a branch map). There are several supported options for incrementing the resync timer (packets, cycle or instruction half-words), and as such updates to the timer do not necessarily coincide with instruction retirements. A small, independent FSM can be used to ensure the required packets are output in the required order, as follows:

- **State 1**: resync_count < max_resync. Transition to **State 2** when resync_count >= max_resync
- **State 2**: output packet of unreported branches if required, transition to **state 3**
- **State 3**: output sync packet, reset resync_count and transition to **State 1**.

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.

9.3. Multiple retirement considerations

As noted earlier in this section, for a single-retirement system the reference algorithm is applied to each retired instruction. 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 1 does not pass through any of the rectangular boxes).

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. 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.

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.

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).

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. 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)

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'. 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. == Parameters and Discovery

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 29 and Table 30.

Depending on the implementation, some parameters may be inherently fixed whilst others may be passed in to the design by some means.

Table 29. 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.

Parameter name	Range	Description
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
fOs_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 if compressed instructions are supported, 2 otherwise
iaddress_width_p		Width of instruction **address** bus. This is the same as *DXLEN*
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

Table 30. 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

Parameter name	Range	Description
sdata_width_p		Width of the **sdata** bus

9.4. Discovery of encoder parameters

To operate correctly, the decoder must be able to determine some of the encoder's parameters at runtime, in the form of discoverable attributes. 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). Table 31 lists the required discoverable attributes for instruction trace.

To access the discoverable attributes, some external entity, for example a debugger or a supervisory hart, must request it from the encoder. The encoder will provide the discovery information in one or more different formats. The preferred format is a packet which is sent over the trace infrastructure. Another format would be allowing the external entity to read the values from some register or memory mapped space maintained by the encoder. Section 9.5 gives an example of how this may be accomplished.

Table 31. 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
fOs_width	0	*fOs_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*

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. In particular, attributes related to filtering capabilities. Table 32 lists the attributes associated with the filtering recommendations discussed in Chapter 5, Table 33 lists attributes related to other instruction trace parameters mentioned in this document, and Table 34 lists attributes related to data trace.

Table 32. Optional filtering attributes

Name	Default	Parameter mapping
comparators	0	*comparators_p* - 1
filters	0	*filters_p* - 1
ecause_choice	5	*ecause_choice_p*

Name	Default	Parameter mapping
filter_context	1	*filter_context_p*
filter_time	1	*filter_time_p*
filter_excint	1	*filter_excint_p*
filter_privilege	1	*filter_privilegep*
filter_tval	1	*filter_tval_p*

Table 33. 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

Table 34. 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

9.5. Example ipxact description

This section provides an example of discovery information represented in the ipxact form.

```
<?xmlversion="1.0" encoding="UTF-8"?>
<ipxact:component
xmlns:ipxact="http://www.accellera.org/XMLSchema/IPXACT/1685-2014"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.accellera.org/XMLSchema/IPXACT/1685-2014
http://www.accellera.org/XMLSchema/IPXACT/1685-2014/index.xsd">
<ipxact:vendor>Siemens</ipxact:vendor>
<ipxact:library>TraceEncoder</ipxact:library>
<ipxact:name>TraceEncoder</ipxact:name>
<ipxact:version>0.8</ipxact:version>
<ipxact:memoryMaps>
```

```
<ipxact:memoryMap>
<ipxact:name>TraceEncoderRegisterMap</ipxact:name>
<ipxact:addressBlock>
<ipxact:name>>TraceEncoderRegisterAddressBlock</ipxact:name>
<ipxact:baseAddress>0</ipxact:baseAddress>
<ipxact:range>128</ipxact:range>
<ipxact:width>64</ipxact:width>

<ipxact:register>
<ipxact:name>discovery_info_0</ipxact:name>
<ipxact:addressOffset>'h0</ipxact:addressOffset>
<ipxact:size>64</ipxact:size>
<ipxact:access>read-only</ipxact:access>
<ipxact:field>
<ipxact:name>version</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>0</ipxact:bitOffset>
<ipxact:bitWidth>4</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>minor_revision</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>4</ipxact:bitOffset>
<ipxact:bitWidth>4</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>arch</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>8</ipxact:bitOffset>
<ipxact:bitWidth>4</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>bpred_size</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>12</ipxact:bitOffset>
<ipxact:bitWidth>4</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>cache_size</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>16</ipxact:bitOffset>
<ipxact:bitWidth>4</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>call_counter_size</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>20</ipxact:bitOffset>
<ipxact:bitWidth>3</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>comparators</ipxact:name>
```

```
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>23</ipxact:bitOffset>
<ipxact:bitWidth>3</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>context_type_width</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>26</ipxact:bitOffset>
<ipxact:bitWidth>5</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>context_width</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>31</ipxact:bitOffset>
<ipxact:bitWidth>5</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>ecause_choice</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>36</ipxact:bitOffset>
<ipxact:bitWidth>3</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>ecause_width</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>39</ipxact:bitOffset>
<ipxact:bitWidth>4</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>filters</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>43</ipxact:bitOffset>
<ipxact:bitWidth>4</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>filter_context</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>47</ipxact:bitOffset>
<ipxact:bitWidth>1</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>filter_excint</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>48</ipxact:bitOffset>
<ipxact:bitWidth>4</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>filter_privilege</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>52</ipxact:bitOffset>
<ipxact:bitWidth>1</ipxact:bitWidth>
```

```
</ipxact:field>
<ipxact:field>
<ipxact:name>filter_tval</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>53</ipxact:bitOffset>
<ipxact:bitWidth>1</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>filter_impdef</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>54</ipxact:bitOffset>
<ipxact:bitWidth>1</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>f0s_width</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>55</ipxact:bitOffset>
<ipxact:bitWidth>2</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>iaddress_lsb</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>57</ipxact:bitOffset>
<ipxact:bitWidth>2</ipxact:bitWidth>
</ipxact:field>
</ipxact:register>

<ipxact:register>
<ipxact:name>discovery_info_1</ipxact:name>
<ipxact:addressOffset>'h4</ipxact:addressOffset>
<ipxact:size>64</ipxact:size>
<ipxact:access>read-only</ipxact:access>
<ipxact:field>
<ipxact:name>iaddress_width</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>0</ipxact:bitOffset>
<ipxact:bitWidth>7</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>ilastsize_width</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>7</ipxact:bitOffset>
<ipxact:bitWidth>7</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>itype_width</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>14</ipxact:bitOffset>
<ipxact:bitWidth>7</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
```

```
<ipxact:name>iretire_width</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>21</ipxact:bitOffset>
<ipxact:bitWidth>7</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>nocontext</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>28</ipxact:bitOffset>
<ipxact:bitWidth>1</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>privilege_width</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>29</ipxact:bitOffset>
<ipxact:bitWidth>2</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>retires</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>31</ipxact:bitOffset>
<ipxact:bitWidth>3</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>return_stack_size</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>34</ipxact:bitOffset>
<ipxact:bitWidth>4</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>sijump</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>38</ipxact:bitOffset>
<ipxact:bitWidth>1</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>taken_branches</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>39</ipxact:bitOffset>
<ipxact:bitWidth>4</ipxact:bitWidth>
</ipxact:field>
<ipxact:field>
<ipxact:name>impdef_width</ipxact:name>
<ipxact:description>text</ipxact:description>
<ipxact:bitOffset>43</ipxact:bitOffset>
<ipxact:bitWidth>5</ipxact:bitWidth>
</ipxact:field>
</ipxact:register>

</ipxact:addressBlock>
<ipxact:addressUnitBits>8</ipxact:addressUnitBits>
```

```
</ipxact:memoryMap>
</ipxact:memoryMaps>
</ipxact:component>
```

Chapter 10. Decoder

Reference Python implementations of both the encoder and decoder can be found at github.com/riscv-non-isa/riscv-trace-spec.

10.1. Decoder pseudo code

```
# global variables
global        pc                          # Reconstructed program counter
global        last_pc                     # PC of previous instruction
global        branches = 0                # Number of branches to process
global        branch_map = 0              # Bit vector of not taken/taken (1/0)
status for branches
global bool   stop_at_last_branch = FALSE # Flag to indicate reconstruction is
to end at the final branch
global bool   inferred_address = FALSE    # Flag to indicate that reported
address from format 0/1/2 was
                                          # not following an uninferable jump
(and is therefore inferred)
global bool   start_of_trace = TRUE       # Flag indicating 1st trace packet
still to be processed
global        address                     # Reconstructed address from te_inst
packets
global        privilege                   # Privilege from te_inst packets
global        poss_next_epc               # Possible next exception address
global bool   pne_valid = FALSE           # Possible next exception address is
valid
global        options                     # Operating mode flags (sijump
onwards from support packet)
global array  return_stack                # Array holding return address stack
global        irstack_depth = 0           # Number of entries on the return
address stack traced
global        irets = 0                   # Number of implicit (i.e.
unreported) returns
global        popped_return_target        # Return address popped from the
stack
global bool   prt_valid = FALSE           # Popped return address is valid
global array  bpred                       # Structure holding branch predictor.
Each entry is 2 bools
global bool   use_bpred = FALSE           # Use branch predictor to determine
branch outcome
global array  cache                       # Array holding cache of jump target
addresses
global struct discovery                   # Structure containing fields from
discovery
global struct previous_te_inst            # Copy of previous te_inst packet

# Process te_inst packet.  Call each time a te_inst packet is received
###################################
function process_te_inst (te_inst)
```

```
  use_bpred = FALSE
  irets = 0
  if (te_inst.format == 3)
    if (te_inst.subformat == 3) # Support packet
      process_support(te_inst)
      return
    if (te_inst.subformat == 2) # Context packet
      return
    if (te_inst.subformat == 1) # Trap packet
      previous_te_inst = te_inst
      report_trap(te_inst)
      if (!te_inst.interrupt) # Exception
        report_epc(exception_address(te_inst))
      if (!te_inst.thaddr) # Trap only - nothing retired
        return

    inferred_address = FALSE
    pne_valid = FALSE
    address       = (te_inst.address << discovery.iaddress_lsb)
    if (te_inst.subformat == 1 or start_of_trace)
      branches    = 0
      branch_map  = 0
    if (is_branch(get_instr(address))) # 1 unprocessed branch if this
instruction is a branch
      branch_map = branch_map | (te_inst.branch << branches)
      branches++
    if (te_inst.subformat == 0 and !start_of_trace and
        !(previous_te_inst.format == 3 and previous_te_inst.subformat == 1 and
!previous_te_inst.thaddr))
      follow_execution_path(address, te_inst)
    else
      pc           = address
      report_pc(pc)
      last_pc      = pc # previous pc not known but ensures correct operation
for is_sequential_jump()
    if (start_of_trace or !options.resync_disabled)
      irstack_depth  = 0
      return_stack[0].count = 0
      bpred_init()
    if ((start_of_trace or !options.resync_disabled or te_inst.subformat == 1)
and is_call(get_instr(pc))
        push_return_stack(pc)
    privilege = te_inst.privilege
    start_of_trace = FALSE

  else # Not format 3
    if (start_of_trace) # This should not be possible!
      ERROR: Expecting trace to start with format 3
      abort
    if (includes_address(te_inst))
      if (options.full_address)
```

```
        address  = (te_inst.address << discovery.iaddress_lsb)
      else
        address += (te_inst.address << discovery.iaddress_lsb)
      stop_at_last_branch = is_branch(get_instr(address))
    else if (includes_cache_index(te_inst))
      address  = cache_lookup(te_inst.index)
      stop_at_last_branch = is_branch(get_instr(address))
    if (includes_bpred_count(te_inst))
      use_bpred = branches == 0
      stop_at_last_branch = (te_inst.branch_fmt == 0)
      branches += te_inst.branch_count + 31
      if (is_failed_bpred(te_inst))
        # Reporting branch following the number of correct predictions
indicated by te_inst.branch_count
        branches++
    else if (includes_branch_map(te_inst))
      stop_at_last_branch = stop_at_last_branch or (te_inst.branches == 0)
      # Branch map will contain <= 1 branch (1 if last reported instruction was
a branch)
      branch_map = branch_map | (te_inst.branch_map << branches)
      if (te_inst.branches == 0)
        branches += 31
      else
        branches += te_inst.branches

    follow_execution_path(address, te_inst)
    return

# Follow execution path to reported address
################################################################
function follow_execution_path(address, te_inst)

  local previous_address = pc
  local bool stop_here = FALSE
  local unproc_ir = FALSE
  while (TRUE)
    if (inferred_address) # iterate again from previously reported address to
find second occurrence
      stop_here = next_pc(previous_address, previous_te_inst)
      report_pc(pc)
      if (stop_here)
        inferred_address = FALSE
        irets = prt_valid ? 1 : 0
    else
      unproc_ir = unprocessed_implicit_returns(te_inst)
      stop_here = next_pc(address, te_inst)
      if (!is_call(get_instr(address)) # Set unproc_ir after popping stack but
before pushing
        unproc_ir = unprocessed_implicit_returns(te_inst)
      report_pc(pc)
      if (branches == 1 and is_branch(get_instr(pc)) and stop_at_last_branch)
        # Reached final branch - stop here (do not follow to next instruction
```

```
as we do not yet know
        # whether it retires)
        if (is_failed_bpred(te_inst))
          # Final branch failed prediction so set branch map accordingly
          branch_map[0] = bpred_taken(pc, te_inst)
        stop_at_last_branch = FALSE
        return
      if (stop_here)
        # Reached reported address following an uninferable discontinuity -
stop here
        if (unprocessed_branches(pc))
          ERROR: unprocessed branches
          abort
        return
      if (!stop_at_last_branch and pc == address and flag_notify(te_inst) and
      !unprocessed_branches(pc) and !unproc_ir)
        # All branches and implicit returns processed, and reached reported
address due to a notification,
        # not as an uninferable jump target.
        return
      if (!stop_at_last_branch and pc == address and
!is_uninferable_discon(get_instr(last_pc) and
          !unprocessed_branches(pc) and !unproc_ir and !flag_updiscon(te_inst)
and te_inst.format != 3))
        # All branches and implicit returns processed, and reached reported
address, but not as an
        # uninferable jump target.  Stop here for now, though flag indicates
this may not be final retired
        # instruction
        previous_te_inst = te_inst
        inferred_address = TRUE
        return
      if (te_inst.format == 3 and pc == address and !unprocessed_branches(pc)
and
          (te_inst.privilege == privilege or
is_return_from_trap(get_instr(last_pc))))
        # All branches processed, and reached reported address
        return

# Compute next PC
################################################################################
##########
function next_pc (address, te_inst)

  local instr   = get_instr(pc)
  local this_pc = pc
  local bool stop_here = FALSE
  local bool exception = te_inst.format == 3 and te_inst.subformat == 1

  if (is_inferrable_jump(instr))
    pc += instr.imm
  else if (is_sequential_jump(instr, last_pc)) # lui/auipc followed by jump
```

```
using same register
    pc = sequential_jump_target(pc, last_pc)
  else if (is_implicit_return(address, instr, te_inst))
    pc = popped_return_target
    if (!exception)
      cache_update(pc)
      if (includes_cache_index(te_inst) and te_inst.index == get_index(pc,
"cache"))
        address = pc
        stop_at_last_branch = is_branch(get_instr(address))

  else if (is_uninferable_discon(instr))
    if (stop_at_last_branch and !is_branch(address))
      ERROR: unexpected uninferable discontinuity
      abort
    else
      pc = address
      if (!exception)
        if (!flag_irreport(te_inst) or address != popped_return_target)
          # Update cache unless address is correctly predicted but reported for
some other reason
          cache_update(pc)
      stop_here = TRUE
  else if (is_branch(instr))
    if (!exception)
      irets = 0
    pc += is_taken_branch(pc, te_inst) ? instr.imm : instruction_size(instr)
  else
    pc += instruction_size(instr)

  if (!exception)
    if (is_call(get_instr(pc)))
      push_return_stack(pc)
    prt_valid = FALSE
    if (is_return(get_instr(pc)) and irstack_depth > 0)
      popped_return_target = pop_return_stack()
      prt_valid = TRUE

  last_pc = this_pc

  return stop_here

# Process support packet.  Call each time a support packet is received
###################################
function process_support (te_inst)

  local bool stop_here = FALSE

  options = te_inst.options
  pne_valid = FALSE
  if (te_inst.qual_status != no_change)
    start_of_trace = TRUE # Trace ended, so get ready to start again
```

```
  if (te_inst.qual_status == ended_ntr and inferred_address)
    local previous_address = pc
    inferred_address       = FALSE
    while (TRUE)
      stop_here = next_pc(previous_address, previous_te_inst)
      report_pc(pc)
      if (stop_here)
        return

  return

# Packet includes an address (formats 0, 1, 2)
################################################################
function includes_address (te_inst)

  if (te_inst.format == 2 or
      (te_inst.format == 1 and te_inst.branches != 0) or
      (includes_bpred_count(te_inst) and te_inst.branch_fmt != 0))
    return TRUE

  return FALSE

# Packet includes branch map
#########################################################################
function includes_branch_map (te_inst)

  if (te_inst.format == 1 or (includes_cache_index(te_inst) and
te_inst.branches > 0))
    return TRUE

  return FALSE

# Packet includes branch predictor count
#################################################################
function includes_bpred_count (te_inst)

  if (te_inst.format == 0 and options.branch_predictor and
      (!includes_f0sub(te_inst) or te_inst.subformat == 0))
    return TRUE

  return FALSE

# Packet includes jump target cache index
#################################################################
function includes_cache_index (te_inst)

  if (te_inst.format == 0 and options.jump_target_cache and
      (!includes_f0sub(te_inst) or te_inst.subformat == (discovery.bpred_size
== 0 ? 0 : 1)))
    return TRUE

  return FALSE
```

```
# Format 0 packet includes optional sub-format field
#######################################################
function includes_f0sub (te_inst)

  if (te_inst.format != 0 or (f0subformats() < 2))
    return FALSE

  return TRUE

# Return the number of supported format 0 sub-formats
#######################################################
function f0subformats ()

  local bpred_present          = discovery.bpred_size > 0 ? 1 : 0
  local cache_present          = discovery.cache_size > 0 ? 1 : 0

  return bpred_present + cache_present

# Determine if instruction is a branch, adjust branch count/map, and return
taken status ##################
function is_taken_branch (address, te_inst)

  local      instr = get_instr(address)
  local bool taken = FALSE

  if (branches == 0)
    ERROR: cannot resolve branch
    abort
  else
    if (use_bpred)
      taken = bpred_taken(address, te_inst)
    else
      taken = !branch_map[0]
      branch_map >> 1
      use_bpred = includes_bpred_count(te_inst)
    bpred_update(address, taken)
    branches--

  return taken

# Determine if instruction is a branch
#########################################################################
function is_branch (instr)

  if ((instr.opcode == BEQ)    or
      (instr.opcode == BNE)    or
      (instr.opcode == BLT)    or
      (instr.opcode == BGE)    or
      (instr.opcode == BLTU)   or
      (instr.opcode == BGEU)   or
      (instr.opcode == C.BEQZ) or
      (instr.opcode == C.BNEZ))
    return TRUE
```

```
  return FALSE

# Determine if branch prediction failed
#####################################################################
function is_failed_bpred(te_inst)

  return (includes_bpred_count(te_inst) and (te_inst.branch_fmt == 0 or
te_inst.branch_fmt == 3))

# Determine if instruction is an inferrable jump
############################################################
function is_inferrable_jump (instr)

  if ((instr.opcode == JAL)   or
      (instr.opcode == C.JAL) or
      (instr.opcode == C.J)   or
      (instr.opcode == JALR and instr.rs1 == 0))
    return TRUE

  return FALSE

# Determine if instruction is an uninferable jump
############################################################
function is_uninferable_jump (instr)

  if ((instr.opcode == JALR and instr.rs1 != 0) or
      (instr.opcode == C.JALR)                  or
      (instr.opcode == C.JR))
    return TRUE

  return FALSE

# Determine if instruction is a return from trap
############################################################
function is_return_from_trap (instr)

  if ((instr.opcode == SRET)      or
      (instr.opcode == MRET))
    return TRUE

  return FALSE

# Determine if instruction is an uninferable discontinuity
################################################
function is_uninferable_discon (instr)

  if (is_uninferable_jump(instr) or
      is_return_from_trap (instr) or
      (instr.opcode == ECALL)     or
      (instr.opcode == EBREAK)    or
      (instr.opcode == C.EBREAK))
    return TRUE
```

```
  return FALSE

# Determine if instruction is a sequentially inferrable jump
################################################
function is_sequential_jump (instr, prev_addr)

  if (not (is_uninferable_jump(instr) and options.sijump))
    return FALSE

  local prev_instr = get_instr(prev_addr)

  if((prev_instr.opcode == AUIPC) or
     (prev_instr.opcode == LUI)   or
     (prev_instr.opcode == C.LUI))
    return (instr.rs1 == prev_instr.rd)

  return FALSE

# Find the target of a sequentially inferrable jump
#########################################################
function sequential_jump_target (addr, prev_addr)

  local instr      = get_instr(addr)
  local prev_instr = get_instr(prev_addr)
  local target     = 0

  if (prev_instr.opcode == AUIPC)
    target = prev_addr
  target += prev_instr.imm
  if (instr.opcode == JALR)
    target += instr.imm

  return target

# Determine if instruction is a call
#########################################################################
# - excludes tail calls as they do not push an address onto the return stack
function is_call (instr)

  if ((instr.opcode == JALR   and instr.rd  == 1 and instr.rs1 != 5) or
      (instr.opcode == JALR   and instr.rd  == 5 and instr.rs1 != 1) or
      (instr.opcode == C.JALR and instr.rs1 != 5)                    or
      (instr.opcode == JAL    and (instr.rd == 1 or instr.rd  == 5)) or
      (instr.opcode == C.JAL)                                        or
      (instr.opcode == CMJALT))
    return TRUE

  return FALSE

# Determine if instruction is a return
#########################################################################
function is_return (instr)
```

```
  if ((instr.opcode == JALR and (instr.rs1 == 1 or instr.rs1 == 5) and instr.rd
!= 1 and instr.rd != 5) or
      (instr.opcode == C.JR and (instr.rs1 == 1 or instr.rs1 == 5)) or
      (instr.opcode == CM.POPRET))
    return TRUE

  return FALSE

# Determine if instruction return address can be implicitly inferred
#####################################
function is_implicit_return (address, instr, te_inst)

  if (!options.implicit_return) # Implicit return mode disabled
    return FALSE

  if (is_return (instr))
    if (flag_irreport(te_inst) and !unprocessed_branches(address) and
te_inst.irets == irets - 1)
      return FALSE
    return prt_valid

  return FALSE

# Check whether notify is present and return flag state
#################################################
function flag_notify (te_inst)

  if (te_inst.format == 3 or !includes_address(te_inst))
    return FALSE

  return (te_inst.notify != get_preceding_bit(te_inst, "notify"))

# Check whether updiscon is present and return flag state
################################################
function flag_updiscon (te_inst)

  if (te_inst.format == 3 or !includes_address(te_inst))
    return FALSE

  return (te_inst.updiscon != get_preceding_bit(te_inst, "updiscon"))

# Check whether irreport is present and return flag state
################################################
function flag_irreport (te_inst)

  if (te_inst.format == 3 or !(includes_address(te_inst) or
includes_cache_index(te_inst)))
    return FALSE

  return (te_inst.irreport != get_preceding_bit(te_inst, "irreport"))

# Check for unprocessed branches
#############################################################################
```

```
function unprocessed_branches (address)
  # Check all branches processed (except 1 if this instruction is a branch)
  return (branches != (is_branch(get_instr(address)) ? 1 : 0))

# Check for unprocessed implicit returns
######################################################################
function unprocessed_implicit_returns (te_inst)

  return (irstack_depth > 0 and flag_irreport(te_inst) and te_inst.irets >
irets)

# Push address onto return stack
##############################################################################
function push_return_stack (address)

  if (!options.implicit_return) # Implicit return mode disabled
    return

  local irstack_depth_max = discovery.return_stack_size  ?
                            2**discovery.return_stack_size :
                            2**discovery.call_counter_size
  local call_counter_max  = discovery.return_stack_size  ?
                            2**discovery.call_counter_size - 1 : 0
  local instr             = get_instr(address)
  local link              = address + instruction_size(instr)

  if (irstack_depth > 0 and
      return_stack[irstack_depth-1].link  == link and
      return_stack[irstack_depth-1].count != call_counter_max)
    # Recursive call and each stack entry has an associated counter - increase
count for top of stack
    return_stack[irstack_depth-1].count++
  else # Push new entry onto the stack
    if (irstack_depth == irstack_depth_max)
      # Delete oldest entry from stack to make room for new entry added below
      irstack_depth--
      for (i = 0; i < irstack_depth; i++)
        return_stack[i] = return_stack[i+1]

    return_stack[irstack_depth].link  = link
    return_stack[irstack_depth].count = 0
    irstack_depth++

  return

# Pop address from return stack
##############################################################################
function pop_return_stack ()

  local link

  link = peek_return_stack()
  if (return_stack[irstack_depth-1].count > 0)
```

```
    return_stack[irstack_depth-1].count--
  else
    irstack_depth-- # function not called if irstack_depth is 0, so no need to
check for underflow

  irets++

  return link

# Return address from top of return stack
#################################################################
function peek_return_stack ()

  return return_stack[irstack_depth-1].link

# Return the address of an exception
#####################################################################
function exception_address (te_inst)

  local instr = get_instr(pc)
  local epc = poss_next_epc

  if (start_of_trace or is_uninferable_discon(instr))
    if (te_inst.thaddr) # This should not be possible!
      ERROR: Expecting EPC address in te_inst
      abort

    return (te_inst.address << discovery.iaddress_lsb)

  poss_next_epc = te_inst.address << discovery.iaddress_lsb
  if (pne_valid)
    pne_valid = !te_inst.thaddr
    return epc
  pne_valid = !te_inst.thaddr

  if ((instr.opcode == ECALL) or (instr.opcode == EBREAK) or (instr.opcode ==
C.EBREAK))
    return pc
  if (!te_inst.thaddr) # pne_valid is false.  te_inst address is EPC of current
trap, not next trap
    return (te_inst.address << discovery_response.iaddress_lsb)

  next_pc(pc, te_inst)
  return pc

# Report ecause and tval (user to populate if desired)
#####################################################
function report_trap (te_inst)

  return

# Report program counter value (user to populate if desired)
############################################
```

```
function report_pc (address)

  return

# Report exception program counter value (user to populate if desired)
####################################
function report_epc (address)

  return

# Initialize branch predictor
################################################################################
function bpred_init ()

  if (discovery.bpred_size == 0)
    return

  for(i = 0; i < 2** discovery.bpred_size; i++)
    bpred[i][1] = FALSE
    bpred[i][0] = TRUE
  return

# Update Branch predictor
################################################################################
##
function bpred_update (address, taken)

  local index

  if (!options.branch_predictor)
    return

  # MSB is predicted next outcome, and changes after 2 failed predictions.  LSB
is current outcome
  index = get_index(address, "bpred")
  if (bpred[index][1] != bpred[index][0])
    bpred[index][1] = taken
  bpred[index][0] = taken

  return

# Return the predicted branch outcome
####################################################################
function bpred_taken (address, te_inst)

  local index
  local taken

  if (!options.branch_predictor) # This should not be possible
    ERROR: Can't predict branch outcome when branch predictor disabled
    abort

  index = get_index(address, "bpred")
```

```
  taken = bpred[index][1]
  if (branches == 1 and te_inst.branch_fmt == 3)
    # Prediction failed on final branch
    taken = !taken

  return taken

# Update jump target cache
################################################################################
#
function cache_update (address)

  local index

  if (options.jump_target_cache)
    index = get_index(address, "cache")
    cache[index] = address

# Look up address in jump target cache
########################################################################
function cache_lookup (index)

  if (!options.jump_target_cache) # This should not be possible
    ERROR: Can't look up cache index when cache is disabled
    abort

  return cache[index]

# Extract lookup index from address
########################################################################
function get_index (address, type)

  local size

  if (type == "bpred")
    size = discovery.bpred_size
  else
    size = discovery.cache_size

  # Index into predictor is bpred_size or cache_size bits of the address,
starting from iaddress_lsb
  return (address >> discovery.iaddress_lsb) & ((1 << size) - 1)
```

Chapter 11. Example code and packets

In the following examples ***ret*** is referred to as uninferable, this is only true if implicit-return mode is off

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:   ...
```

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>
```

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:   ....
```

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]

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 12. Code fragment and transport

This section shows fragments of code, and associated data from one of the architectural tests in the repository. For the individual fragments the ingress signals are shown and the corresponding packets generated. It further shows how the packets are transported via on-chip transport fabric. The fragments shown below are extracted from the test whilst it is being executed. In order to give some context to the fragment of interest, code prior to and after the fragment is also given.

12.1. Illegal Opcode test

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.

12.1.1. Code fragment

```
1:
******************************************************************************
******
2: ****************** Fragment 0x80000222 - 0x80000226:illegal_opcode
******************
3:
******************************************************************************
******
4:   KEY: ">" means pre-fragment execution, "<" means post-fragment execution
5: ^^^^^^^^^^^^^^^^^^^^^^^^^ Part 1 of 1 ^^^^^^^^^^^^^^^^^^^^^^^^^
6:
7: elf:
8:  >  0000000080000104 <j_exception_stimulus>:
9:  >      80000104:    00000297            auipc   t0,0x0
10: >      80000108:    11e28293            addi    t0,t0,286 # 80000222
<bad_opcode>
11: >      8000010c:    8282                    jr  t0
12: >      80000154:    9282                    jalr    t0
13:     0000000080000222 <bad_opcode>:
14:         80000222:   0000                    unimp
15:         80000224:   0000                    unimp
16:         80000226:   b709                    j   80000128
<j_target_end_fail>
17: <  00000000800001b0 <machine_trap_entry>:
18: <      800001b0:    a805                    j   800001e0
<machine_trap_entry_0>
19: <  00000000800001e0 <machine_trap_entry_0>:
20: <      800001e0:    342023f3            csrr    t2,mcause
21: <      800001e4:    fff0031b            addiw   t1,zero,-1
22: <      800001e8:    137e                    slli    t1,t1,0x3f
23:
24: trace_spike:
```

```
25: ******** Data from br_j_asm.spike_pc_trace line 5029 ********
26: >  ADDRESS=80000154, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
27: >  ADDRESS=80000104, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
28: >  ADDRESS=80000108, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
29: >  ADDRESS=8000010c, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
30:    ADDRESS=80000222, PRIVILEGE=3, EXCEPTION=1, ECAUSE=2, TVAL=0,
INTERRUPT=0
31: <  ADDRESS=800001b0, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
32: <  ADDRESS=800001e0, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
33: <  ADDRESS=800001e4, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
34: <  ADDRESS=800001e8, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
35:
36: encoder_input:
37: ******** Data from br_j_asm.encoder_input line 5029 ********
38: >  UNINFERABLE_JUMP, cause=0, tval=0, priv=3, iaddr_0=80000154, context=0,
ctype=0, ilastsize_0=2
39: >  ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=80000104, context=0,
ctype=0, ilastsize_0=4
40: >  ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=80000108, context=0,
ctype=0, ilastsize_0=4
41: >  UNINFERABLE_JUMP, cause=0, tval=0, priv=3, iaddr_0=8000010c, context=0,
ctype=0, ilastsize_0=2
42:    EXCEPTION, cause=2, tval=0, priv=3, iaddr_0=80000222, context=0,
ctype=0, ilastsize_0=2,

----------> NOT RETIRED
43: <  ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=800001b0, context=0,
ctype=0, ilastsize_0=2
44: <  ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=800001e0, context=0,
ctype=0, ilastsize_0=4
45: <  ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=800001e4, context=0,
ctype=0, ilastsize_0=4
46: <  ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=800001e8, context=0,
ctype=0, ilastsize_0=2
47:
48: te_inst:
49: ******** Data from br_j_asm.te_inst_annotated line 5071 ********
50: >  next=80000154 curr=80000150 prev=8000014c
51: >  next=80000104 curr=80000154 prev=80000150
52: >  next=80000108 curr=80000104 prev=80000154
53: >  format=1, address=80000104, branches=1, branch_map=0, irreport=0,
notify=0, updiscon=0,
                                                  Reason[prev_updiscon]
```

```
Payload[05 04 01 00 80 00]
54: >  next=8000010c curr=80000108 prev=80000104
55:    next=80000222 curr=8000010c prev=80000108
56:    format=2, address=8000010c, irreport=0, notify=0, updiscon=0,
Reason[exc_only]
                                                    Payload[32 04 00 00 02]
57: <  next=800001b0 curr=80000222 prev=8000010c
58: <  format=3, subformat=TRAP, address=80000222, branch=1, context=0,
ecause=2, interrupt=0,
                                                 privilege=3, thaddr=0,
tval=0, Reason[prev_updiscon, curr_exc_only]
                                                    Payload[77 00 00 00 00 81
88 00 00 20]
59: <  format=3, subformat=START, address=800001b0, branch=1, context=0,
                                              privilege=3,
Reason[exception_prev, reported]
                                                 Payload[73 00 00 00 00 6c 00
00 10]
60: <  next=800001e4 curr=800001e0 prev=800001b0
61: <  next=800001e8 curr=800001e4 prev=800001e0
```

12.1.2. Packet data

The output from the encoder for the fragment of interest is given in line 56. The least significant byte is output first, this means 32 is byte 0, 04 is byte 1 and and the final value 02 is byte 4.

12.1.3. Siemens transport

The packet is encapsulated according to the Unformatted Trace & Diagnostic Data Packet Encapsulation for RISC-V Specification, with the following attributes:

- Header - 1 byte
- SrcID - N bits. As an example use 6 bits and the value of 1.
- This example has no timestamp
- A 2-bit type field with '10' meaning instruction trace
- trace_payload - [0x32 0x04 0x00 0x00 0x02]

Since the Siemens transport is byte stream based the data seen will be:

```
[0x06][0x81][0x32 0x04 0x00 0x00 0x02]
```

12.1.4. ATB transport

The packet is encapsulated according to the Unformatted Trace & Diagnostic Data Packet Encapsulation for RISC-V Specification, with the following attributes:

- Header - 1 byte carried via ATDATA bus.
- SrcID - 0 bits as the SrcID is carried via the ATID bus.

- This example has no timestamp
- No type field (encoder has no data trace support)
- trace_payload - [0x32 0x04 0x00 0x00 0x02] carried via ATDATA bus.

Assuming a 32 bit ATB transport results in the following ATB transfers

```
[ATID=1] [ATBYTES = 3] [ATDATA = 0x00043205]
[ATID=1] [ATBYTES = 1] [ATDATA = 0x00000200]
```

12.2. Timer Long Loop

12.2.1. Code fragment

```
1:
*******************************************************************************
*******
2: ****************** Fragment 0x800001a2 - 0x800001b0:timer_long_loop
******************
3:
*******************************************************************************
*******
4:   KEY: ">" means pre-fragment execution, "<" means post-fragment execution
5: ^^^^^^^^^^^^^^^^^^^^^^^^^ Part 443 of 445 ^^^^^^^^^^^^^^^^^^^^^^^^^
6:
7: elf:
8: >      80000194: fab50ce3           beq a0,a1,8000014c
<timer_interrupt_return>
9: >      80000198: 40430333           sub t1,t1,tp
10: >      8000019c:    34402473           csrr    s0,mip
11: >      800001a0:    8c21                   xor s0,s0,s0
12:         800001a2:   300024f3           csrr    s1,mstatus
13:         800001a6:   8ca5                   xor s1,s1,s1
14:         800001a8:   fe0310e3           bnez    t1,80000188
<timer_interrupt_long_loop>
15:         800001ac:   bfb5                   j   80000128
<j_target_end_fail>
16:         800001ae:   0001                   nop
17:    00000000800001b0 <machine_trap_entry>:
18:         800001b0:   a805                   j   800001e0
<machine_trap_entry_0>
19: <  00000000800001e0 <machine_trap_entry_0>:
20: <      800001e0:    342023f3           csrr    t2,mcause
21: <      800001e4:    fff0031b           addiw   t1,zero,-1
22: <      800001e8:    137e                   slli    t1,t1,0x3f
23: <      800001ea:    031d                   addi    t1,t1,7
24:
25: trace_spike:
26: ******** Data from br_j_asm.spike_pc_trace line 5000 ********
27: >  ADDRESS=80000194, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
```

```
INTERRUPT=0
28: >  ADDRESS=80000198, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
29: >  ADDRESS=8000019c, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
30: >  ADDRESS=800001a0, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
31:    ADDRESS=800001a2, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
32:    ADDRESS=800001a6, PRIVILEGE=3, EXCEPTION=1, ECAUSE=8000000000000007,
TVAL=0, INTERRUPT=1
33:    ADDRESS=800001b0, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
34: <  ADDRESS=800001e0, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
35: <  ADDRESS=800001e4, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
36: <  ADDRESS=800001e8, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
37: <  ADDRESS=800001ea, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
38:
39: encoder_input:
40: ******** Data from br_j_asm.encoder_input line 5000 ********
41: >  NONTAKEN_BRANCH, cause=0, tval=0, priv=3, iaddr_0=80000194, context=0,
ctype=0, ilastsize_0=4
42: >  ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=80000198, context=0,
ctype=0, ilastsize_0=4
43: >  ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=8000019c, context=0,
ctype=0, ilastsize_0=4
44: >  ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=800001a0, context=0,
ctype=0, ilastsize_0=2
45:    ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=800001a2, context=0,
ctype=0, ilastsize_0=4
46:    INTERRUPT, cause=7, tval=0, priv=3, iaddr_0=800001a6, context=0,
ctype=0, ilastsize_0=2,

----------> NOT RETIRED
47:    ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=800001b0, context=0,
ctype=0, ilastsize_0=2
48: <  ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=800001e0, context=0,
ctype=0, ilastsize_0=4
49: <  ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=800001e4, context=0,
ctype=0, ilastsize_0=4
50: <  ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=800001e8, context=0,
ctype=0, ilastsize_0=2
51: <  ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=800001ea, context=0,
ctype=0, ilastsize_0=2
52:
53: te_inst:
54: ******** Data from br_j_asm.te_inst_annotated line 5038 ********
```

```
55: >  next=80000194 curr=80000192 prev=80000190
56: >  next=80000198 curr=80000194 prev=80000192
57: >  next=8000019c curr=80000198 prev=80000194
58: >  next=800001a0 curr=8000019c prev=80000198
59:    next=800001a2 curr=800001a0 prev=8000019c
60:    next=800001a6 curr=800001a2 prev=800001a0
61:    format=1, address=800001a2, branches=15, branch_map=21845, irreport=0,
notify=0, updiscon=0,
                                                  Reason[exc_only] Payload[bd
aa aa 68 00 00 20]
62:    next=800001b0 curr=800001a6 prev=800001a2
63: <  next=800001e0 curr=800001b0 prev=800001a6
64: <  format=3, subformat=TRAP, address=800001b0, branch=1, context=0,
ecause=7, interrupt=1,
                                                  privilege=3, thaddr=1,
Reason[prev_exception]
                                                     Payload[77 00 00 00 80 33
6c 00 00 20]
65: <  next=800001e4 curr=800001e0 prev=800001b0
66: <  next=800001e8 curr=800001e4 prev=800001e0
67: <  next=800001ea curr=800001e8 prev=800001e4
```

12.2.2. Packet data

The output from the encoder for the fragment of interest is given in line 61. The least significant byte is output first, this means 77 is byte 0, 00 is byte 1 and and the final value 20 is byte 9.

12.2.3. Siemens transport

The packet is encapsulated according to the Trace & Diagnostic Data Packet Encapsulation for RISC-V Specification, with the following attributes:

- Header - 1 byte
- SrcID - N bits. As an example use 6 bits and the value of A.
- This example has no timestamp
- A 2-bit type field with '10' meaning instruction trace
- trace_payload - [0xBD 0xAA 0xAA 0x68 0x00 0x00 0x20]

```
[0x8][0x8A][0xBD 0xAA 0xAA 0x68 0x00 0x00 0x20]
```

12.2.4. ATB transport

The packet is encapsulated according to the Unformatted Trace & Diagnostic Data Packet Encapsulation for RISC-V Specification, with the following attributes:

- Header - 1 byte carried via the ATDATA bus.
- SrcID - 0 bits as the SrcID is carried via the ATID bus.
- This example has no timestamp

- No type field (encoder has no data trace support)
- trace_payload - [0xBD 0xAA 0xAA 0x68 0x00 0x00 0x20] carried via the ATDATA bus

Assuming at 32 bit ATB transport results in the following ATB transfers

```
[ATID=0xA] [ATBYTES = 3] [ATDATA = 0xAAAABD07]
[ATID=0xA] [ATBYTES = 3] [ATDATA = 0x20000068]
```

12.3. Startup xrle

12.3.1. Code fragment

```
1:
**********************************************************************************
****
2: ****************** Fragment 0x20010522 - 0x20010528:startup_xrle
******************
3:
**********************************************************************************
****
4:   KEY: ">" means pre-fragment execution, "<" means post-fragment execution
5: ^^^^^^^^^^^^^^^^^^^^^^^^^ Part 1 of 1 ^^^^^^^^^^^^^^^^^^^^^^^^^
6:
7: elf:
8:     20010522 <main>:
9:     20010522:    1141                    addi    sp,sp,-16
10:    20010524:    c606                    sw  ra,12(sp)
11:    20010526:    c422                    sw  s0,8(sp)
12:    20010528:    0800                    addi    s0,sp,16
13: <  2001052a:    800107b7            lui a5,0x80010
14: <  2001052e:    6721                    lui a4,0x8
15: <  20010530:    e8670713            addi    a4,a4,-378 # 7e86
<__heap_size+0x7686>
16: <  20010534:    1ae7aa23            sw  a4,436(a5) # 800101b4
<_sp+0xfffffbfc>
17:
18: trace_spike:
19: ******** Data from xrle.spike_pc_trace line 2 ********
20:    ADDRESS=20010522, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
21:    ADDRESS=20010524, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
22:    ADDRESS=20010526, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
23:    ADDRESS=20010528, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
24: <  ADDRESS=2001052a, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
25: <  ADDRESS=2001052e, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
```

```
26: <  ADDRESS=20010530, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
27: <  ADDRESS=20010534, PRIVILEGE=3, EXCEPTION=0, ECAUSE=0, TVAL=0,
INTERRUPT=0
28:
29: encoder_input:
30: ******** Data from xrle.encoder_input line 2 ********
31:    ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=20010522, context=0,
ctype=0, ilastsize_0=2
32:    ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=20010524, context=0,
ctype=0, ilastsize_0=2
33:    ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=20010526, context=0,
ctype=0, ilastsize_0=2
34:    ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=20010528, context=0,
ctype=0, ilastsize_0=2
35: <  ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=2001052a, context=0,
ctype=0, ilastsize_0=4
36: <  ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=2001052e, context=0,
ctype=0, ilastsize_0=2
37: <  ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=20010530, context=0,
ctype=0, ilastsize_0=4
38: <  ITYPE_NONE, cause=0, tval=0, priv=3, iaddr_0=20010534, context=0,
ctype=0, ilastsize_0=4
39:
40: te_inst:
41: ******** Data from xrle.te_inst_annotated line 2 ********
42: >  format=3, subformat=SUPPORT, enable=1, encoder_mode=0, options=4,
qual_status=0 Payload[1f 04]
43:    next=20010522
44:    next=20010524 curr=20010522
45:    format=3, subformat=START, address=20010522, branch=1, context=0,
                                            privilege=3, Reason[ppccd]
                                                 Payload[73 00 00 00 00 91 82
00 10]
46:    next=20010526 curr=20010524 prev=20010522
47:    next=20010528 curr=20010526 prev=20010524
48: <  next=2001052a curr=20010528 prev=20010526
49: <  next=2001052e curr=2001052a prev=20010528
50: <  next=20010530 curr=2001052e prev=2001052a
51: <  next=20010534 curr=20010530 prev=2001052e
```

12.3.2. Packet data

The output from the encoder for the fragment of interest is given in line 45. The least significant byte is output first, this means 73 is byte 0, 00 is byte 1 and and the final value 10 is byte 8.

12.3.3. Siemens transport

The packet is encapsulated according to the Unformatted Trace & Diagnostic Data Packet Encapsulation for RISC-V Specification, with the following attributes:

- Header - 1 byte
- SrcID - N bits. As an example use 6 bits and the value of 5.
- This example has no timestamp
- A 2-bit type field with '10' meaning instruction trace
- trace_payload - [0x73 0x00 0x00 0x00 0x00 0x91 0x82 0x00 0x10]

```
[0xA][0x85][0x73 0x00 0x00 0x00 0x00 0x91 0x82 0x00 0x10]
```

12.3.4. ATB transport

The packet is encapsulated according to the Unformatted Trace & Diagnostic Data Packet Encapsulation for RISC-V Specification, with the following attributes:

- Header - 1 byte carried via the ATDATA bus.
- SrcID - 0 bits as the SrcID is carried via the ATID bus.
- This example has no timestamp
- No type field (encoder has no data trace support)
- trace_payload - [0x73 0x00 0x00 0x00 0x00 0x91 0x82 0x00 0x10] carried via the ATDATA bus.

Assuming at 32 bit ATB transport results in the following ATB transfers

```
[ATID=0x5] [ATBYTES = 3] [ATDATA = 0x00007309]
[ATID=0x5] [ATBYTES = 3] [ATDATA=0x82910000]
[ATID=0x5] [ATBYTES = 1] [ATDATA = 0x00001000]
```

Chapter 13. Standard Support Packet v1.0 extension to Efficient Trace for RISC-V v2.0



This chapter is in the Ratified state.

No changes are allowed. Any desired or needed changes can be the subject of a follow-on new extension. Ratified extensions are never revised.

The Efficient Trace for RISC-V v2.0 specification defines a support packet (see Section 7.5) and when it must be sent. The general requirements for the content of this packet are described, but the exact details of the payload is left to the implementor. The original thinking was that this left flexibility for different encoder implementations to provide support information best suited to their specific capabilities, as well as allowing implementation-specific customizations.

However, this flexibility presents difficulties for applications such as Self-hosted Trace, where the operating system and decoding software need a single, consistent, standardized format. Furthermore, correct trace decoding is dependent on several discoverable attributes of the trace encoder. But in a Self-hosted context, the decoder will not be aware of which physical encoder instance is producing the trace it is decoding, and so will be unable to discover these attributes. This issue can be addressed by including these attributes directly in the support packet.

Additionally, the fully orthogonal Implicit Return and Multi-Memory Access and Compare and Swap extensions introduce revised **te_inst** and **te_data** packet formats, but the monotonically increasing protocol version does not provide a mechansim to indicate independent presence or absence of these extentions (see trTeImpl.trTeProtocolMajor and trTeImpl.trTeProtocolMinor in RISC-V Trace Control Interface Specification). This can resolved by indicating the presence of thes extensions directly in the support packet.

If this extension is implemented, the protocol version must be 2.1 or higher; if the protocol version is 2.1 or higher, this extension must be implemented.

This extension is required by the following extensions:

- Smstrce* and Ssstrce* (Self-Hosted Trace)
- Implicit Return extension to Efficient Trace for RISC-V v2.0 (see Chapter 14)
- Data Trace of Multiple-Memory Access and Compare and Swap extension to Efficient Trace for RISC-V v2.0 (see Chapter 15)

13.1. Standard Support Packet format

Table 35. Packet format 3, subformat 3 suitable for self-hosted applications

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	2	Identifies trace algorithm. Currently Branch trace is the only mode defined, indicated by the value 0.

Field name	Bits	Description
qual_status	2	00 (no_change): No change to filter qualification 01 (ended_rep): Qualification ended, preceding **te_inst** sent explicitly to indicate final qualification instruction 10 (trace_lost): One or more instruction trace packets lost. 11 (ended_ntr): Qualification ended, preceding **te_inst** would have been sent anyway due to an updiscon, even if it wasn't the final qualified instruction
sijump	1	When 1, targets of sequentially inferrable jumps not reported (see Section 3.2.4).
implicit_return	1	When 1, implicit return mode enabled (see Section 3.2.5).
branch_predictor	1	When 1, branch predictor enabled (see Section 3.2.6).
jump_target_cache	1	When 1, jump target cache enabled (see Section 3.2.7).
implicit_except	1	When 1, implicit exception mode enabled (see Section 3.2.3).
full_iaddress	1	When 1, **te_inst** packets always contain full (i.e. non-differential) addresses (see Section 3.2.2).
resync_disabled	1	When 1 indicates that resynchronization is disabled (trTeControl.trTeInstSyncMode == 0).
iret_ext	1	When 1, Implicit Return extension to E-trace v2.0 implemented. [**Add link when extensions merged**]
time_width	3	The width of the time field in sync-start and sync-trap packets (*time_width_p*) is 16 * **time_width** bits.
f0s_width	2	The value of *f0s_width_p*
return_stack_size	3	The number of entries in the return address stack is $2^{\text{return_stack_size}}$ (no stack if 0).
call_counter_size	4	The number of bits in the call counter is $2^{\text{call_counter_size}}$ (no counter if 0).
bpred_size	3	The number of entries in the branch predictor is $2^{\text{bpred_size}}$ (no predictor if 0).
cache_size	3	The number of entries in the jump target cache is $2^{\text{cache_size}}$ (no cache if 0).
denable	1	Indicates if data trace is enabled (if supported).
dloss	1	One of more data trace packets lost (if supported).
mmacas_ext	1	When 1, Multi-Memory Access and Compare & Swap extension to E-trace v2.0 implemented. [**Add link when extensions merged**]
noaddr	1	When 1, **te_data** packets exclude addresses (just report data).
nodata	1	When 1, **te_data** packets exclude data (just report addresses).
full_daddress	1	When 1, addresses in **te_data** packets are always full (i.e non-differential).
full_data	1	When 1, data values in **te_data** packets are always full (i.e non-differential).
other	>=0	Any other implementation specific information

Chapter 14. Implicit Return v1.0 extension to Efficient Trace for RISC-V v2.0



This chapter is in the Ratified state.

No changes are allowed. Any desired or needed changes can be the subject of a follow-on new extension. Ratified extensions are never revised.

The Implicit Return mode information in the format 0, 1 and 2 packets defined in the Efficient Trace for RISC-V 2.0 specification does not produce correct trace decoding in some cases. This extension corrects this by replacing the **irdepth** field with **irets**.

If this extension is implemented, the Standard Support Packet extention to Efficient Trace for RISC-V v2.0 (see Chapter 13) must also be implemented.

14.1. Background

As described in Section 3.2.5, when Implict Return mode is enabled, the target address of function returns is not usually reported. When it *is* reported (for example because it's the last instruction retired before a trap), there are some circumstances where the decoder cannot determine precisely which return target it is without some additional information from the encoder.

One such circumstance occurs when decoding recursive functions. Each recursively called function will have the same return target address, and including the depth of the return address stack allowed the decoder to determine which one has been reported. However, it is now known that providing the stack depth results in incorrect decoding in other cases unrelated to recursive functions. Consider the following illustration:

A function is executed, and returns normally to the predicted target, so nothing is reported. Execution continues without any branches or uninferrable calls/jumps to a second function which also contains no branches or uninferrable calls/jumps. The return address of this 2nd function is reported for some reason (e.g. it is followed by a trap). Note that the stack depth is the same after the 1st function returns and after the 2nd function returns. This means that when the decoder reconstructed the execution path based on **irdepth**, it encountered the 1st return and because the stack depth in the *te_inst* packet matched the current stack depth it stopped decoding at this point, rather than carrying on to the second return.

Had there been any branches between the two functions or inside the 2nd function, decoding would have continued until all branches were processed (the 1st return would have been correctly treated as inferred), and stopped at the right place: the return target of the 2nd function. Similarly if there were uninferrable calls/jumps.

So in summary, the decoder needs additional information to correctly determine which return target has been reported when there have been multiple returns without any branches or uninferrable calls/jumps between them. This is true for all cases, including recursive functions.

14.2. Revised packet formats

The following packet definitions replace those detailed in Chapter 7. All formats and fields are the same, except **irdepth** is replaced by **irets** throughout.

Table 36. 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]**.
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 address stack, or the last retired before an exception, interrupt, privilege change or resync because it is necessary to report the implicit return count.
irets	8	If the value of **irreport** is different from **updiscon**, this field indicates the number of implicit returns that have occurred since the last branch, or *te_inst* packet if there have been no branches since then. If **irreport** is the same value as **updiscon**, all bits in this field will also be the same value as **updiscon**.

14.2.1. Format 2 irreport and irets

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. If Implicit Return mode is enabled, the encoder keeps track of function calls and returns, and does not generate *te_inst* packets for return target addresses that can be inferred by the decoder, as described in Section 3.2.5.

When reporting an address via *te_inst* when Implicit Return mode is enabled, the packet will include details of the number of implicit returns (i.e. return targets that have not been reported explicitly) since the last branch or *te_inst* packet under the following conditions:

- If the reported address is the instruction following a return, and:
  - it is mispredicted, or;
  - there have been 255 unreported returns, or;
  - it is correctly predicted but is being reported for some other reason (e.g. it is the instruction before a trap, privilege change, resync or is the final qualified instruction), or;
  - the return stack is empty, but the number of implicit returns is non-zero.
- If the address is not the instruction following an updiscon but is being reported for some other reason (as listed above), and the number of implicit returns is non-zero.

Under the above conditions, **irreport** is set to the oppositive value to **updiscon**.

Table 37. Packet format 1 - address, branch map

Field name	Bits	Description
format	2	01 (diff-delta): includes branch information and may include differential address

Field name	Bits	Description
branches	5	Number of valid bits **branch_map**. The number of bits of **branch_map** is determined as follows: 0: (cannot occur for this format) 1: 1 bit 2-3: 3 bits 4-7: 7 bits 8-15: 15 bits 16-31: 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: 0 : branch taken 1 : 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]**.
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 implicit return count.
irets	8	If the value of **irreport** is different from **updiscon**, this field indicates the number of implicit returns that have occurred since the last branch, or *te_inst* packet if there have been no branches since then. If **irreport** is the same value as **updiscon**, all bits in this field will also be the same value as **updiscon**.

Table 38. 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: 0: 31 bits, no **address** in packet 1-31: (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: 0 : branch taken 1 : branch not taken

14.2.2. Format 1 irreport and irets fields

See Section 14.2.1.

Table 39. 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. 11 (addr-fail): Packet contains an **address** that points to a branch which failed the prediction. 00, 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]**.
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 implicit return count.
irets	8	If the value of **irreport** is different from **updiscon**, this field indicates the number of implicit returns that have occurred since the last branch, or *te_inst* packet if there have been no branches since then. If **irreport** is the same value as **updiscon**, all bits in this field will also be the same value as **updiscon**.

Table 40. 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:+ 0: (cannot occur for this format) 1: 1 bit 2-3: 3 bits 4-7: 7 bits 8-15: 15 bits 16-31: 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

Field name	Bits	Description
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 implicit return count.
irets	8	If the value of **irreport** is different from **branch_map[MSB]**, this field indicates the number of implicit returns that have occurred since the last branch, or *te_inst* packet if there have been no branches since then. 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]**.

Table 41. 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 implicit return count.
irets	8	If the value of **irreport** is different from **branches[MSB]**, this field indicates the number of implicit returns that have occurred since the last branch, or *te_inst* packet if there have been no branches since then. If **irreport** is the same value as **branches[MSB]**, all bits in this field will also be the same value as **branches[MSB]**.

14.2.3. Format 0 irreport and irets fields

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). Purpose and behavior is as described in Section 14.2.1.

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. An implementation could omit these if all implicit return failures are reported using format 1.

Chapter 15. Data Trace of Multiple-Memory Access and Compare and Swap v1.0 extension to Efficient Trace for RISC-V v2.0



This chapter is in the Ratified state.

No changes are allowed. Any desired or needed changes can be the subject of a follow-on new extension. Ratified extensions are never revised.

The RISC-V Hart to Trace Interface (RHTI) Specification is based on chapter 4 of the Efficient Trace for RISC-V v2.0 specification. It adds **dtype** encodings to support data trace of multi-memory access instructions as defined by the Zcmp and Vector extensions, as well as Compare and Swap as defined by the Zacas extension. This information also needs to be included in *te_data* packets generated by the Trace Encoder, but there are no available unused encodings in the existing packets defined by the Efficient Trace for RISC-V v2.0 specification. This extension defines revised packet formats that address this.

If this extension is implemented, the Standard Support Packet extention to Efficient Trace for RISC-V v2.0 (see Chapter 13) must also be implemented.

15.1. Revised packet formats

The following packet definitions replace those defined in Chapter 8. The **format** and **subtype** fields have been restructured but all other fields are the same.

15.1.1. format field

Data trace packets are classified by the **format** field. This field is a maximum of 2 bits wide, but may be omitted completely if only unified loads and stores are supported, or reduced to 1 bit if unified multi-memory access load and stores are also supported. 2 bits are only required when tracing of atomics or CSRs is required, or if loads are split.

Table 42. Packet format for Unified load or store, with address and data

Field name	Bits	Description
format	clog2(*dtype_width_p*)	Transaction class: 00: Load or store 01: Multi-memory access load or store (other codes select other packet formats)
subformat	2	Transaction type: 00: Unified load address, aligned 01: Unified load address, unaligned 10: Store, aligned address 11: Store, unaligned address
size	max(1, clog2(clog2(*data_width_p*/8 + 1)))	Transfer size is 2^{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

Field name	Bits	Description
address	*daddress_width_p*	Byte address if format is unaligned, otherwise shift left by **size** to recover byte address

The following packet format is used when the encoder is configured to omit data values from data trace packets (trTeDataControl.trTeDataNoValue is 1).

Table 43. Packet format for Unified load or store, with address only

Field name	Bits	Description
format	clog2(*dtype_width_p*)	Transaction class: 00: Load or store 01: Multi-memory access load or store (other codes select other packet formats)
subformat	2	Transaction type: 00: Unified load address, aligned 01: Unified load address, unaligned 10: Store, aligned address 11: Store, unaligned address
size	max(1, clog2(clog2(*data_width_p*/8 + 1)))	Transfer size is 2^{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

The following packet format is used when the encoder is configured to omit address values from data trace packets (trTeDataControl.trTeDataNoAddr is 1).

Table 44. Packet format for Unified load or store, with data only

Field name	Bits	Description
format	clog2(*dtype_width_p*)	Transaction class: 00: Load or store 01: Multi-memory access load or store (other codes select other packet formats)
subformat	2	Transaction type: 00: Unified load address, aligned 01: Unified load address, unaligned 10: Store, aligned address 11: Store, unaligned address
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

The following packet is used for the request phase of a split load.

Table 45. Packet format for Split load - request phase

Field name	Bits	Description
format	2	Transaction class: 00: Load or store 01: Multi-memory access load or store (other codes select other packet formats)
subformat	2	Transaction type: 00: Split load address, aligned 01: Split load address, unaligned
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

The following packet is used for the data phase of a split load and split atomic.

Table 46. Packet format for Split load or atomic - data phase

Field name	Bits	Description
format	2	Transaction type 11: 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

Table 47. Packet format for Unified atomic with address and data

Field name	Bits	Description
format	2	Transaction class: 10: Atomic or CSR (other codes select other packet formats)
subformat	4	Atomic sub-type 1000: Swap 1001: ADD 1010: AND 1011: OR 1100: XOR 1101: MAX 1110: MIN (other codes select other packet formats)
size	max(1, clog2(clog2(*data_width_p*/8 + 1)))	Transfer size is 2^{size} bytes
diff	2	00: Full address and data (sync) 01: Differential address, XOR-compressed data/operand 10: Differential address, full data/operand 11: Differential address, differential data/operand
op_len	size	Number of bytes of operand is **op_len** + 1

Field name	Bits	Description
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

Table 48. Packet format for Unified CAS failure

Field name	Bits	Description
format	2	Transaction class: 10: Atomic or CSR (other codes select other packet formats)
subformat	4	Atomic sub-type 0111: Compare and Swap failure+ (other codes select other packet formats)
size	max(1, clog2(clog2(*data_width_p*/8 + 1)))	Transfer size is 2^{size} bytes
diff	2	00: Full address and data (sync) 01: Differential address, XOR-compressed operand 10: Differential address, full operand 11: Differential address, differential operand
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

The following packet format is used when the encoder is configured to omit data values from data trace packets (trTeDataControl.trTeDataNoValue is 1)

Table 49. Packet format for Unified atomic with address only

Field name	Bits	Description
format	2	Transaction class: 10: Atomic or CSR (other codes select other packet formats)
subformat	4	Atomic sub-type 0111: compare and swap failure 1000: Swap 1001: ADD 1010: AND 1011: OR 1100: XOR 1101: MAX 1110: MIN 1111: conditional store failure (other codes select other packet formats)
size	max(1, clog2(clog2(*data_width_p*/8 + 1)))	Transfer size is 2^{size} bytes
diff	1	0: Full address 1: Differential address
address	*daddress_width_p*	Address, aligned and encoded as per size

The following packet format is used when the encoder is configured to omit address values from data trace packets (trTeDataControl.trTeDataNoAddr is 1).

Table 50. Packet format for Unified atomic with data and operand only

Field name	Bits	Description
format	2	Transaction class: 10: Atomic or CSR (other codes select other packet formats)
subformat	4	Atomic sub-type 1000: Swap 1001: ADD 1010: AND 1011: OR 1100: XOR 1101: MAX 1110: MIN (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/operand (XOR if 2 bits) 10: reserved 11: Differential data/operand
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

The following packet is used for the request phase of a split atomic:

Table 51. Packet format for Split atomic with operand and address

Field name	Bits	Description
format	2	Transaction class: 10: Atomic or CSR (other codes select other packet formats)
subformat	4	Atomic sub-type 1000: Swap 1001: ADD 1010: AND 1011: OR 1100: XOR 1101: MAX 1110: MIN (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	2	00: Full address and operand (sync) 01: Differential address, XOR-compressed operand 10: Differential address, full operand 11: Differential address, differential operand
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

The following packet format is used when the encoder is configured to omit address values from data trace packets (trTeDataControl.trTeDataNoAddr is 1).

Table 52. Packet format for Split atomic with operand only

Field name	Bits	Description
format	2	Transaction class: 10: Atomic or CSR (other codes select other packet formats)
subformat	4	Atomic sub-type 1000: Swap 1001: ADD 1010: AND 1011: OR 1100: XOR 1101: MAX 1110: MIN (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	2	00: Full operand (sync) 01: XOR-compressed operand (XOR if 2 bits)+ 10: reserved 11: Differential operand
operand	*data_width_p*	Operand. Value from rs2 before operator applied

Table 53. Packet format for Unified CSR, with address, data and operand

Field name	Bits	Description
format	2	Transaction class: 10: Atomic or CSR (other codes select other packet formats)
subformat	4	CSR sub-type 0100: RW 0101: RS 0110: RC (other codes select other packet formats)
diff	1 or 2	00: Full data/operand (sync) 01: Compressed data/operand (XOR if 2 bits) 10: reserved 11 : Differential data/operand
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]

Table 54. Packet format for Unified CSR, with address and read-only data (as determined by addr[11:10] = 11)

Field name	Bits	Description
format	2	Transaction class: 10: Atomic or CSR (other codes select other packet formats)
subformat	4	CSR sub-type 0100: RW 0101: RS 0110: RC (other codes select other packet formats)

Field name	Bits	Description
diff	1 or 2	00: Full data (sync) 01: Compressed data/operand (XOR if 2 bits) 10: reserved 11 : Differential data/operand
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]

Table 55. Packet format for Unified CSR, with address only

Field name	Bits	Description
format	2	Transaction class: 10: Atomic or CSR (other codes select other packet formats)
subformat	4	CSR sub-type 0100: RW 0101: RS 0110: RC (other codes select other packet formats) 11: reserved
diff	1	0: Full address 1: Differential address
addr_msbs	6	Address[11:6]
addr_lsbs	6	Address[5:0]