RISC-V Processor Trace
Version 1.0.1-Draft
bd814b8a97bbf7a9d605d47872f42f5945ea4567

Gajinder Panesar, Iain Robertson
<gajinder.panesar@ultrasoc.com>, <iain.robertson@ultrasoc.com>

UltraSoC Technologies Ltd.

August 24, 2020

Contents

List of Figures

List of Tables

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 successfully create 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 havea 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)
- **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
- **exception**: an unusual condition occurring at run time associated with an instruction in a RISC-V hart
- **hart**: a RISC-V hardware thread

- **interrupt**: an external asynchronous event that may cause a RISC-V hart to experience an unexpected transfer of control
- **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

Need to add definitions for individual fields here.

Fields are categorized into the following groups:

- M: Mandatory
- O: Optional
- MD: Mandatory if data trace is supported
- OD: Optional for data trace
- F: Optional for filtering.

Other abreviations used in the tables are:

- **W** column heading indicates field width
- **G** column heading indicates field group
- **Rst** column heading indicates field reset value
- SD in **Rst** column indicates a system dependent reset value

2.1 Basic Control

The following fields control basic encoding behaviour.

Table 2.1: Basic Control

Field	W	G	RW	Rst	Description
Active	1	M	RW	0	Master enable for trace system. When 0, the trace system may have clocks gated off or be powered down, and other register locations may be inaccessible. Hardware may take arbitrarily long to process power-up or power-down and will indicate completion when the read value of this bit matches the value written.
iEnable	1	M	RW	0	Instruction trace enable. Setting to 0 flushes any queued instruction trace.
dEnable	1	MD	RW	0	Data trace enable. Setting to 0 flushes any queued data trace.
iTrigEnable	1	O	RW	0	When 1, allows **iEnable** to be set or cleared by *trace-on* and *trace-off* Debug module triggers respectively (see 4.2.4).
dTrigEnable	1	OD	RW	0	When 1, allows **dEnable** to be set or cleared by *trace-on* and *trace-off* Debug module triggers respectively (see 4.2.4).
stallEnable	1	O	RW	0	When 0, if the encoder cannot accept trace input from the RISC-V hart, trace is lost, and is indicated via the *Support* trace packet (see table 6.3). When 1, the **stall** output signal is asserted to stall the RISC-V hart until trace can be accepted (see table 4.6).
Empty	1	O	R	1	Reads as 1 when all trace has been emitted. Note: this status is also indicated via the *Support* trace packet (see table 6.3).
ResyncMode	2	M	RW	SD	Selects the resynchronization mechanism. At least one non-zero mechanism must be implemented. 0: Off 1: Count trace packets 2: Count clock cycles 3: Count instruction half-words
ResyncMax	4	O	RW	SD	The maximum interval (in units determined by **ResyncMode**) between synchronization packets (see tables 6.2 and 6.3) is $2^{\mathbf{ResyncMax}+4}$.

2.2 Optional Modes

See section 3.2 for details of the modes covered in this section. **Author note:** data trace mode descriptions need adding.

Table 2.2: Optional and run-time configurable modes

Field	W	G	RW	Rst	Description
FullAddress	1	O	RW	SD	Send only full (non-differential) addresses when set
ImplicitExcept	1	O	RW	SD	When set, do not send exception address, only exception cause in Exception packets (see table 6.3)
siJump	1	O	RW	SD	Do not treat sequentially inferrable jumps as uninferable PC discontinuities when set.
ImplicitReturn	1	O	RW	SD	Do not treat returns as uninferable PC discontinuities when set.
BranchPrediction	1	O	RW	SD	Branch predictor enabled when set.
JumpTargetCache	1	O	RW	SD	Jump target cache enabled when set.

2.3 Filtering

Details to follow

2.4 Buffering and Transport

Details to follow

2.5 Message mapping to register proposal

Table 2.3: Mapping UST messages to Register based control used in Nexus

Field	W	Description	Register	Offset	Message	Offset	UST Name
teActive	1	Master Enable	**teControl**	0	**set_enabled**	16	**module_enable**
teEnable	1	Trace Enable	**teControl**	1	**set_trace.instruction**	8	**trace_enable**
teTriggered	1	Enable external triggers to change teEnable	**teControl**	TBD	**set_trace.instruction**	9	**trigger_enable**
teTracing	1	trace being generated	**teControl**	2			
teEmpty	1	All trace emitted	**teControl**	3			Not necessary for RISC-V WG trace, as **te_inst** support message issued after all trace emitted.
teInstruction	?	trace mode enables	teControl	4	**set_trace.instruction**	37 38 39 40 41 42 43	**encoder_mode** (2 bits) **full_address** **implicit_return** **implicit_except** **sijump** **branch_prediction** **jump_target_cache** (others for proprietary extensions)
teStallEnable	1	Enable lossless behaviour	**teControl**	13	**set_trace.instruction**	35	lossless
teStallOnWrap	1	Circular buffer control	**teControl**	14	-		
teInhibitSrc	1	Inhibit SrcID	**teControl**	15	-		Inappropriate in a message based architecture
teSyncMax	4	Resync interval	**teControl**	16	**set_trace.instruction**	13	**max_resync**
teSyncMode	2	Resync mode	**teControl**	TBD	**set_trace.instruction**	17	**resync_mode**
teMessageFormat	3	Trace format	**teControl**	26	-		
teSink	3	Sink selector	**teControl**	28	-	UST trace routing is distributed and control mechanism is incompatible with this.	

Table 2.4: Mapping UST messages to Register based teImpl regster used in Nexus

Field	W	Description	Register	Offset	Message	Offset	UST Name
teFilter	16	I-trace filter enable vector	TBD	TBD	**set_trace.instruction**	19	**trigger**
version	4	TE version	**teImpl**	0	**discovery_response**	TBD	arch
hasSRAMSink	1	TE has SRAM sink	**teImpl**	4	-		
hasATBSink	1	TE has ATB sink	**teImpl**	5	-		
hasPIBSink	1	TE has PIB sink	**teImpl**	6	-		
hasSBASink	1	TE has SBA sink	**teImpl**	7	-		
hasFunnelSink	1	TE has funnel sink	**teImpl**	8	-		
reserved	11	9					
SrcID	4	Source ID	**teImpl**	20	-		
nSrcBits	3	Width of source ID	**teImpl**	24			

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 occured in the instruction sequence, so the trace must report the address where normal program flow ceased, as well as give an indication of the asynchronous destination which may be as simple as reporting the exception type. 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;
- 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 section 4.2.4);
- The matching criteria for any filtering capabilities implemented by the encoder may no longer be met;
- The encoder may be disabled.

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 9.1 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 ***utvec/stvec/mtvec*** CSR registers. In some RISC-V implementations, the lower address bits are stored in the ***ucause/scause/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 Section 4.1.1 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 Section 4.1.1 for definitions). The counter will not over or underflow, and is reset to 0 whenever a synchronization packet is sent. Returns will be treated as inferable and will not generate a trace packet if the count is non-zero (i.e. the associated call was already reported in an earlier packet).

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.

Other predictors, such as the gShare predictor (see Hennessy & Patterson), should be considered. Some further experimentation is needed to determine the benefits of different lookup table sizes and predictor algorithms.

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. 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 at its index in the cache. 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.

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

Chapter 4

Hart to encoder interface

4.1 Instruction Trace Interface requirements

This section describes in general terms the information which must be passed from the RISC-V hart to the trace encoder for the purposes of Instruction Trace, and distinguishes between what is mandatory, and what is optional.

The following information is mandatory:

- The number of instructions that are being retired;
- Whether there has been an exception or interrupt, and if so the cause (from the ***ucause/scause/mcause*** CSR) and trap value (from the ***utval/stval/mtval*** CSR);
- The current privilege level of the RISC-V hart;
- The *instruction_type* of retired instructions for:
  - Jumps with a target that cannot be inferred from the source code;
  - Taken and nontaken branches;
  - Return from exception or interrupt (****ret*** instructions).
- The *instruction_address* for:
  - Jumps with a target that *cannot* be inferred from the source code;
  - The instruction retired immediately after a jump with a target that *cannot* be inferred from the source code (also referred to as the target or destination of the jump);
  - Taken and nontaken branches;
  - The last instruction retired before an exception or interrupt;
  - The first instruction retired following an exception or interrupt;
  - The last instruction retired before a privilege change;
  - The first instruction retired following a privilege change.

The following information is optional:

- Context information:
  - The context and/or hart ID;
  - The type of action to take when context changes.
- The *instruction_type* of instructions for:
  - Calls with a target that *cannot* be inferred from the source code;
  - Calls with a target that *can* be inferred from the source code;
  - Tail-calls with a target that *cannot* be inferred from the source code;
  - Tail-calls with a target that *can* be inferred from the source code;
  - Returns with a target that *cannot* be inferred from the source code;
  - Returns with a target that *can* be inferred from the source code;
  - Co-routine swap;
  - Jumps which don't fit any of the above classifications with a target that *cannot* be inferred from the source code;
  - Jumps which don't fit any of the above classifications with a target that *can* be inferred from the source code.
- If context is supported then the *instruction_address* for:
  - The last instruction retired before a context change;
  - The first instruction retired following a context change.
- Whether jump targets are sequentially inferable or not.

The mandatory information is the bare-minimum required to implement the branch trace algorithm outlined in Chapter 8. The optional information facilitates alternative or improved trace algorithms:

- Implicit return mode (see Section 3.2.5) requires the encoder to keep track of the number of nested function calls, and to do this it must be aware of all calls and returns regardless of whether the target can be inferred or not;
- A simpler algorithm useful for basic code profiling would only report function calls and returns, again regardless of whether the target can be inferred or not;
- Branch prediction techniques can be used to further improve the encoder efficiency, particularly for loops (see Section 3.2.6). This requires the encoder to be aware of the address of all branches, whether they are taken or not.
- Uninferable jumps can be treated as inferable (which don't need to be reported in the trace output) if both the jump and the preceding instruction which loads the target into a register have been traced.

4.1.1 Jump classification and target inference

Jumps are classified as *inferable*, or *uninferable*. An *inferable* jump has a target which can be deduced from the binary executable or representation thereof (e.g. ELF). For the purposes of this specification, the following strict definition applies:

If the target of a jump is supplied via a constant embedded within the jump opcode, it is classified as *inferable*. Jumps which are not *inferable* are by definition *uninferable*.

However, there are some jump targets which can still be deduced from the binary executable by considering pairs of instructions even though by the above definition they are classified as uninferable. Specifically, jump targets that are supplied via

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

Such jump targets are classified as *sequentially inferable* if the pair of instructions are retired consecutively (i.e. the ***auipc***, ***lui*** or ***c.lui*** immediately precedes the jump). Note: the restriction that the instructions are retired consecutively is necessary in order to minimize the additional signalling needed between the hart and the encoder, and should have a minimal impact on trace efficiency as it is anticipated that consecutive execution will be the norm. Support for sequentially inferable jumps is optional.

Jumps may optionally be further classified according to the recommended calling convention:

- *Calls*:
  - ***jal*** x1;
  - ***jal*** x5;
  - ***jalr*** x1, rs where rs != x5;
  - ***jalr*** x5, rs where rs != x1;
  - ***c.jalr*** rs1 where rs1 != x5;
  - ***c.jal***.
- *Tail-calls*:
  - ***jal*** x0;
  - ***c.j***;
  - ***jalr*** x0, rs where rs != x1 and rs != x5;
  - ***c.jr*** rs1 where rs1 != x1 and rs1 != x5.
- *Returns*:
  - ***jalr*** rd, rs where (rs == x1 or rs == x5) and rd != x1 and rd != x5;
  - ***c.jr*** rs1 where rs1 == x1 or rs1 == x5.

- *Co-routine swap*:
  - ***jalr*** x1, x5;
  - ***jalr*** x5, x1;
  - ***c.jalr*** x5.
- *Other*:
  - ***jal*** rd where rd != x1 and rd != x5;
  - ***jalr*** rd, rs where rs != x1 and rs != x5 and rd != x0 and rd != x1 and rd != x5.

4.1.2 Relationship betweem RISC-V core and the encoder

The encoder is intended to encode the instructions executed on a single hart.

It is however commonplace for a RISC-V core to contain multiple harts. This can be supported by the core in several different ways:

- Implement a separate instance of the interface per hart. Each instance can be connected to a separate encoder instance, allowing all harts to be traced concurrently. Alternatively, external muxing may be used in conjunction with a single encoder in order to trace one particular hart at a time;
- Implement a singe interface for the core, with muxing inside the core to select which hart to connect to the interface.

(Whilst it is technically feasible to use a single encoder with multiple harts operating in a fine-grained multi-threaded configuration, the frequent context changes that would occur as a result of thread-switching would result in extremely poor encoding efficiency, and so this configuration is not recommended.)

4.2 Instruction Trace Interface

This section describes the interface between a RISC-V hart and the trace encoder that conveys the information described in the section 4.1. Signals are assigned to one of the following groups:

- M: Mandatory. The interface must include an instance of this signal.
- O: Optional. The interface may include an instance of this signal.
- MR: Mandatory, may be replicated. For harts that can retire a maximum of N taken branches per clock cycle, the interface must include N instances of this signal.
- OR: Optional, may be replicated. For harts that can retire a maximum of N taken branches per clock cycle, the interface must include zero or N instances of this signal.

- BR: Block, may be replicated. Mandatory for harts that can retire multiple instructions in a block. Replication as per OR. If omitted, the interface must include SR group signals instead.
- SR: Single, may be replicated. Mandatory for harts that can only retire one instruction in a block. Replication as per OR (see section 4.2.2). If omitted, the interface must include BR group signals instead.

Tables 4.1 and 4.2 list the signals in the interface designed to efficiently support retirement of multiple instructions per cycle. The following discussion describes the multiple-retirement behavior. However, for harts that can only retire one instruction at a time, the signalling can be simplified, and this is discussed subsequently in Section 4.2.1.

The information presented in a block represents a contiguous block of instructions starting at **iaddr**, all of which retired in the same cycle. Note if **itype** is 1 or 2 (indicating an exception or an interrupt), the number of instructions retired may be zero. **cause** and **tval** are only defined if **itype** is 1 or 2. If **iretire**=0 and **itype**=0, the values of all other signals are undefined.

iretire contains the number of half-words represented by instructions retired in this block, and **ilastsize** the size of the last instruction. Half-words rather than instruction count enables the encoder to easily compute the address of the last instruction in the block without having access to the size of every instruction in the block.

itype can be 3 or 4 bits wide. If *itype_width_p* is 3, a single code (6) is used to indicate all uninferable jumps. This is simpler to implement, but precludes use of the implicit return mode (see section 3.2.5), which requires jump types to be fully classified.

Whilst **iaddr** is typically a virtual address, it does not affect the encoder's behavior if it is a phyical address.

For harts that can retire a maximum of N branches per clock cycle, the signal groups MR, OR and either BR or SR must be replicated N times. Signal group 0 represents information about the oldest instruction block, and group N-1 represents the newest instruction block. The interface supports no more than one privilege, context, exception or interrupt per cycle and so signals in groups M and O are not replicated. Furthermore, **itype** can only take the value 1 or 2 in one of the signal groups, and this must be the newest valid group (i.e. **iretire** and **itype** must be zero for higher numbered groups). If fewer than N branches are retired in a cycle, then lower numbered groups must be used first. For example, if there is one branch, use only group 0, if there are two branches, instructions up to the 1st branch must be reported in group 0 and instructions up to the 2nd branch must be reported in group 1 and so on.

sijump is optional and may be omitted if the hart does not implement the logic to detect sequentially inferable jumps. If the encoder offers an **sijump** input it must also provide a parameter to indicate whether the input is connected to a hart that implements this capability, or tied off. This is to ensure the decoder can be made aware of the hart's capability. Enabling sequentially inferable jump mode in the encoder and decoder when the hart does not support it will prevent correct reconstruction by the decoder.

The **context** field can be used to convey any additional information to the decoder. For example:

- The software thread ID;

Table 4.1: Instruction interface signals

Signal	Group	Function
itype[*itype_width_p*-1:0]	MR	Termination type of the instruction block (see Section 4.1.1 for definitions of codes 6 - 15): 0: Final instruction in the block is none of the other named **itype** codes; 1: Exception. An exception that traps occurred following the final retired instruction in the block; 2: Interrupt. An interrupt that traps occurred following the final retired instruction in the block; 3: Exception or interrupt return; 4: Nontaken branch; 5: Taken branch; 6: Uninferable jump if *itype_width_p* is 3, reserved otherwise; 7: reserved; 8: Uninferable call; 9: Inferrable call; 10: Uninferable tail-call; 11: Inferrable tail-call; 12: Co-routine swap; 13: Return; 14: Other uninferable jump; 15: Other inferable jump.
cause[*ecause_width_p*-1:0]	M	Exception or interrupt cause (***ucause/scause/mcause***). Ignored unless **itype**=1 or 2.
tval[*iaddress_width_p*-1:0]	M	The associated trap value, e.g. the faulting virtual address for address exceptions, as would be written to the **utval/stval/mtval** CSR. Future optional extensions may define **tval** to provide ancillary information in cases where it currently supplies zero. Ignored unless **itype**=1 or 2.
priv[*privilege_width_p*-1:0]	M	Privilege level for all instructions retired on this cycle.
iaddr[*iaddress_width_p*-1:0]	MR	The address of the 1st instruction retired in this block. Invalid if **iretire**=0
context[*context_width_p*-1:0]	O	Context for all instructions retired on this cycle.
ctype[*ctype_width_p*-1:0]	O	Reporting behavior for **context**: 0: Don't report; 1: Report imprecisely; 2: Report precisely; 3: Report as asynchronous discontinuity.
sijump	OR	If **itype** indicates that this block ends with an uninferable discontinuity, setting this signal to 1 indicates that it is sequentially inferable and may be treated as inferable by the encoder if the preceding ***auipc***, ***lui*** or ***c.lui*** has been traced. Ignored for **itype** codes other than 8, 10, 12 or 14.

Table 4.2: Instruction interface signals - multiple retirement per block

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

Table 4.3: Instruction interface signals - single retirement per block

Signal	**Group**	**Function**
iretire[0:0]	SR	Number of instructions retired in this block (0 or 1).

- The process ID from an operating system;
- It could be used to convey the values of CSRs to the decoder by setting **context** to the CSR number and value when a CSR is written;
- In cases where a single encoder is being shared amongst multiple harts (see section 4.1.2), it could also be used to indicate the hart ID, in cases where the hart ID can be changed dynamically.

Table 4.4 specifies the actions for the various **ctype** values. A typical behaviour would be for this signal to remain zero except on the 1st retirement after a context change. *ctype_width_p* may be 1 or 2. The reduced width option only provides support for reporting context changes imprecisely.

Table 4.4: Context type **ctype** values and corresponding actions

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

4.2.1 Simplifications for single-retirement

For harts that can only retire one instruction at a time, the interface can be simplified to the signals listed in tables 4.1 and 4.3. The simplifications can be summarized as follows:

- As the number of instructions that are retired in a block is only 0 or 1, the encoder does not need information to enable it to deduce the address of the last instruction retired (it is the same as the 1st and only instruction retired). So **ilastsize** is not necessary, and **iretire** simply indicates whether an instruction retired or not.

The parameter *retires_p* which indicates to the encoder the maximum number of instructions that can be retired per cycle can be used by an encoder capable of supporting single or multiple retirement to select the appropriate interpretation of **iretire**. The **ilastsize** encoder input must be tied low when attached to a single-retirement hart that does not provide these outputs.

4.2.2 Alternative multiple-retirement interface configurations

For a hart that can retire multiple instructions per cycle, but no more than one branch, the preferred solution is to use one instance of signals from groups BR, MR and OR. However, if the hart can retire N branches in a cycle, N instances of signals from groups MR, OR and either SR or BR must be used (each instance can be either a single instruction or a block).

If the hart can retire N instructions per cycle, but only one branch, it is allowed (though not recommmended) to provide explicit details of every instruction retired by using N instances of signals from groups SR, MR and OR.

4.2.3 Optional sideband signals

Optional sideband signals may be included to provide additional functionality, as described in tables 4.5 and 4.6.

Note, any user defined information that needs to be output by the encoder will need to be applied via the **context** input.

Table 4.5: Optional sideband encoder input signals

Signal	Group	Function
impdef[*impdef_width_p*-1:0]	O	Implementation defined sideband signals. A typical use for these would be for filtering (see Chapter 5.
trigger[2:0]	OR	A pulse on bit 0 will cause the encoder to start tracing, and continue until further notice, subject to other filtering criteria also being met. A pulse on bit 1 will cause the encoder to stop tracing until further notice. See section 4.2.4).
halted	O	Hart is halted. Upon assertion, the encoder will output a packet to report the address of the last instruction retired before halting, followed by a support packet to indicate that tracing has stopped. Upon deassertion, the encoder will start tracing again, commencing with a synchronization packet.
reset	O	Hart is in reset. Provided the encoder is in a different reset domain to the hart, this allows the encoder to indicate that tracing has ended on entry to reset, and restarted on exit. Behavior is as described above for halt.

Table 4.6: Optional sideband encoder output signals

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

4.2.4 Using trigger outputs from the Debug Module

The debug module of the RISC-V hart may have a trigger unit. This defines a match control register (***mcontrol***) containing a 4-bit **action** field, and reserves codes 2 - 5 of this field for trace use. These action codes are hereby defined as shown in table 4.7. If implemented, each action must generate a pulse on an output from the hart, on the same cycle as the instruction which caused the trigger is retired.

Table 4.7: Debug Module trigger support (***mcontrol* action**)

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

Trace-on and Trace-off actions provide a means for the hart to control when tracing occurs. Trace-notify provides means to ensure that a specified instruction is explicitly reported. This capability is sometimes known as a watchpoint.

4.2.5 Example retirement sequences

Table 4.8: Example 1 : 9 Instructions retired over four cycles, 2 branches

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

4.3 Data Trace Interface requirements

This section describes in general terms the information which must be passed from the RISC-V hart to the trace encoder for the purposes of Data Trace, and distinguishes between what is mandatory, and what is optional.

If Data Trace is not needed in a system then there is no requirement for the RISC-V hart to supply any of the signals in section 4.4.

Data trace supports up to four data access types: load, store, atomic and CSR. Support for atomic and CSR accesses is optional.

The signalling protocol can take one of two forms, depending on the needs of the RISC-V hart: *unified* or *split*.

Unified is the simplest form, suitable for simpler, in-order harts. In this form, all information about a data access is signalled by the RISC-V hart in the same cycle that the associated data access instruction is reported on the instruction trace interface.

For harts with out of order or speculative execution capabilites, many loads may be in progress simultaneously, and this approach is not practical as it would require the hart to maintain a large amount of state relating to all the in-progress loads. For this reason, the interface also supports splitting loads into two parts:

- The *request* phase provides all information about the load that originates from the hart (address, size, etc.) when the instruction retires;
- The *response* phase provides the data and response status when it has been returned to the hart from the memory system.

The two parts of a split load are associated by use of a transaction ID.

4.4 Data Trace Interface

This section describes the interface between a RISC-V hart and the trace encoder that conveys the information described in the section 4.3. Signals are assigned to one of the following groups:

- M: Mandatory. The interface must include an instance of this signal;
- U: Unified. Mandatory for unified signalling;
- S: Split. Mandatory for split load signalling;
- O: Optional. The interface may include an instance of this signal.

All signals in M, U and O groups are only valid when **dretire** is high. Signals in the S group are valid as indicated in table 4.9.

The maximum value of *dtype_width_p* is 4. However, if only loads and stores are supported, *dtype_width_p* can be 1. If CSRs are supported but atomics are not, *dtype_width_p* can be 3.

Atomic and CSR accesses have either both load and store data, or store data and an operand. For CSRs and unified atomics, both values are reported via **data**, with the store data in the LSBs and the load data or operand in the MSBs.

Table 4.9: Data interface signals

Signal	Group	Function
dretire	M	Data access retired
dtype[*dtype_width_p*-1:0]	M	Data access type: 0: Load 1: Store 2: reserved 3: reserved 4: CSR read-write 5: CSR read-set 6: CSR read-clear 7: reserved 8: Atomic swap 9: Atomic add 10: Atomic AND 11: Atomic OR 12: Atomic XOR 13: Atomic max 14: Atomic min 15: reserved
daddr[*daddress_width_p*-1:0]	M	The data access address
dsize[*dsize_width_p*-1:0]	M	The data access size is $2^{\mathbf{dsize}}$ bytes
data[*data_width_p*-1:0]	U	The data
iaddr_lsbs[*iaddr_lsbs_width_p*-1:0]	O	LSBs of the data access instruction address. Required if *retires_p* > 1
dblock[*dblock_width_p*-1:0]	O	Instruction block in which the data access instruction is retired. Required if there are replicated instruction block signals
lrid[*lrid_width_p*-1:0]	S	Load request ID. Valid when **dretire** is high
lresp[*lresp_width_p*-1:0]	S	Load response: 0: None 1: reserved 2: Okay. Load successful; **ldata** valid 3: Error. Load failed; **ldata** not valid
lid[*lrid_width_p*-1:0]	S	Split Load ID. Valid when **lresp** is non-zero
sdata[*sdata_width_p*-1:0]	S	Store data. Valid when **dretire** is high
ldata[*ldata_width_p*-1:0]	S	Load data. Valid when **lresp** is non-zero

lrid_width_p is determined by the maximum number of loads that can be in progress simultaneously, such that at any one time there can be no more than one load in progress with a given ID.

iaddr_lsbs and **dblock** are provided to support filtering of which data accesses to trace based on their instruction address. This is best illustrated by considering the following instruction sequence:

1. load
2. <some non data access instruction>
3. load
4. <some non data access instruction>
5. <some non data access instruction>

Suppose the hart is capable of retiring up to 4 instructions in a cycle, via a single block. Instruction trace is enabled throughout, but the requirement is to collect data trace for the 1st load (instruction 1), and filtering is configured to match the address of this instruction only. However, information about instruction addresses is passed to the encoder at the block level, and the block boundaries are invisible to the decoder. For instruction trace, all instructions in a block are traced if any of the instructions in that block match the filtering criteria. That is fine for instruction trace - the address of the 1st and last traced instruction are output explicitly. There will be some fuzziness about precisely what those addresses will be depending on where the block boundaries fall, but this is not a concern as everything is always self-consistent.

However, that is not the case for data trace. Consider two scenarios:

- Case 1: 1st block contains instructions 1, 2, 3; second block contains 4, 5
- Case 2: 1st block contains instructions 1,2; second block contains 3, 4, 5

Given that **dvalid** is high in the same cycle that the data access retires, the encoder knows the address of the 1st and last instructions in a block, but does not know precisely where in the block the data access is. In both cases, the first block matches the filtering criteria (it contains the address of instruction 1), and the second block does not. But if the encoder traced all the data accesses in the matching block, then in case 1 it would trace both instructions 1 and 3, whereas in the second case it would trace only instruction 1. The decoder has no visibility of the block boundaries so cannot account for this. It is expecting only instruction 1 to be traced, and so may misinterpret instruction 3. If this code is in a loop for example, it will assume that the 2nd traced load is in fact instruction 1 from the next loop iteration, rather than instruction 3 from this iteration.

Providing the LSBs of the data access instruction address allows the decoder to determine precisely whether the data access should be traced or not, and removes the dependency on the block sizes and boundaries. The number of bits required is one more bit than the number required to index within the block because blocks can start on any half-word boundary.

For harts that replicate the block signals to allow multiple blocks to retire per cycle it is also necessary to indicate which block each data access is associated with, so the encoder knows which block address to combine with the LSBs in order to construct the actual data access instruction address. 1 bit for 2 blocks per cycle, 2 bits for 4, and so on.

Chapter 5

Filtering

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 provides:

- Comparators offering a range of arithmetic options (<, >, =, !=, etc) for **iaddress**, **context** and **tval** inputs;
- Multiple choice selection for **priv** and **cause** inputs;
- Masked matching for **interrupt** and **impdef** inputs;
- The ability to enable tracing when **trigger[0]** is asserted, and continue tracing until **trigger[0]** is asserted (see Section 4.2.4).

Chapter 6

Trace Encoder Output Packets

The bulk of this section describes the payload of packets output from the 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 specified. However, the following information must be provided to the encapsulator:

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

Two example transport schemes are the UltraSoC Messaging Infrastructure, and the Arm Trace Bus. Figure 6.1 shows the encapsulation used for the UltraSoC infrastructure:

- The header byte contains a 5-bit field specifying the payload length in bytes, a 2-bit field indicating the "flow" (destination routing indicator), and a bit to indicate whether an optional 16-bit timestamp is present;
- The index field indicates the source of the packet. The number of bits is system dependent, And the initial value emitted by the trace encoder is zero (it gets adjusted as it propagates through the infrastructure);
- An optional 2-byte timestamp;
- The packet payload.

Figure 6.1: Example encapsulated packet format

Alternatively, for ATB, the source of the packet is indicated by the **ATID** bus field, and there is no equivalent of "flow", so an example encapsulation might be:

- A 5-bit field specifying the payload length in bytes
- A bit to indicate whether an optional 16-bit timestamp is present;
- An optional 2-byte timestamp;
- The packet payload.

It may be desirable for packets to start aligned to an ATB word, in which the **ATBYTES** bus field in the last beat of a packet can be used to indicate the number of valid bytes.

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.

6.1 Format 3 packets

Format 3 packets are used for synchronization, 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.

6.2 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 a the first in an exception handler), and when resynchronization has been scheduled by expiry of the resynchronisation timer.

Table 6.1: 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
context	*context_width_p*, or 0 if *nocontext_p* is 1	The instruction context
address	*iaddress_width_p* - *iaddress_lsb_p*	Full instruction address. Address alignment is determined by *iaddress_lsb_p* Address must be left shifted in order to recreate original byte address.

6.2.1 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 8.1.

6.3 Format 3 subformat 1 - Exception

This packet also contains all the information the decoder needs to fully identify an instruction. It is sent following an exception, and as well as reporting the address of the exception handler, it also includes the exception cause and the address of the faulted instruction.

If the implicit exception mode is enabled (see section 3.2.3), the address is omited.

Table 6.2: Packet format 3, subformat 1

Field name	Bits	Description
format	2	11 (sync): synchronisation
subformat	2	01 (exception): Exception 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.
context	*context_width_p*, or 0 if *nocontext_p* is 1	The instruction context.
ecause	*ecause_width_p*	Exception cause.
interrupt	1	Interrupt.
address	*iaddress_width_p* - *iaddress_lsb_p*	Full instruction address. Address alignment is determined by *iaddress_lsb_p* Address must be left shifted in order to recreate original byte address.
tvalepc	*iaddress_width_p*	Exception address if **ecause** is 2 and **interrupt** is 0 (illegal instruction exception), or trap value otherwise.

6.3.1 Format 3 tvalepc field

This field reports the address of illegal instructions, or the trap value otherwise. This ensures that the address of the faulting instruction is reported for all required cases. The trap value is set to the address of the faulting instruction for hardware breakpoints, access or page faults and instructions, loads or stores that are mis-aligned, but not for illegal instructions (for which it is set to the opcode).

6.4 Format 3 subformat 2 - Context

This packet contains only the context, and is output when the context changes and can be reported imprecisely (see Table 4.4).

Table 6.3: 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.
context	*context_width_p*	The instruction context.

6.5 Format 3 subformat 3 - Support

This packet provides supporting information to aid the decoder. It is issued when

- Trace is enabled or disabled;
- The operating mode changes;
- One or more trace packets cannot be sent (for example, due back-pressure from the packet transport infrastructure).

The **options** field is a placeholder that must be replaced by an implementation specific set of individual bits - one for each of the optional modes supported by the encoder.

6.5.1 Format 3 subformat 3 qual_status field

When tracing ends, the encoder reports the address of the last traced instruction, and follows this with a format 3, subformat 3 (supporting information) packet. Two codes are provided for indicating that tracing has ended: **ended_rep** and **ended_upd**. This relates to exactly the same ambiguous case described in detail in section 6.6.2, and in principle, the mechanism described in that section can be used to disambiguate when the last 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_upd** 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;

Table 6.4: Packet format 3, subformat 3

Field name	Bits	Description
format	2	11 (sync): synchronisation
subformat	2	11 (support): Supporting information for the decoder
enable	1	Indicates if the encoder is enabled
encoder_mode	N	Identifies trace algorithm Details and number of bits implementation dependent. Currently Branch trace is the only mode defined, indicated by the value 0.
qual_status	2	Indicates qualification status 00 (no_change): No change to filter qualification 01 (ended_rep): Qualification ended, preceding **te_inst** sent explicitly to indicate last qualification instruction 10: (trace_lost): One or more packets lost. 11 : (ended_upd): Qualification ended, preceding **te_inst** would have been sent anyway due to an updiscon, even if it wasn't the last qualified instruction)
options	N	Values of all 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)

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**.

6.6 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 6.5: Packet format 2

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

6.6.1 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.

6.6.2 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 last 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 final retired instruction:

- If **updiscon** == !**notify**, this indicates case 2. The decoder must continue until it encounters looplabel a 2nd time;
- If **updiscon** == **notify**, the decoder cannot yet distinguish cases 1 and 3, and must wait for the next packet.
  - If the next packet is a format 3, this is case 3. The decoder has already reached the correct instruction;
  - If the next packet is a format 1/2, this is case 1. The decoder must continue until it encounters looplabel a 2nd time.
- If **notify** == !**address**[**MSB**], this indicates case 4, 1st iteration. The decoder has reached the correct instruction.

This example uses an exception at looplabel + 4, but anything that could cause a format 3 for looplabel + 4 would result in the same behavior: a privilege change, or the expiry of the resync timer. It could also occur if looplabel was the last traced instruction (because tracing was disabled for some reason). See section 6.5.1 for further discussion of this point.

Note: 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.

6.6.3 Format 2 irreport and irdepth

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 last instruction 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 last call (in which case the decoder will correctly stop in the innermost call), or
  - There has been at least one unreported branch since the last 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.

6.7 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 whe 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).

6.7.1 Format 1 updiscon field

See section 6.6.2.

6.7.2 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. The exception to this is when the instruction immediately prior to the final branch causes an uninferable discontinuity, in which case **branches** is set to 31.

Table 6.6: Packet format 1 - address, branch map

Field name	Bits	Description
format	2	01 (diff-delta): includes branch information and may include differential address
branches	5	Number of valid bits **branch_map**. The number of bits of **branch_map** is determined as follows: 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]** (see section 4.2.4).
updiscon	1	If the value of this bit is different from the MSB of **notify**, it indicates that this packet is reporting the instruction following an uninferable discontinuity and is also the instruction before an exception, privilege change or resync (i.e. it will be followed immediately by a format 3 *te_inst*).
irreport	1	If the value of this bit is different from **updiscon**, it indicates that this packet is reporting an instruction that is either: following a return because its address differs from the predicted return address at the top of the implicit_return return address stack, or the last retired before an exception, interrupt, privilege change or resync because it is necessary to report the current address stack depth or nested call count.
irdepth	*return_stack_size_p* + (*return_stack_size_p* > 0 ? 1 : 0) + *call_counter_size_p*	If the value of **irreport** is different from **updiscon**, this field indicates the number of entries on the return address stack (i.e. the entry number of the return that failed) or nested call count. If **irreport** is the same value as **updiscon**, all bits in this field will also be the same value as **updiscon**.

Table 6.7: 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

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.

6.7.3 Format 1 irstatus and irdepth fields

See section 6.6.3.

6.8 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. 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.

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 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 6.8: 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 6.8.1	0 (correctly predicted branches)
branch_count	32	Count of the number of correctly predicted branches, minus 31.
branch_fmt	2	00 (no-addr): Packet does not contain an **address**, and the branch following the last correct prediction failed. 01-11: (cannot occur for this format)

6.8.1 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 9.1). 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 6.5. This provision allows additional formats to be added in future without reducing the efficiency of the existing formats.

6.8.2 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 when **branch_count** is 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.

Table 6.9: 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 6.8.1	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]** (see section 4.2.4).
updiscon	1	If the value of this bit is different from **notify**, it indicates that this packet is reporting the instruction following an uninferable discontinuity and is also the instruction before an exception, privilege change or resync (i.e. it will be followed immediately by a format 3 *te_inst*).
irreport	1	If the value of this bit is different from **updiscon**, it indicates that this packet is reporting an instruction that is either: following a return because its address differs from the predicted return address at the top of the implicit_return return address stack, or the last retired before an exception, interrupt, privilege change or resync because it is necessary to report the current address stack depth or nested call count.
irdepth	*return_stack_size_p* + (*return_stack_size_p* > 0 ? 1 : 0) + *call_counter_size_p*	If the value of **irreport** is different from **updiscon**, this field indicates the number of entries on the return address stack (i.e. the entry number of the return that failed) or nested call count. If **irreport** is the same value as **updiscon**, all bits in this field will also be the same value as **updiscon**.

Table 6.10: 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 6.8.1	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: 0: branch taken 1: 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]**.

Table 6.11: 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 6.8.1	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: no **branch_map** in packet 1-31: (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]**.

6.8.3 Format 0 irstatus and irdepth 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 6.6.3.

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 7

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 (UltraSoC use a 2-bit **msg_type** field to differentiate different trace types from the same source).

By default, all trace packets include both address and data. However, provision is made for run-time configuration options to exclude either the address or 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 6.5). The following sections include examples for all three cases.

7.1 Load and Store

7.1.1 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 LSBs of the address.

Table 7.1: 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	clog2(*data_width_p*/8)	Transfer size is $2^{\textbf{size}}$ bytes
diff	2	00: Full address and data (sync) 01: Differential address, XOR-compressed data 10: Differential address, full data 11: Differentail address, differential data
data_len	**size**	Number of bytes of data is **data_len** + 1
data	8 * (**data_len** + 1)	Data
address	*iaddress_width_p*	Byte address if format is unaligned, otherwise shift left by size to recover byte address

Table 7.2: 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	clog2(*data_width_p*/8)	Transfer size is $2^{\textbf{size}}$ bytes
diff	1	0: Full address (sync) 1: Differential address
address	*iaddress_width_p*	Byte address if format is unaligned, otherwise shift left by size to recover byte address

Table 7.3: 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	clog2(*data_width_p*/8)	Transfer size is $2^{\textbf{size}}$ 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

Table 7.4: Packet format for Split load - Address only

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

Table 7.5: Packet format for Split load - Data only

Field name	**Bits**	**Description**
format	3	Transaction type 100: split load data (other codes select other packet formats)
size	clog2(*data_width_p*/8)	Transfer size is $2^{\mathbf{size}}$ bytes
index	*index_width_p*	Transfer index
resp	2	00: Error (no data) 01: XOR-compressed data 10: Full data 11: Differential data
data	*data_width_p*	Data

7.1.2 size field

The width of this field is 2 bits if max size is 64-bits ($data\_width\_p < 128$), 3 bits if wider.

7.1.3 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.

If only one data compression type is offered, the **diff** field can be 1 bit wide rather than 2 for table 7.3.

7.2 Atomic

7.2.1 size field

Only 1 bit as atomics are either 32 or 64 bits.

7.2.2 diff field

See section 7.1.3.

Table 7.6: Packet format for Unified atomic with address and data

Field name	**Bits**	**Description**
format	3	Transaction type 110: Unified atomic or split atomic address (other codes other packet formats)
subtype	3	Atomic sub-type 000: Swap 001: ADD 010: AND 011: OR 100: XOR 101: MAX 110: MIN 111: reserved
size	1	Transfer size is $2^{\textbf{size}+2}$ bytes
diff	2	00: Full address and data (sync) 01: Differential address, XOR-compressed data 10: Differential address, full data 11: Differential address, differential data
op_len	2 + **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	*iaddress_width_p*	Address, aligned and encoded as per size

Table 7.7: Packet format for Unified atomic with address only

Field name	**Bits**	**Description**
format	3	Transaction type 110: Unified atomic or split atomic address (other codes other packet formats)
subtype	3	Atomic sub-type 000: Swap 001: ADD 010: AND 011: OR 100: XOR 101: MAX 110: MIN 111: reserved
size	1	Transfer size is $2^{\textbf{size}+2}$ bytes
diff	1	0: Full address 1: Differential address
address	*iaddress_width_p*	Address, aligned and encoded as per size

Table 7.8: Packet format for Unified atomic with data only

Field name	Bits	Description
format	3	Transaction type 110: Unified atomic or split atomic address (other codes other packet formats)
subtype	3	Atomic sub-type 000: Swap 001: ADD 010: AND 011: OR 100: XOR 101: MAX 110: MIN 111: reserved
size	1	Transfer size is $2^{\textbf{size}+2}$ bytes
diff	1 or 2	00: Full data (sync) 01: Compressed data (XOR if 2 bits) 10: reserved 11: Differential data
op_len	2 + **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

Table 7.9: Packet format for Split atomic with operand only

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

Table 7.10: Packet format for Split atomic load data only

Field name	Bits	Description
format	3	Transaction type 111: Split atomic data other codes other packet formats
index **resp**	*index_length_p* 2	Transfer index 00: Error (no data) 01: XOR-compressed data 10: full data 11: differential data
data_len	2 + **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)

7.3 CSR

Discussion points:

CSR accesses are not typically that frequent, and so there may be little benefit from offering compression at all. Removing compression may simplify the implementation.

Likewise it may not be worth offering a dedicated format without the operand for read-only accesses.

The benefits of splitting the address are also likely to be minimal, in which case a single 12-bit address is an alternative for consideration.

7.3.1 diff field

See section 7.1.3.

7.3.2 data_len field

2 bits wide if hart has 32-bit CSRs, 3 bits if 64-bit.

7.3.3 addr fields

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

Table 7.11: Packet format for Unified CSR, with address, data and operand

Field name	Bits	Description
format	3	Transaction type 101: CSR (other codes other packet formats)
subtype	2	CSR sub-type 00: RW 01: RS 10: RC 11: reserved
diff	1 or 2	00: Full data (sync) 01: Compressed data (XOR if 2 bits) 10: reserved 11: Differential data
data_len	2 or 3	Number of bytes of data is **data_len** + 1
data	8 * (**data_len** + 1)	Data
addr_msbs	6	Address[11:6]
op_len	2 or 3	Number of bytes of data is **op_len** + 1
operand	8 * (**op_len** + 1)	Operand. Value from rs2 before operator applied
addr_lsbs	6	Address[5:0]

Table 7.12: Packet format for Unified CSR, with address and read-only data (as determined by addr[11:10] = 11)

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

Table 7.13: Packet format for Unified CSR, with address only

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

Chapter 8

Reference Compressed Branch Trace Algorithm

The contents of this chapter are informative only.

A reference algorithm for compressed branch trace is given in figure 8.1. In the diagram, the following terms are used:

- *te_inst.* The name of the packet type emitted by the encoder (see Chapter 6);
- *inst.* Abbreviation for 'instruction';
- *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;
- *e_ccd.* An exception has been signalled, or context has changed and should be treated as an uninferable PC discontinuity (see Table 4.4);
- *ppch.* Privilege has changed, or context has changed and needs to be reported precisely (see Table 4.4);
- *ppch_br.* As above, but branch map not empty;
- *er_ccdn.* Instruction retirement and exception signalled on the same cycle, or context has changed and should be treated as an uninferable PC discontinuity, or context notification (see Table 4.4);
- *exc_only.* Exception signalled without simultaneous retirement;

- *cci.* context change that can be reported imprecisely (see Table 4.4);
- *rep_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);
- *resync count.* A counter used to keep track of when it is necessary to send a synchronization packet (see Section 8.2);
- *max_resync.* The resync counter value that schedules a synchronization packet (see Section 8.2);
- *resync_br.* The resync counter has reached the maximum value and there are entries in the branch map that have not yet been output (see Section 8.2).

Figure 8.1 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.

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 6.5) 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 Table 4.5) 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*);

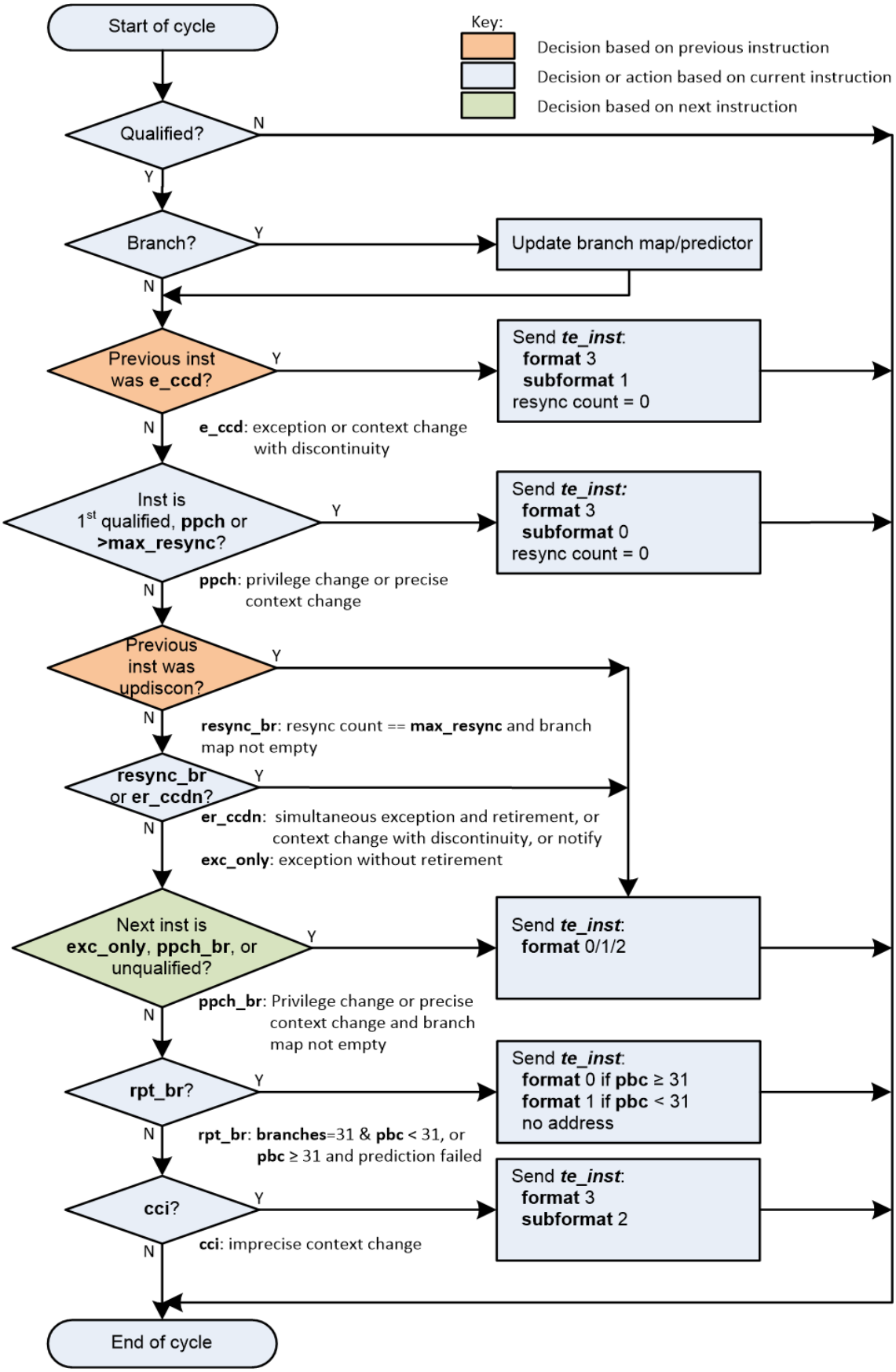

Figure 8.1: Instruction delta trace algorithm

8.1 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 6.8.

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 6.8);
- 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 6.

8.2 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 1 packet for the preceding instruction if there are any unreported branches (because format 3 does not contain a branch map). 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.

8.3 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 8.1 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.

Chapter 9

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 9.1.

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

Table 9.1: Parameters to the encoder

Parameter name	Range	Description
arch_p		The architecture specification version with which the encoder is compliant (0 for initial version).
bpred_size_p		Number of entries in the branch predictor is $2^{\textbf{bpred\_size\_p}}$. 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 $2^{\textbf{cache\_size\_p}}$. Minimum number of entries is 2, so a value of 0 indicates that there is no jump target cache implemented.
call_counter_size_p		Number of bits in the nested call counter is $2^{\textbf{call\_counter\_size\_p}}$. 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
ecause_width_p		Width of exception cause bus
ecause_choice_p		Number of bits of exception cause to match using multiple choice
f0s_width_p		Width of the **subformat** field in format 0 *te_inst* packets (see section 6.8.1).
filter_context_p	0 or 1	Filtering on context supported when 1
filter_excint_p		Filtering on exception cause or interrupt supported when non_zero. Number of nested exceptions supported is $2^{\text{filter\_excint\_p}}$
filter_privilege_p	0 or 1	Filtering on privilege supported when 1
filter_tval_p	0 or 1	Filtering on trap value supported when 1 (provided *filter_excint_p* is non-zero)
iaddress_lsb_p		LSB of instruction address bus to trace. 1 is compressed instructions are supported, 2 otherwise
iaddress_width_p		Width of instruction address bus. This is the same as *DXLEN*
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
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 $2^{\textbf{return\_stack\_size\_p}}$. 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
taken_branches_p		Number of times **iretire**, **itype** etc. are replicated
impdef_width_p		Width of implementation-defined input bus

9.1 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 9.2 lists the required discoverable attributes.

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 9.2 gives an example of how this may be accomplished.

Table 9.2: Required 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
ecause_width	3	*ecause_width_p* - 1
f0s_width	0	*f0s_width_p*
iaddress_lsb	0	*iaddress_lsb_p* - 1
iaddress_width	31	*iaddress_width_p* - 1
nocontext	0	*nocontext*
privilege_width	2	*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 9.3 lists the attributes associated with the filtering recommendations discussed in Chapter 5, and Table 9.4 lists attributes related to other parameters mentioned in this document.

Table 9.3: Optional filtering attributes

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

Table 9.4: Other recommended attributes

Name	Default	Description
ctype_width	1	*ctype_width_p* - 1
ilastsize_width	0	*ilastsize_width_p* - 1
itype_width	4	*itype_width_p* - 1
iretire_width	2	*iretire_width_p* - 1
retires	0	*retires_p* - 1
taken_branches	0	*taken_branches_p* - 1
impdef_width	0	*impdef_width_p* - 1

9.2 Example ipxact description

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

```
<?xml version="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>UltraSoC</ipxact:vendor>
   <ipxact:library>TraceEncoder</ipxact:library>
   <ipxact:name>TraceEncoder</ipxact:name>
   <ipxact:version>0.8</ipxact:version>
   <ipxact:memoryMaps>
      <ipxact:memoryMap>
         <ipxact:name>Trace Encoder Register Map</ipxact:name>
         <ipxact:addressBlock>
            <ipxact:name>>Trace Encoder Register Address Block</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

Future Directions

The current focus is the compressed branch trace, however there a number of other types of processor trace that would be useful (detailed below in no particular order). These should be considered as possible features that maybe added in the future, once the current scope has been completed.

10.1 Data trace

The trace encoder will output packets to communicate information about loads and stores to an off-chip decoder. To reduce the amount of bandwidth required, reporting data values will be optional, and both address and data will be able to be encoded differentially when it is beneficial to do so. This entails outputting the difference between the new value and the previous value of the same transfer size, irrespective of transfer direction.

Unencoded values will be used for synchronisation and at other times.

10.2 Fast profiling

In this mode the encoder will provide a non-intrusive alternative to the traditional method of profiling that requires the processor to be halted periodically so that the program counter can be sampled. The encoder will issue packets when an exception, call or return is detected, to report the next instruction executed (i.e. the destination instruction). Optionally, the encoder will also be able to report the current instruction (i.e. the source instruction).

10.3 Inter-instruction cycle counts

In this mode the encoder will trace where the hart is stalling by reporting the number of cycles between successive instruction retirements.

10.4 Transport

After the current charter has been satisfied the transport mechanism should be defined and standardised. This will include Aurora based serdes, PCIe and Ethernet.

Chapter 11

Decoder

This decoder implementation assumes there is no branch predictor or return address stack (*return_stack_size_p* and *bpred_size_p* both zero).

Reference C-code implementations of both the encoder and decoder can be found at https://github.com/riscv/riscv-trace-spec/tree/master/te_codec/src.

11.1 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 messages
global       options                     # Operating mode flags
global array return_stack                # Array holding return address stack
global       irstack_depth = 0           # Depth of the return address stack
```

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

    inferred_address = FALSE
    address       = (te_inst.address << discovery_response.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)
      follow_execution_path(address, te_inst)
    else
      pc           = address
      last_pc      = pc # previous pc not known but ensures correct
                        #  operation for is_sequential_jump()
    start_of_trace = FALSE
    irstack_depth  = 0

  else
    if (start_of_trace) # This should not be possible!
      ERROR: Expecting trace to start with format 3
      return
    if (te_inst.format == 2 or te_inst.branches != 0)
      stop_at_last_branch = FALSE
      if (options.full_address)
        address  = (te_inst.address << discovery_response.iaddress_lsb)
      else
        address += (te_inst.address << discovery_response.iaddress_lsb)
    if (te_inst.format == 1)
      stop_at_last_branch = (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)
```

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

  local previous_address = pc
  local stop_here        = FALSE
  while (TRUE)
    if (inferred_address) # iterate again from previously reported address to
                          #   find second occurrence
      stop_here = next_pc(previous_address)
      if (stop_here)
        inferred_address = FALSE
    else
      stop_here = next_pc(address)
      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)
        stop_at_last_branch = FALSE
        return
      if (stop_here)
        # Reached reported address following an uninferable discontinuity - stop here
        if (branches > (is_branch(get_instr(pc)) ? 1 : 0))
          # Check all branches processed (except 1 if this instruction is a branch)
          ERROR: unprocessed branches
        return
      if (te_inst.format != 3 and pc == address and !stop_at_last_branch and
        (te_inst.notify != get_previous_bit(te_inst, "notify")) and
        (branches == (is_branch(get_instr(pc)) ? 1 : 0)))
          # All branches processed, and reached reported address due to notification,
          # not as an uninferable jump target
        return
      if (te_inst.format != 3 and pc == address and !stop_at_last_branch and
        !is_uninferable_discon(get_instr(last_pc)) and
        (te_inst.updiscon == get_previous_bit(te_inst, "updiscon")) and
        (branches == (is_branch(get_instr(pc)) ? 1 : 0)) and
        ((te_inst.irreport == get_previous_bit(te_inst, "irreport")) or
         te_inst.irdepth == irstack_depth))
          # All branches 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
        inferred_address = TRUE
        return
      if (te_inst.format == 3 and pc == address and
        (branches == (is_branch(get_instr(pc)) ? 1 : 0)))
        # All branches processed, and reached reported address
        return
```

```
# Compute next PC #
function next_pc (address)

  local instr     = get_instr(pc)
  local this_pc   = pc
  local stop_here = FALSE

  if (is_inferable_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(instr))
    pc = pop_return_stack()
  else if (is_uninferable_discon(instr))
    if (stop_at_last_branch)
      ERROR: unexpected uninferable discontinuity
    else
      pc        = address
      stop_here = TRUE
  else if (is_taken_branch(instr))
    pc += instr.imm
  else
    pc += instruction_size(instr)

  if (is_call(instr))
    push_return_stack(this_pc)

  last_pc = this_pc

  return stop_here

# Process support packet #
function process_support (te_inst)

  local stop_here = FALSE

  options = te_inst.options
    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_upd and inferred_address)
      local previous_address = pc
      inferred_address       = FALSE
      while (TRUE)
        stop_here = next_pc(previous_address)
        if (stop_here)
          return
    return
```

```
# Determine if instruction is a branch, adjust branch count/map,
#   and return taken status #
function is_taken_branch (instr)
  local bool taken = FALSE

  if (!is_branch(instr))
    return FALSE

  if (branches == 0)
    ERROR: cannot resolve branch
  else
    taken = !branch_map[0]
    branches--
    branch_map >> 1

  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 instruction is an inferable jump #
function is_inferable_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 an uninferable discontinuity #
function is_uninferable_discon (instr)

  if (is_uninferable_jump(instr) or
      (instr.opcode == URET)      or
      (instr.opcode == SRET)      or
      (instr.opcode == MRET)      or
      (instr.opcode == DRET)      or
      (instr.opcode == ECALL)     or
      (instr.opcode == EBREAK)    or
      (instr.opcode == C.EBREAK))
    return TRUE

  return FALSE

# Determine if instruction is a sequentially inferable 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 inferable 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) or
      (instr.opcode == C.JALR)                 or
      (instr.opcode == JAL  and instr.rd == 1) or
      (instr.opcode == C.JAL))
    return TRUE

  return FALSE

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

  if (options.implicit_return == 0) # Implicit return mode disabled
    return FALSE

  if ((instr.opcode == JALR and instr.rs1 == 1 and instr.rd == 0) or
      (instr.opcode == C.JR and instr.rs1 == 1))
    if ((te_inst.irreport != get_previous_bit(te_inst, "irreport")) and
        te_inst.irdepth == irstack_depth)
      return FALSE
    return (irstack_depth > 0)

  return FALSE
```

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

  if (options.implicit_return == 0) # Implicit return mode disabled
    return

  local irstack_depth_max = discovery_response.return_stack_size ?
                            2**discovery_response.return_stack_size :
                            2**discovery_response.call_counter_size
  local instr             = get_instr(address)
  local link              = address

  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]

  link += instruction_size(instr)

  return_stack[irstack_depth] = link
  irstack_depth++

  return

# Pop address from return stack #
function pop_return_stack ()

  irstack_depth-- # function not called if irstack_depth is 0, so no need
                  #  to check for underflow
  local  link = return_stack[irstack_depth]

  return link
```

Chapter 12

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]