RISC-V Processor Trace
Version 0.026-DRAFT
a0620d0f6d2b7606e5f5380393c234fae86645eb

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

UltraSoC Technologies Ltd.

May 22, 2019

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 and one method of achieving this is via Processor Branch Trace.

This works by tracking execution from a known start address and sending messages about the 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.

Software, known as a decoder, will take this compressed branch trace and reconstruct the program flow. This can be done off-line or whilst the system is executing.

In RISC-V, all instructions are executed unconditionally or at least their execution can be determined based on the program, the instructions between the deltas are assumed to be executed sequentially. This characteristic means that there is no need to report them via 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 source code.

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 executed beforehand must be traced.

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.

1.0.1 Nomenclature

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

Items in *italics* refer to parameters either built into the hardware or configurable hardware values.

A decoder is a piece of software that takes the packets emitted by the encoder and is able to reconstruct the execution flow of the code executed in the RISC-V core.

Chapter 2

Branch Trace

2.1 Instruction Delta Tracing

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 trace delta modes provide 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 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. There is no need for either assembly or high-level source code to be available, although such source code will aid the debugger in presenting the decoded trace.

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.

2.1.1 Sequential Instructions

For instruction set architectures where all instructions are executed unconditionally or at least their execution can be determined based on the program, the instructions between the deltas are assumed to be executed sequentially. This characteristic means that there is no need to report them via the trace, only whether the branches were taken or not and the addresses of taken indirect jumps.

2.1.2 Uninferable PC Discontinuity

If the program counter is changed by an amount that cannot be inferred 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 jumps, where the next instruction address is determined by the contents of a register rather than a constant embedded in the source code.

2.1.3 Branches

When a branch occurs, the decoder must be informed of whether it was taken or not. For a direct branch, this is sufficient. There are no indirect branches in RISC-V; an indirect jump is an uninferable PC discontinuity.

2.1.4 Interrupts and Exceptions

Interrupts are a different type of delta, they 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 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 executed beforehand must be traced. Following this, for an interrupt or exception, the next valid instruction address (the first of the interrupt or exception handler) must be traced in order to instruct the trace decoder to classify the instruction as an indirect jump even if it is not.

2.1.5 Synchronization

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

The instruction trace encoder needs to synchronise fully:

- After a reset.
- When tracing starts.
- If the instruction is the first of an interrupt service routine or exception handler (hardware context change).
- After a prolonged period of time.

2.1.6 Optional and run-time configurable modes

The following modes are optional, and if present must be run-time selectable. The active run-time options must be reported in the *te_support* packet, which is issued by the encoder whenever the encoder configuration is changed.

2.1.6.1 Full address

All packet formats apart from format 3 output addresses in differential form by default. An option to output full addresses for all packet formats is a useful debugging aid for software decoder developers. It will always result in less efficient trace encoding.

2.1.6.2 Implicit exception

The exception handler base address is specified by ***utvec/stvec/mtvec***, and in some RISC-V implementations the lower address bits can be specified by ***ucause/scause/mcause***. By default, both these values are reported when an exception or interrupt occurs, via the the format 3, subformat 1 packet. The 'implicit exception' option omits the trap handler address, and will improve efficiency in cases where the decoder can infer the address of the trap handler from just the exception cause.

2.1.6.3 Implicit return

Although a function return is usually an indirect jump, well behaved programs following a calling convention return to the point in the program from which the function was called, and as such it is possible to determine the execution path without being explicitly notified of the destination address of the return. The 'implicit return' option can result in very significant improvements in trace encoder efficiency. It utilizes 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 3.1.1 for definitions). The counter will not over or underflow, and is reset to 0 whenever a format 3 *te_inst* 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 a *te_inst* message).

2.1.6.4 Branch prediction

Whilst recording the taken/not-taken status of each branch in a branch map is efficient, there are some cases where this can result in a relatively large volume of trace. For example:

- Executing tight loops of straight-line code. 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.

The prediction scheme implemented in the encoder will need to be modelled in the decoder software. The predictor shall comprise a lookup table of 2^N entries, where N is specified by a parameter. Each entry is indexed by bits N:1 of the instruction address (or N+1:2 if compressed instructions aren't supported), and each contains a 2-bit prediction state:

- 00: predict 0, transition to 01 if prediction fails;
- 01: predict 0, transition to 00 if prediction succeeds, else 11;
- 11: predict 1, transition to 10 if prediction fails;
- 10: predict 1, transition to 11 if prediction succeeds, else 00.

We could also consider the gShare predictor (see Hennessy & Patterson). Some further experimentation is needed to determine the benefits of different lookup table sizes and predictor algorithms.

The lookup table entries are initialized to 00 when a format 3 *te_inst* packet is sent.

Chapter 3

Ingress Port

3.1 Interface Requirements

This section describes in general terms the information which must be passed from the RISC-V core to the trace encoder, 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 core;
- The *instruction_type* of retired instructions for:
  - Jumps with a target that cannot be inferred from the source code;
  - Taken branches;
  - Return from exception or interrupt (****ret*** instructions).
- The *instruction_address* for:
  - Jumps with a target that *cannot* be inferred from the source code;
  - Taken branches;
  - The instruction executed immediately after a jump or taken branch (also referred to as the target or destination of the jump or taken branch);
  - The last instruction executed before an exception or interrupt;
  - The first instruction executed following an exception or interrupt;
  - The last instruction executed before a privilege change;
  - The first instruction executed following a privilege change;
  - The first and last instruction being retired.

- The number of nontaken branches being retired.

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;
  - Nontaken branches.
- If context is supported then the *instruction_address* for:
  - The last instruction executed before a context change;
  - The first instruction executed following a context change.

The mandatory information is the bare-minimum required to implement the branch trace algorithm outlined in Chapter 5. The optional information facilitates alternative or improved trace algorithms:

- Implicit return mode (see Section 2.1.6.3) 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 2.1.6.4). This requires the encoder to be aware of the address of all branches, whether they are taken or not.

3.1.1 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). This means the target of the jump is supplied via

- a constant;
- a register which contains a constant (e.g. the destination of an ***lui*** or ***c.lui***);
- a register which contains a constant offset from the PC (e.g. the destination of an ***auipc***).

Jumps which are not *inferable* are by definition *uninferable*.

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

- *Calls*:
  - ***jal*** x1;
  - ***jal*** x5;
  - ***jalr*** x1, rs where rs != x1;
  - ***jalr*** x5, rs where rs != x5;
  - ***c.jalr*** rs1.
- *Tail-calls*:
  - ***jalr*** x0, rs where rs != x1 and rs != x5;
  - ***c.jr*** rs1 where rs1 != x1 and rs1 != x5.
- *Returns*:
  - ***jalr*** x0, rs where rs == x1 or rs == x5;
  - ***c.jr*** rs1 where rs1 == x1 or rs1 == x5.
- *Co-routine swap*:
  - ***jalr*** x1, x1;
  - ***jalr*** x5, x5.
- *Other*:
  - ***jal*** rd where rd != x1 and rd != x5;
  - ***jalr*** rd, rs where rd != x0 and rd != x1 and rd != x5.

Table 3.1: Core-Encoder signals - common

Signal	Function
itype[*itype_width_p*-1:0]	Termination type of the instruction block (see Section 3.1.1 for definitions of codes 6 - 15): 0: Final instruction in the block is none of the other named **itype** codes; 1: Exception. An exception occurred following the final retired instruction in the block; 2: Interrupt. An interrupt occurred following the final retired instruction in the block; 3: Exception return; 4: Nontaken branch; 5: Taken branch; 6: reserved; 7: Co-routine swap; 8: Uninferable call; 9: Inferrable call; 10: Uninferable tail-call; 11: Inferrable tail-call; 12: Uninferable return; 13: Inferrable return; 14: Other uninferable jump; 15: Other inferable jump.
cause[*ecause_width_p*-1:0]	Exception or interrupt cause (***ucause/scause/mcause***), Ignored unless **itype**=1 or 2.
tval[*iaddress_width_p*-1:0]	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]	Privilege level for all instructions in this block.
context[*context_width_p*-1:0]	Context and/or Hart ID for all instructions in this block.
iaddr[*iaddress_width_p*-1:0]	The address of the 1st instruction retired in this block. Invalid if **iretire**=0
context_type[*context_type_width_p*-1:0]	Behavior type of **context** 0: Context change with discontinuity; 1: Precise context change; 2: Imprecise context change; 3: Notification.

Table 3.2: Core-Encoder signals - multiple retirement

Signal	Function
iretire[*iretire_width_p*-1:0]	Number of halfwords represented by instructions retired in this block.
ilastsize[*ilastsize_width_p*-1:0]	The size of the last retired instruction. For cases where the address of the last retired instruction is needed.

Table 3.3: Core-Encoder signals - multiple non-taken branches

Signal	Function
ntkn[*ntkn_width_p*-1:0]	Number of nontaken branches in this block.

3.2 Instruction Interface

This section describes the interface between a RISC-V core and the trace encoder that conveys the information described in the previous section.

Tables 3.1, 3.2 and 3.3 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 cores that can only retire one instruction at a time, the signalling can be simplified, and this is discussed subsequently in Section 3.2.1.

The information presented on the ingress port 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.

If address translation is enabled, **iaddr** is a virtual address, else it is a physical address. Virtual addresses narrower than *iaddress_width_p* bits must be sign-extended to make computation of differential addresses easier, and physical addresses narrower than *iaddress_width_p* bits must be zero-extended.

Cores can retire multiple non-taken branches per clock cycle, indicated via **ntkn**. However, a consequence of this is that the encoder will be unaware of the addresses of some non-taken branches, which will prevent the use of a branch predictor to improve compression (see Section 2.1.6.4. For cores that can only retire a maximum of one non-taken branch per clock cycle, **ntkn** can be omitted, provided all non-taken branches are indicated via **itype**. The number of non-taken branches is **ntkn** if **ntkn** is non-zero, or 1 if **itype** = 4 and **ntkn** is zero. In other words, if for example **ntkn** is 2 and **itype** = 4, the encoder will interpret this as 2 non-taken branches, not 3.

For cores that can retire a maximum of N taken branches per clock cycle, the signal group (**iretire**, **itype**, **ntkn** (if present), **ilastsize**, **iaddr**) must be replicated N times. Signal group 0 represents

Table 3.4: Core-Encoder signals - single retirement

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

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 **priv**, **context**, **context_type**, **cause** and **tval** 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 taken branches are retired in a cycle, then lower numbered groups must be used first. For example, if there is one taken branch, use only group 0, if there are two taken branches, instructions up to the 1st taken branch must be reported in group 0 and instructions up to the 2nd taken branch must be reported in group 1 and so on.

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

- The Hart ID;
- The software thread ID;
- 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.

Table 3.5 specifies the actions for the various **context_type** values.

3.2.1 Simplifications for single-retirement

For cores that can only retire one instruction at a time, the interface can be simplified to the signals listed in tables 3.1 and 3.4. 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;
- As the number of non-taken branches retired is never more than 1, and can always be indicated via **itype**, **ntkn** is not necessary.

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**. **ilastsize** and **ntkn** encoder inputs must be tied low when attached to a single-retirement core that does not provide these outputs.

Table 3.5: Call/return **context_type** values and corresponding actions

Type	**Value**	**Actions**
Context change with discontinuity	0	An example would be a change of Hart. Need to report the last instruction executed on the previous context, as well as the 1st on the new context. Treated the same as an exception.
Precise context change	1	Need to output the address of the 1st instruction, and the new context. If there were unreported branches beforehand, these need to be output first. Treated the same as a privilege change.
Imprecise context change	2	An example would be a SW thread 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).
Notification	3	An example would be a watchpoint. Need to output the address of the watchpoint instruction. The context itself is not output.

3.2.2 Alternative Multiple-retirement interface configurations

For a core that can retire multiple instructions per cycle, but no more than one taken branch, the preferred solution is to use one of each of the signals from tables 3.1, 3.2 and optionally 3.3. However, an alternative approach would be to provide explicit details of every instruction retired by using N sets of the signal group (**iretire**, **itype**) from tables 3.1 and 3.4 with the groups detailing one instruction each (replicating the single retirement example N times).

3.2.3 Example Retirement Sequences

3.2.4 Sideband signals

In some circumstances there will be some sideband signals which may affect the encoder's behavior, for example to start and/or stop encoding. There will sometimes be cases where the encoder may be required to affect the behaviour of the core, for example stalling.

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

Table 3.6: Example 1 : 9 Instructions retired over three cycles, 2 branches

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

Table 3.7: User Sideband Encoder Ingress signals

Signal	**Function**
user [*user_width_p*-1:0]	Filtering sideband signals (see Chapter 4)
halted	Core is stalled or halted
reset	Core in reset

3.2.5 Parameters

The encoder will have some configurable or variable parameters. Some of these are related to port widths whilst others may indicate the presence or otherwise of various features, e.g. filter or comparators. Table 3.9 outlines the list of parameters.

How the parameters are input to the encoder is implementation specific. The number range of some of the parameters may be implementation specific.

Table 3.8: User Sideband Encoder Egress signals

Signal	**Function**
stall	Stall request to core

Table 3.9: Parameters to the encoder

Parameter name	**Range**	**Description**
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.
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.
context_type_width_p	2	Width of the **context_type** 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
filter_context_p	0 or 1	Filtering on context supported when 1
filter_ecause_p		Filtering on exception cause supported when non_zero. Number of nested exceptions supported is $2^{\text{filter\_ecause\_p}}$
filter_interrupt_p	0 or 1	Filtering on interrupt supported when 1
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
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 *XLEN*
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
notval_p	0 or 1	Exclude trap value from *te_inst* packets if 1
ntkn_width_p		Width of the **ntkn** bus
privilege_width_p		Width of privilege bus
retires_p		Maximum number of instructions that can be retired per block
taken_branches_p		Number of times **iretire**, **itype**, **ntkn** is replicated
user_width_p		Width of user-defined filter qualifier input bus

3.2.6 Discovery of parameter values

The parameters used by the encoder must be discoverable at runtime. Some external entity, for example a debugger or a supervisory hart, would issue a discovery command to the encoder. The encoder will provide the discovery information as encapsulated in the following parameters in one or more different formats. The preferred format would be in 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.

- *minor_revision*. Identifies the minor revision.
- *version*. Identifies the module version.
- *comparators*. The number of comparators is *comparators* + 1.
- *filters*. Number of filters is *filters* + 1.
- *bpred_size*. Number of entries in the branch predictor is $2^{\mathbf{bpred\_size}}$. No predictor if 0.
- *call_counter_size*. Width of the nested call counter is $2^{\mathbf{call\_counter\_size}}$. No counter if 0.
- *context_type_width*. Width of the **context_type** bus is *context_type_width* + 1.
- *context_width*. Width of context input bus is *context_width* + 1.
- *ecause_choice*. Number of LSBs of the ecause input bus that can be filtered using multiple choice.
- *ecause_width*. Width of the ecause input bus is *ecause_width* + 1.
- *filter_context*. Filtering on the *context* input bus supported when 1.
- *filter_ecause*. Filtering on the ecause input bus supported when non-zero. Number of nested exceptions supported is $2^{\frac{filter\_ecause}{}}$.
- *filter_interrupt*. Filtering on the interrupt input signal supported when 1.
- *filter_privilege*. Filtering on the privilege input bus supported when 1.
- *filter_tval*. Filtering on the tval input bus supported when 1.
- *iaddress_lsb*. LSB of iaddress output in trace encoder data messages is *iaddress_lsb* + 1.
- *iaddress_width*. Width of the iaddress input bus is *iaddress_width* + 1.
- *ilastsize_width*. Width of the **ilastsize** bus is *ilastsize_width* + 1.
- *itype_width*. Width of the **itype** bus is *itype_width* + 1.
- *iretire_width*. Width of the **iretire** bus is *iretire_width* + 1.
- *nocontext*. Context ignored when 1.

- *notval*. Trap value ignored when 1.
- *ntkn_width*. Width of the **ntkn** bus is *ntkn_width* + 1.
- *privilege_width*. Width of the privilege input bus is *privilege_width* + 1.
- *retires*. Maximum number of instructions that can be retired per block is *retires* + 1.
- *rv32*. ISA is RV32 when 1.
- *taken_branches*. Number of times **iretire**, **itype**, **ntkn** is replicated is *taken_branches* + 1.
- *user_width*. Width of the **user** bus is *user_width* + 1.

Chapter 4

Filtering

The instruction trace encoder must be able to filter on the following inputs to the encoder:

- The instruction address
- The context
- The exception cause
- Whether the exception is an interrupt or not
- The privilege level
- Tval
- User specific signals

Internal to the encoder will be several comparators and filters. The actual number of these will vary for different classes of devices. The filters and comparators must be configured to provide the trace and filtering required. There will be three command types needed to set up the filtering operation.

1. Set up comparator
    - Which input bus to compare
        (a) address
        (b) context
        (c) tval
    - Which comparator(s) to use which filtering operation to enable
        (a) *eq*
        (b) *neq*
        (c) *lt*
        (d) *lte*

(e) *gt*
(f) *gte*
(g) *always*

2. Value e.g. start address
3. Set up filter
4. Set match
    - Configure matching behaviour for exception, privilege and user sideband

The user may wish to:

1. Trace instructions between a range of addresses
2. Trace instruction from one address to another
3. Trace interrupt service routine
4. Start/stop trace when in a particular privilege level
5. Start/stop trace when context changes or is a particular value
    - This can be HARTs and/or software contexts. If the latter this would be
    - Start/stop trace when specific instruction
    - Start/stop based on **user** sideband signals
    - This could be the specific CSR value being presented to the Encoder

4.1 Using trigger outputs from Debug Module

The debug module of the RISC-V core may have a trigger unit. This exposes a 4-bit field as shown in table 4.1.

Table 4.1: Debug module trigger support (mcontrol)

Value	**Description**
2	Trace on
3	Trace off
4	Trace single. The 'single' action for an instruction trigger could cause just that instruction to be traced if connected to a **user** input; Alternatively it could be used to assert the 'Notification' **context_type** to generate a watchpoint trace.

Chapter 5

Example Algorithm

An example algorithm for compressed branch trace is given in figure 5.1. In the diagram, the following terms are used:

- *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, or a non-zero **ntkn**);
- *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;
- *inst.* Abbreviation for 'instruction';
- *resync count.* A counter used to keep track of when it is necessary to send a synchronization packet (see Section 2.1.5, final bullet). The exact mechanism for incrementing this counter 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 last synchronization message was sent;
- *max_resync.* The resync counter value that schedules a synchronization packet;
- *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);
- *te_inst.* The name of the packet type emitted by the encoder (see Chapter 6);
- *e_ccd.* An exception has been signalled, or context has changed and should be treated as an uninferable PC discontinuity (see Table 3.5);
- *ppch.* Privilege has changed, or context has changed and needs to be reported precisely (see Table 3.5);
- *ppch_br.* As above, but branch map not empty;
- *resync_br.* The resync counter has reached the maximum value and there are entries in the branch map that have not yet been output. These must be output before the subsequent synchronization packet, which does not report branch map history;

- *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 3.5);
- *exc_only.* Exception signalled without simultaneous retirement;
- *cci.* context change that can be reported imprecisely (see Table 3.5).

Figure 5.1 shows instruction by instruction behavior, as would be seen in a single-retirement system only. Whilst the ingress port allows the RISC-V core 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 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 (or last) 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 Chapter 6) 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 last 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 should 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 3.7) the encoder will behave as if it has received an unqualified instruction (output *te_inst* reporting the address of the last instruction, followed by *te_support*);

5.1 Full vs Differential Addresses

Addresses can be output in one of two ways: *full* or *differential*.

- The *full* address is the actual address of the current instruction;
- The *differential* address is 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.

Packet formats 1 and 2 include a differential address, whilst format 3 includes the full address.

5.2 Format selection

In all cases but one, the packet format (3) is determined only by a 'yes' outcome from the associated decision. The choice between formats 1 or 2 for the case in the middle of the diagram needs further explanation.

If there are no branches that need to be reported, packet format 2 is used.

If there are branches to report, format 1 is used.

If branch prediction is supported and is enabled, then there is a choice of whether to output a full branch map, or a count of correctly predicted branches. In order to choose the count, the number of correctly predicted branches must be 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. If all 31 unreported branches were correctly predicted, then the encoder starts counting subsequent correct predictions, and will output a count under the following conditions:

- A branch is mis-predicted. The count value will be the number of correctly predicted branches, minus 31. **branch_fmt** will be 01, indicating that the next branch failed its prediction. No address information is provided;
- 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) with **branch_fmt** set to 10. The packet also contains **mispred**, indicating whether prediction of the next branch failed.
- The branch count reaches its maximum value (0xffff). Again, **branch_fmt** will be set to 10. Strictly speaking an address isn't required for this case, but it will occur so rarely that the bandwidth impact can be ignored

Packet formats 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.

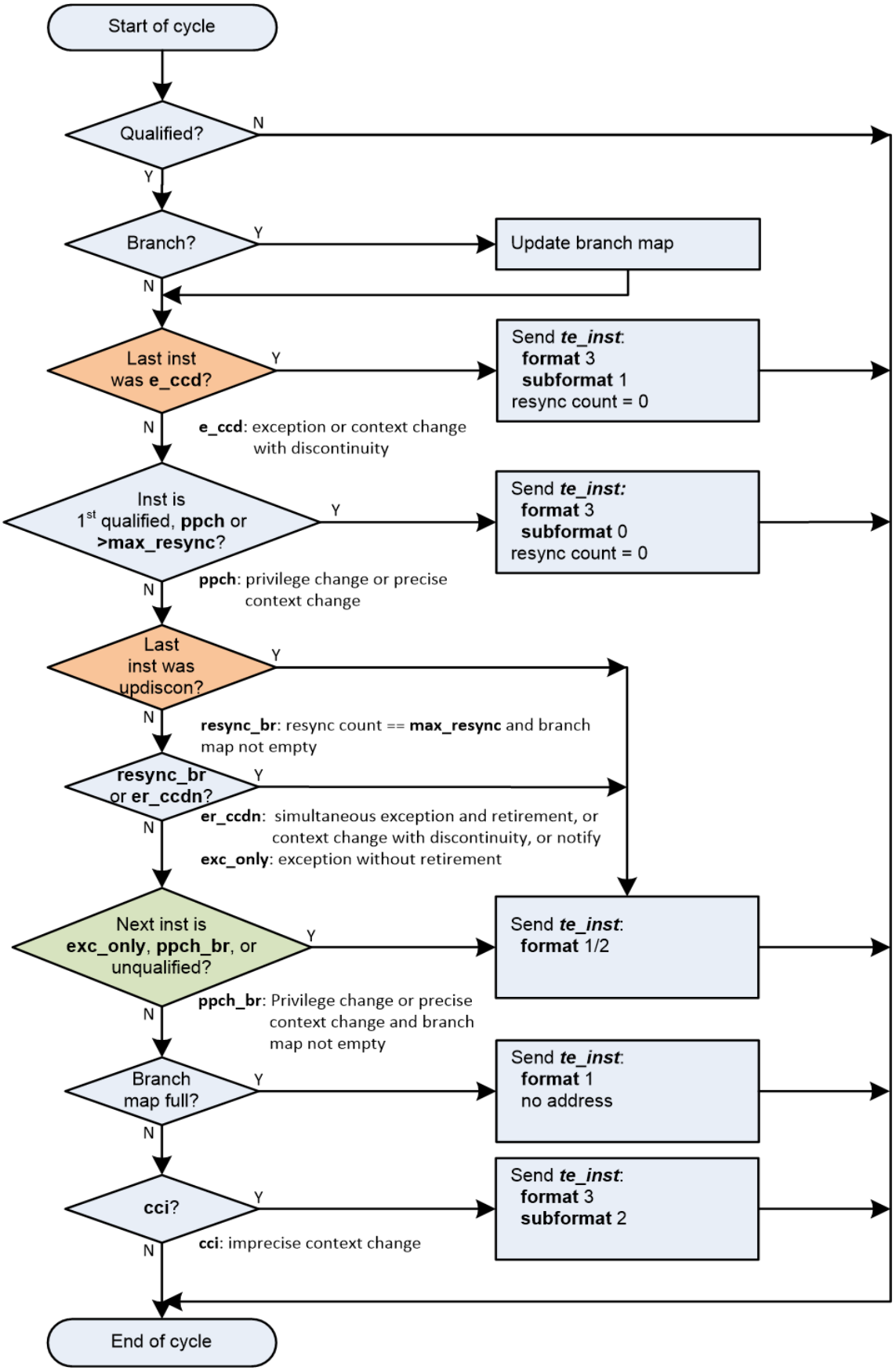

Figure 5.1: Delta Mode 1 instruction trace algorithm

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. An example of how this technique is used to choose between address formats is given in Section 5.1. The same principle can be applied to the entire packet, and the length (typically given in bytes) adjusted accordingly.

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.

Table 6.1: Packet Payload Format 1 - with address

Field name	**Bits**	**Description**
format	2	01 (diff-delta): includes branch map and may include differential 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 2-9: 9 bits 10-17: 17 bits 18-25: 25 bits 26-31: 31 bits For example if branches = 12, the branch-map is 17 bits long, and the 12 LSBs are valid. In most cases when the branch map is full there is no need to report an address, and 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.
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.
updiscon	1	If the value of this bit is different from the MSB of **address**, it indicates that this packet is reporting the instruction following an uninferable iscontinuity 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*).

Table 6.2: Packet Payload Format 1 - no address, branch map

Field name	Bits	Description
format	2	01 (diff-delta): includes branch map 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
branch_fmt	2	Both bits set to the same value as **branch_map[MSB]** indicates that the preceding field is **branch_map**.

Table 6.3: Packet Payload Format 1 - no address, branch count

Field name	Bits	Description
format	2	01 (diff-delta): includes branch map 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 31-1: (cannot occur for this format)
branch_count	16	Count of the number of correctly predicted branches, minus 31.
reserved	15	zero
branch_fmt	2	Set to 001, indicates that the packet contains a **branch_count** field, no **address** field, and that the next branch failed prediction.

Table 6.4: Packet Payload Format 1 - differential address, branch count

Field name	Bits	Description
format	2	01 (diff-delta): includes branch map 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 31-1: (cannot occur for this format)
branch_count	16	Count of the number of correctly predicted branches, minus 31.
address (LSBs) **branch_fmt**	15 2	15 LSBs of differential instruction address. Set to 10, indicates that the packet contains a **branch_count** field and an **address** field. This will be the case if the packet is output because it is necessary to report an address (e.g. following an updiscon, or if the next instruction is an exception), or because **branch_count** has reached 0xffff).
mispred	1	Set to 1 if next branch failed prediction.
address (MSBs)	*iaddress_width_p - iaddress_lsb_p - 15*	MSBs of the differential instruction address.

Table 6.5: Packet Payload Format 2

Field name	Bits	Description
format	2	10 (addr-only): differential address and no branch map
address	*iaddress_width_p - iaddress_lsb_p*	Differential instruction address.
updiscon	1	If the value of this bit is different from the MSB of **address**, 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*).

6.1 Further notes on packet format details

Some of the packet fields warrant further explanation.

6.1.1 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

Table 6.6: Packet Payload Format 3, subformat 0

Field name	Bits	Description
format	2	11 (sync): synchronisation
subformat	2	00 (start): Start of tracing, or resync
context	*context_width_p*, or 0 if *nocontext_p* is 1	The instruction context
privilege	*privilege_width_p*	The current privilege level
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
branch	1	If the address points to a branch instruction, the branch is not taken if the value of this bit is different from the MSB of **address**. Set to the same value as the MSB of **address** if the branch is taken or the instruction is not a branch.

Table 6.7: Packet Payload Format 3, subformat 1

Field name	Bits	Description
format	2	11 (sync): synchronisation
subformat	2	01 (exception): Exception cause and trap handler address
context	*context_width_p*, or 0 if *nocontext_p* is 1	The instruction context
privilege	*privilege_width_p*	The current privilege level
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
ecause	*ecause_width_p*	Exception cause
interrupt	1	Interrupt
tval	*iaddress_width_p*, or 0 if *notval_p* is 1	Trap value
branch	1	If the address points to a branch instruction, the branch is not taken if the value of this bit is different from the MSB of **tval**. Set to the same value as the MSB of **tval** if the branch is taken or the instruction is not a branch.

Table 6.8: Packet Payload Format 3, subformat 2

Field name	Bits	Description
format	2	11 (sync): synchronisation
subformat	2	10 (context): Context change
context	*context_width_p*	The instruction context

Table 6.9: Packet Payload 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 encoder is enabled
encoder_mode	N	Identifies trace algorithm Details implementation dependent. Currently Branch trace is the only mode defined.
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: (packet_lost): One or more packets lost. 11 : (ended_ntr): Qualification ended, no unreported instructions (so preceeding **te_inst** would have been sent anyway, 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 - '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 field* - 'branch prediction' Branch predictor enabled - 'full address' Always output full addresses (SW debug option)

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 instruction that matches the filtering criteria is a branch, and this is then followed immediately by an exception. This results in format 3 packets being generated on two consecutive cycles. 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 5.1.

This bit is encoded so that most of the time it will take the same value as the MSB of the preceding field, and will therefore compress away, in order to minimize the efficiency impact. Branches are unlikely to be reported using a format 3 packet apart from if the 1st traced instruction is a branch, or if the instruction reported when the resync timer expires is a branch.

6.1.2 Format 1/2 updiscon 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 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 three 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** == **address**[**MSB**]. The next packet will be a format 1/2;
2. Code executes through to the end of the 1st loop iteration, but there is an exception, privilege change or resync at the instruction following the ***JALR*** (i.e. at looplabel + 4). In this case, the encoder reports looplabel using format 1/2 following the ***JALR***, with **updiscon** == !**address**[**MSB**], and the next packet is a format 3;
3. An exception occurs after the 1st execution of looplabel. In this case, the encoder reports looplabel using format 0/1/2 and again, **updiscon** == **address**[**MSB**], and the next packet is a format 3.

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 **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** == !**address**[**MSB**], this indicates case 2. The decoder must continue until it encounters looplabel a 2nd time;
- If **updiscon** == **address**[**MSB**], 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.

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 next section for further discussion of this point.

6.1.3 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_ntr**. This relates to exactly the same ambiguous case described in the previous section, and in principle, the mechanism described in that section can be used to disambiguate when the last traced instruction is at looplabel + 4. 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 priviledge change or exception is straightforward, but determining whether the next instruction meets the filtering criteria is much more involved, and this information won't

typically be available, at least not without adding an additional pipeline stage, which is expensive. This means a different mechanism is required, and that is provided by having two codes to indicate that tracing has ended:

- **ended_rep** indicates that the preceding packet would not have been issued if tracing hadn't ended, which means that tracing stopped after executing looplabel in the 1st loop iteration;
- **ended_ntr** indicates that the preceding packet would have been issued anyway, which means that tracing stopped after executing looplabel in the 2nd loop iteration;

If the encoder implementation does have early access to the filtering results, and the designer chooses to use the **updiscon** when thet last qualified instruction is also the instruction following an uninferable PC discontinuity, loss of qualification shoujld always be indicated using **ended_ntr**.

6.1.4 Format 1 branch_fmt field

This is encoded so that it will take the same value as the MSB of the **branch_map** field, so that extra bits are only required when reporting predicted branch counts, and reporting a branch map is unaffected. Although there are 15 unused bits when reporting a branch count without address, branch counts will by their nature be reported much less frequently, so this is not a significant cost. Furthermore, even for the most pathological case (32 correctly predicted branches followed by a misprediction), the total number of bits used is still fewer than if using just the branch map format.

Chapter 7

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 future, once the current scope has been completed.

7.1 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.

7.2 Fast profiling

In this mode the encoder will provide a non-intrusive alternative to the traditional method of profiling, which 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).

7.3 Inter-instruction cycle counts

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

7.4 Using a jump target cache to further improve efficiency

The encoder could include a small cache of uninferable jump targets, managed using a least-recently-used (LRU) algorithm. When an uninferable PC discontinuity occurs, if the target address is present in the cache, report the index number of the cache entry (typically just a few bits) rather than the target address itself. The decoder would need to model the cache in order to know the target address associated with each cache entry.

DISCUSSION POINT:

This mode needs more analysis before we commit. The primary concern is whether it will result in an overall gain in efficiency. Packet formats 0 and 2 are used to report differential addresses with and without branch history information respectively. Format 0 is currently reserved. This could be redefined as being equivalent to either format 0, or format 2, but reporting a jump target cache index instead of a differential address. However, ideally we would want the ability to output the equivalent of both format 1 and format 2 with a jump target cache index. In order to do that we will need to add an extra bit somewhere. These are the options:

- Define format 0 as jump target cache index without branch information, and add a bit to format 1 to indicate whether it contains an address or a jump target cache index;
- Define format 0 as jump target cache index with branch information, and add a bit to format 2 to indicate whether it contains an address or a jump target cache index;
- Leave format 0 reserved, and add a bit to both formats 0 and 2 to indicate whether they contain an address or a jump target cache index.

For this mode to be useful, the overall efficiency gain achieved as a result of jump target cache indexes requiring fewer bits to encode than differential addresses needs to be enough to overcome the efficiency loss of adding a bit to every packet.

Chapter 8

Decoder

8.1 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 uninferrable
                                         #     jump (and is therefore inferred)
global bool  start_of_trace = TRUE       # Flag indicating 1st trace message still
                                         #   to be processed
global       call_counter = 0            # Count of number of nested calls being traced
global array return_stack                # Array holding return address stack
```

```
# Process te_inst message.  Call each time a message is received #
function process_te_inst (te_inst)
  local address
  if (te_inst.format == 3)
    inferred_address = FALSE
    address       = (te_inst.address << discovery_response.iaddress_lsb)
    pc            = address
    call_counter  = 0
    if (te_inst.subformat == 0 and !start_of_trace)
      follow_execution_path(address, te_inst)
      start_of_trace = FALSE

    if (is_branch(get_instr(address))) # 1 unprocessed branch if this instruction is a branch
      branches      = 1
      branch_map[0] = te_inst.branch
    else
      branches      = 0
      branch_map[0] = 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
      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
  while (TRUE)
  if (inferred_address) # iterate again from previously reported address to find
                        #   second occurrence
      next_pc(previous_address)
      if (pc == previous_address)
        inferred_address = FALSE
    else
      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)
        return
      if (pc == address and is_uninferrable_jump(get_instr(last_pc)))
        # Reached reported address following an uninferrable jump - 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 (pc == address and (te_inst.updiscon == te_inst.address[MSB]) and
          (branches == (is_branch(get_instr(pc)) ? 1 : 0)))
            # All branches processed, and reached reported address, but not as an
            #   uninferrable jump target
            # Stop here for now, though flag indicates this may not be
            #  final retired instruction
        inferred_address = TRUE
        return
```

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

  local instr   = get_instr(pc)
  local this_pc = pc

  if (is_inferrable_jump(instr))
    pc += instr.imm
  else if (is_sequential_jump(instr, last_pc)) # lui/auipc followed by
                                               #  jump using same register
    pc = sequential_jump_target(pc, last_pc)
  else if (is_implicit_return(instr))
    pc = pop_return_stack()
  else if (is_uninferrable_discon(instr))
    if (stop_at_last_branch)
      ERROR: unexpected uninferrable discontinuity
    else
      pc = address
  else if (is_taken_branch(instr))
    pc += instr.imm
  else
    pc += instruction_size(instr)

  if (is_call(instr))
    push_return_stack(pc)

  last_pc = this_pc


# 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 inferrable jump #
function is_inferrable_jump (instr)

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

  return FALSE

# Determine if instruction is an uninferrable jump #
function is_uninferrable_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 uninferrable discontinuity #
function is_uninferrable_discon (instr)

  if (is_uninferrable_jump(instr) or
      (instr.opcode == URET)      or
      (instr.opcode == SRET)      or
      (instr.opcode == MRET)      or
      (instr.opcode == DRET))
    return TRUE
    # Note: The exception reporting mechanism means it is not
    #   necessary to include
    # ECALL, EBREAK or C.EBREAK in this test

  return FALSE

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

  if (not is_uninferrable_jump(instr))
    return FALSE

  local prev_instr = get_instr(prev_addr)

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

  return FALSE

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

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

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

  return target
```

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

  if ((instr.opcode == JALR and instr.rd == 1) 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 (te_support.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))
    return (call_counter > 0)

  return FALSE
```

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

  local call_counter_max = 2**(discovery_response.call_counter_width + 2)
  local instr            = get_instr(address)
  local link             = address

  if (call_counter == call_counter_max)
    # Delete oldest entry from stack to make room for new entry added below
    call_counter--
    for (i = 0; i < call_counter; i++)
      return_stack[i] = return_stack[i+1]

  link += instruction_size(instr)

  return_stack[call_counter] = link
  call_counter++

  return

# Pop address from return stack #
function pop_return_stack ()

  local link = return_stack[call_counter]

  call_counter-- # function not called if call_counter is 0, so no need
                 #  to check for underflow
  return link
```