RISC-V Processor Trace
Version 0.026-DRAFT
fda72fe21e3947aeffce62ffac54144c40044448

Gajinder Panesar <gajinder.panesar@ultrasoc.com>, UltraSoC Technologies Ltd.

March 13, 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 **font** 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 delta.

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 address of taken indirect jump.

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 needs 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.

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 ***mcause/scause*** CSR) and trap value (from the ***mtval/stval*** 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:

- Significant improvements in trace efficiency can be achieved by not reporting the return addresses of functions where the associated function call has been traced. This 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. This requires the encoder to be aware of the address of nontaken branches.

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 source code. 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

Signal	**Function**
iretire[*iretire_width_p*-1:0]	Number of halfwords represented by instructions retired in this block.
ntkn[*ntkn_width_p*-1:0]	Number of nontaken branches in this block.

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: Uninferable jump;
	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 (***mcause/scause***), 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 **mtval/stval** 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 **iretires**=0
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.
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.

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.

Table 3.1 lists the signals in the interface. 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.

For cores that can retire a maximum of N taken branches per clock cycle, the signal group (**iretire**, **itype**, **ntkn**, **ilastsize**, **iaddr**) 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 **priv**, **context**, **context_type**, **cause** and **tval** are not replicated. Futhermore, **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. **iretires** 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 upto the 1st taken branch must be reported in group 0 and instructions upto the 2nd taken branch must be reported in group 1 and son 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 3.2 specifies the actions for the various **context_type** values.

3.2.1 Example Interface Configurations

The ingress interface may be used in a number of different configurations. For example:

Table 3.2: 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.

- For a core that retires no more than one instruction per cycle, the simplest solution is to provide details of every retired instruction explicitly:
  - *iretire_width_p* = 2;
  - Set **iretire** to 1 for 16-bit instructions and 2 for 32-bit instructions;
  - *ntkn_width_p* = 1;
  - Set **ntkn** to 1 if the instruction is a nontaken branch, zero otherwise;
- 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 Table 3.1, with **iretire** indicating the total number of instructions retired. However, an alternative approach would be to provide explicit details of every instruction retired by using N sets of the signal group (**iretire**, **itype**, **ntkn**, **ilastsize**, **iaddr**) with the groups detailing one instruction each (replicating the single retirement example N times).

3.2.2 Example Retirement Sequences

3.2.3 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

Table 3.3: 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

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 3.4: 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

Table 3.5: User Sideband Encoder Egress signals

Signal	**Function**
stall	Stall request to core

3.2.4 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 feature, e.g. filter or comparators. Table 3.6 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 3.6: Parameters to the encoder

Parameter name	**Range**	**Description**
context_width_p		Width of context bus
ecause_width_p		Width of exception cause bus
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*
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
privilege_width_p		Width of privilege 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
user_width_p		Width of user-defined filter qualifier input bus
taken_branches_p		Number of times **iretire**, **itype**, **ntkn** is replicated
itype_width_p		Width of the **itype** bus
iretire_width_p		Width of the **iretire** bus
ilastsize_width_p		Width of the **ilastsize** bus
ntkn_width_p		Width of the **ntkn** bus
context_type_width_p	2	Width of the **context_type** bus

3.2.5 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 may 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.
- *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 $2^{\underline{filter\_ecause}}$.
- *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.
- *nocontext*. Context ignored when 1.
- *notval*. Trap value ignored when 1.
- *privilege_width*. Width of the privilege input bus is *privilege_width* + 1.
- *rv32*. ISA is RV32 when 1.
- *itype_width*. Width of the **itype** bus is *itype_width* + 1.
- *iretire_width*. Width of the **iretire** bus is *iretire_width* + 1.
- *ntkn_width*. Width of the **ntkn** bus is *ntkn_width* + 1.
- *ilastsize_width*. Width of the **ilastsize** bus is *ilastsize_width* + 1.
- *taken_branches*. Number of times **iretire**, **itype**, **ntkn** is replicated is *taken_branches* + 1.
- *context_type_width*. Width of the **context_type** bus is *context_type_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
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 are 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 disconinuity. 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 6, 7, 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 3.2);
- *ppch.* Privilege has changed, or context has changed and needs to be reported precisely (see Table 3.2);
- *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 notify (see Table 3.2);
- *exc_only.* Exception signaled without simultaneous retirement;
- *cci.* context change that can be reported imprecisely (see Table 3.2).

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 two further packet types, not shown on the diagram for clarity (see Chapter 6):

- a *te_support* packet is sent when the encoder is enabled or disabled, or its configuration is changed, and also after the last qualified instruction has been traced. This informs the decoder of the operating mode of the encoder, and also that tracing has stopped;
- a *packet_lost* packet is sent 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 *packet_lost* packet. Following this, tracing will resume with a sync packet.

Note: if the **halted** or **reset** sideband signals are asserted (see Table 3.4) 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

In all cases but one, the packet format is determined only by a 'yes' outcome from the associated decision. The choice between format 0, 1 or 2 for the case in the middle of the diagram needs further explanation.

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 0 and 1 include the full or differential address respectively, along with a branch map. If the branch map is not empty, The choice of which format to use is determined as follows:

1. Calculate the differential address (current address minus previously reported address);
2. Count the number of identical most-significant bits for both the full and differential addresses;
3. Choose the address which has the highest number of identical bits.

For example, suppose the current address is 0xffff1234, and the address of the instruction most recently output in a packet is 0xffff1200. The differential address is 0x00000034. This has 26 identical MSBs. The full address has ony 16 identical MSBs, so the differential address is chosen.

If the branch map is empty and does not need to be reported, packet format 2 is used. This always contains a differential address. The rationale here is that if there are no branches to report then the instruction address is unlikely to differ by much from the previously reported address, and so the differential form is much more likely to be optimal. Offering a choice between full and differential would require an extra bit in the packet, which would reduce the efficiency.

The packet formats are organized so that the address is always 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.

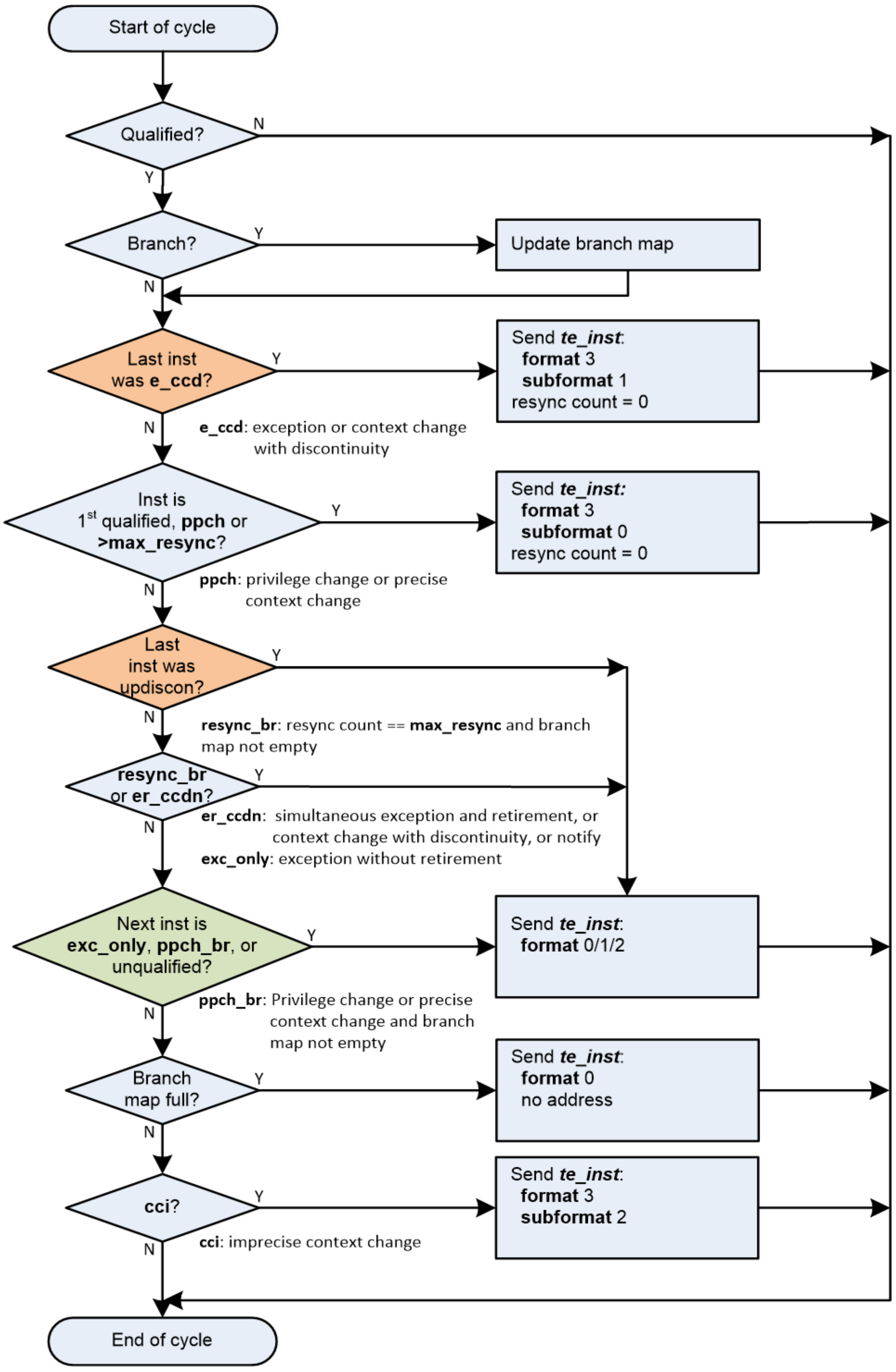
Start of cycle
Qualified?
N
Y
Branch?
Y
Update branch map
N
Last inst was e_ccd?
Y
Send te_inst: format 3 subformat 1 resync count = 0
N
e_ccd: exception or context change with discontinuity
Inst is 1st qualified, ppch or >max_resync?
Y
Send te_inst: format 3 subformat 0 resync count = 0
N
ppch: privilege change or precise context change
Last inst was updiscon?
Y
N
resync_br: resync count == max_resync and branch map not empty
resync_br or er_ccdn?
Y
N
er_ccdn: simultaneous exception and retirement, or context change with discontinuity, or notify
exc_only: exception without retirement
Next inst is exc_only, ppch_br, or unqualified?
Y
Send te_inst: format 0/1/2
N
ppch_br: Privilege change or precise context change and branch map not empty
Branch map full?
Y
Send te_inst: format 0 no address
N
cci?
Y
Send te_inst: format 3 subformat 2
N
cci: imprecise context change
End of cycle

Figure 5.1: Delta Mode 1 instruction trace algorithm

Chapter 6

Trace Encoder Output Packets

Figure 6.1 gives an example of a possible basic structure for packets emerging from the encoder, in network order (1st transmitted byte is on the left). This shows how the payload maybe encapsulated. The header includes the packet length, and the index identifies the source of the packet. Different instantiations or implementations may have different encapsulating structures. This will typically be dependent upon such things as the trace routing infrastructure within the SoC. In order to support this, the encoder must provide the following information to the encapsulator:

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

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 transmited first, and multi-bit fields are transmitted LSB first.

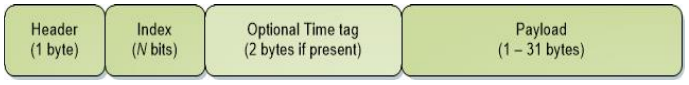

Figure 6.1: Example Encapsulated Packet Format

This packet payload format is used to output encoded instruction trace. Four 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, but ideally will be applied to the whole packet. 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 principal 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.

Table 6.1: Packet Payload Format 0 and 1

Field name	**Bits**	**Description**
format	2	00 (full-delta): includes branch map and full address 01 (diff-delta): includes branch map and differential address
branches	5	Number of valid bits in branch-map. The length of branch-map is determined as follows: 0: 31 bits (**address** is not valid) 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 unpredictable discontinuity.
branch-map	Number of bits 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	Number of bits is *iaddress_width_p* - *iaddress_lsb_p*	Differential or full instruction address, according to **format**. When branches is 0, the address is invalid, and is formed by sign extending branch-map.

Table 6.2: Packet Payload Format 2

Field name	**Bits**	**Description**
format	2	10 (addr-only): address and no branch map
address	Total number of bits for address is *iaddress_width_p* - *iaddress_lsb_p*.	Address is always differential unless the encoder has been configured to only use full-address

Table 6.3: Packet Payload Format 3

Field name	**Bits**	**Description**
format	2	11 (sync): synchronisation
subformat	2	Sync sub-format omits fields when not required: 00 (start): ecause, interrupt and tval omitted 01 (exception): All fields present 10 (context): **address, branch, ecause, interrupt** and **tval** omitted 11 : reserved
context	Total number of bits for context is *context_width_p* unless *nocontext_p* is 1, in which case it is 0	The instruction context
privilege	Number of bits is *privilege_width_p*	The current privilege level
branch	1	If the address points to a branch instruction, set to 1 if the branch was not taken. Has no meaning if this instruction is not a branch.
address	number of bits is *iaddress_width_p* - *iaddress_lsb_p*, unless subformat is 10,	Full instruction address. Address alignment is determined by *iaddress_lsb_p* Address must be left shifted in order to recreate original byte address
ecause	Number of bits is *ecause_width_p* if subformat is 01, or 0 otherwise (no exception).	Exception cause
interrupt	Number of bits is 1 if subformat is 01, or 0 otherwise (no exception).	Interrupt
tval	Number of bits is *iaddress_width_p* if subformat is 01 and *notval_p* is 0, or 0 otherwise (no exception).	Trap value

Table 6.4: te_support payload

Field name	Bits	Description
support_type	4	Set to 0 to indicate instruction trace status
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: Reserved 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 - Always output full addresses (SW debug option) - Exclude address from format 3, sub-format 1 *te_inst* packets if trap vector can be determined from *ecause field* - Don't report function return addresses

Table 6.5: packet_lost payload

Field name	Bits	Description
high_loss	1	Qualitative indication of loss severity. Heuristic is implementation specific.
lost_stream	N	Identifies the source of lost packets Number of bits and encoding implementation specific. Provides for encoders to independently indicate loss of different types of packets (e.g. instruction trace, data trace, etc.)

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 outputs 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 Don't report the addresses of function returns

In well behaved programs, the return from a function adjusts the PC back to the instruction immediately following the original function call. As long as the encoder was enabled when the function call occurred (i.e. the call was traced), the decoding software will be able to infer the return address of the function. As function returns are usually performed using indirect jumps (e.g. *jalr*) they would normally be treated as uninferable. This run-time optional mode would treat them as inferable, and would typically result in a significant increase in trace efficiency.

7.4 Don't report the addresses of exceptions

The exception handler base address is specified by ***stvec/mtvec***, and optionally the lower address bits can be specified by (***mcause/scause***. As the latter is reported by the trace encoder when an exception occurs, the target of the exception handler does not need to be reported explicitly, and an option to eliminate this would improve efficiency.

7.5 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.

7.6 Using branch prediction to further improve efficiency

Rather than outputting branch maps detailing the taken/not taken state of every branch, the encoder could include a branch predictor. This would allow the encoder to report "N branches executed that matched the prediction", which would be a lot more efficient in some circumstances, for example tight loops. The decoder would need to model the same branch prediction algorithm in software.

7.7 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.