RISC-V Processor Trace
Version 0.026-DRAFT
0a920060d7cb7b586195e21e2bf223d0b9aed054

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

January 31, 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 Unpredictable PC Discontinuity

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 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 unpredictable 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 Instruction Interface

This section describes the interface between a RISC-V core and the trace encoder. The trace interface conveys information about instruction-retirement and exception events.

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 at 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 halfwords represented by instructions retired in this bundle. Half words rather than instruction count enables the encoder to easily compute the 'from' address of a branch or exception without having access to the opcode.

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**) 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 exception or interrupt per cycle and so **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.

Table 3.2 specifies the instructions for the various **itype** values.

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 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: Reserved 5: Reserved 6: Taken branch 7: Reserved 8: Unpredictable call 9: Predictable call 10: Unpredictable return 11: Predictable return 12: Unpredictable non call/return jump 13: Predictable non call/return jump
cause [*context_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_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: Call/return **itype** values and corresponding instructions

Type	**Value**	**Instructions**
Unpredictable call	8	***jalr*** x1/x5, rs unless preceded by ***auipc*** rs, ***lui*** rs or ***c.lui*** rs ***c.jalr*** rs1 unless preceded by ***auipc*** rs1, ***lui*** rs1 or ***c.lui*** rs1 ***jalr*** x0, rs where rs !=x1/x5, unless preceded by ***auipc*** rs, ***lui*** rs or ***c.lui*** rs (tail call) ***c.jr*** rs1 where rs1 !=x1/x5, unless preceded by ***auipc*** rs1, ***lui*** rs1 or ***c.lui*** rs1 (tail call)
Predictable call	9	***jal*** x1/x5 ***c.jal*** ***jalr*** x1/x5, rs preceded by ***auipc*** rs, ***lui*** rs or ***c.lui*** rs ***c.jalr*** rs1 preceded by ***auipc*** rs1, ***lui*** rs1 or ***c.lui*** rs1 ***jalr*** x0, rs where rs !=x1/5 preceded by ***auipc*** rs, ***lui*** rs or ***c.lui*** rs (tail call) ***c.jr*** rs1 where rs1 !=x1/x5 preceded by ***auipc*** rs1, ***lui*** rs1 or ***c.lui*** rs1 (tail call)
Unpredictable return	10	***jalr*** x0, x1/x5 unless preceded by ***auipc*** x1/x5, ***lui*** x1/x5 or ***c.lui*** x1/x5 ***c.jr*** x1/x5 unless preceded by ***auipc*** x1/x5, ***lui*** x1/x5 or ***c.lui*** x1/x5
Predictable return	11	***jalr*** x0, x1/x5 preceded by ***auipc*** x1/x5, ***lui*** x1/x5 or ***c.lui*** x1/x5 ***c.jr*** x1/x5 preceded by ***auipc*** x1/x5, ***lui*** x1/x5 or ***c.lui*** x1/x5
Unpredictable non call/return jump	12	***jalr***, ***c.jalr***, ***c.jr*** not matching any of the above
Predictable non call/return jump	13	***jal*** not matching any of the above

Table 3.3 specifies the actions for the various **context_type** values.

Table 3.3: 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

3.1.1 Example Signal Blocks

Table 3.4: Example 1 : 9 Instructions retired in one cycle, 3 branches

Retired	**Instruction Trace Bundle**
1000: ***divuw*** 1004: ***add*** 1008: ***or*** 100C: ***c.jalr***	**iretire**=7, **iaddr**=0x1000, **ntkn**=0, **itype**=4
0940: ***addi*** 1004: ***c.beq*** 1008: ***c.bnez***	**iretire**=4, **iaddr**=0x0940, **ntkn**=1, **itype**=6
0988: ***lbu*** 098C: ***csrrw***	**iretire**=4, **iaddr**=0x0988, **ntkn**=0, **itype**=0

3.1.2 Side band signals

In some circumstances there will be some side band signals which may affect the encoder's behaviour, 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 3.5: User Sideband Encoder Ingress signals

Signal	**Function**
user [*user_width_p*-1:0]	Sideband signals
halted	Core is stalled or halted
reset	Core in reset

Table 3.6: User Sideband Encoder Egress signals

Signal	**Function**
stall	Stall request to core

3.1.3 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 3.7 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 3.7: Parameters to the encoder

Parameter name	Range	Description
context_width_p	1-32	Width of context bus
ecause_width_p	1-16	Width of exception cause bus
iaddress_lsb_p	0-3	LSB of instruction address bus
iaddress_width_p	2-128	Width of instruction address bus. This is the same as *XLEN*
nocontext_p	0 or 1	Ignore context if 1
notval_p	0 or 1	Ignore trap value if 1
privilege_width_p	1-4	Width of privilege bus
ecause_choice_p	0-6	Number of bits of exception cause to match using multiple choice
filter_context_p	0,1	Filtering on context supported when 1
filter_ecause_p	0-15	Filtering on exception cause supported when non_zero. Number of nested exceptions supported is $2^{\text{filter-ecause-p}}$
filter_interrupt_p	0,1	Filtering on interrupt supported when 1
filter_privilege_p	0,1	Filtering on privilege supported when 1
filter_tval_p	0,1	Filtering on trap value supported when 1
user_width_p	0-256	Width of user-defined filter qualifier input bus
taken_branches_p	1-8	Number of times **iretire**, **itype**, **ntkn** is replicated
itype_width_p	3-4	Width of the **itype** bus
iretire_width_p	2-8	Width of the **iretire** bus
ntkn_width_p	1-5	Width of the **ntkn** bus
context_type_width_p	2	Width of the **context_type** bus

3.1.4 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 enity 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.
- *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.
- *iretire_width*. Width of the **iretire** bus.
- *ntkn_width*. Width of the **ntkn** bus.
- *ilastsize_width*. Width of the **ilastsize** bus.
- *taken_branches*. Number of times **iretire**, **itype**, **ntkn** is replicated
- *context_type_width*. Width of the **context_type** bus

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 sideban

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 using User specific 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 figure

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 a synchronisation point to be recorded; for a data trigger it could be a watchpoint trace.

Chapter 5

Example Algorithm

An example algorithm for compressed branch trace is given in figure 5.1. In the diagram, *unpredis-con* is short for unpredictable discontinuity; when the program counter changes in a manner that cannot be predicted from the source code alone. *te_inst* is the name of a type of packet emitted by the encoder.

Context is a generic term that refers to the threadID. This could be the HART ID and/or the software context id. This is reported when there is a change.

Figure 5.1 shows instruction by instruction behaviour, as would be seen in a single-retirement system only.

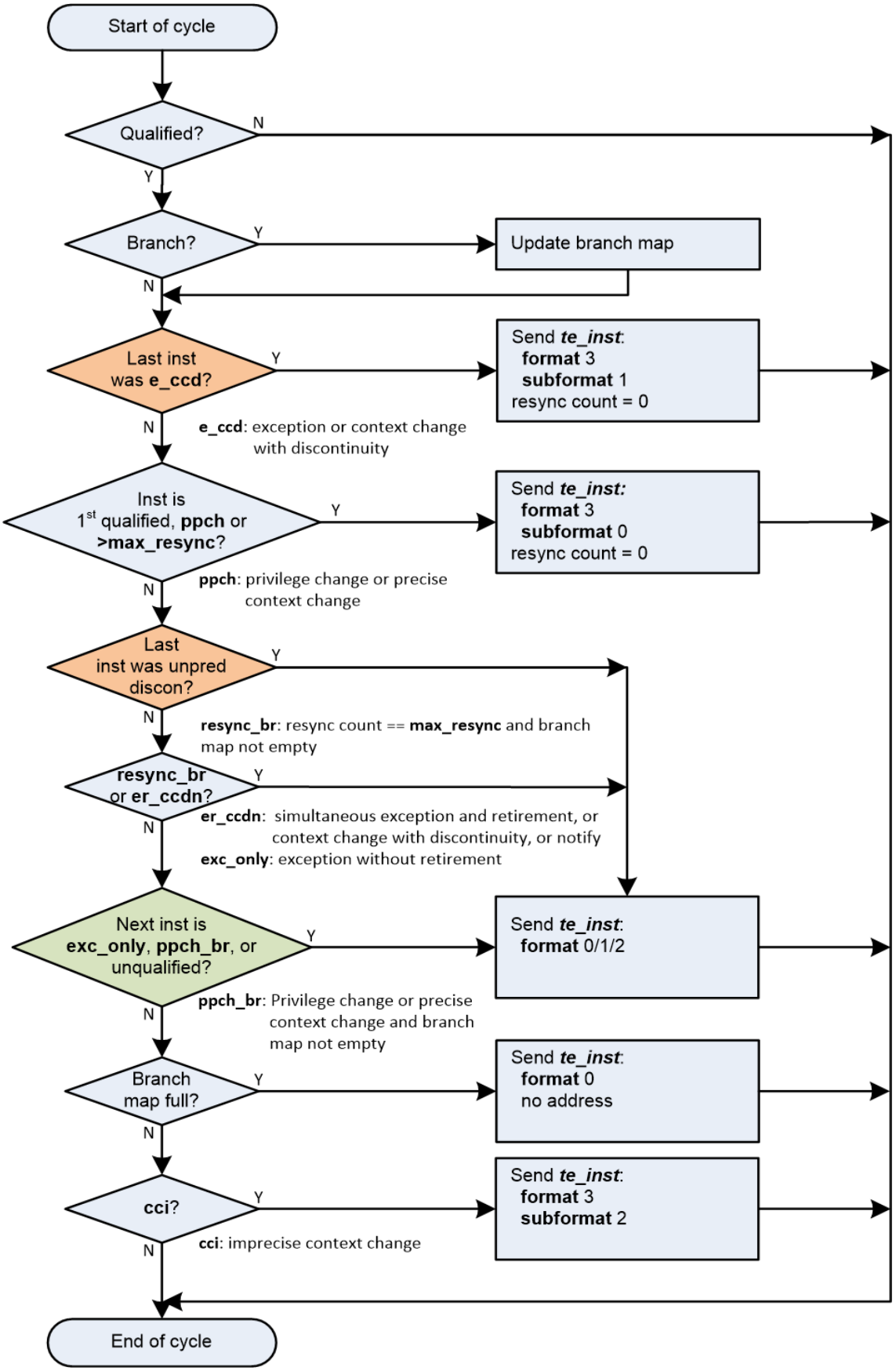

Figure 5.1: Delta Mode 1 instruction trace algorithm

Chapter 6

Trace Encoder Output Packet

Figure 6.1 gives an example basic structure of the packet energing from the encoder. This gives an example of how the payload maybe encapsulated. Different instantiations or implementations may have different encapsulating structures. These would typically be dependent upon such things as the trace routing infrastructure within the SoC.

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

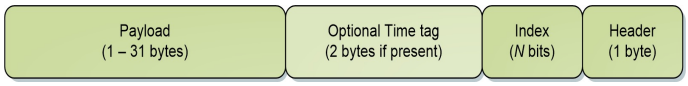

Figure 6.1: Packet Format

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. not including the optional timestamp and not including the the index and header.

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**
msg_type	2	Set to 0x0
control_code	6	Set to 0x34
support_type	4	Set to 0 to indicate *itrace_status*
enable	1	Value of **trace_enable**
encoder_mode	2	Value of **encoder_mode**
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_instr** would have been sent anyway, even if it wasn't the last qualified instruction)

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. 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 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.