RISC-V Processor Trace
Version 0.022-DRAFT

Gajinder Panesar <gajinder.panesar@ultrasoc.com>

October 23, 2018

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In complex systems understanding program behaviour 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 behaviour 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: Core-Encoder signals

Signal	**Function**
ivalid	Instruction has retired or trapped (exception)
iexception	Exception
interrupt	0 if the exception was synchronous; 1 if interrupt
cause [*context_width_p*-1:0]	Exception cause
tval[*XLEN*-1]	Exception data
priv[*privilege_width_p*-1:0]	Privilege mode during execution
context[*context_width_p*-1:0]	Context and/or Hart ID
iaddr [*XLEN*-1:0]	The address of the instruction
instr[*insn_width_p*-1:0]	The instruction

Table 3.1 lists the signals in the interface. The **valid** signal is 1 if and only if an instruction retires or traps (either by generating a synchronous exception or taking an interrupt). The remaining fields in the word are only defined when valid is 1.

The **iaddr** bus holds the address of the instruction that retired or trapped. If address translation is enabled, it is a virtual address, else it is a physical address. Virtual addresses narrower than *XLEN* bits are sign-extended to make computation of differential addresses easier, and physical addresses narrower than *XLEN* bits are zero-extended.

The **instr** bus holds the instruction that retired or trapped. For instructions narrower than the maximum width, e.g., those in the RISC-V C extension, the unused high-order bits are zero-filled. The length of the instruction can be determined by examining the low-order bits of the instruction, as described in The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1 [1]. The width of the **insn** bus, *ILEN*, is 32 bits for current implementations.

The **priv** bus indicates the privilege mode at the time of instruction execution. (On an exception, the next valid ingress word's **priv** bus gives the privilege mode of the activated trap handler.) The width of the **priv** bus, *PRIVLEN*, is typically 3. The actual vaues do not have any special meaning within the encoder other than when filtering. When the level changes the encoder will output the new value in a packet.

The exception bus is 0 if this ingress word corresponds to a retired instruction, or 1 if it corresponds to an exception or interrupt. In the former case, the cause and interrupt buses are undefined and the tval bus is zero. In the latter case, the buses are set as follows:

- **interrupt** is 0 for synchronous exceptions and 1 for interrupts.
- **cause** supplies the exception or interrupt cause, as would be written to the lower *CAUSELEN* bits of the mcause CSR.
- **tval** supplies the associated trap value, e.g., the faulting virtual address for address exceptions, as would be written to the **mtval** CSR. Future optional extensions may define **tval** to provide ancillary information in cases where it currently supplies zero.

For cores that can retire N instructions per clock cycle, this interface is replicated N times. Lower-numbered entries correspond to older instructions. If fewer than N instructions retire, the valid ingress words need not be consecutive, i.e., there may be invalid ingress words between two valid ingress words. If one of the instructions is an exception, no younger instruction will be valid.

3.1.1 Instruction Opcode at Fetch

Some cores may not keep hold of the instruction through the pipeline, in those cases the instruction must be provided at the fetch stage. Pre-decoded instructions will then be queued through a FIFO in the encoder for use at retirement.

To support speculative execution, a *retract count* input will allow a specified number of queued instructions to be deleted from the input side of the queue within the encoder.

The signals in Table 3.2 will be presented to the encoder in such systems. Note **fvalid** only applies to the **insn**, all other signals in Table 3.1 are still valid when **ivalid** is high.

Table 3.2: Core-Encoder optional signals

Signal	**Function**
fvalid	Instruction presented is valid at fetch stage
retract[*RETRACTLEN*-1:0]	Number of newest fetched instructions to discard

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 halting.

Table 3.3: User Sideband Encoder Ingress signals

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

Table 3.4: User Sideband Encoder Egress signals

Signal	**Function**
halt	Halt 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 comparitors.

How the parameters are input to the encoder is implementation specific.

3.1.4 Discovery of parameter values

The parameters used by the encoder must be discoverable at runtime.

Table 3.5: 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*
itrace_fast_p	0 or 1	Support fast profiling only (no delta mode 1)
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
privilege_reset_p	0-15	Default privilege authorization value
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
retires_p	1-4	Maximum number of instructions that can retire simultaneously
insn_width_p	32, 64, 128	Width of instruction 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

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.

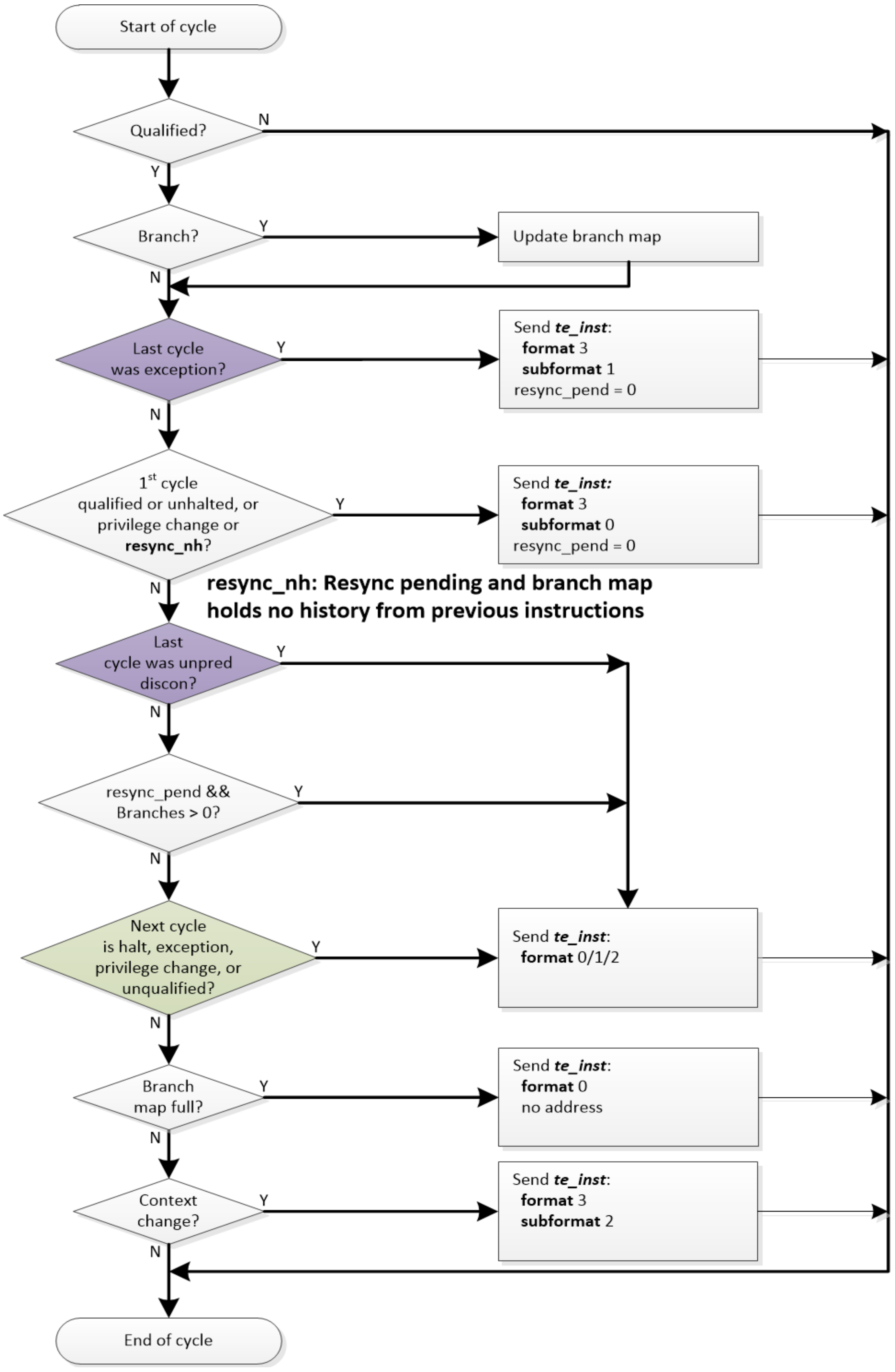
Start of cycle
Qualified?
N
Y
Branch?
Y
Update branch map
N
Last cycle was exception?
Y
Send te_inst: format 3 subformat 1 resync_pend = 0
N
1st cycle qualified or unhalted, or privilege change or resync_nh?
Y
Send te_inst: format 3 subformat 0 resync_pend = 0
N
resync_nh: Resync pending and branch map holds no history from previous instructions
Last cycle was unpred discon?
Y
N
resync_pend && Branches > 0?
Y
N
Next cycle is halt, exception, privilege change, or unqualified?
Y
Send te_inst: format 0/1/2
N
Branch map full?
Y
Send te_inst: format 0 no address
N
Context change?
Y
Send te_inst: format 3 subformat 2
N
End of cycle

Figure 5.1: Delta Mode 1 instruction trace algorithm

Chapter 6

Trace Encoder Output Packet

The packets emerging from the encoder can be up to 35 bytes long including 2 header bytes, an optional 2 byte timestamp and a variable length payload. This can be seen in figure 6.1

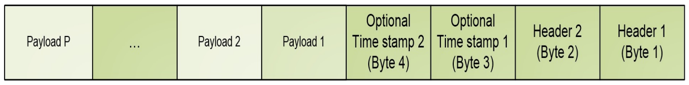

Figure 6.1: Packet Format

This packet 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 2 byte header.

Table 6.1: Packet 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 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 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