RISC-V Processor Trace
Version 0.01-DRAFT

Gajinder Panesar <gajinder.panesar@ultrasoc.com>

October 3, 2018

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Terminology

1.1.1 Context

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 a 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 branch was taken or not and the address of taken indirect jump.

2.1.2 Unpredictable PC Discontinuities

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 Synchronisation

In order to make the trace robust there needs to be regular synchronisation points within the trace. Synchronisation 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 synchronisation is a trade-off between robustness and trace bandwidth.

The instruction trace encoder needs to synchronise fully:

- After a reset.
- When it becomes qualified[1] and instructions have been executed since losing qualification which needed logging by the instruction or data trace encoder.
- If the instruction is the first of an interrupt service routine or exception handler (hardware context change).
- After a prolonged period of time.

[1] Qualification is when certain conditions have been met so that a function can be enabled and executed

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 **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 packet are only defined when valid is 1.

The **iaddr** field 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, and physical addresses narrower than XLEN bits are zero-extended.

The **insn** field 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** field, ILEN, is 32 bits for current implementations.

The **priv** field indicates the privilege mode at the time of instruction execution. (On an exception, the next valid trace packet's **priv** field gives the privilege mode of the activated trap handler.) The width of the **priv** field, PRIVLEN, is 3, and it is encoded as shown in Table 3 2 (TODO: this reference doesn't exist).

The exception field is 0 if this packet corresponds to a retired instruction, or 1 if it corresponds to an exception or interrupt. In the former case, the cause and interrupt fields are undefined and the tval field is zero. In the latter case, the fields 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. For current implementations, CAUSELEN=log2XLEN.

- **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 packets need not be consecutive, i.e., there may be invalid packets between two valid packets. If one of the instructions is an exception, no younger instruction will be valid.

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 [CAUSELEN-1:0]	Exception cause
tval[XLEN-1]	Exception data
priv[PRIVLEN-1:0]	Privilege mode during execution
iaddr [XLEN-1:0]	The address of the instruction
instr[ILEN-1:0]	The instruction

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.

Chapter 4

Example Algorithm

An example algorithm for compressed branch trace is given in figure 4.1

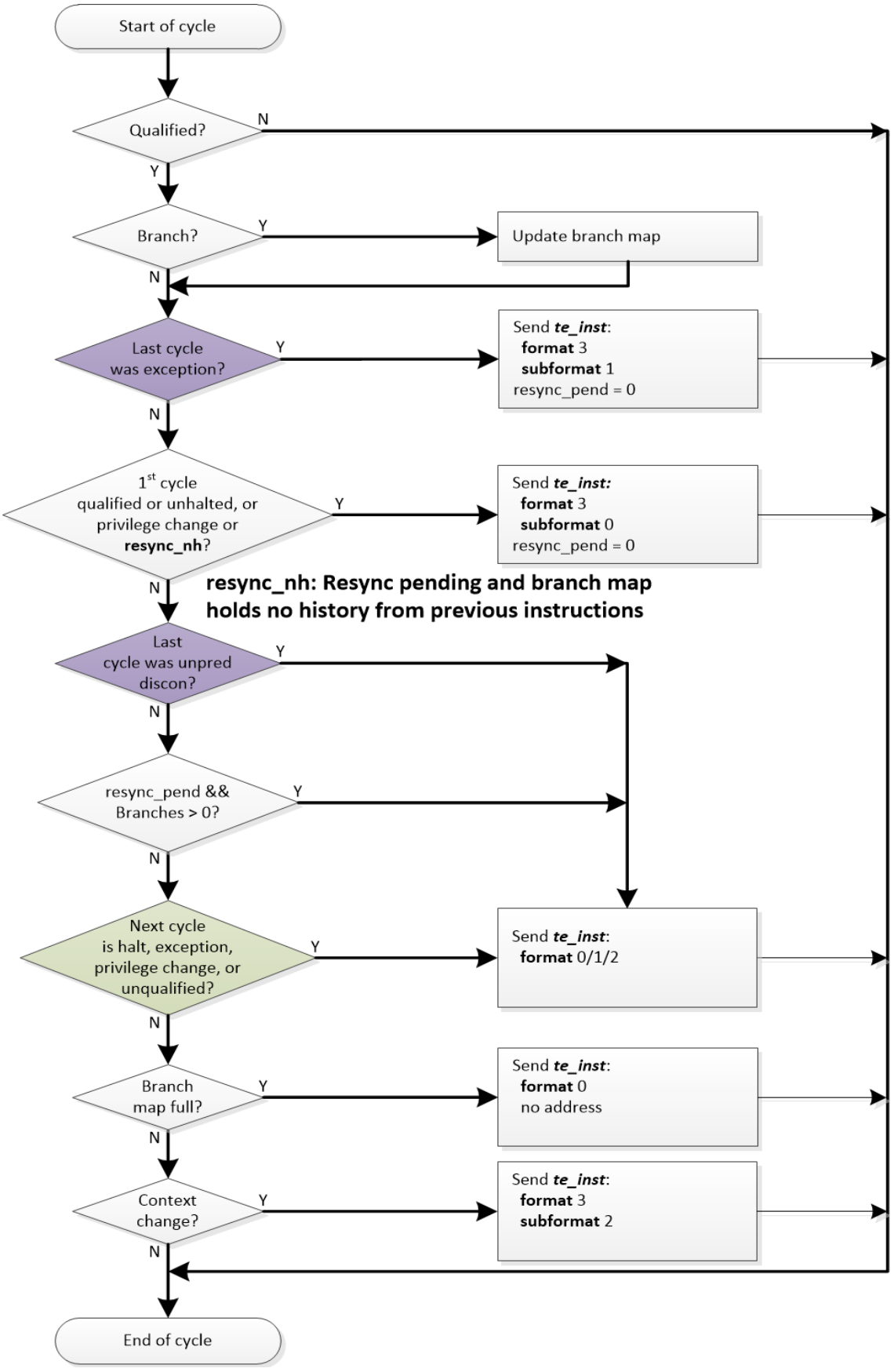

Figure 4.1: Delta Mode 1 instruction trace algorithm

Chapter 5

Trace Encoder Output Packet

Used to output encoded instruction trace. Four different formats are used according to the needs of the encoding algorithm.

Table 5.1: 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: 0-1: 1 bit 2-9: 9 bits 10-17: 17 bits 18-25: 25 bits 26-31: 31 bits
branch-map	See[3]	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	See[4]	Differential or full instruction address, according to format. When branches is 31 (indicating a full address map), the address is only valid if the instruction immediately prior to the final branch causes an unpredicatable discontinuity.

Table 5.2: Packet Format 2

Field name	**Bits**	**Description**
format	2	10 (addr-only): address and no branch map
address	See[6]	Address Address is always differential unless full-address (TODO: needs to be explained) is set

Table 5.3: 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 (address omitted if implicit-except in set-trace is 1 (TODO: needs to be explained)) 10 (context): address, branch, ecause, interrupt and tval omitted 11 : reserved
context	See[13]	The instruction context
privilege	See[14]	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	See[15]	Full instruction address. Address alignment is determined by iaddress-lsb-p (TODO: needs to be explained).
ecause	See[16]	Exception cause
interrupt	See[17]	Interrupt
tval	See[18]	Trap value