Typical Use Cases (for 'self-hosted' trace …)

- Trace single user process.
  - On every process suspend/resume trace is started/stopped.
    - Very similar to storing/restoring debug-triggers.
  - Process may resume at different hart (but it should not matter much …).
    - With local storage you may have very fragmented trace.
  - It is possible that in the same system, several 'users' (not knowing about each other …) may want to trace several user-level processes.
    - In this case trace may be 'fragmented' as different trace may interleave in different buffers …
- Trace a set of (heavily cooperating …) processes/threads.
  - In this (practical case …) we need to track which process needs to be traced and handle it appropriately (at suspend/resume).
- Trace system/kernel/machine.
  - To assess performance (system calls/page faults/task switching)
    - Only certain events should be traced (using Instrumentation Trace).
    - We may want HW support for tracing of performance counters …
  - For 'kernel panic' analysis (often you need trace from more than one core!)
    - Trace should be active all the time and trace 'everything'.

Key Assumptions

- We should NOT re-invent the wheel but try to see how hard (or easy …) we can re-use what we have.
- We should share as much as possible between 'external' and 'self-hosted' trace.
  - We like it or not, but external trace is useful and is being used (for Linux …).
  - Some vendors do NOT use SoC without trace = no driving without insurance 😊 !
- Here what we have:
  - Ingress port (E-Trace definition was not perfect, but it is fixed now).
    - N-Trace is the newest (**discussed with ARC just today**!).
  - We have **RISC-V Trace Control** (defined as set of 32-bit registers).
    - Defined as MMIO addresses, but spec is permitting OTHER methods (if trace component 'base' and 12-bit address can be used).
  - We have defined several trace components
    - Located 'outside' of a hart, but it does not mean access from hart is not possible.

Topics …

- Self-hosted trace was always considered (taken from Control SPEC):

> Trace control register access by an internal debugger:
>
> - Through loads and stores performed by one or more harts in the system. Mapping the control interface into physical memory accessible from a hart allows that hart to manage a trace session independently from an external debugger. A hart may act as an internal debugger or may act in cooperation with an external debugger. Two possible use models are collecting crash information in the field and modifying trace collection parameters during execution. If a system has physical memory protection (PMP), a range can be configured to restrict access to the trace system from hart(s).
>
> 
>
> Additional control path(s) may also be implemented, such as extra JTAG registers or devices, a dedicated DMI debug bus or message-passing network. Such an access (which is NOT based on System Bus) may require custom implementation by trace probe vendors as this specification only mandates probe vendors to provide access via SBA commands.

- CSRs are not mentioned above, but CSR access trace should not be the hard
  - I personally do not see accessing MMIO as a deal-breaking limitation.
    - We do not have CSR to access UART/SPI/PCIe and it somehow works 😊.
- Currently there are two types of trace memory.
  - Shared system RAM (memory mapped) – this **MUST be** physically continuous.
  - Dedicated RAM (read via ‘window’) – this **may NOT be** physically continuous.
- IMO trace sub-system is not much different from DMA …
  - Some ‘pre-programmed’ external agent is sending data to memory (for later consumption).
  - DMA controller is NOT working on virtual addresses.
    - How physical pages are allocated (for massive PCIe read for example?)
- The current trace sub-system does NOT provide a way to generate an interrupt when trace buffer is FULL.
  - It should be ‘almost full’ to allow place for ‘flush’.
  - We may want to automatically disable trace on ‘trace full’ interrupt.
    - Appropriate settings can be added.
  - No matter what, but it may take some time to flush the trace.
    - We may also ‘inject a marker’ and next trace.

Trace Storage

- Proper trace memory management
  - Tracing to physical memory (pool of pages).
    - Currently trace memory is SINGLE continuous page.
    - HW should be extended to automatically 'switch' to next page (scatter-gather ...).
  - Reading in virtual memory (physical pages mapped into virtual space of a hart).
- Shared buffer is easier for **Trace Over Functional Interface** trend (USB-C ...)
- How to obtain physical memory pages from a system with multiple OS-es?
  - IMO this is somehow solved by SBI as memory for guest OS-es is (rather ...) not statically allocated?
- Virtual address (visible to hart) is 'statically' mapped into physical address.
- Trace may write to physical memory pool, but read it in virtual memory pool.
- To assure security, access to collected trace [memory] should be provided by API.
  - That API call must extract the trace (from shared, physical pages) to user-s space.
  - This should be done as a S-mode driver (cooperating with SBI).
- Do we want to 'swap-buffers' if process running on hart is changing?
  - It makes buffer pool management a bit more complex
  - It makes trace writing to N different buffers at the same time.
    - It is possible to have "one hart ➔ one encoder ➔ one sink" configuration.
  - It may be difficult to correlate interleaved execution (only timestamps ...)
- Trace control settings must be 'virtualized'.
  - Like debug triggers ...

Some more ideas (how to 'harness' this …)

I was developing trace probes (and tools …) since 1990-s. These 3 layers are present in every trace tool. It would be nice to 'standardize' them (***RISC-V Trace API***).

Trace Control API

- Start with several simple functions.
  - Config, enable, disable, start, stop, status, simple filters.
  - Using MMIO accesses to trace registers (as defined in **RISC-V Trace Control**).
    - Method of accessing registers can be changed (if needed).
- Advanced configuration functions/parameters to define what should be traced:
  - Single process (with or without potential child processes/threads).
  - Arbitrary set of processes (assuming permissions are correct).
  - Kernel (different levels – user process may 'see' system calls made).
  - Several quest OS-es and/or Hypervisor (very advanced …)
- More complex functions to define advanced filters and triggers.
  - Trace filters cooperate with debug triggers.

Trace Read API

- This should abstract how (and where …) trace is stored/collected.
  - It does not mean it must include copying memory (it may just provide access …)
- It should not provide full access for everyone (permissions, security).
- Reading should allow 'seek' (so decoding may start 'from the middle').
- Trace can be read 'live' (while trace is still being collected)
  - New trace may be added at the end.
  - Old trace may 'disappear' from the beginning (as trace memory may be 'limited')

Trace Decode API

- Set different parameters.
  - Key control setting used to collect trace (some affect trace format …).
  - Code being traced (one or more files or reading from memory directly).
- DecodeInit() ➔ Initialize decoder
- DecodeChunk(buf, size) ➔ Decode trace from a chunk of raw trace.
  - This may be called several times to decode more and more trace.
  - Each buffer can be obtained from 'Trace Read API'.
  - What is an output of decoded trace needs to be defined.
- This should be the only piece which is different between E-Trace and N-Trace.
  - Reference trace decoders exist (for N-Trace it is written in C).