Different trace methods and efficient ways to utilize them

Robert Chyla
Thomas Andersson
Agenda

- What is Trace
- Why do I want trace?
- Different types of trace
- Analyze your trace
- HW and SW tools
RISC-V Trace specifics

• Specifications of standard RISC-V trace are “in the making”
  – Processor Trace TG define trace encoder packets and core ➔ encoder interface
  – More work is needed to make all aspects of trace standard (trace control & trace export)

• Goal is to get on par with what is already existing on more mature architectures

• Done right it enables easy adoption of existing trace viewers, hardware trace probes and trace analysis tools

• Some implementations are already around
  – RISC-V architecture deserves good trace in every device from IoT to servers
  – Even simple, standard trace is better than no trace …
What is trace

• In contrast to traditional debugging trace is more like non intrusively observing your application
• Trace can include full PC flow (no need for printf nor UART …)
• It is non-intrusive to your application
• Go back in time
• Quickly isolate exceptions/hard faults
  – Find bugs that are rare and dependent on order-of-execution
• But trace is not only about finding bugs
• Performance and coverage monitoring
  – Live trace streams can be integrated in your debugger
Seeing every instruction

• With integrated support for trace in your development tools every day code development/debugging can really be enhanced
  – See how you arrived at your current execution point
  – Go back in time
• Power measurements can be correlated with program flow
• Do all of these things in a multicore environment
• But tracing can be like finding a needle in a haystack when looking for a bug
  – Just a few seconds of execution time can produce hundreds of millions of instructions
  – Instrumentation may enhance visibility
  – Advanced navigation and search capabilities is essential
  – If your compiler/debugger tools have it, use Trace triggers to constrain trace data to only what you need
Example of power debugging and profiling

- Power samples is one form of trace too
- Especially if it is synced with your PC

Includes power measurements

- USB_Reset()
- USB_RealizeEp()
- USB_Rese
- US
- USB_LP_IRQHandler()
Why do I want Trace?

Implementing trace IP in your device gives you the possibility to non-intrusively track your product as it is running.
Types of trace

• Serial
  – Enough for PC sampled trace (good for statistical code profiling)
  – Light instrumentation, RTOS monitoring, variable tracing etc.
  – With a good probe it is still possible to reach speeds up to several M bytes/s

• High speed parallel interfaces (4 to 16-bit dual-edge)
  – Capture everything (clock speed can be very high)
  – Traces via “breadcrumbs” left when control flow diversion occurs
  – Guarantees you every single instruction executed (may need optional stall)
  – Trace breadcrumbs are stored on debugger probe

• RAM Buffer
  – Either small dedicated RAM or shared with system memory
  – Even 4KB of trace RAM can provide enough to be really useful

• High speed serial
  – Speeds of 10 Gbits/s or higher
  – Mainly suitable for bigger, complex systems

• Trace over functional interfaces (USB3.0 provides a lot of bandwidth!)
  – Use cases are limited – not an option for small IoT devices
Debugging exceptions / faults

• Exceptions/unhandled faults can be caused by:
  – Pointer problems
  – Illegal instructions
  – Data aborts

• Typically, your stack (and call-frame information) gets trashed

• You have full application history with trace
  – By using trace, you can inspect the program flow up to a specific state, for instance an application crash, and use the trace data to locate the origin of the problem

• Trace data can be useful for locating programming errors that have irregular symptoms and occur sporadically
  – Some “million-dollar” bugs can be found here
HW and SW tools Integration

The best approach is to integrate trace analysis capacity already in your everyday development environment.
Parallel Trace (4-bit) from RISC-V FPGA Implementation

- Standard FPGA board running RISC-V core
- Adapter combining JTAG and 4-bit trace into standard connector
- Trace and JTAG via generic FPGA pins
- Debug and Trace probe (high performance, live streaming capable)
Call Stack and interrupt timeline

Example of a Timeline combining the call stack and interrupts and variable logs

- Startup code
- Complex calls (twice)
- Longer function
- Call level
- Tooltip (for one call)
Code quality

• Performance monitoring
  – Trace can help you see where your application is spending its execution time
    o Excessive interrupts?
    o Not responding fast enough?
  – Trace can provide cycle-accurate counts

• Code coverage
  – Proves code has been exercised at least once
  – Isolate dead code / show test deficiencies
  – Functional safety certifications strongly recommend code coverage

• Trace and a static code analysis tools is a good complement
  – Ensures code compliance with branch specific standards and best programming practices

• Silicon vendors provide a lot of BSP code – you better know how it really runs!
Everything together (synchronized)
Thank you for your attention!

iar.com