



# Different trace methods and efficient ways to utilize them

Robert Chyla  
Thomas Andersson

# Agenda

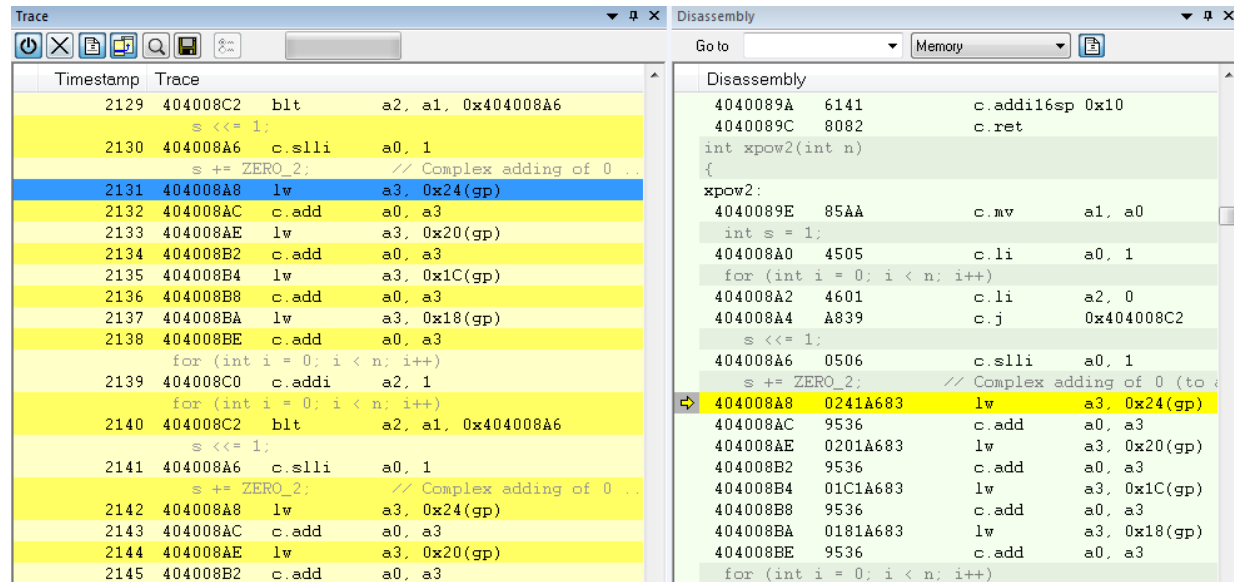
- What is Trace
- Why do I want trace?
- Different types of trace
- Analyze your trace
- HW and SW tools

# 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



The image shows two windows from a debugger. The left window is titled 'Trace' and displays a list of instructions with columns for 'Timestamp', 'Trace', and 'Instruction'. The right window is titled 'Disassembly' and shows the assembly code for the same instructions, with columns for 'Address', 'Disassembly', and 'Comment'.

Timestamp	Trace	Instruction
2129	404008C2	blt a2, a1, 0x404008A6
		s <<= 1;
2130	404008A6	c.slli a0, 1
		s += ZERO_2; // Complex adding of 0 ..
2131	404008A8	lw a3, 0x24(gp)
2132	404008AC	c.add a0, a3
2133	404008AE	lw a3, 0x20(gp)
2134	404008B2	c.add a0, a3
2135	404008B4	lw a3, 0x1C(gp)
2136	404008B8	c.add a0, a3
2137	404008BA	lw a3, 0x18(gp)
2138	404008BE	c.add a0, a3
		for (int i = 0; i < n; i++)
2139	404008C0	c.addi a2, 1
		for (int i = 0; i < n; i++)
2140	404008C2	blt a2, a1, 0x404008A6
		s <<= 1;
2141	404008A6	c.slli a0, 1
		s += ZERO_2; // Complex adding of 0 ..
2142	404008A8	lw a3, 0x24(gp)
2143	404008AC	c.add a0, a3
2144	404008AE	lw a3, 0x20(gp)
2145	404008B2	c.add a0, a3

Address	Disassembly	Comment
4040089A	6141	c.addi16sp 0x10
4040089C	8082	c.ret
		int xpow2(int n)
		{
		xpow2:
4040089E	85AA	c.mv a1, a0
		int s = 1;
404008A0	4505	c.li a0, 1
		for (int i = 0; i < n; i++)
404008A2	4601	c.li a2, 0
404008A4	A839	c.j 0x404008C2
		s <<= 1;
404008A6	0506	c.slli a0, 1
		s += ZERO_2; // Complex adding of 0 (to i
404008A8	0241A683	lw a3, 0x24(gp)
404008AC	9536	c.add a0, a3
404008AE	0201A683	lw a3, 0x20(gp)
404008B2	9536	c.add a0, a3
404008B4	01C1A683	lw a3, 0x1C(gp)
404008B8	9536	c.add a0, a3
404008BA	0181A683	lw a3, 0x18(gp)
404008BE	9536	c.add a0, a3
		for (int i = 0; i < n; i++)

# Seeing every instruction

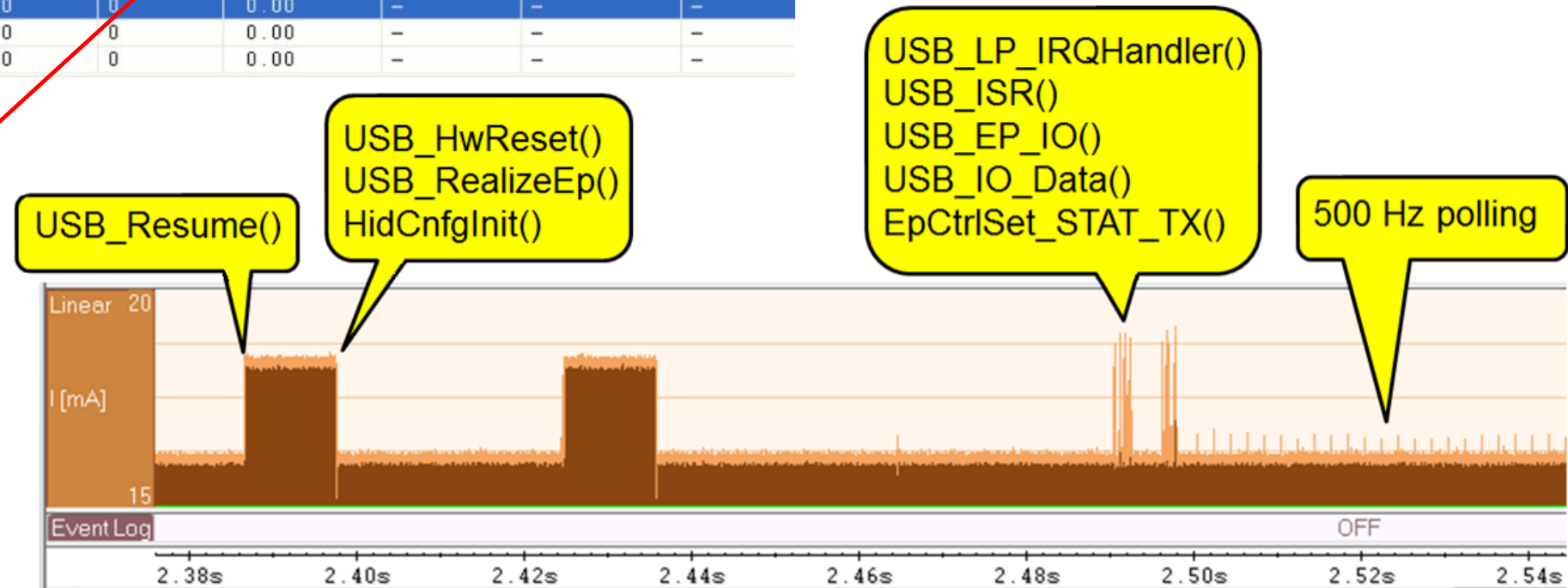
- With integrated support for trace in your development tools every day code development/debugging can really be enhanced
  - See how you arrived at your current execution point
  - Go back in time
- Power measurements can be correlated with program flow
- Do all of these things in a multicore environment
- 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

Function	PC Sam...	PC Sample...	Power S...	Energy (%)	Avg Curre...	Min Current [...]	Max Curr...
putchar	0	0.00	0	0.00	--	--	--
out	0	0.00	0	0.00	--	--	--
memcmp	164	14.35	2	13.01	196.0	193	199
main()	151	13.21	2	13.64	205.5	199	212
exit	0	0.00	0	0.00	--	--	--
abort	0	0.00	0	0.00	--	--	--
<b>_main</b>	<b>0</b>	<b>0.00</b>	<b>0</b>	<b>0.00</b>	<b>--</b>	<b>--</b>	<b>--</b>
_exit	0	0.00	0	0.00	--	--	--
_write	0	0.00	0	0.00	--	--	--

Includes power measurements

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



# 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



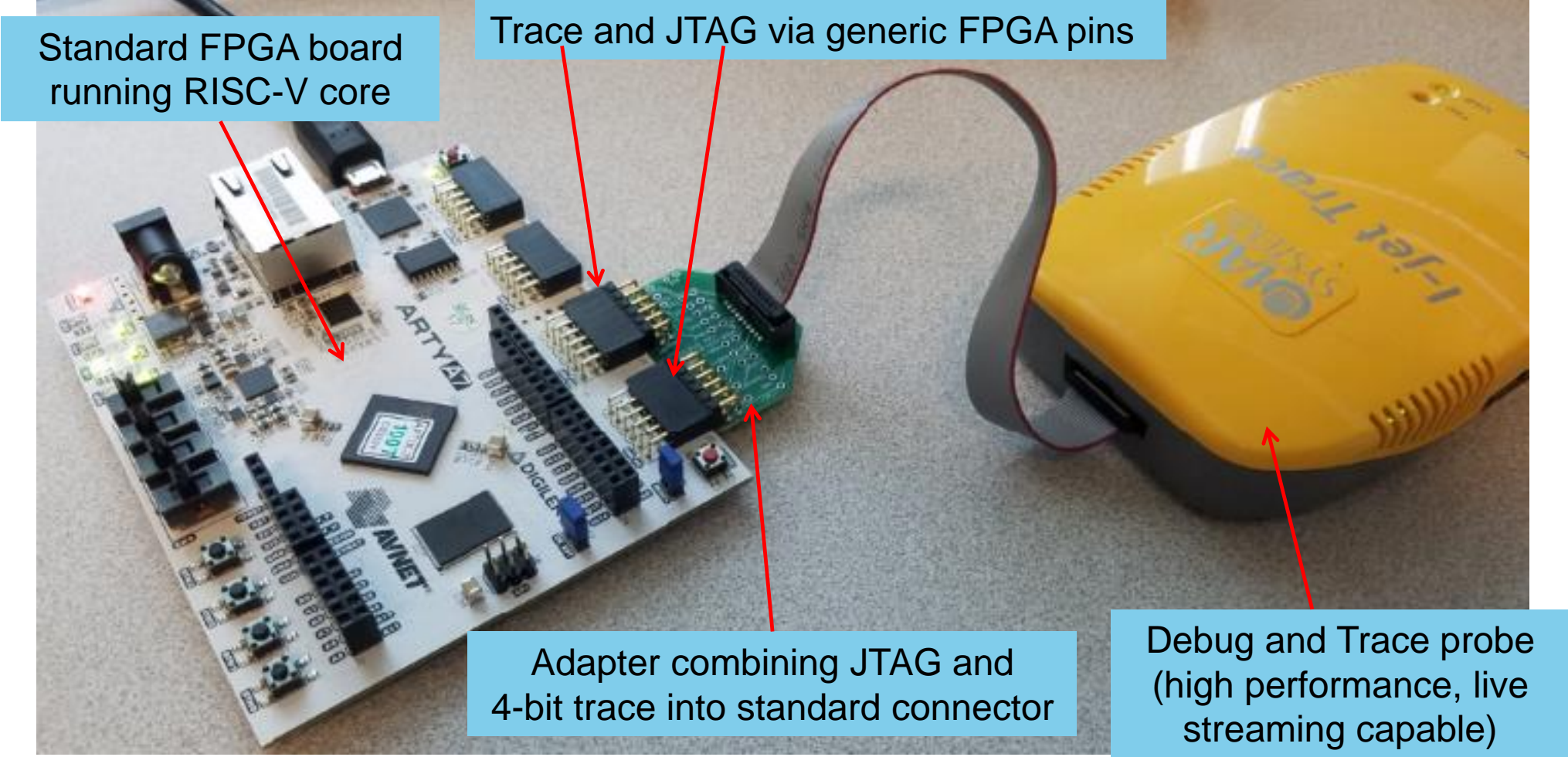
# Debugging exceptions / faults

- 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

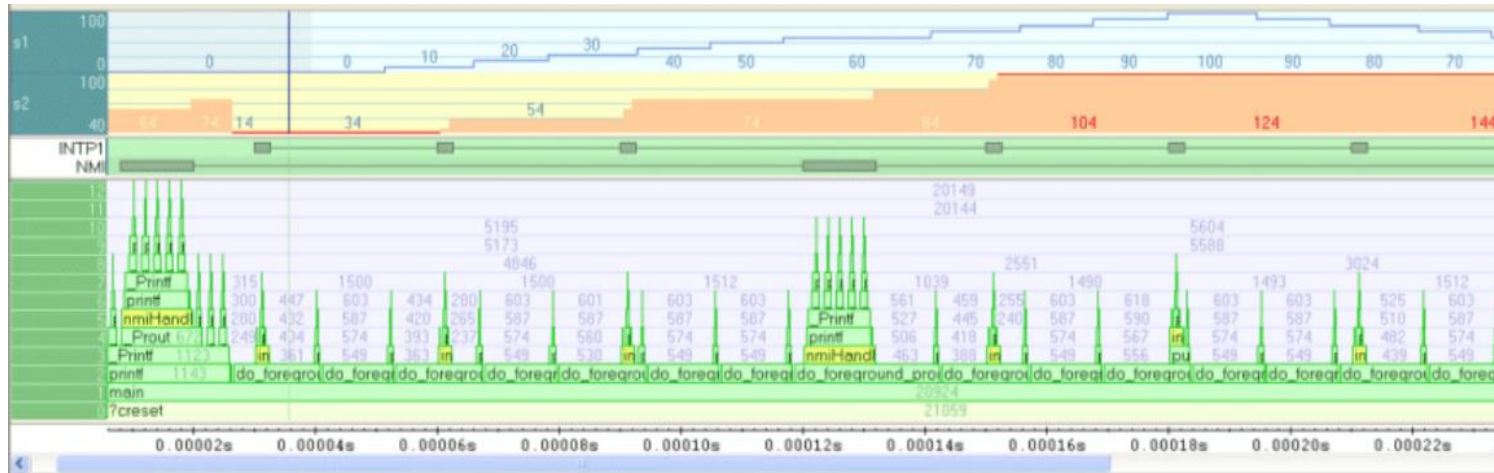
# HW and SW tools Integration

The best approach is to integrate trace analysis capacity already  
in your everyday development environment

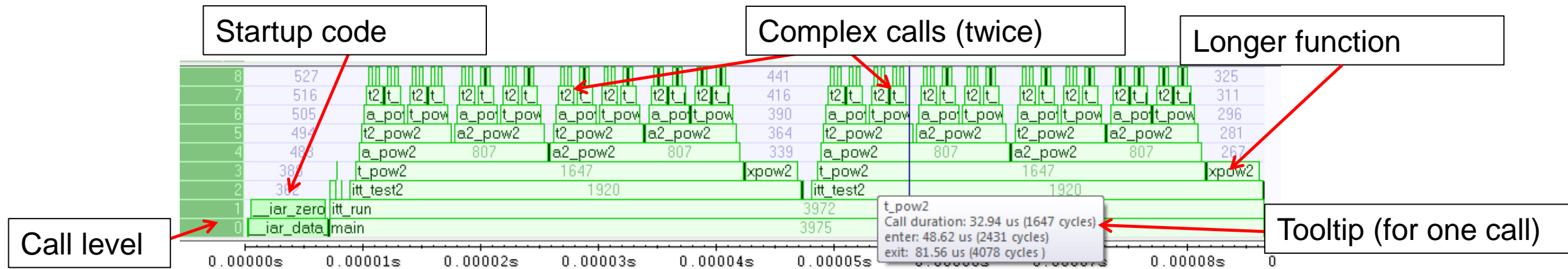
# Parallel Trace (4-bit) from RISC-V FPGA Implementation



# Call Stack and interrupt timeline

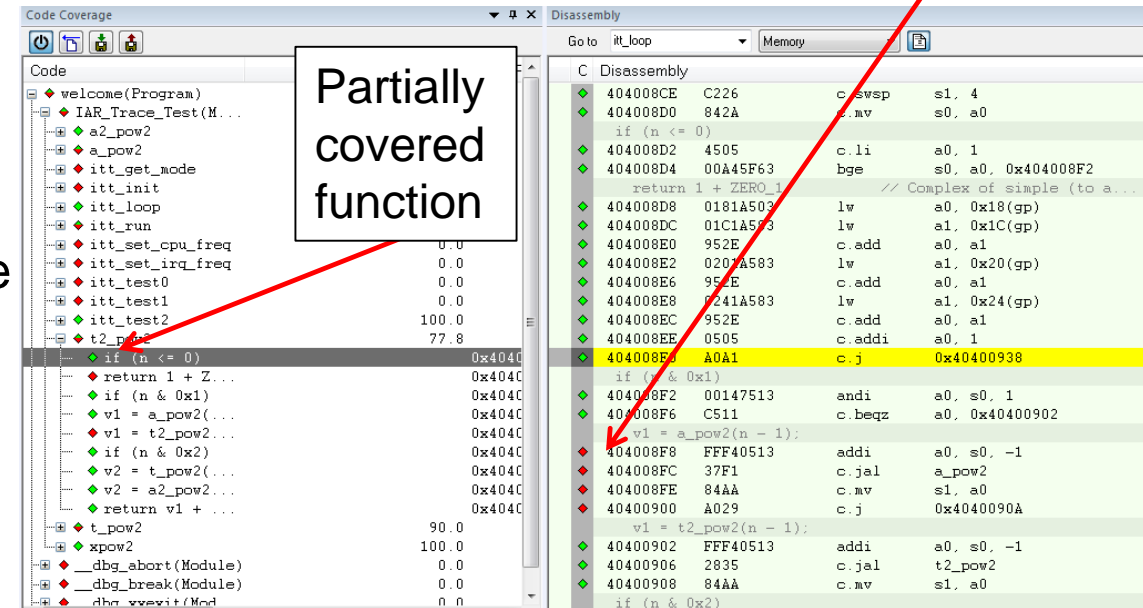


Example of a Timeline combining the call stack and interrupts and variable logs



# Code quality

- Performance monitoring
  - Trace can help you see where your application is spending its execution time
    - Excessive interrupts?
    - 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)

The screenshot displays three synchronized windows in the IAR Embedded Workbench IDE:

- Trace Window:** Shows a list of instructions with their timestamps and addresses. The current instruction is `c.jal a_pow2` at address `40400858` with timestamp `16470`.
- Code Coverage Window:** Shows a tree view of code blocks with their coverage percentages. The `if (n <= 0)` block is highlighted, showing a coverage of 90.0%.
- Disassembly Window:** Shows the assembly code for the selected instruction. The instruction `c.li a1, 1` at address `4040082E` is highlighted.

The Trace window also includes a Timeline view at the bottom, showing the execution flow of the program over time, with various instructions and their durations.

Thank you for your attention!

[iar.com](http://iar.com)