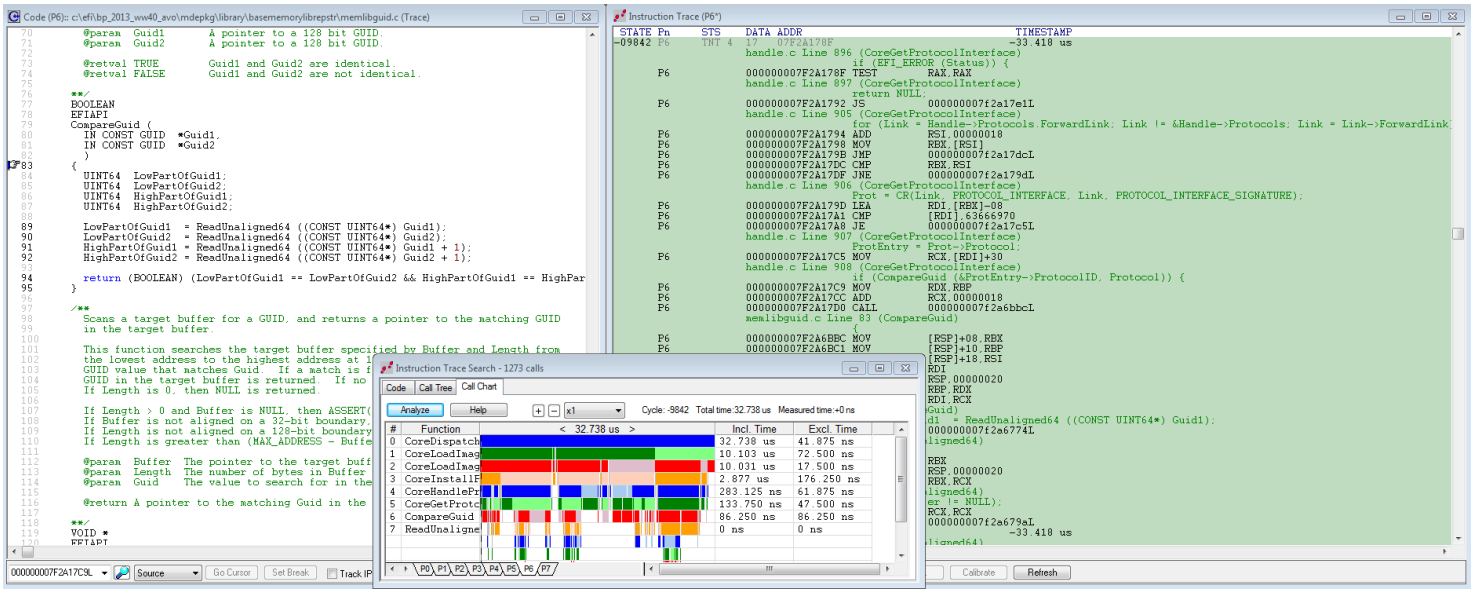
HARDWARE-ASSISTED DEBUG AND TRACE WITHIN THE SILICON



EBOOK

BY LARRY OSBORN

SOURCEPOINT™
More visibility. Software Debug and Trace.

By Larry Osborn

Larry Osborn, SourcePoint™ Product Manager, at ASSET InterTech, has over 30 years of experience in product management, hardware/software product design and development, product delivery to the marketplace and user support. Over the years, Larry has a proven track record for identifying user needs and opportunities in the marketplace, providing innovative solutions and exceeding the expectations of users. At ASSET, Larry is responsible for the profit and loss for a product group. Prior to ASSET, he has held positions with Lockheed Martin, OCD Systems, Windriver, Hewlett-Packard, Ford Aerospace and Intel® Corporation. He holds a Bachelor's Degree in Computer Science from the University of Kansas and various technical and marketing training certifications.

SOURCEPOINT™
More visibility. Software Debug and Trace.

Table of Contents

Figures/Tables

© 2015 ASSET InterTech, Inc.

ASSET and ScanWorks are registered trademarks and the ScanWorks logo, SourcePoint and Arium are trademarks of ASSET InterTech, Inc. All other trade and service marks are the properties of their respective owners.

SOURCEPOINT™
More visibility. Software Debug and Trace.

Executive Summary

Hardware-assisted debug and trace can help identify bugs quicker today than in years past by taking advantage of intellectual property (IP) integrated into chips by silicon vendors. Taking advantage of this IP is key to accelerating the last 20 percent of the software development/debug cycle when engineers are chasing the most difficult bugs. This is the stage in product development when schedules are blown and profitability suffers the most. Plenty of studies have analyzed the growth of code and the impact this has had on product development costs and profitability. Understanding the value of trace and how to setup and apply this technology could be the difference between profit and loss.

There are three areas that need focus during the product design phase. The silicon, software debuggers, and vendor support. Too often the focus is solely upon the silicon without regard to the needs of software developers. They need powerful features so they can debug from within the silicon. In fact, selecting silicon with powerful debug features will likely be the most cost-effective move for getting software delivered on time. A debugger that can access the silicon's trace IP in an effective manner and display the information provided by the chip is valuable. In fact, recently several of the more prominent silicon vendors have integrated new trace IP to make it even more valuable. Focus on the debug capabilities of the silicon and understanding the specific importance of trace will help to deliver a more robust product on time.

How did we get here?

When software developers start writing code, their objective is always error-free or bug-free code. In reality, that never happens. Bugs are introduced due to product schedule pressures, poor requirements, failing to adhere to programming standards or best practices, and the list goes on. Management often struggles with the progress of software development, especially when it goes relatively smoothly for a time and then hits a bug that takes weeks to find and fix. Generally, these show-stopper bugs are discovered in the last 20 percent of the time allocated for product development. Then, during the weekly or daily progress meetings that were triggered by the project delays caused by the bug, the status of the problem inevitably does not change and management becomes even more frustrated. So, it is important that the software team has

SOURCEPOINT™
More visibility. Software Debug and Trace.

sufficient tools and contextual knowledge of when to use which tool to be effective in eliminating bugs as quickly as possible.

When it comes time to attack software bugs, the first tool that most developers pull out of their tool kit is printf(). Sometimes, it is used quite liberally throughout the code. After all, they are just debugging statements. Yet if code instrumentation is mentioned to these same developers, they cringe, as they know it is difficult to back out this instrumentation code and it changes the code execution properties by introducing execution delays and alterations to the code flow. Unfortunately, they either ignore or don't realize that printf() involves instrumenting the code and printf() can introduce timing problems that may cause months of delay and all too often changes the timing of the code. Programs will work with printf() present, but once removed, the problem is manifest and becomes extremely difficult to find. There has to be a better way and there is! The first effective method would be trace and the second would be ARM's System Trace Macrocell (STM).

This eBook focuses on trace, but an excellent book on STM in the context of printf() is available on the ASSET InterTech website's eResources page.

Trace vs. printf()

Not only does the printf() change the code execution timing, it also adds code bloat because libraries must be included to support the printf() structure. At some point, a hardware console driver must also be connected. An open source driver for a particular console might be available, but that would introduce open source code into the program and many companies are not ready to deal with GPL. If a driver is not available, then more development time will be spent extracting the required data from the device manual, writing low-level code and testing the interface. This isn't necessarily a particularly difficult task, but it can be time consuming and it doesn't add any value to the end product if it is only used as a debug tool. In some embedded product environments, application memory space is traded off to support the printf() functionality. Below is a table that highlights just some of the difference between printf() and trace. Examples of how trace can capture programing errors are presented later in this eBook.

SOURCEPOINT™
More visibility. Software Debug and Trace.

Table 1: Comparing printf and trace.

Item	Printf	Trace
Requires special Silicon IP	No	Yes (but available in commercial silicon)
Must instrument user code	Yes	No
Changes code behavior	Yes	No
Longer code execution time	Yes	No*
Timing measurements	No	Yes
Analysis tools available	No	Yes

*negligible as documented in silicon manuals

Utilizing silicon IP, tools and knowledge to gain insight

Considering the increasing complexity of today's hardware and software designs, engineers should fully utilize all of the resources available to them. Within silicon there are typically two types of trace IP. They are:

- Instruction trace
- Data trace

A careful evaluation of the silicon specification will be required to determine which trace IP is supported or if both are available. Instruction trace captures code flow. Data trace provides context for code behavior. That is, data values are captured and these values are what the code relies upon to make flow-control and other decisions.

Hardware-assisted debuggers like ASSET's SourcePoint™ take two fundamental approaches to providing developers insight into how the code behaves in relation to the behavior the developer expected. These two approaches are:

- Static code analysis or run control
- Dynamic code analysis or trace

Static code analysis or run control debugging involves start, stop, stepping, setting break points and running to break points. By its very nature, this is intrusive to code execution, but it has its place in software debug. Examples include viewing data structures, examining pointers, checking the stack, stopping execution to allow experiments by the developer and so on.

SOURCEPOINT™
More visibility. Software Debug and Trace.

Dynamic code analysis or trace captures code flow and program data and is the tool of choice for providing rapid insight into real code execution. Code execution always seems to be where the most insidious and the most difficult to find bugs hide. Execution bugs are manifest in various ways like code exceeding range boundaries on arrays or data structures, uninitialized pointers, wild writes to memory by incorrectly formed addresses and timing issues. Timing issues might involve flows from hardware to software, software to software or software to system. All of these execution bugs are prime candidates to be found by a trace tool.

Understanding trace

There are different ways to use trace or different types of trace operations. Trace types can generally be categorized as one of the following:

- Trace Before – The trace is captured prior to a line of code or event. This type of trace is also referred to as Trace Until. This is the best type of trace for examining a crash, because the last data captured in the trace buffer will be the point where the crash occurred, which is the beginning of the bread crumb trail leading back to the cause of the crash.
- Trace After – The trace starts after a line of code or event.
- Trace About – The trace about a line of code or event.
- Trace Event – This trace type finds a specific event for the debug engineer like writing to a particular memory location.
- Trace Sequence – This trace type helps when the engineer is looking for a specific sequence of events prior to triggering the start of a trace. The complexity of the sequence is limited to the capabilities of the debugger or the engineer's imagination. Trace sequence can be the most powerful of the trace types as it will mimic the events leading up to the bug's manifestation and then capture the most relevant information.

Intel and ARM provide trace capabilities which vary with the silicon. Each vendor has different devices (IP) by various names and various control schemes, but they all boil down to instruction trace, data trace or both trace capabilities. Trace schemes are implemented to be as unobtrusive as possible. Minimal buffers within the silicon may be available for trace, but most trace capabilities in silicon generally support a means for storing trace data in target memory or off-

SOURCEPOINT™
More visibility. Software Debug and Trace.

target memory when deeper trace storage is needed. It is necessary to take these factors into account before a debug methodology is adopted. Silicon trace features in the past introduced delays in code execution, but the silicon vendors' trace IP has improved greatly recently. A deeper discussion of the tradeoffs involving on-die buffer, target memory or hardware-assisted tools versus the code being developed is beyond the scope of this eBook, but could be covered in a future eBook.

ARM has a variety of trace implementations, including Embedded Trace Macrocell (ETM), Program Trace Macrocell, Instrumentation Trace Macrocell, AHB Trace Macrocell and the previously mentioned System Trace Macrocell (STM). This link goes to a page on the ARM website that discusses why trace is recommended for SoCs with ARM cores. The topics include why ETM is used, performance benchmarks and specifications.

Intel has released information about Intel Processor Trace (Intel PT) in its latest "Intel® 64 and IA-32 Architectures Software Developer's Manual." Volume 3, Chapter 36, of this manual describes Intel PT as an extension of the Intel® Architecture. It states that Intel PT captures information about software execution using dedicated hardware facilities that cause only minimal performance perturbation to the software being traced. This trace information is collected in data packets. The first implementation of Intel PT offers control flow tracing, which collects timing and program flow information, such as branch targets and branch taken/not taken indications, and program-induced mode-related information. Another resources is a presentation from the Intel Developers Forum 2014 by Beeman Strong of Intel that discusses the introduction and use of Intel® PT. Here is the link.

Regardless of the silicon being considered, examining the code to determine the degree to which it is data driven should play an important factor in silicon selection. Some applications are mostly flow dependent, where tracking interrupt flow and overall code flow is the major concern. Other applications are dependent on dealing with large amounts of real-time hardware-driven data. Whether porting code or developing new code, knowing the code debug requirements will help select the processor that includes the correct resources needed by the software development team to diagnose the inevitable bugs.

SOURCEPOINT™
More visibility. Software Debug and Trace.

Debugger requirements

Setup

Software debuggers must be able to configure the trace IP available on a SoC from a more abstract view of the device and present that view to the debug engineer. Otherwise the engineer is left digging through a thousand page manual, for example, to find the single register with the single bit that must be enabled to turn on or off tracing or to find multiple registers to configure the trace stream.

Display

The debugger must present the trace captured data in a view correlated to the code view. The raw trace data can include instructions, code addresses, hex data and time stamps. All of this information needs to be presented in a form relevant to the developers' point of view. Also extremely valuable is a view that locks trace data with the code so that while scrolling through trace data, the engineer can view the underlying code that triggered the trace data. An example of this view follows in Figure 1.

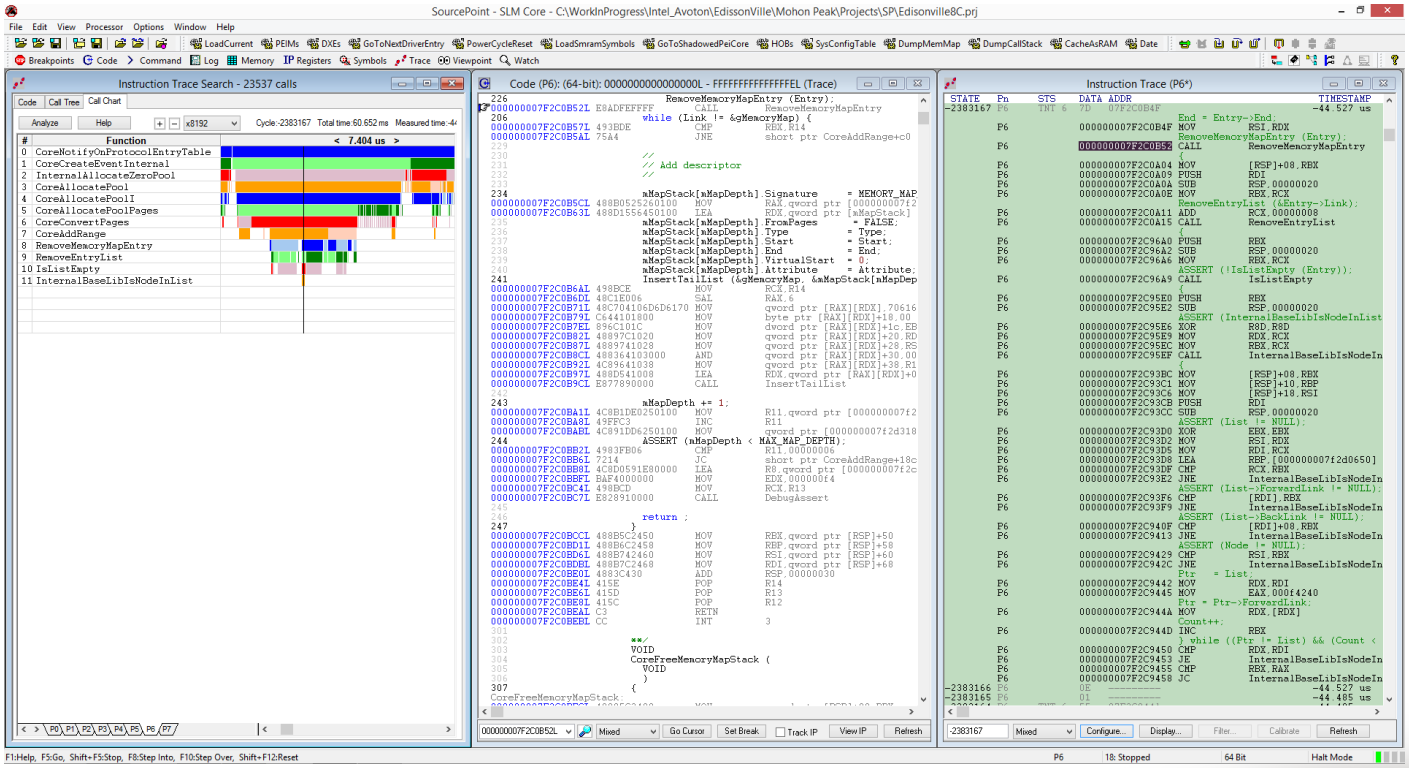


Figure 1: A view with trace data synched to the underlying code.

SOURCEPOINT™
More visibility. Software Debug and Trace.

Support

The debugger vendor needs to be very knowledgeable of the trace support within the silicon, as it may have multiple trace implementation. Consider the following example of how important knowledge of a device's trace resources and the debugger's capabilities can be. During final test of a new product a bug is discovered. The manifestation is clear, but the root cause is not obvious. Starting from what is known, a trace might use one implementation such as instruction trace to narrow the focus from 1,000 lines of code to a module of 100 lines by just following code execution. This reduces the problem from haystack-size to a bale of hay-size. Next, switching to data trace will capture the data that the code is making incorrect decisions about to cause the bug. The needle or bug is found. Support experience on the part of the debugger's supplier in configuring the trace capture can be invaluable. Knowing how to drill down and when to use what resource are skills that the development team will develop over time. With support and training from the supplier of the debugger, these skills and information transfer will happen much quicker enabling the developers to be more productive.

Example: Using the power of trace

The following more comprehensive example is based be on an ARM target running Linux. Once the kernel with application is loaded, including debug symbols, code execution can be examined via trace. Trace can display lots of data, but it is extremely useful if the trace data view can lock with a code view so the engineer can traverse the trace and see the exact code reference. It will also be helpful when the source code is integrated into the trace view. However, the trace data is often cluttered with so much information that the source code gets lost. During the postmortem, at least in the early stages of analysis, less information is more useful. As a result, a link to the code view can be just the ticket for quick, high-level analysis.

The following are several views of traces captured from the environment described above. The first view (Figure 2) is a raw capture of a function executing. The debugger's display parameters are set to view only the state transitions and associated assembly. In this case, the trace data is not of much value for quickly understanding what is being debugged.

SOURCEPOINT™
More visibility. Software Debug and Trace.

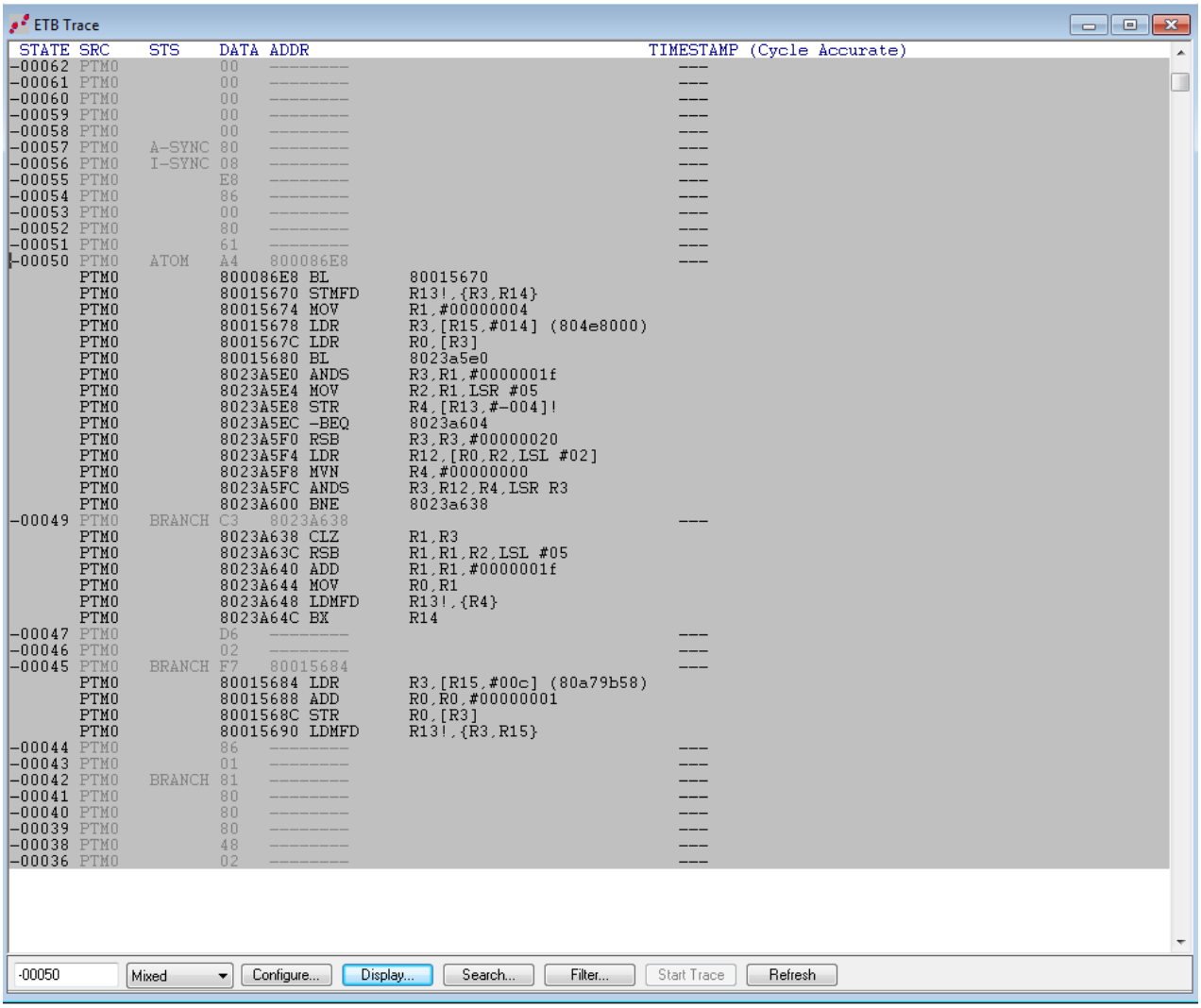


Figure 2: An executing function with only state transitions and associated assembly.

This is what the processor is capable of capturing but the display of the data still isn't informative enough. Changing the debugger's display parameters can add information about the source code to the view (Figure 3). Now, a clearer picture emerges. This is an example of how the vendor of a debugger that has working knowledge of the silicon can provide multiple views of the trace information.

SOURCEPOINT™
More visibility. Software Debug and Trace.

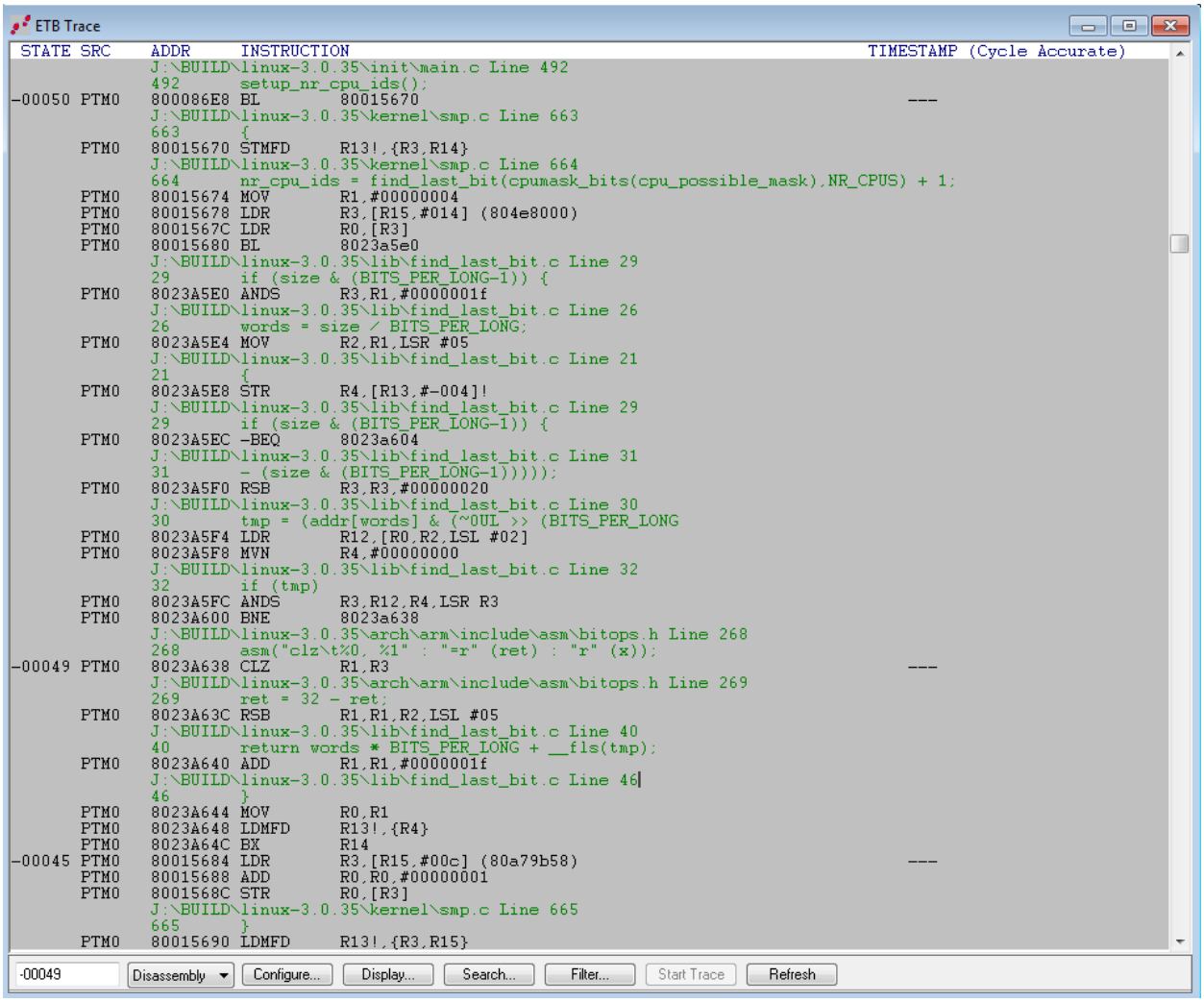


Figure 3: A simultaneous view of an executing function and source code.

Figure 3 shows that the first line captured is “setup_nr_cpu_ids()” followed by a branch into the function. With this view, the code can be analyzed at a detailed level. The trace view should allow the engineer to scroll forward to the end of the trace or backwards to the beginning of the trace.

Better yet, some debuggers are able to add another level of clarity to this picture. This is done by opening a code tracking view to show the context within the code, rather than just a code fragment with all the extra data. With code tracking, the bigger context of the code can be seen. Interlocking the code tracking view with the trace view allows the engineer to move through the code in the trace window and see the context clearly in the code window (Figure 4).

SOURCEPOINT™
More visibility. Software Debug and Trace.

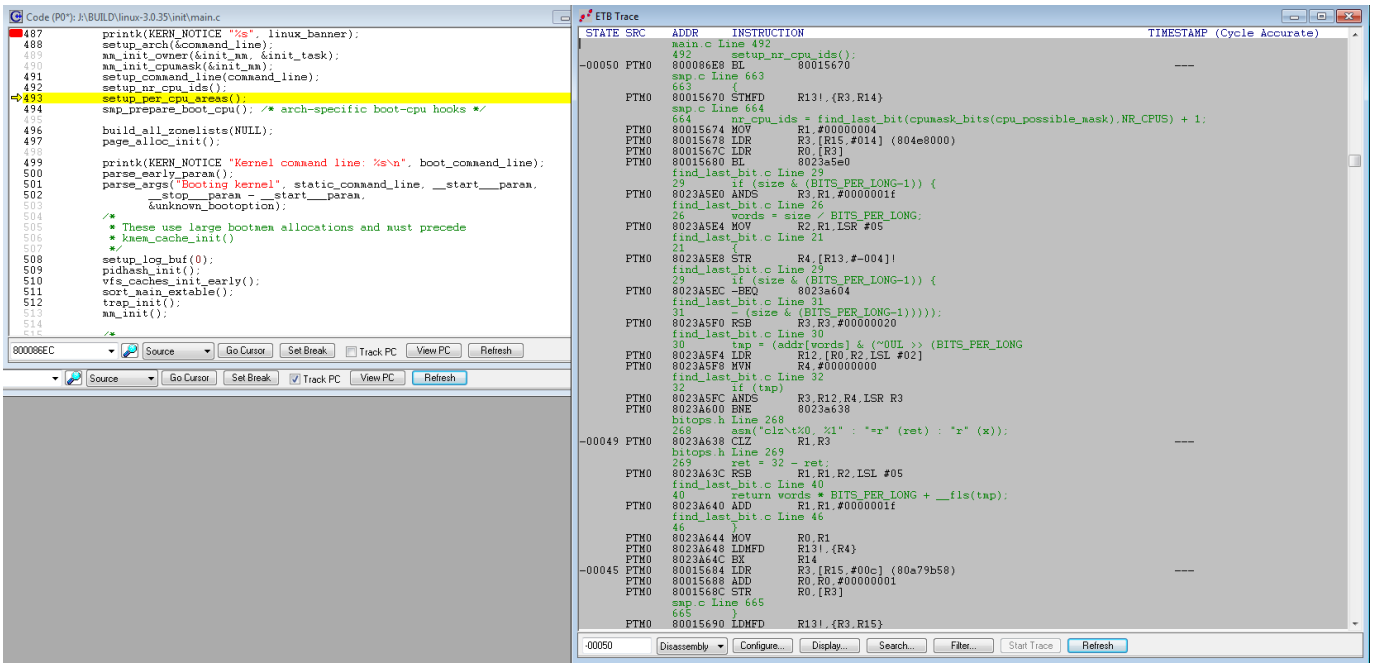


Figure 4: Interlocking code tracking view and trace view displays code context.

Figure 4 shows that a breakpoint has been set at line 487. By stepping through the code and stopping at line 493, the whole execution of the function “setup_nr_cpu_ids” is captured. Code tracking clearly shows the context and also the depths of the code execution. Neither the source view nor the code tracking view will show that the starting point is in main.c and then switches to smp.c for this function. By traversing the trace the engineer sees that the function calls several other functions that are not within the scope of the code view. This simple view provided by the trace opens a clearer view of the true execution behavior of the code.

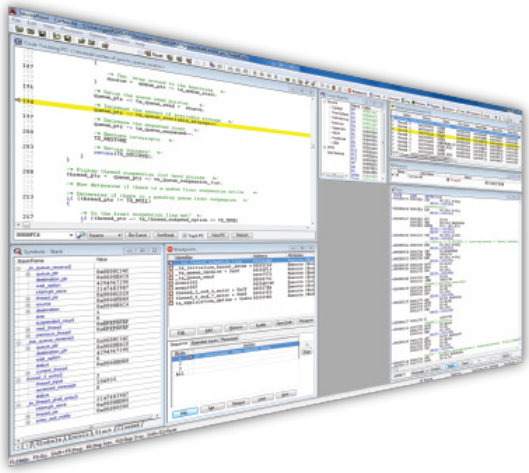
SOURCEPOINT™
More visibility. Software Debug and Trace.

Conclusions

Using trace can help ensure a timely product delivery. Getting to the needle in the haystack is a matter of choosing the right silicon that supports the software debug efforts, selecting a software debugger that provides clear insight into the trace data captured and a support team that can assist the development team when they encounter the most difficult of real-time execution bugs. Using trace is like building muscle through daily exercises. That is, by analyzing a myriad of problems and repetitive use of a trace tool and performing numerous debug experiments, software developers will have the means to make complex measurements to pinpoint problems quickly. In the beginning of this process, engineers will only be using 20 percent of the functionality of the trace tool, but as their skills are developed, they will rapidly get to a point where they will be using 80 percent of the tool's functionality. Accelerating this learning curve in the setup and use of trace can be accomplished by investing in training from the tool provider as well as target-specific training. This small investment will provide big payback in increasing the quality of the software and getting products to market sooner.

Learn More

You can experience the value of trace firsthand through a demo of ASSET's SourcePoint™ for either ARM® or Intel®



SourcePoint ARM® Demo

SourcePoint Intel® Demo