



Why Trace? or Software Development Using an ICE

An Application Note

www.arium.com

Overview

The last two cars in the train of software engineering are development and debug. The purpose of this document is to clearly describe how the trace memory of an in-circuit emulator (ICE) can help software and firmware engineers develop and troubleshoot code. Numerous examples will be given to elucidate specific points.

Although this document focuses on the Pentium[®] processor, the same results can be achieved with P6-class processors.

Background

American Arium has provided microprocessor development and debug tools for over 17 years. Working with Intel, in-circuit emulation tools were developed to support the Pentium processor in 1991. Since then, American Arium has provided ICE tools with Intel first silicon.

In-Circuit Emulator Features

Minimally, any ICE should allow you to:

1. Control (run/stop/single-step) the processor
2. Display code
3. Set a variety of breakpoints including memory fetch, code execution and I/O access
4. View and change register values
5. View and change memory
6. Download code from your host to your target machine

In Pentium processor based systems, these actions are done by the ICE via the Test Access Port (TAP) and Boundary Scan Architecture. Although details of the TAP are beyond the scope of this document, more information can be found in the *Pentium Family Developer's Manual, Volume 1: Chapters 11 and 15*, available from Intel.

TAP Limitations

Although TAP is a useful basic development and debug tool, it won't allow an ICE to:

1. Stop the processor on unique values written to (or read from) memory and I/O ports
2. Trigger breakpoints on specific activities seen on the processor bus
3. Trace bus transactions for later review

Lots of words...what do they mean?

In the Pentium processor world, breakpoints (i.e., instances where you want to temporarily freeze execution of the

processor) are set (via the TAP) by placing address values in Debug Registers DR0 - DR3, then placing a breakpoint-type value in DR7. To quote from the *Pentium Family Developer's Manual, Volume 3: Chapter 17*:

“...An exception is generated when a memory or I/O operation is made to one of these addresses. A breakpoint is specified for a particular form of memory or I/O access, such as an instruction fetch, doubleword memory write operation or a word I/O read operation. The debug registers support both instruction breakpoints and data breakpoints.”

This is how a breakpoint is set up: If you want to stop the processor at port 80h (an I/O port used by BIOS developers to display the number of the current Power On Self-Test, or POST), the ICE would move an 80h to DR0, and 22400h to DR7 (indicating a breakpoint when a one-byte I/O access is made at whatever address got loaded into DR0).

Herein lies the problem. Assume, as an example, a large number of values written to port 80h. What happens if you want to freeze the processor **only** when the value of AAh is written to the port? Since the debug registers don't allow you the granularity of specifying a value written to the I/O port (or memory location), *the processor will stop every time port 80h is accessed*. When it stops, you must interrogate the port (or the AL register). If it's value you're looking for, great; if not, start the processor and wait for the next time it triggers.

That's a lot of starting and stopping. That may also be unacceptable if the code (or bug) is time sensitive.

A better to handle this problem is to watch the processor bus and, if you see an AAh write to port 80h ... stop the processor. Doing that without slowing down the bus is known as ***non-intrusive bus monitoring***, which is beyond the TAP's capabilities.

Another limitation of the TAP is the ability to stop on specific bus activities, such as an Interrupt Acknowledge signal (IACK) or Halt condition. If your software is an interrupt handler, you'll want to stop the processor just as the processor enters your code. Once the processor is frozen, you can view register and memory values.

Finally, if your routine is called by other routines, you need to figure out:

1. Who called the routine, and more importantly,
2. WHY was the routine called?

You can always look at the stack to figure out who made the call to the routine. The stack, however, will rarely tell you which of the decision-making branches were taken to actually call the routine.

By storing the processor bus transactions in trace memory, you can discover who called the routine, why it was called and what else happened to the target while the processor was zooming along. At that point you can convert the information into changes you may need to make to the software.

An analogy of TAP debug versus a “full-featured” ICE with trace debug can be made by imagining a field of snow. Walk to the center of the field and you can see trees, the creek, more snow, etc. That's like being able to inspect memory, registers, code, etc. in either the TAP or “full” ICE. What you can't do with the TAP that you can do with the full ICE is look over your shoulder and see your “footprints in the snow,” i.e., the path you took to get to your current location.

The guarantee: Judicious use of trace will shorten the time you take to develop and debug your project.

Turning On Trace

In an ICE with trace memory, trace is always “turned on.” By the nature of the product, all bus transactions are watched and stored in trace memory (or “traced”).

However, Branch Trace Messages (BTMs) **do** need to be turned on whenever the processor is reset so you can view disassembled code in the trace.

TR12.TR (bit 1 in the TR12 register) controls the Branch Trace Message Special Cycle. If your are using assembly language, set the bit with an **OR TR12, 2h**. A BTM is generated whenever a branch is taken (such as an IRET or RET, JMP, conditional branch, CALL or software interrupt). Or **use the BTMON command** in the ICE. The special cycle that's generated tells the ICE that code was executed in a specific memory range.

The ICE sees a BTM cycle in the trace memory, gets the data from the memory range specified by the cycle,

disassembles, and displays the code. This operation is performed for every BTM cycle stored in the trace memory (which, potentially, could be the only thing stored in trace memory).

So Far, So Good

Ok, if you've read this far, you have a rough understanding of:

- Typical ICE features
- TAP debug and its limitations
- The purpose of trace memory
- Branch Trace Message Cycles

Now we can proceed into how this information is used in code development.

If you're part of the typical hardware-based project, you've (hopefully) got a bunch of specs from the hardware engineers, a couple of databooks relevant to the chip set, and a fairly good idea of where you fit into the hardware initialization process. Now it's time to write a code template, or start modifying an existing piece of code. That initial code will most likely contain stubs (i.e., procedures that don't do much, maybe only a return to the caller) predominantly around the new hardware initialization you need to write.

The first thing you want to do is verify your assumptions. You think certain values in certain addresses will yield certain results. Instead of blasting out a piece of code, why not fill in the values by hand and look at the bus transactions in trace with your handy hardware designer? After everyone is satisfied with the results, then you can code like crazy and have some assurance your code is meeting hardware expectations.

Looking At Code and Trace

The following example displays the first few instructions of typical BIOS (just after the JMP at the reset vector):

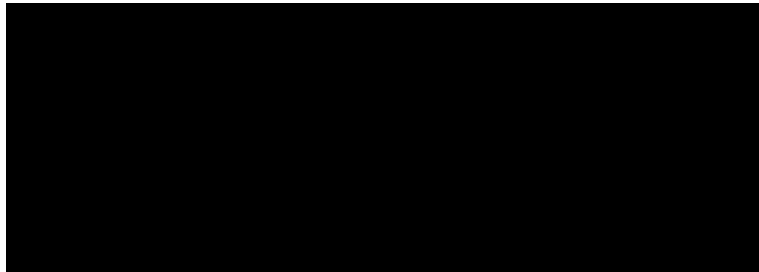


Figure 1: BIOS Code

The following illustration displays the bus transactions with the code disassembly. For this example, a breakpoint was set to trigger at the first write to port 80h and BTMs were turned on.

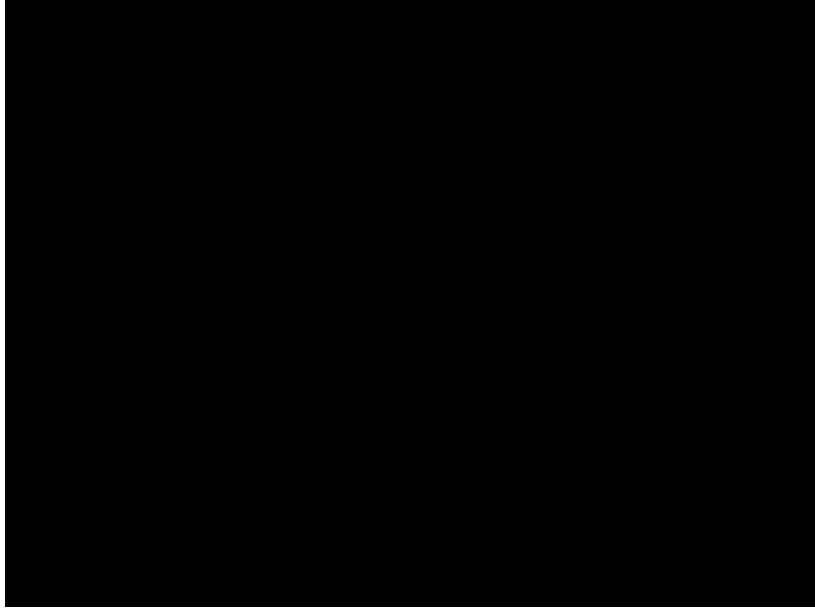


Figure 2: BIOS Trace

The trace memory in Figure 2 displays:

- The first JMP generated a BTM, hence it was displayed in the trace. (Note: Remember you're looking over your shoulder, so the closest step is -1 and the furthest is -19)
- The code that you see disassembled at -15 began executing. (State -14 showed the address from where it branched.) At -9 a write of 80h to I/O port 70h occurred 930ns into code execution, and at -5 a write of C0h to I/O port 80h occurred 1000ns into code execution.
- Our breakpoint triggered four bus transactions after the write to port 80h. The delay between the I/O write, and the ICE recognizing the breakpoint and stopping the processor, is known as *bus slide*. Since the ICE was tracing at the standard processor bus speed of 66Mhz, though, everything the processor did to the system is in trace memory; therefore nothing could have happened without your knowledge.
- Finally, you see timestamp information for all the bus cycles. The timestamp shows how long each transaction was on the processor bus. That data can be interpreted to show "bus hogs": areas of your code that took longer to execute than expected. After the code has been profiled, it can be tuned for maximum throughput.

And This Affects Me...How?

The previous example was used to show how information between two points can be captured in trace and interpreted. The points of interest were the first line of BIOS code and the first port 80h I/O access. You may be interested in how long it takes to execute a specific section of code, like this:

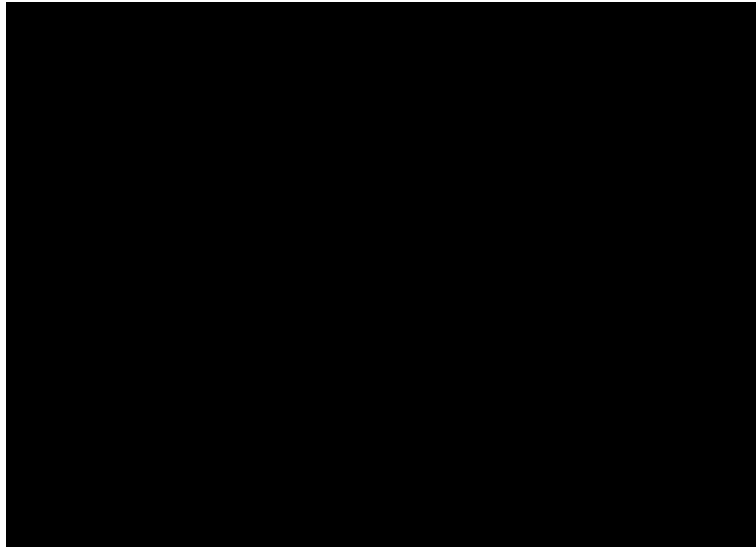


Figure 3: C Source

In this case, a breakpoint was set at the beginning of the C code (**int main**) and at the first line of the **while** loop. The trace information will tell us the amount of overhead accrued in the program's initialization:

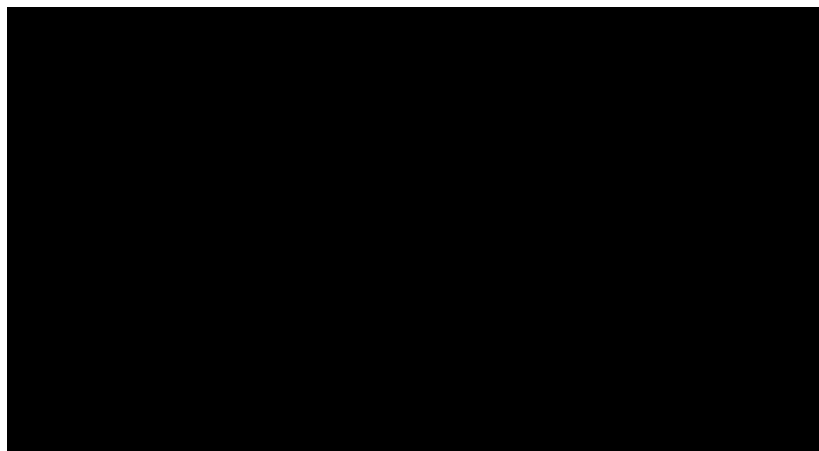


Figure 4: Disassembly and State Trace Information from the C Source

Don't be side-tracked by bus slide or by the processor pre-fetching code. (Also, some engineers prefer to see the results in assembly; others prefer to view source in the trace window.)

In state -2, note that a 78h is being written to memory location CA5Ch; that's the **MOV [EBP+FC], EAX**. The **FETCH** at states -4, -3 and -1 and the **TRIG** is getting the next code to execute (the first part of the **while** loop). The question you've got to ask yourself in addition, then, is "What are my measurement parameters?"

As a side issue, note that the timestamp information displayed in Figure 2 consists of the *delta*, or difference, between bus transactions. Figure 4 uses an *accumulation* of all bus transactions in the timestamp column. Deltas make it easy to understand time variances. Accumulation makes it easy to understand total times.

Besides Profiling

Profiling is just one way to use ICE trace memory in code development. As mentioned earlier, you can also manually initialize the hardware, then review the state information with the hardware designer for accuracy. If the traced actions don't match the expected actions, you may want to consider working side-by-side with the hardware developer. It may be much easier for you to duplicate motherboard error states in software than by changing the hardware - and results can be saved in trace.

You may ask, “Is it possible to save only *specific* results in trace?” Absolutely! **Sequencing and Data Qualification** allow the conditional filtering of bus cycle information as it's being received. The only information stored in the trace buffer, then, is that which meets specific filtering criteria.

But First...“Triggers” and “Breakpoints”

A **trigger** is a unique instance of an event that causes a **breakpoint** in the normal operations of the processor. Remember when we wanted to see the POST codes in section *TAP Limitations*? The I/O write to port 80h was a trigger, the event that caused the processor to freeze. An ICE may start or stop tracing, run a macro or perform any number of user-define actions when the specified event occurs, i.e., “a breakpoint was triggered.” There are three different types of breakpoints: software, debug register and bus event.

Software breakpoints are created when the first byte of some code line is temporarily replaced by an INT 3 (opcode: CCh). Since an INT 3 is a call to any resident debug tool, any attempt to execute that code line will result in control passing to the ICE. The drawbacks are:

1. They can't be used to debug firmware, since the code is either flashed or burned in ROM and therefore not writeable.
2. They can't be used to debug I/O accesses (or memory reads and writes) unless the breakpoint is set on the code line that does the write, not the change of the actual I/O port or memory address.
3. It's easy to get confused with an excessive number of breakpoints.

Debug register breakpoints were discussed in the section *TAP Limitations*. Briefly, you're using features inherent to the processor via it's registers. Breakpoints can be set on code in memory or ROM. Additionally, breakpoints can be set on actual I/O port accesses, not just code line writes, such as software breakpoints.

Why not use debug register breakpoints in all cases? First, there's a limitation of four debug registers. Second, the processor runs slower, so timing-specific problems (such as data communications) may temporarily disappear at the slower speed. Third, breakpoints can't be set for *only* a particular value written, or *only* a particular value read. Finally, note that debug register breakpoints will trigger only for the processors that are set up in multi-processor systems.

Bus event breakpoints occur when something you've triggered on goes flying by on the processor bus. You can set all the breakpoints like the debug registers PLUS:

- Processors run at full speed.
- Specific value writes or reads to memory or I/O will trigger.
- Breakpoints can be set on code in memory or ROM.
- Breakpoints can be set on bus transactions like an interrupt acknowledge (IACK), special bus (SPBUS), deferred reply (DREPLY).
- A breakpoint will occur due to any processor in a multi-processor system causing the trigger event.

The tradeoff is three-fold:

- Breakpoint limitations (American Arium's products allow six bus event breakpoints).
- No triggers on-chip cache events.
- There's a delay, called **bus slide**, between the time the ICE spotted the event occurring and the time it took to stop the processor. Typical bus slide can be from 10 and 20 clocks. That sounds like a lot of information is being lost, BUT recall that it's all being stored in trace memory. Again, nothing happens without your knowledge.

Introducing Sequencing...

Let's assume you have four-levels of triggering, which means up to four unique events can occur before some specified action(s) happen. The ICE may allow you:

- T1: "If the event specified as trigger 1 occurs, then do...."
- T1 then T2: "If and only if the event specified as trigger 1 occurs THEN the event specified as trigger 2 occurs, then do...."
- T1 then T2 then T3: "If and only if the event specified as trigger 1 occurs THEN the event specified as trigger 2 occurs THEN the event specified as trigger 3 occurs, then do...."
- T1 then T2 then T3 then T4: "If and only if the event specified as trigger 1 occurs THEN the event specified as trigger 2 occurs THEN the event specified as trigger 3 occurs THEN the event specified as trigger 4 occurs, then do...."

Exponential confusion and costs are reasons most ICE manufacturers typically allow around four event triggers.

- T1 and not T2: "If trigger 1 happens at the same time trigger 2 DOESN'T happen, then do..."
- Count of T1: "If it's the n^{th} iteration of trigger 1, then do..."

If the term "the event specified as trigger 1 occurs" is too confusing, replace it with "I see Larry." Replace trigger 2 with "I see Nancy", trigger 3 with "I see Jeff" and trigger 4 with "I see Wendy." In that case, it becomes:

- T1 then T2 then T3 then T4: "If and only I see Larry THEN I see Nancy THEN I see Jeff THEN I see Wendy, then go to the beach."
- Count of T1: "If it's the 5^{th} time I see Larry, then go to the beach."

And if you're more comfortable with the PCI register set, use CF8h instead of Larry and so on. (You may "go to the beach" at your own discretion.)

This entire analogy describes **sequencing**. Sequencing provides a method for defining a breakpoint that requires more complexity than a simple software, debug register or bus event breakpoint.

This "A then B" type of ordered breakpoint sequencing can be very useful in triggering on cause-effect situations where *the event occurs due to several different causes, but only one cause is of interest*.

...And Data Qualification

The purpose of data qualification is to increase the effective **trace depth** (trace-dedicated memory located in the ICE) via a real-time filter of the **trace events** (bus cycles) being stored. The American Arium ICE provides two different methods of data qualification, each suited to a different task.

The first method filters the trace data based upon a single **specifier**. On an event-by-event basis, bus events matching this specifier are stored in trace, and bus events not matching the specifier are discarded. The specifier is extremely flexible, allowing any or all signals being collected to be used in determining which trace events to store. A typical example would be storing only memory writes to a particular address. Without data qualification the trace buffer might hold only a handful of this particular event, while the rest of the trace events would be of no interest.

The second type of data qualification is useful when a **snapshot of sequential bus cycles** need to be stored repeatedly. One example where this can be used is tracing the repeated execution of an interrupt routine. The data qualification can be set up to start storing when the interrupt vector is read, and stop storing when the address of

the return instruction appears on the bus. Using data qualification in this fashion prevents the trace from storing the bus activity of anything besides the interrupt routine.

Obviously you can combine sequencing and data qualification to create an incredibly powerful set of development, profiling, troubleshooting, and failure analysis tools.

Bringing It Home

You've learned a bunch of terms and theoretical practicalities; now let's put them to work in a few real world situations. A problem will be given, along with an interpretation of the actions required of the ICE.

Real-World Case #1

You've just written an assembly language procedure called by your power-on sequencer, and you want to see how much time it takes to run your code. You wouldn't mind seeing the time for the individual lines of code, but you're mostly interested in how long it takes to run all the whole code.

Interpretation: You want to stop at the first line of your code to clear trace memory. Then you want to run until the last line of your code is executed, and see the amount of accumulated time it took to run the code. Based on that information, you can tune your code.

Easy stuff. Here's the beginning of your code:

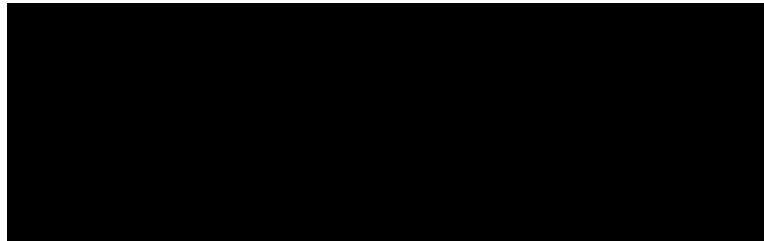


Figure 5: Code Start at FC00:1A09

And here's the end of your code:

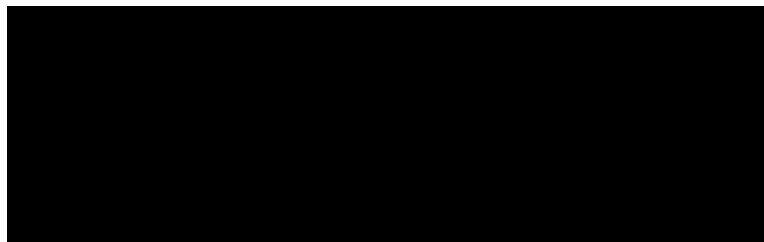


Figure 6: Code End at FC00:1B43

American Arium's WinDb software uses an arrow (Figure 5) to indicate the current location of the instruction pointer (CS:EIP), which means we're currently sitting at the first line of your code. Since we're not interested in the JMP, just how long the rest of the code takes to run, we'll just let it run until it hits the breakpoint (Figure 6) and then look at trace.

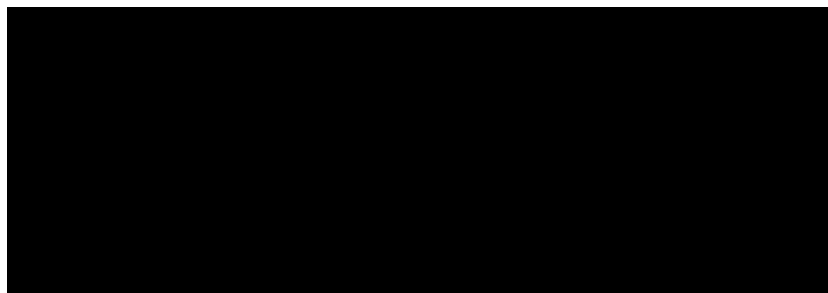


Figure 7: Code Trace with Accumulation Timings

We see the last lines of the code at state -5, and the FETCH at state -1 is the **MOV AL, BL**. Let's go to the top of trace to see how long ago the first line was executed:



Figure 8: Top of the Trace

The first lines of code executed 3.387ms (or 3387us) ago.

Caveat coder: Spitting out each of the BTM cycles took some processor time, so the code execution time in a standard environment (i.e., your customer's site) would be slightly faster than 3.387ms. In other words, your time and mileage may vary.

Real-World Case #2

You want your machine to bootstrap as fast as possible, so you're interested in how long it takes each BIOS POST to execute.

Interpretation: You want to trace all I/O writes to port 80h (remember - the BIOS POST port) and see the time delta between each write. Based on that information, you can tune your code.

The first set is to set a trigger on I/O writes to port 80h. Note that we're not setting a breakpoint in this case, since we don't want the processor to stop, we're just setting a trigger so it can trace. That means that the I/O write to port 80h will be set up as a **qualifier**, not a breakpoint:

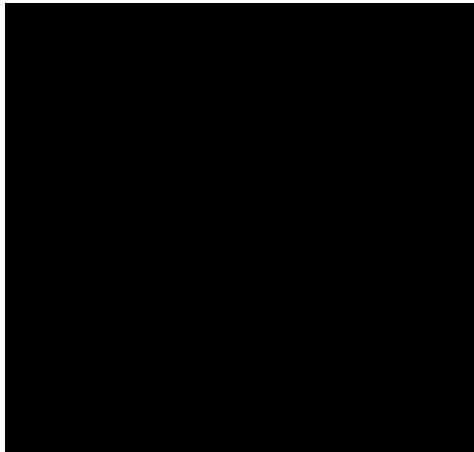


Figure 9: Setting a qualifier

And we can now let **WinDb** know only to trace qualifier Q1, the port 80h I/O write:

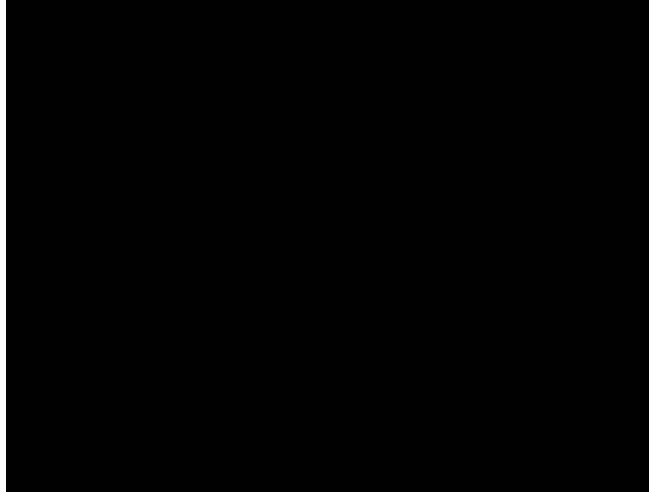


Figure 10: Recording a qualifier

Only writes to port 80h will be stored in trace memory:

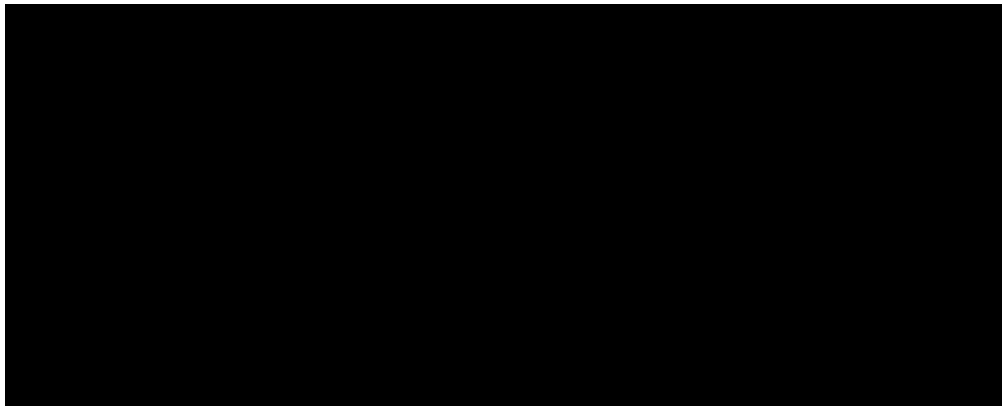


Figure 11: Timing of Data Qualification of Writes to I/O Port 80h

The BIOS POST number is seen in the second **DATA** column. With a little interpretation, the **TIMESTAMP** information (using delta times, not an accumulation) shows the routines that take the longest. Remember that a delta timestamp is the amount of elapsed time since the last trigger. That means the time between POST 7Ch (state -18) and POST 7Eh (state -17), or *the elapsed time to run POST 7Ch*, was 505.170 us (seen in the timestamp at state -17).

Real-World Case #3

For some reason your new motherboard is causing a reset **much** more frequently than you expect. You need to find what is resetting your unit and how it's being done.

Interpretation: You want to set a breakpoint on a reset, and look at trace to find out how you got there. Based on that information, you can change your expectations, motherboard or code.

Just set a breakpoint on reset (or a fetch to F000:FFF0h, the reset vector) and look at the trace:

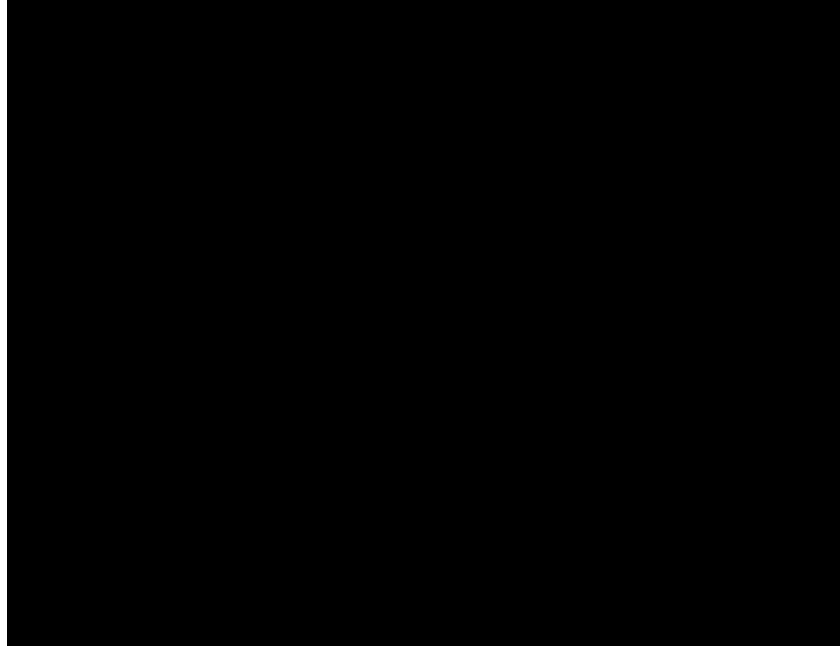


Figure 12: Pre-Reset Trace

Real-World Case #4

Your device driver works fine, but it's being called when it shouldn't be called. You need to find where your device driver is loaded, what routine is calling it and why.

Interpretation: You want to set breakpoint so that when your code is started you can stop all processes and review the trace. The trace will show you the calling routine and the steps it took to determine a call to your code was in order.

Since the operating system may have loaded your code anywhere, we'll use a passive technique. In effect, instead of searching around where the driver is located and setting a breakpoint (active technique), we want sit back and let the *driver* stop the ICE (passive technique).

Since we saw all the POST codes used by our target's BIOS in **Real-World Case #2**, we know if we write the value **AAh** to I/O port 80h at, say, the first line of our device driver, we won't conflict with the BIOS. Further, that means if the ICE triggers when an AAh is written to I/O port 80h, we'll know the location of the first line of our device driver.

So, briefly, we can accomplish our goal with an AAh written to I/O port 80h by our device driver and an I/O Write breakpoint set in the ICE, like this:

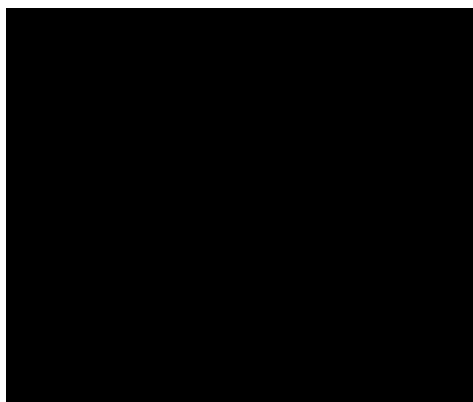


Figure 13: I/O Bus Event Breakpoint

Remember that since we're setting a bus event breakpoint, this will even work in a multi-processor system (where you can't even count on which processor is going to execute your code).

Completing this example by obtaining trace data is left as an exercise for the reader.

Real-World Case #5

Power-On Self-Test 3, a very small routine that seems too simple to fail, caused some unexpected results. You need to find and resolve the problem.

Interpretation: You need to trace between POST 3 and POST 4, then review the trace. Further information in the trace may result in further sequencing and data qualification, followed by possible target changes.

This is similar to Real-World Case #1: You set the breakpoints at the beginning and end of POST 3, and review the trace. If it's an interrupt, you may want to set a breakpoint on the IACK to figure out why it was called. If writing to a memory-mapped I/O register swapped out your ROM and swapped in memory, you may want to set a breakpoint on the I/O write and single-step the code (or connect additional analysis tools).

The paths you take for additional investigation is, as always, highly dependent on the data you find in the trace.

Summary

In this paper we've reviewed some basic definitions used in the world of In-Circuit Emulation, including:

- ICE, TAP, JTAG, debug port and Boundary Scan
- Triggers, breakpoints, sequencing and data qualification
- Trace, BTMs, Non-intrusive bus monitoring and bus slide

We've also discussed quite a few concepts, such as:

- Expectations of an In-Circuit Emulator
- Debug limitations of the TAP
- Pros and cons of setting software, debug register and bus event breakpoints
- Using trace memory to profile code execution, develop, debug and analyze failures



14281 Chambers Road
Tustin, CA 92780
Voice: 714-731-1661
Fax: 714-731-6344
Web: www.arium.com
E-mail: info@arium.com

Pentium is a registered trademark of the Intel Corporation.

WinDb is a trademark of American Arium.

Copyright© 1998, American Automation dba American Arium