SYNOPSYS®
Accelerating Innovation

Real-Time Trace: A Better Way to Debug Embedded Applications

November 2014

Authors

James Campbell
CAE, Synopsys Inc.

Valeriy Kazantsev
CAE, Synopsys Inc.

Hugh O'Keeffe
Engineering Director, Ashling Microsystems

Abstract

Firmware and application software development is often the critical path for many embedded designs. Problems that appear in the late phases of the development can be extremely difficult to track down and debug, thus putting project schedules at risk. Traditional debug techniques cannot always help to localize the issue.

This whitepaper presents the concept of debugging with 'real-time trace' hardware assistance and shows its benefits, including how it can vastly reduce the amount of time needed to track down problems in the code. In addition, this paper introduces the other benefits available with a real-time trace system, such as hot-spot profiling and code coverage.

Introduction

A study from Cambridge University [1] has estimated that software developers spend almost 50% of their time debugging code, at a total cost of $312 billion per year in terms of developer salaries and overhead. Debugging native desktop applications is already a complex task, but cross-debugging an embedded system can be significantly more challenging for a number of reasons: real-time execution, lack of visibility into the target resources, dependencies on external peripherals, etc. For many embedded designs, firmware and application development is often on the critical path and the last phase of the design cycle. Problems with the firmware can delay the entire release and cause cost overruns in the best case and completely missed market opportunity windows in the worst case. Given these challenges, it's absolutely critical for engineering managers to provide their developers with powerful debugging tools that can help to quickly resolve even the most challenging bugs.

Traditional Embedded Debugging Techniques

How do engineers typically debug? Are the tools at their disposal effective? Traditionally, a couple of different techniques are used to debug embedded systems, but these have a number of drawbacks.

"printf" Debugging

If code is not behaving as expected, engineers add diagnostic print statements to the code, which emits the relevant data to the terminal as the code runs (see Figure 1). They can then examine this in the hope of finding clues about the cause of the problem (for example, the value of a certain variable at a given point in time is not what is expected). This technique can work in cases where the bug is caused by a deterministic algorithmic error and the developer is quite sure of the approximate place in the code where the problem is occurring. However, this can be a time-consuming process and is not guaranteed to localize the problem.

Some disadvantages of this method include:

- It is almost useless in real-time code, since inserting print statements changes the program's timing and code paths and may entirely mask the problem
- Many deeply embedded systems do not have UARTs connected to debug consoles and semi-hosted I/O via the debugger also affects real-time performance
- Many edit+compile+run iterations are needed; this can be time consuming if the code base takes a long time to compile or if the code needs to run for a long time to hit the problem area

Surprisingly, a number of developers still rely on this debugging technique, which likely contributes to the huge debugging costs reported in the research.

```
for (iter = 1; iter <= ITERATIONS; iter++) {
      Count = 0;
      for (I = 0; I <= SIZE; I++)
            Primes[I] = True;
      for (I = 0; I <= SIZE; I++)
      if (Primes[I]) {
            Prime=2*I+3;
printf("Prime:%d\n", Prime);
      K = I+Prime;
      while (K <= SIZE) {
            Primes[K] = False;
printf("K: %d\n", K);
            K += Prime;
      }/*endwhile*/
      Count++;
}/*endif*/
}/*endfor*/
```

Figure 1: Code instrumented with printf()

Run-time Debugging

Run-time debugging is a common technique used by most embedded developers. The embedded CPU has JTAG pins exported to a header on the development board that are then connected to a small debug "pod" that serves as an interface between the target and the host PC. The developer runs a cross-debugger on the host, which then communicates with and controls the target CPU over the JTAG connection (see Figure 2). The debugger typically offers functionality like:

- Single stepping through source or disassembly
- Inserting code and data breakpoints, and free-running to them
- Viewing and changing local and global variables
- Viewing and changing target memory and registers

Although this technique allows for much more in-depth debugging than the "printf" method, one major drawback is that the target must be halted in order to examine its resources. In some cases, depending on the CPU, certain items can be accessed while the target is running, but most items are only available when the target is halted. In a real-time system, this can be detrimental since stopping the target immediately changes real-time behavior; in many cases the problem being investigated will disappear when code is being single-stepped, or halted. Stopping the target causes further problems if the application is communicating with peripherals or external nodes that are not aware that the CPU has halted and are still either sending or receiving data. If the system gets notified via interrupts about external events like incoming packets from a network MAC or mailbox messages from another CPU, the interrupts risk 'getting lost' and the remote side may 'time out' if the target is stopped for too long. Another significant drawback to this debugging technique is that it can only show the current state of the system (current values of registers, variables, memory, etc.) at a particular point. There is no ability to see any history of the code's execution.

Figure 2: An example of using JTAG to debug an embedded target

A Better Type of Debugging — Real-Time Trace

Given the disadvantages inherent with the printf and run-time debugging techniques, these tools are insufficient for finding the most challenging bugs. With only these tools at the developer's disposal, the debugging effort could add weeks or more to the development schedule. This is where real-time trace (RTT) debugging can be used to great advantage.

With this method, the embedded SoC contains an RTT module that sits next to the processor. As the target runs, the trace module records information about the execution on the target. This may include:

- Sequence of program counter (PC) values
- Memory reads/writes and associated data values
- Register reads/writes

Combined together, the above information provides a complete picture, or trace history, of exactly what happened on the target during the execution. Importing this data into the appropriate analysis tools on the host side allows the engineer to do a post mortem debug to quickly pin-point the issues and uncover exactly why the code is not performing as expected.

The following are some keys advantages of real-time trace:

- Developers can see why/how execution arrived at a certain point, via a back-trace or instruction history, and can now easily answer questions like "Why and how did I end up in this function?" and "Why did my code crash?"
- Developers can easily correlate the corruption event with a specific instruction to see exactly where data corruption occured
- Trace information can usually be captured non-intrusively, meaning that the application's real-time performance is not affected
- Trace information allows developers to profile their code to find out where time is actually being spent and to determine if deadlines are being met
- Trace can capture and record intermittent behaviors that may not always occur. Once captured, developers can easily track the root cause

Despite real-time trace's numerous advantages, there are some things to keep in mind when investigating trace solutions:

- The RTT logic block consumes space on the SoC. Although modern process technologies make this less important, it's still a consideration. It's important to select an RTT technology that is build-time scalable so it can be configured to match the nature and typical use case of the SoC
- The RTT logic block consumes extra power. It's therefore important to choose an RTT solution that allows the RTT block to be gated off when not in use
- An external debug probe is often needed. Therefore, choosing a solution that works with an efficient and cost-effective debug probe is important

Choosing a Real-Time Trace Module

There are many types of RTT modules that differ in functionality, area, power, and performance. Depending on the application, various types of RTT modules may be used.

A deeply embedded processor running a single process might take advantage of a lightweight trace module that only records the last eight or sixteen branches in the code. Such a module would have very low area overhead, and still typically allows reconstruction of a trace of at least 100 instructions.

On the other hand, a multicore, high-performance embedded processor running Linux requires a more comprehensive solution, as traces of tens of thousands of instructions or more may be needed to debug complex issues in a multi-process environment.

The following sections describe the tradeoffs in more detail.

Configurability

Typically an RTT module can be configured to save only the information necessary for debugging:

- Small trace: save only non-sequential PC values
- Medium trace: save PC values and memory reads/writes
- Full trace: save PC values, memory reads/writes, and register writes

These options must be chosen at design-time because additional hardware must be added to connect the RTT module to the internals of the processor.

An RTT module that records a full trace makes debugging easier than if it records only a small trace. However, these configuration capabilities ultimately affect the overall gate count and performance of the RTT module, as well as the amount of memory required to store the trace history. Designers can therefore make area vs. functionality vs. performance tradeoffs.

Filters

No matter how the RTT module is configured, it will generate a huge amount of data during the capture, since it is potentially recording trace history for a processor that could be executing at hundreds of MHz.

One technique to reduce the amount of data generated is to use compression, but a full trace can still consume gigabytes of space.

A better solution is to make use of the filters within the trace module that allow developers to constrain the trace to regions of interest within the code. Some examples of how filters can be used to restrict the trace:

- Trace only when function func() is executing
- Start tracing when function bar() begins execution and stop when bar() finishes, tracing all activity, including sub-functions during that time
- Trace memory accesses only when global variable X is written
- Trace memory accesses when value 1234 is written to address Y
- Trace register writes for r3, r4, r5 and r6

With the detailed controls that filtering offers, developers can focus the tracing results on specific areas of the application, which ultimately facilitates application debugging.

Trace Storage

Memory to store the trace history is the essential component of the RTT module. In general, trace data can either be stored locally in on-chip or in on-board memory for later retrieval by the host, or can be exported directly from the target in real time using a dedicated high-speed interface. Table 1 lists some different storage options and where they can typically be used.

Storage type	Memory type	Typical size	Storage good for		
			Small trace	Medium trace	Full trace
Local buffer	Flip-flops	32-512 Bytes	✓		
Dedicated on-chip memory	SRAM	1..128 kBytes	✓	✓	
Shared system memory	SRAM (on-chip)/ DRAM (on-board)	Up to a few GB		✓	✓
External trace pod	Any (off-board)	Up to a few GB		✓	✓

Table 1: RTT trace storage options

A local buffer made of flip-flops is typically implemented inside the RTT module. It has the fastest possible access time, but the memory size is very limited.

When the size of the local buffer is not enough, an RTT module with a dedicated on-chip memory may be used. The RTT module can be connected to the memory either directly or via a dedicated bus fabric.

When even larger storage is required, the RTT module can use shared system memory. In this case it is connected to an existing bus fabric as a typical master. However, use of this type of memory is not non-intrusive, as the RTT module will compete for the bus access with other masters.

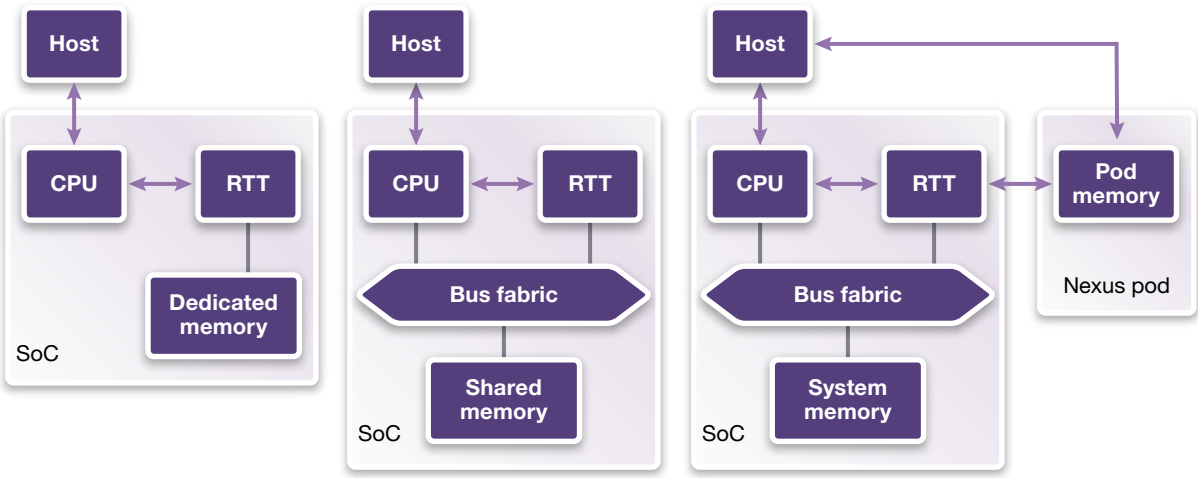

Figure 3: Different types of trace storage

When it is required to have very large storage and still maintain non-intrusive operation, an 'off-board' memory can be connected to the RTT module via a dedicated high-speed debug interface. Figure 3 illustrates different approaches to connecting trace storage to the RTT module.

Nexus Interface

Nexus (IEEE-ISTO 5001™) is an open industry standard that provides a general-purpose interface for the software development and debug of embedded processors [2]. Nexus is a packet-based protocol that can use either a typical JTAG port or a dedicated high-speed auxiliary port.

If an RTT module has an option to export the auxiliary port of the Nexus interface, it can be used to dump trace history to the external debug pod in real time.

The advantages of an RTT with Nexus:

- No need to have a dedicated trace memory – saves area and power
- No need to save trace history in a shared system memory – avoids intrusive memory transactions

The disadvantages of an RTT with Nexus:

- Pin count increase (Nexus typically adds 7-19 pins, depending on the data width)
- A special debug probe is required

Multicore Trace

An RTT module that natively supports multicore systems allows designers to connect a single trace module to multiple processor cores. Eliminating the need to instantiate separate trace modules for each core saves silicon area. It also saves pin count when the Nexus interface is used, as only one interface per multi-core processor is required.

Power Saving Features

An RTT module is used for debugging only. When programs are working correctly, the RTT is not needed, yet continues to consume precious power. It is therefore essential to minimize power consumption while the RTT is idle. Using an RTT with the following features can help:

- Clock gating allows systems to minimize dynamic power usage. The RTT clock is switched off whenever the RTT is not used
- Power gating allows systems to minimize leakage power by switching off the power supply for certain areas of a chip. Depending on the implementation of the RTT module, it may support partial or full shutdown. This is particularly important for RTT modules with large dedicated trace memories

Trace Probes

An RTT module with a small on-chip memory does not require any special trace probes. Virtually any low-cost JTAG probe can be used to export the trace to the host where it will be analyzed. Even if the size of the trace exceeds tens of megabytes, it is typically a matter of minutes to read it via JTAG.

Using a Nexus interface requires a dedicated Nexus-compatible high-speed trace probe, such as the Ashling Ultra-XD probe.

Ashling Ultra-XD

The Ultra-XD is a powerful high-speed trace and run-time control debug probe. The Ultra-XD allows capturing and viewing of program-flow and data-accesses in real-time, non-intrusively, and also supports program download and exercising (go, step, halt, breakpoints, interrogate memory, registers and variables). This means that a single probe can be used for both tracing and regular 'run-time' debugging.

The Ultra-XD captures up to 4 GB of trace frames allowing long tracing periods before the trace buffer overflow. The Ultra-XD trace buffer can be optionally configured as a circular buffer, thereby always recording the last 'n' trace events, as opposed to events from the beginning of the session up until the buffer reached capacity.

The Ultra-XD features automatic trace clock and data skew adjustment ("AUTOLOCK") to ensure integrity of captured high-speed data and it automatically calibrates itself to the embedded target's trace data port. Ultra-XD supports single- and multicore designs and is compatible with hardware debug standards including cJTAG, JTAG, and RTT NEXUS Trace. Figure 4 shows the Ultra-XD connected in a typical system.

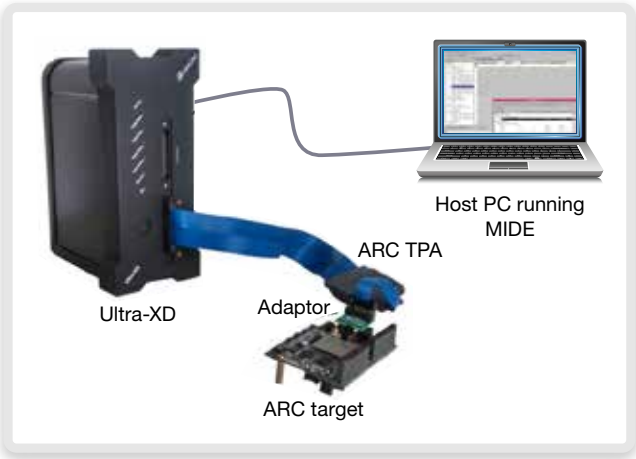

Figure 4: Ashling Ultra-XD probe

Trace Analysis Software

Raw trace history data on its own is not very useful. Typically the gathering and analysis of trace captures is entirely controlled by RTT-aware debuggers.

The debug process works as follows:

1. Developers set up trace sources and filters and select the storage memory if multiple choices are available (for example: on-chip memory or a Nexus-compatible debug pod), then activate tracing. A typical configuration window is shown in Figure 5.
2. After the tracing is activated, developers run their applications under debugger control as normal.
3. After execution ends (or is stopped), developers acquire the trace data for display within the debugger.

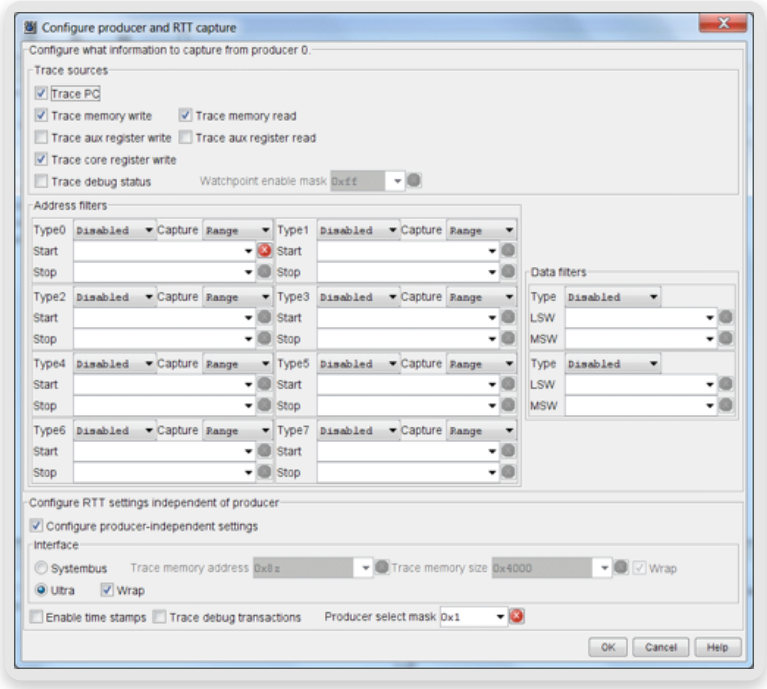

Figure 5: Typical RTT configuration window

The trace data shows the sequence of instructions that were executed. If configured, memory and register accesses are shown interleaved with the instructions. This is essentially an 'instruction history' display that shows the sequence of events that ultimately ends at the current program counter.

Here's an example of how developers can make use of the captured trace data: The code is hitting an exception and is halting inside an exception handler. The engineer needs to understand what sequence of events caused this exception to occur. After the trace is configured, the developer would set a breakpoint on the exception handler and run the code until the breakpoint hits. After acquiring the trace, the developer can scroll backwards through the trace log to find the sequence of events and function calls that led to the exception.

Debugging with Trace Information

A large trace log can be difficult and time consuming to parse through. Therefore, for complex problems, the debug tools should be able to assist with the trace analysis. Some debuggers offer the ability to 'debug' through the trace log. In other words, typical 'run-time' debugging techniques can be used, but with the captured trace data. Developers can capture a non-intrusive trace then have the debugger's full 'execution' capabilities at their disposal to analyze the trace. Developers have access to standard run-time debugging features like breakpoints, watchpoints (data breakpoints), stepping, etc. In addition, they can use backwards execution, a unique feature not available with standard debugging. When using trace, developers typically need to understand why they have arrived at a certain point in the code. Reverse execution can help them to 'unwind' the execution and see where the program took an unexpected path or exactly why some data corruption occurred. For example, they can track variables, memory and register values, observing how they change during the backwards execution.

Profiling

The trace data contains information about the execution and many trace-aware debuggers can analyze this data to generate profiling information about the target's execution during the trace capture. For example, it's helpful to understand where the particular 'hotspots' are within an application, to see if there are any unexpected bottlenecks and to target future optimization efforts. Profiling data can be available both on a per-function basis and on a per-line basis.

Real-Time Trace Options for ARC Processors

There are two different trace solutions available for Synopsys' DesignWare ARC® processor families, allowing designers the flexibility to choose a solution most suitable for their project:

- ARC SmaRT (Small Real-time Trace)
- ARC RTT (Real-Time Trace)

SmaRT

SmaRT is a low-gate count solution that captures instruction trace history until the processor halts. It uses an internal buffer to store the addresses of any non-contiguous instruction that is executed. When the target halts, this captured information is then uploaded to the ARC MetaWare Debugger via a standard JTAG probe. The debugger can then reconstruct the execution and present the developer with an instruction history display, showing a back-trace that can help the developer understand how execution arrived at a certain point. The size of the SmaRT buffer is configurable at design time, which allows developers to trade-off between the size of the module and the length of the back-trace.

ARC RTT

ARC RTT is a complete and highly configurable trace solution available for the ARC HS and ARC EM families of cores. It supports three different trace modes, allowing developers to choose what they want to include in the traces, ranging from only program counter values up to a full trace including program counter, memory accesses, and register accesses. ARC RTT can be configured to use either on-chip memories or can offload trace data in real time using the Ashling Ultra-XD pod.

ARC RTT module includes two different types of filters:

- Address filters – allows filtering control based on the addresses of various trace sources
- Data filters – allows filtering based on the data associated with various trace sources

These filters allow full control over the trace information that's captured, allowing tracing to be focused on particular problem areas in the application.

The ARC MetaWare Debugger is fully RTT-aware and has full integration with the Ashling Ultra-XD probe. Developers control and analyze the trace directly within the debugger, by first setting up the trace parameters and filters, and then viewing the captured trace data. The ARC Replay feature provides the ability to 'debug' the captured trace, using either forward or backward execution, breakpoints, stepping, etc.

Conclusion

Real-time trace is a powerful debugging technique that can help developers find even the most difficult bugs, and can vastly reduce the amount of time spent debugging an application. It's important to select a trace solution that is scalable, so that designers can make appropriate area vs. functionality vs. performance tradeoffs for their application. Different trace storage options are available, such as on-chip storage or real-time offload via a probe like the Ashling Ultra-XD. In addition to the trace hardware module itself, a complete solution needs to also provide advanced analysis tools within the debugging environment. The DesignWare ARC processors support two different trace modules, SmaRT and RTT, each with different capabilities, both of which are tightly integrated with the ARC MetaWare Debugger software. The debugger includes a number of power analysis and profiling tools including a Replay capability, which facilitate the analysis of the trace data.

References

[1] Reversible Debugging Software, University of Cambridge, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.370.9611

[2] Nexus 5001™ Forum, http://www.nexus5001.org

11/14.AP.CS5015.