WHITE PAPER

Debug and Trace for Multicore SoCs

How to build an efficient and effective debug and trace system for complex, multicore SoCs

William Orme September 2008

Abstract

As SoC designs become ever more complex with levels of system integration increasing, multi-functional, multicore SoC are now the fastest growing section of ASIC/ASSP design starts. Hence, designers are faced with three debug and trace options:

i. *Bury your head in the ground, hope it will all be alright (it won't! Software will take forever to get qualified, products will be less reliable and perform worse than the competition).*
ii. *Follow the old route used for simple, single core designs (massive and unnecessary cost in gates, pins and tools and no overall system visibility).*
iii. *Use a joined up SoC debug and trace IP and tools solution that gives the software developer just what he had before (nice 'n' easy) but utilizing massive amounts of improvements "under the hood" (work ARM has already done for designers).*

This article will discuss how to implement debug and trace solutions for simpler single core systems and more complex multicore solutions, explaining the trade-off decisions between device costs (silicon area and pin count) and debug and optimization functionality. The article will show the resulting benefits to the developer by example of ARM CoreSight IP on-chip.

Keeping visibility despite rising complexity

The embedded community, those that put microprocessors in everyday things such as cars, phones, cameras, TVs, MP3 players, printers as well as the communications infrastructure the public doesn't get to see, knows how important it is that these products should "just work" and preferably better than the competitors' products. But as the SoC systems behind these continue to increase in complexity that simple goal gets harder and harder. The challenge increases with the rise of multicore systems.

To get systems to work well means giving engineers, all the way along the design and test cycle, visibility in to what their systems are doing. At the modeling stage visibility is provided in the modeling tool, however, once you move to a physical implementation the designer must include mechanisms to provide visibility. Choosing which mechanism to provide should be a direct response to the needs of the different engineers doing hardware bring-up, low level system software, RTOS and OS porting, application development, system integration, performance optimization, production test, in-field maintenance, returns failure analysis, etc. All need to be satisfied and although their respective tools may handle and present the data in different ways, they all rely on getting debug and trace data from the target SoC.

Making trade off decisions

The easy answer is to fit everything, give full visibility to everything happening on-chip in real time. Most processors offer good debug and trace capabilities as do the interconnect fabrics and if you have custom cores they can have custom debug capabilities added. A system such as the ARM® CoreSight™ Architecture (Fig. 1) can be used to integrate all these together, handling many of the issues of multicore SoCs. However, the costs in IP design time or licensing fees,

silicon area, pins and tools may need strong justification to fit in to tight budgets. This article takes you through the options and reasons for fitting each element so that you, the ASIC designer or product specifier, can decide what is obligatory or highly beneficial and what is only a-nice-to-have or simply not needed for the engineers (listed above) working on your product over its lifetime.

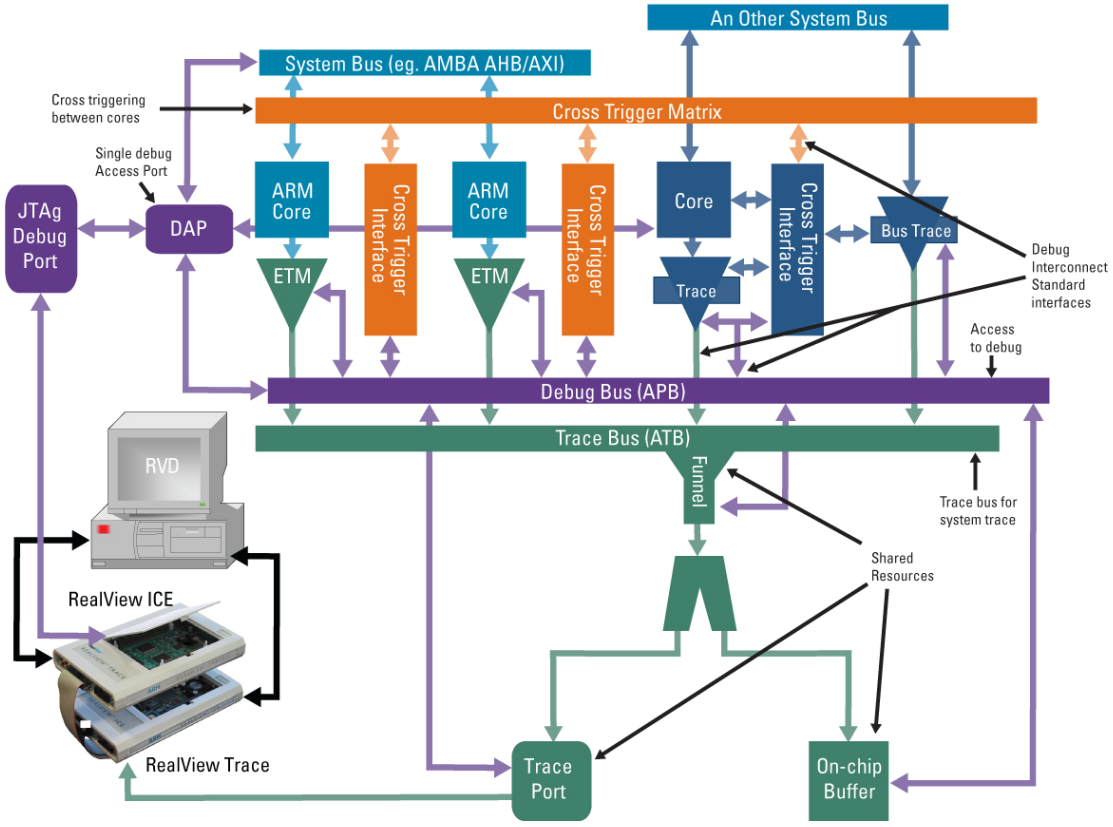

Fig.1 An SoC using the CoreSight Architecture

Run control debug

Almost all SoC designs will need to enable basic run control debug, where the core can be halted at any instruction or data access and system state examined and changed if required. This 'traditionally' uses the JTAG port. However, the number of pins can now be reduced to two, using technology such as the ARM Serial Wire Debug (one bi-directional data pin plus an externally provided clock overlaid on TMS and TCK). Where boundary scan test is not employed or separate debug and test JTAG ports are implemented this can save 2-5 pins (TDO, TDI, nTRST, nSRST and RTCLK). Where boundary scan test is employed, the redundant pins can be re-assigned when not in test mode. Where re-assignment occurs to pins for a trace port this does not even cast a "test shadow".

For multicore SoCs where cores are placed in multiple clock and power domains (mainly for energy management) a traditional JTAG daisy-chain should be replaced with a system able to maintain debug communications between the debug tool and the target, despite any individual core being powered down or in sleep mode. The CoreSight Debug Access Port (DAP) is an

example of a bridge between the external debug clock and multiple domains for cores in the SoC (Fig. 2). This has the secondary advantage of being able to maintain debug communications with any core at the highest frequency supported rather the slowest frequency of all cores on a JTAG daisy-chain. For those requiring ultra fast code download or access to memory-mapped peripheral registers while the core is running, the ASIC designer should connect up a direct memory access from the DAP to the system interconnect, so that the debug tool can become a bus master on the system bus. For remote debug of in-field products or large batch testing where a debug tool seat per device under test is unrealistic, the designer can also connect the DAP into a processor's peripheral map allowing the target resident software to set up its own debug and trace configurations.

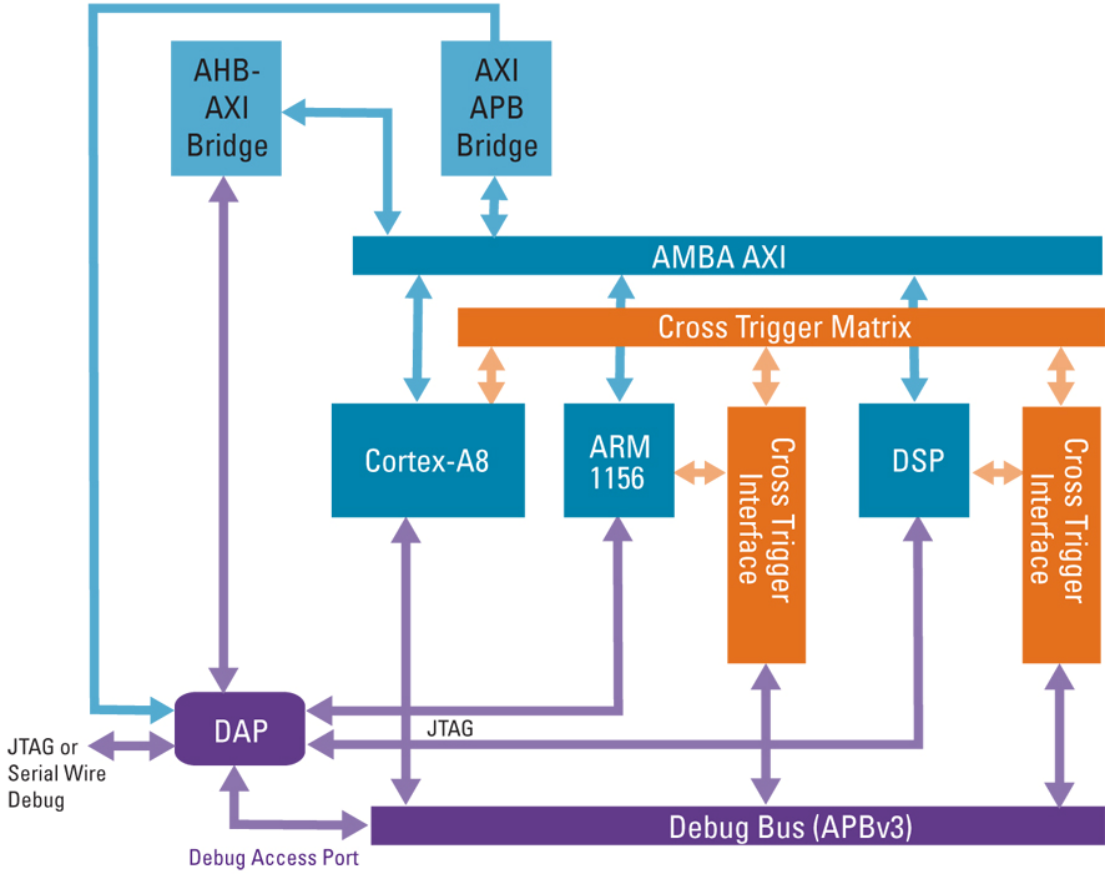

Fig.2 – Debug Connectivity for Multicore SoC

A common criteria for the debug of embedded systems is to be able to debug from reset and through partial power cycles. This requires careful design of power domains and reset signals. Critically, reset of the debug control register should be separated from that of the functional (non-debug) system. Power-down can be handled in different ways such as ignoring power-down signals when debugging or maintaining the debug logic in different power domains which are not powered down while debugging.

For multicore systems where there is any form of inter process communication or shared memory the ability to stop and start all cores synchronously is extremely valuable. To ensure this synchronization is within a few cycles, a cross trigger matrix should be fitted (Fig. 2). If on the

other hand the cores have widely separated and non-interfering tasks it may be sufficient to synchronize stopping and starting of cores with the debug tools, which will inevitably lead to 100s of cycles of skid between cores stopping. Synchronous starting of cores can be achieved with either a cross-triggering mechanism or via the TAP controller of each core.

Fitting multiple debug ports, one for each core, has obvious silicon and pin overheads, and leaves the synchronization and power-down issue to be managed by the tools (which most likely they don't). This approach only has merit where completely different cores with separate tool chains are employed and the re-engineering cost of sharing a single debug port with a single JTAG emulator box are substantial higher than the costs of duplicating debug ports and debug tool seats. This may be sufficient where two separate systems co-reside on the same piece of silicon, but debugging both systems simultaneously is rare. An example might be an MCU plus a dedicated DSP or data engine, where the DSP or data engine is not re-programmed by applications, but a set of fixed functions developed independently.

How to size your trace subsystem

After run-control debug, trace is the next most important debug feature, by which I mean the passive recording of the system execution while it is executing. This is obligatory in hard real-time, electro-mechanical systems where halting the control system is just not an option, e.g. hard disk drives and engine/motor control systems. However, it is also highly beneficial for debugging any system that reacts with another system (e.g. the real world) in a data dependent or asynchronous manner. And that covers just about any complex embedded system! Trace allows the capture of errant corner cases that system validation pre-tapeout just could not cover. Three other very important usage cases for trace are: performance optimization of an application; efficiency of software and system development; and accountability – hard evidence as to the cause, and thus responsibility for, a product failure. Choosing the level of trace has the largest impact on the cost of implementing the on-chip debug system. The good news is that for multicore SoCs the cost per CPU can actually be reduced. The first question to ask is: who is going to use the trace data and with what tool?

Software trace

The simplest and cheapest form of trace is that generated by the software executing on the cores themselves. Traditionally this data was written to an area of system memory, while a separate task emptied and sent back the data to the debug tools via any available communication channel, the ARM Debug Comms Channel (DCC) over JTAG being a common choice. Recent optimizations on this approach write to a peripheral such as the CoreSight Instrumentation Trace Macrocell (ITM) (Fig. 3) which streams the trace data direct to a trace buffer, with the benefit of minimizing, and making deterministic, the number of cycles taken to instrument the code. It also provides a higher bandwidth channel to allow more instrumentation points and enables very deep off-chip buffers. The biggest drawbacks of this approach are the intrusiveness on the application execution time and limited trace bandwidth. However, it is a good approach where all the target resident software and the debug tools to interpret the trace data have been written with this mechanism in mind. For multi-processing systems, instrumentation trace has the advantage of understanding its own context (e.g. which thread am I?) and can add a higher level semantic that is extremely useful to a software application developer. Additionally, the processor has access to registers, such as the Performance Monitor Unit (PMU) of an ARM core, which can provide valuable system performance profiling data. Given the relatively low implementation costs and high potential benefits, instrumentation trace is an obvious candidate to fit in any multicore SoCs.

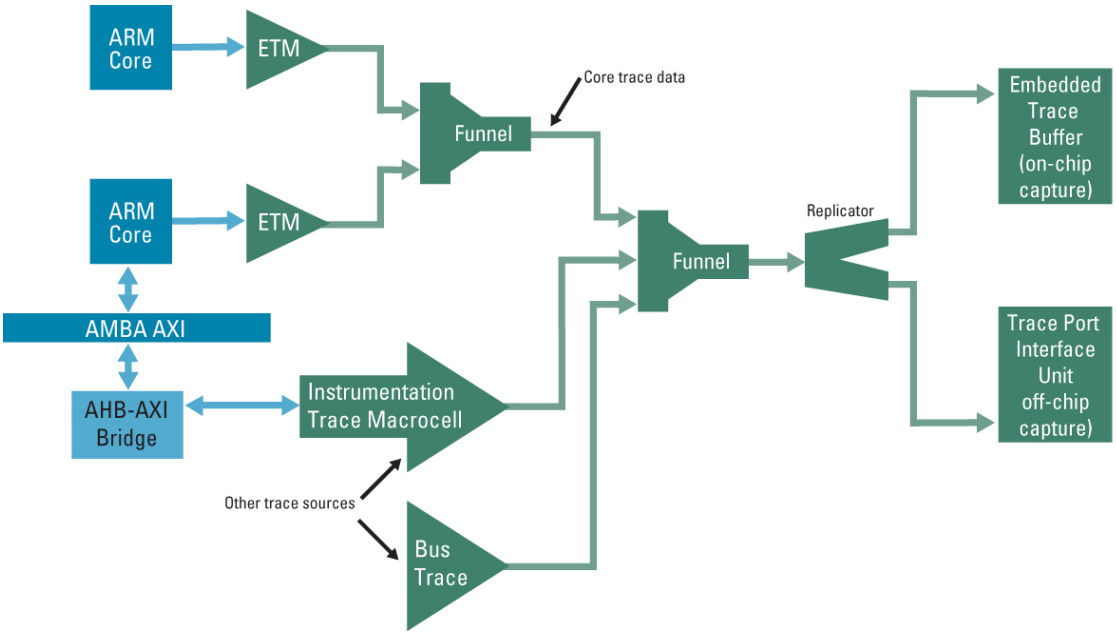

Fig.3 – Multiple Trace Sources Combined with a CoreSight Trace Funnel

Hardware Trace

Where more detail is required or code instrumentation is not adopted, hardware trace, such as ARM Embedded Trace Macrocells (ETM™) is extremely popular (uptake amongst licensees of both ARM11 and Cortex families of processors is >90%). Hardware trace, by which I mean logic that watches the address, data and control signals within the SoC compresses that information and emits to a trace buffer, itself can be subdivided in to three main categories: program/instruction trace; data trace; and bus (or interconnect fabric) trace. Each of these functions has different usage models and different costs.

Program trace is highly valuable for both hardware and software debugging and the main source data required for many profiling tools. In terms of implementation costs program only trace macrocells can be quite small, the ARM Cortex-M3 processor has a program trace only ETM of ~7K gates, and the data compresses well, requiring only ~1 bit/instruction/CPU thus the bandwidth requirements for a trace port are not too high, even for a 4xCPU multicore SoC with 500 MHz - 1GHz CPU clock. Where on-chip trace buffers are implemented, a 4K RAM can hold over 30,000 lines of assembler code execution, that's a lot of code for an embedded developer to review! Additionally profiling tools, such as the ARM RealView® Profiler, can continuously process program trace data in real-time for cores up to 450 MHz for runs off several hours, even days, if required. The addition of cycle-accurate instruction trace, useful for close correlation of the interaction of multiple processors, increases the bandwidth to ~4 bits/instruction, substantially increasing the required frequency and width of a trace port.

However, there are some classes of bug, where it is necessary to see the data (data addresses and/or data values). Many process control algorithms are driven by the data and watching parameters over time is important. Some, difficult to replicate system bugs are the result of data coherency errors (in hardware, system configuration or software) where a trace of the data values, or who has accessed a shared location and when, can soon track down the problem. Several debug tools contain a powerful feature where all windows of the debugger, including processor register and memory values, can be re-created from the trace data, allowing a programmer to step forwards (or backwards) through code actually executed in real-time in the real environment, showing it real (mis)behavior. Unfortunately, the cost of implementing data trace is the highest of all: trace macrocells need to be larger, data is more difficult to compress (data trace from an ARM ETM typically requires 1-2 bytes/instruction); trace buffers need to be larger and trace ports faster. However, the upside of higher levels of SoC integration is that the gates required can be squeezed in to ever smaller areas, so even high performance, multicore systems can have data trace capabilities if required. Multiple on-chip trace buffers can be implemented or trace ports using high speed PHYs can now support multiple gigabit lanes. Today's technology supports up to 6 lanes at 6 Gbit/s, enough for full, cycle-accurate simultaneous program and data trace of three ARM cores running at ~600 MHz.

Sizing the trace port, is another key task for the ASIC designer and another trade-off decision derived from understanding the cost of implementation versus the level of trace functionality (described above). For multicore SoCs the best approach may be a combination of solutions. For example by fitting three parallel Trace Funnels any subset of trace data may be sent to one of three destinations: a very high bandwidth interface to on-chip trace buffers; a medium bandwidth trace port to a very deep off-chip buffer; a very narrow (even single pin) interface for continuous monitoring. This gives a trace solution that can provide for almost any usage case from hardware fault analysis where the instruction-by-instruction code and data is recorded over a period of 1000s of cycles, through software debug and profiling of multicore code over trillions of instructions, to a high-level application-generated trace available even from the most connectivity-challenged end product.

Multiple Trace Sources

As with debug ports, fitting multiple trace ports, one for each core, has obvious silicon and pin overheads. One solution is to use a CoreSight Trace Funnel that combines multiple, asynchronous, heterogeneous trace streams into one for output via a single trace port or trace buffer (Fig. 3). This provides better visibility, a higher bandwidth port or deeper buffer, for single core use as well as while reducing the implementation overhead (area, pins and tools) substantially when simultaneous trace of multiple cores is required. Furthermore this is an ideal mechanism for extending support to any source of trace data. Other sources of trace data may come from trace macrocells for DSPs, bus monitors, embedded logic analyzers or in-silicon validation logic, such as synthesized assertions. The essence of the system is to provide a data path from trace data generation through to a file on the developer's workstation for use by the tool that configures and displays the data for the respective trace generator,