Debug support for complex systems on-chip: a review

A.B.T. Hopkins and K.D. McDonald-Maier

Abstract: The introduction of complex systems-on-chip (SoC) devices with multiple processor cores presents new challenges for embedded systems developers. Novel development tools specifically targeting complex SoC will help overcome these challenges, but are typically limited by inadequate debug support facilities within the SoC. High-quality debug support with advanced features is essential to take full advantage of complex SoC devices in challenging applications while simultaneously reducing development time. Here, existing strategies for providing comprehensive SoC debug support targeting hard real-time applications, such as automotive control, where development challenges are overwhelming are reviewed. This overview includes an evaluation of the available solutions and their suitability for use with the next generation of complex SoC based on multiple processor cores. It is shown that many existing solutions do not readily permit developers to take advantage of the complex features integrated into the next generation of SoC. The essential features of debug support for multiple processor core SoCs are summarised and discussed. Recommendations are made for SoC designers and for the future direction of research in this area, with the aim of providing a more suitable foundation for new development tools. Such tools are badly needed for all hard real-time embedded systems and are paramount to managing the development complexity introduced by SoC devices with multiple highly interactive processor cores and active peripherals.

1 Introduction

Advanced mechanical systems and multimedia applications are increasingly required to provide hard real-time performance while maintaining low power. These requirements are leading to SoC solutions with multiple processor cores and active peripherals. Systems are further constrained by requirements to operate in harsh environments such as within a vehicle's engine bay or even its gearbox. Development of embedded computer systems, where missed deadlines can result in physical damage to the mechanics, is an aggressive challenge. Efficient tools such as debuggers and profilers are an essential part of overcoming these challenges and are vital in the development of dependable embedded systems.

Advancements in technology now permit the integration of an entire system onto a single silicon chip, known as a system-on-chip (SoC), which results in the existing external interfaces used extensively for development purposes being moved on-chip. Traditionally, the communications within an embedded system take place using the processor's external system bus that is realised as tracks on a printed circuit board (PCB). The PCB supports convenient attachment points for the development tools that require a physical connection, such as logic analysers and storage oscilloscopes. The placement of the previously external interfaces on-chip leaves increasingly few points of attachment for external analysis tools, rendering developers 'blind' to the SoC internal state. Without a reliable and consistent view of an embedded system's state, defects or bugs in the system can become difficult or even impossible to find. The primary solution to this lack of external interfaces is to make the internal nodes observable again from outside the system. There are many different approaches to achieving the required visibility; this review outlines them.

Debug support can be defined as the strategy of placing access points within a system such as in Fig. 1, so that its internal nodes become observable and controllable from outside with the intention of improving the overall system development process. Application software receives special focus, as it contains much of the complexity and may differ between system designs that contain the SoC. The infrastructure provided by debug support is a foundation for development tools and activities such as debuggers, profilers and calibration.

Debug support should not be confused with other design and development activities such as hardware/software co-design [1, 2] modelling [3], simulation [4], formal verification [5] and design for testability (DFT) [6–8]. These related activities are important parts of the development process and may include the debugging of code [9]. With the exception of DFT, they occur mainly at the conceptual design or modelling stages. DFT is the process of inserting infrastructure so that the SoC structure and containing PCB can be tested, normally in a dedicated structural testing mode where the system is not active. To test blocks of combinational logic, the connected flip-flops (FFs) are used to assert the inputs with test patterns and provide monitoring of the outputs under the test stimuli [7]. Testing can be externally driven or built-in-self-test (BIST)-based, but is often a combination of techniques [6, 7]. Some BIST and self-checking

IEE Proceedings online no. 20050194

doi:10.1049/ip-cdt:20050194

Paper first received 18th November 2005 and in revised form 15th March 2006

The authors are with the Department of Computer Science, University of Essex, Colchester CO4 3SQ, UK

E-mail: ahopkins@essex.ac.uk

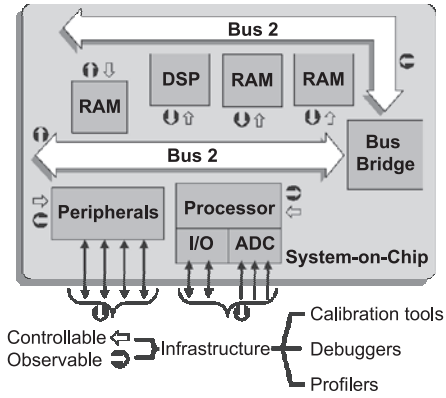

Fig. 1 *Debug support for SoC*

techniques can operate in parallel to the system, highlighting the close divide between functional testing and system level debugging. Furthermore, the test infrastructure supports debugging of SoC devices by allowing diagnosis of design errors such as misconnected wiring and layouts leading to poor manufacturing yield.

In contrast, debug support concerns the removal of defects and bugs from overall systems, including software, while operating in their normal functional mode as prototype or production units. Debug support is a vital part of the embedded system's development flow, as no matter how much care is taken at the design or verification stages, bugs are frequently introduced, which can only be found when the system is operational. Sources of bugs include incorrectly defined or misunderstood specifications, late hardware changes at the manufacturing stage and, the most common source of all, human error.

The importance of good debug support and DFT support is demonstrated by the allocation of approximately half the total system development effort to verification tasks after first silicon leaves the foundry [10]. These verification tasks not only include the test and debug of the silicon when it leaves the foundry, but also the debugging of the entire system as a blend of hardware and software. Furthermore, it has been estimated by the US National Institute of Standards and Technology that bugs and glitches cost the US economy alone approximately $59.9 billion a year [11]. Maturing a product through increased testing and verification can reduce this cost by up to one-third. It has been reported that 77% of electronic failures in automobiles were software-related [12]. The challenges of debugging the system will only increase as technology advances, whereas silicon debug is already one of the fastest growing costs per unit [6].

SoC devices are often associated with high-tech devices such as mobile phones and set-top boxes; however, it is at the heart of hard real-time systems where they prevail. Such systems are some of the most difficult to realise, as they generally control mechanical parts in addition to the challenges of achieving high device utilisation and short time to market. Current design approaches for hard real-time systems rely on deterministic task execution times; even a small change in timing can adversely impact mechanical parts. If an engine controller misses a deadline or triggers a pulse too early, for example, then the engine can be catastrophically damaged. Non-determinism in hard real-time systems is so problematic that many system integrators do not even use performance enhancing cache memory and would seldom use a real-time operating system [13].

2 Overview of debug support strategies

The fundamental requirements of effective debug support are as follows.

- The debug support should not significantly change the device behaviour.
- Infrastructure for external observation of the internal system state and other critical nodes.
- External access to control the system state and resources including complex peripherals.
- Limited cost impact on the SoC in terms of device pins or chip area.

Design re-use is critical in SoC designs for maintaining good time to market, so the debug support must be easily adaptable to different system architectures. Debug support architectures for SoC have three stylistic classifications as defined by Huang and Kao [14], shown in Fig. 2. Further to these styles, prior art typically uses one of four debug support infrastructure types, as outlined in Fig. 2.

There are also different debugging techniques that are normally supported by the debug infrastructure. Independent of the chosen debug support strategy, the underlying architecture usually contains a number of core parts. The following sections overview these parts which loosely fit within the abstract system shown in Fig. 3.

One technique is postmortem debugging that stops an SoC in response to an event such as access to a program or data location. In most systems, bugs take time to manifest themselves, they may leave 'evidence' when the system is halted, making the cause straightforward to find. Unfortunately, a small proportion of errors or bugs are long gone by the time the SoC halts, making them tricky to find and costly in development effort. Postmortem debugging is a traditional way of debugging embedded systems with a single processor [15], but has severe limitations. An alternative approach is to observe the system while it runs. The four debug infrastructure types are now overviewed; Table 1 provides a summary.

2.1 Software-based debug support with basic hardware support

Debug support resources can be software or hardware-based. Software-based debug support is added using monitor routines [7, 16] which is where software compiled or assembled with the program code instruments a processor based system. Software instrumentation can be platform-independent and is often supported in processors by debug support instructions, a debug mode or interrupt handler [17]. An example of software instrumentation is using 'print' procedures to display or log some of the system's state variables or its progress through the program code. Monitor programs provide a limited set of access points that must be memory-mapped resources visible by the processor.

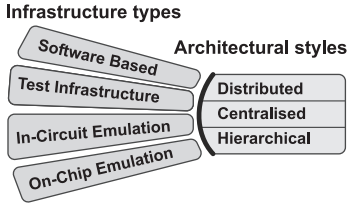

Fig. 2 *Stylistic classifications*

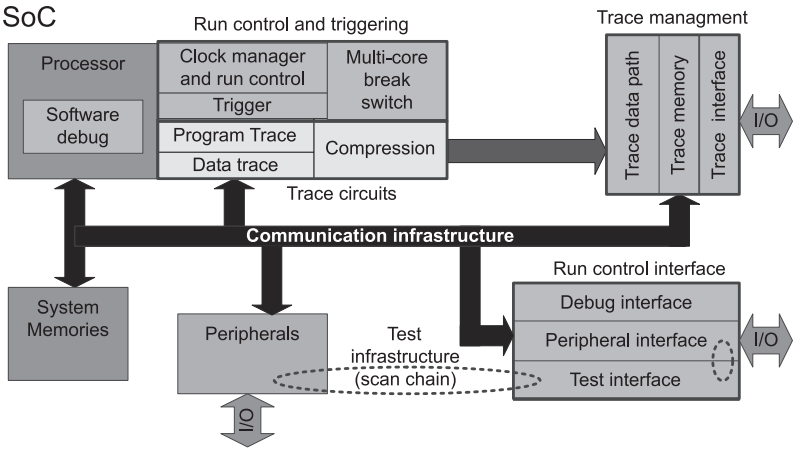

Fig. 3 *Parts of the abstract system*

Software debug support is almost always supplemented by basic hardware debug [17], including both traditional bench equipment and basic on-chip debug support. Bench equipment includes logic analysers and storage oscilloscopes, triggered by internal detection units that observe the input signals or by on-chip debug support resources that signal events via a dedicated break output pin. Basic on-chip debug support provides facilities for remotely located development tools to run and control an embedded processor's program via a development interface known as a 'run-control' interface. Basic on-chip debug support provides a minimal interface to control the SoC. Software support can be active at runtime but will be in contention with the system tasks, substantially impacting the system's behaviour. Once halted, monitor routines provide a reliable means of observation and control. The process of stopping and starting the SoC, however, is not always practical, especially for real-time systems. Stopping and resuming usually changes the relationship between the system and the environment. When the SoC is stopped, the environment continues unless being simulated as in the case of a test bench.

The intrusive nature of software instrumentation makes it unsuitable for hard real-time systems when used with the 'real' production mechanics. Any instrumentation included during debugging would be removed in the product, thereby changing the overall behaviour [18]. A second disadvantage of this strategy is that it only supports 'postmortem' debugging that provides observation once the errors are visible, not when they actually occur. Software instrumentation is, however, very useful in some applications and is much safer to use in systems with soft deadlines, such as those not controlling the actual mechanics, particularly if the system is not operating at maximum capacity. Furthermore, a recent study has shown that for a hard real-time system designed to withstand the non-determinism of using cached memory, carefully optimised light weight software instrumentation only slightly reduces determinism when profiling high-level tasks [19].

2.2 Debugging using device test features

It is now a common practice to include DFT within most integrated circuits to support manufacturing test and silicon debug [7, 20]. They are usually scan chain-based and may conform to a relevant standard discussed later. The scan chains pass through many of SoC's critical data and control paths by replacing the FFs with scan-enabled versions, allowing read-out of the flip-flops' state. In combination, scanned data, software-based debug support and run-control features provide a simple observation and control tool. The primary advantage of a scan chain-based solution is that it re-uses the DFT infrastructure, so does not increase manufacturing costs.

Table 1: Summary of debug infrastructure types

Type	Software with	Test feature based		In-circuit	On-chip
Application	basic hardware	Basic	Improved	emulation	emulation
Postmortem support	low	low	high	very high	very high
Runtime support	low	none	medium	high	very high
High internal frequency compatible	yes	–	–	no	yes
Observation point coverage	medium	very high	very high	medium	high
Hard real-time support	none	none	none	medium	high
Support for debugging in application	high	low	low	very low	very high
Production SoC	low	none	high	verylow	medium
Behavioural impact	very high	medium	medium	high	low
Modifiable by developer	yes	no	no	no	no
Suitable applications	initial stages of development and optimised profiling	initial stages of development		excellent for very high volume low frequency SoCs	excellent for hard real-time and reduce volume. Can be used in most applications

Conventional scan chains use a single scan path to reduce routing overhead, but this limits the performance. A further drawback is that they are only normally accessible in testing mode, as the scan chains are multiplexed to device pins used for another purpose in functional mode. Using a dedicated test interface overcomes this. In practice, manufacturing test overcomes the performance limitations by multiplexing multiple scan chains onto spare functional pins [21]; however, there are no spare pins in functional mode, so this is unsuitable for debug support. One solution is to introduce a further debugging mode where the scan chains are concatenated together [22] or multiplexed [23], so that they are accessible via a dedicated single scan line interface.

Operating scan chains cause the data held by the FFs to move in the chain, potentially changing the output of the FFs. Using scan chains while the system is inactive avoids state corruption but prevents system debugging. Scan data corruption can be partly overcome by replicating the normal FFs with shadow scan FFs that can capture their state. Such an approach allows read-out of the state at a specific point in time, providing key information at runtime, without corruption. This technique was used successfully in the UltraSPARC^{TM}-III microprocessor for access to parts of the pipeline, including the program counter [20], and is useful to help debug fatal hardware errors, such as queue counter overflow. Supplementing the shadow latches with on-chip trigger circuits that initiate state capture helps overcome some of the latency and runtime problems of scan chain-based control, as once the data is captured, the data can be read out at a convenient time or offline. Such an approach builds on 'postmortem' debugging to provide very limited runtime debug, which is unsuitable for real-time operation. For a large SoC, the time taken to read out the data will consume development time, and the cost of replicating many FFs with shadow latches can be prohibitive.

Another potential source of corruption is data invalidation where signals cross clock domains [24]. Although clock management circuitry allows reliable stopping of a clock domain, it does not maintain the relationship between different clock domains, creating the potential for one clock domain to capture old data from another clock domain that has already stopped. This signal slippage is called data invalidation and is liable to occur when the receiving domain is higher in frequency than the sender, as the sending domain can stop before the receiving domain has made its final clock tick. Detection circuits to help overcome data invalidation during debugging have been designed to increase the integrity of captured data [24]. Crucially, restarting still causes a slight change in behaviour as the alignment between the unsynchronised clocks is changed.

2.3 In-circuit emulation devices

An in-circuit emulator (ICE) is a specially produced development-specific part that replaces the normal production SoC. An ICE helps overcome the limitations of test infrastructure-based debug or basic hardware and software instrumentation debug by providing additional connectivity through an increased footprint. The many additional package pins provided by the expanded footprint are not used by the system in functional mode; instead, they are connected to internal signals to allow observation using a logic analyser. This type of ICE is often called a 'bond-out' chip, as they have additional wire bonds connecting the chip to its package. The extra data provide substantially increased observation over software instrumentation and can be from points not visible to a processor core. Furthermore, the information is potentially available while the program runs normally, so there is in principle no impact on SoC behaviour.

The disadvantages of 'bond-out' chips are that some of the extra connections do not come from conventional bonding pads in the chip's periphery. Connections are often made to other regions on the top metal layer, sometimes using abnormally long bonding wires. This leads to a larger and more fragile device, unsuitable for harsh automotive environments, such as an engine test bay or within a gearbox. Furthermore, driving the extra bond wires loads the internal signals, compromising the performance and changing the development part's behaviour compared with that of the production part, and most likely introducing extra bugs. Modern flip-chip packaging may help eliminate some of the fragility, and optimisation may overcome the negative impact of driving the bond-out lines, but in practice the external pins still limit the performance.

Internal operating frequencies now exceed the frequency of conventional external pins for harsh environments [10], creating a mismatch in performance, which prevents the most interesting internal signals from being observed in real time. Reducing the ICE frequency overcomes this mismatch, but introduces a behavioural mismatch between ICE and production parts. Provided internal frequencies are $\leq$80 MHz, full speed operation is maintained making bond-out ICEs an effective debugging solution.

Changing the footprint requires two different PCB designs, consuming the development resources and changing the behaviour such that bugs are introduced to the development design but not to the production design. Furthermore, when the debug strategy focuses on an enhanced development part, debugging in the final product becomes a major challenge. Rising gate counts also act to make 'bond-out' chips undesirable, as there are too many critical observation points to directly connect the raw signals to external pins. The rising cost of mask sets and design effort for state-of-the-art integration processes has now made custom bond-out ICEs prohibitively expensive.

2.4 On-chip emulation-based debug support

An increasingly accepted alternative to bond-out ICEs is to make the SoC emulate itself by extending on-chip emulation beyond basic run-control features to create extensive data capture tools within the SoC itself [25]. Such capture tools typically include trigger logic for breakpoints that halt the program when a predefined instruction address or other condition is reached and watch-points that operate on the data address. Hardware resources operate in parallel to the normal SoC functionality; therefore the system behaviour is almost unaltered. Consistent system behaviour makes self-emulation-based debugging strategies suitable for most applications, including hard real-time systems development.

On-chip emulation techniques scale well with technology, as increased integration supports a rise in infrastructure gates comparable to that of system gates. Larger gate counts enable features like tracing, which provide developers with sufficient information about the runtime activity to analyse the behaviour offline, when the system is safely shutdown. A further advantage of tracing is its ability to capture the events building up to a problem rather than just its effects. Maintaining the correct temporal order of events is also

essential in providing the developer with reliable information and provides scope for further research. On-chip tracing differs significantly from bond-out ICEs that only provide raw signals. On-chip tracing adds the repackaging and compression necessary to observe multiple high speed embedded processors and data access.

2.5 Debug infrastructure summary

As integrated circuits have evolved so has debug support; each of the four styles overviewed has advantages and disadvantages as summarised in Table 1. The current trend is to heavily rely upon tracing for the hard to find sporadic bugs, as they can be impossible to find with postmortem techniques. Profiling and optimising real-time systems is increasingly reliant on hardware tracing.

3 Control of SoC with multiple processor cores

Complex multiple processor core SoC devices combined with pressure for increased design re-use presents new challenges for SoC creators. SoC is not simply the integration of the circuits once contained on the PCB, but a new design approach, requiring evolution of design practices.

3.1 Run-control and memory access interfaces

The traditional SoC development interface is the run-control interface. It is used to control the operation of a microcontroller or single processor core SoC, specifically it allows the developer to run, halt and step units like processors. Some SoC devices use a proprietary interface [26]; however, a widely adopted solution is to re-use the existing boundary scan testing port, although there are several approaches. The industry standard boundary scan testing port as defined by the IEEE Joint Testing Action Group (JTAG) 1149.1 standard [27] has up to five pins. Re-using a widely adopted and standardised testing interface for debug has the advantages that it is well known and requires no extra device pins, giving this approach the potential for adoption across competing platforms. One advantage is that the JTAG Test Access Port (TAP) includes an externally driven clock that allows construction of low cost but low performance tools based around a workstation's parallel port or an existing Universal Serial Bus (USB) chipset.

Systems with multiple processors on a PCB have previously included multiple debug interfaces, often based on IEEE JTAG 1149.1, that was originally intended for structural testing of digital PCBs. Early SoC devices with multiple processor cores were constructed by integrating most of the once separate components from a system PCB onto a single chip. Many of the cores were hard macros with embedded TAP controllers, raising the issue of how to manage a plurality of TAP controllers while maintaining compliance with the IEEE 1149.1. One specific challenge is maintaining a single FF in the devices' 'bypass' mode register, which is a requirement of the standard, even though most tools will work with deeper registers. Fig. 4 shows six solutions as overviewed below: there are three distributed solutions *a*–*c*, two hierarchical solutions, *d* and *e*, and one hybrid solution, *f*. Sub-sections (*g*)–(*i*) discuss other solutions.

(*a*) Make no changes, directly re-use each core, connecting its TAP controller to a dedicated set of device pins.

+, straightforward connection to each core's development tools;
+, scalable bandwidth;
−, replicating interface requires many device pins.

(*b*) Connect the TAP controllers in a chain, just as on a PCB [27].

−, approach logically unchanged;
−, non-standard, multiple FFs in SoC bypass register;
+, problems resolvable by changing standard to allow virtual components.

(*c*) Select TAP using multiplexers and external select input [14].

+, compatible with JTAG 1149.1;
−, requires extra device pins;
−, incompatible with existing development tools.

(*d*) Use TAP linking module [28] to interface embedded TAP controllers.

+, compatible with JTAG 1149.1;
−, requires modification to the TAP controllers, which is not possible for hard cores;
−, not directly compatible with debugging tools.

(*e*) Several other designs centre on a chip level TAP for conventional boundary scan testing which then provides access to the embedded TAP controllers during debugging [25, 29, 30]. This approach avoids many complications of approaches (*a*)–(*d*).

+, compatible with JTAG 1149.1;
+, no extra device pins;
+, no need to modify core level TAP;
−, could scale better, not system-centric.

(*f*) *Agent-based approach*. The next evolutionary step is to adopt a more system-centric approach, where each SoC design aims for a single TAP [31]. Existing hard cores cannot be changed, so a good solution must support these legacy cores until they are phased out. A single chip level TAP controller that communicates with each core's generic debug support provides a more scalable architecture.

One specific implementation called JTAG+, as shown in Fig. 4*f*, combines aspects of both central and distributed architectures. It uses a central TAP controller and scan chains to communicate with a single active debug agent core, which is a master on the system interconnect/bus [25]. It uses the system interconnects to access memory-mapped resources from any number of processors, including each core's debug resources. This differs significantly from solutions like EJTAG [32] where the debug core is tied to each processor core and accessed using scan chains. Legacy hard cores are supported by connecting the scan chains to the central TAP controller that provides selection between the embedded TAP controllers and the debug agent in a similar way to conventional scan-based approaches. In some applications, mixing debug and system traffic on the bus may unduly impact system determinism, but is avoided by using a dedicated bus. The agent-based approach's pros and cons are summarised below.

+, compatible with JTAG 1149.1;
+, no extra device pins;
+, no need to modify core level TAP;
+, scales well, system infrastructure-centric;
+, compatible with past and present solutions;
−, may impact determinism without dedicated bus.

(*g*) *NEXUS (IEEE-ISTO 5001-2001 std.)*. Increased pressures to reduce form factor and cost while increasing performance now drive a growing trend towards debug focused interfaces. The NEXUS 5001 forum has defined

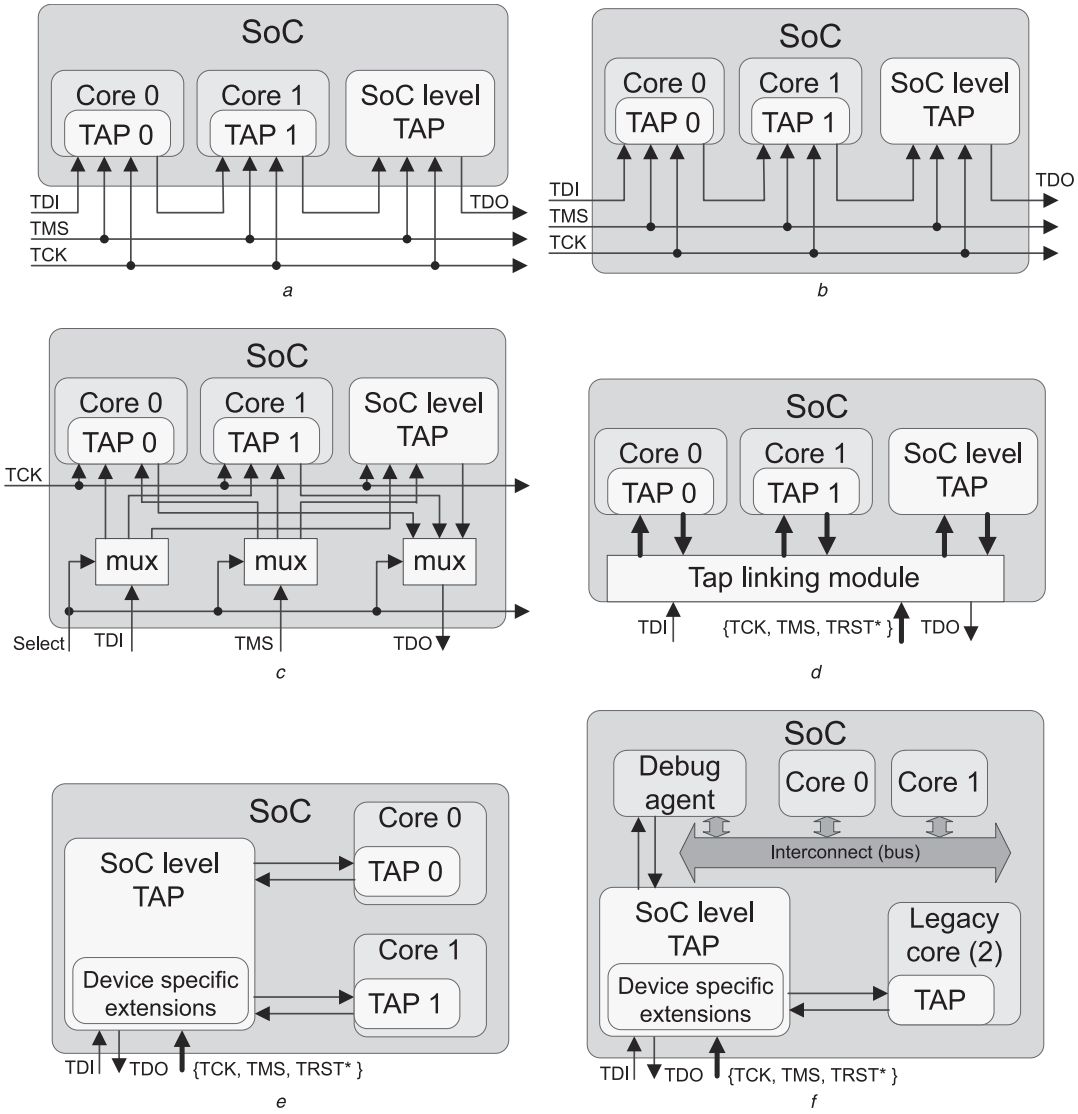

Fig. 4 *Different solutions*

a–c Distributed solutions
d, *e* Hierarchical solutions
f Hybrid solution

and standardised a debug support interface for embedded processors targeting control applications [33], where the challenges are highest. The NEXUS standard described later in more detail specifies multiple classifications of debug support. They range from an IEEE 1149.1-based solution to more comprehensive solutions that centre on the scalable NEXUS auxiliary interface. Fig. 5 shows the JTAG and NEXUS auxiliary port interfaces. Options include free-running clock inputs for input and output data paths, a reset input, an event output signal and message synchronisation bits. Like most system-centric approaches, multiple external tools are supported through an external arbiter that keeps some complexity outside the SoC, conserving production device resources. NEXUS is characterised by:

+, design time options including bandwidth;
+, developer configurable, best compromise between bandwidth and pins;
+, scalable, system centric with single interface;
−, messaging protocol fairly complicated.

(*h*) *Low pin count interfaces*. In the pursuit of further pin reductions, some IP creators now offer run-control interfaces that provide comparable functionality to a JTAG interface, but with only a single pin [34]. Some interfaces use a single bi-directional data pin with a clock encoding scheme [35]. Others maintain compatibility with existing JTAG-based tools by using external hardware to convert between the single pin interface and the JTAG interface [34]. Single pin interfaces appear to be lower in bandwidth than for a conventional JTAG-based solution, as all five JTAG pins are multiplexed onto a single pin, but it is claimed that little bandwidth is lost compared with standard JTAG, because of a more optimised coding scheme [34]. In most cases, slow pin count interfaces are a trade-off between performance and other factors such as size and cost. Their other characteristics include:

+, ultra small form factor;
+, can be made compatible with JTAG;
−, low performance;
+, suitable for mobile/ubiquitous computing applications.

(*i*) *Peripheral interfaces*. One further approach is to use a system interface or dedicated widely available peripheral

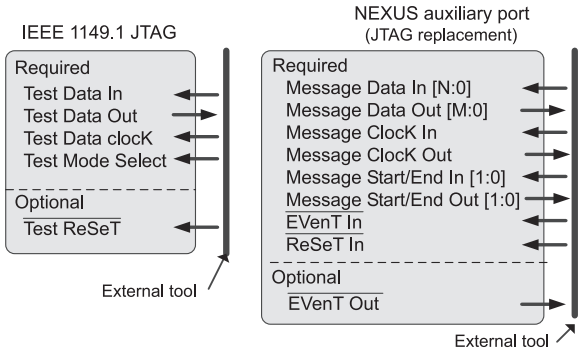

Fig. 5 *JTAG and NEXUS auxiliary port interfaces*

interface such as controller area network (CAN), USB or even Bluetooth. Using a system peripheral requires no extra pins, but greatly changes the system behaviour for two reasons: (1) it consumes interface bandwidth and (2) it consumes processing time, as the peripherals have low-level driver routines. The high level of operation also makes debugging low-level hardware and software interactions impractical, unless dedicated peripheral and processing resources are used. For this approach, the challenge is overcoming or avoiding the cost of these additional resources. This approach is characterised by the following points:

+, Low pin cost;

−, requires costly dedicated resources to avoid impacting the system;

−, high latency without dedicated event pin;

−, suitable for high level debugging, including profiling and calibration;

3.2 Triggering SoC devices with multiple cores and clocks

In most modern SoC devices, breakpoints are supported by hardware triggers; they also support data access watch-points, allowing control and capture of data transfers. In systems with tracing, triggers can also be used to start and stop a trace and filter the messages to conserve trace bandwidth and storage. This filtering process is known as trace qualification, because the most interesting messages are identified so that they remain in the trace. Trigger implementations often have several compare units that are configurable in a group. This allows detection of when a bussed signal's value is equal to a preloaded compare register or when its value is between that of two compare registers. Triggers that combine the outputs of several detection units are known as cross-triggers. They are extremely useful, especially in multitasking systems where it is desirable not only to qualify a trace down to a specific program address space, but also to specific tasks accessing that region. Counters support detection of basic sequences, but a state machine known as a sequencer is better suited to more complex sequences [18]. Sequencers are similar to logic analyser triggers. One of the challenges with debug support is balancing the resources between triggers and trace infrastructure. Adding more triggers can help minimise the trace size, but requires more on-chip resources. The best compromise depends on the application, as does the way triggers are configured. Further research focusing on reconfigurable approaches for multicore cross-triggering will lead to improved solutions.

Most complex SoC devices now have multiple clock domains that add the complexity of synchronising the inter-domain signals; however, this also impacts debugging, as stopping a running system is no longer straightforward. With a single clock, the whole SoC is stoppable at once; but with different or unsynchronised clocks, there is potential for slippage between the domains causing data invalidation [24]. Although data invalidation can be detected, the problem of restarting the SoC remains. It is still useful to suddenly stop and step an SoC, provided it is not controlling mechanics. To avoid changing the behaviour, all units should ideally be stopped at the same instant in time and restarted with the exact clock phase separations as when stopped. Such precise clock control is unrealistic; chip wide break events are further limited by signal skew that is a major factor in highly integrated processes where wire delays are significant [10]. A practical solution is to switch trigger signals from a small number of closely located units and then link them to build a larger structure when required [36]. An alternative and more distributed solution is to build a distributed switch in the form of multipath break and suspend busses [37]. This has the advantage that modules connect to a generic unit, acting as a central hub to deterministically route break and suspend signals with minimal skew. Using a switch or bus allows the developer to configure which cores will respond to break and suspend signals. Centralised configuration also leads to a more area-efficient implementation requiring less configuration accesses. Eventually, interconnect delays will motivate a more hierarchical approach, with regionalised multicore break and suspend units. Reliable cross-triggers between specific signals will also only be practical for small groups of close-coupled cores. Low-swing differential drivers have been proposed for network-on-chip (NoC) to help overcome wire delay, and could be applied to debug triggers. It has been suggested that an NoC could transfer interrupts and therefore trigger signals directly with little loss in performance [38]. Current trigger solutions are adequate, however, further innovation is required for future SoC devices with more than four processor cores.

4 Support for tracing

The debug support features reviewed so far are primarily concerned with the control and observation of the system state at single points in time. To help find the challenging bugs, developers need to view parts of the system state in the build up to an event. To gain accesses to this information, the SoC can internally capture and compress the state data in real time, to form a trace. The trace is then made available to the external development tools either via a trace port [16, 32, 33, 35, 37, 39] or by storing it in an embedded trace memory that can be read offline using the run-control interface [32, 35]. Tracing is not just useful for debugging single core systems, it is essential for debugging the interactions between multiple processor cores and active peripherals.

Conventional traces just log the program flow for a processor core [39] listing the type of instructions executed in their order of occurrence. Data access trace is also possible [33], which helps to find the bugs relating to shared variables, locks and other interactions where event ordering is important. Data tracing of all active units is becoming increasingly important, as not all interactions involve a processor [18]. A basic program trace may contain only sufficient information to reconstruct the program flow offline using the assembly and program code. This can be supplemented by details of the active context, which is the running task or interrupt. Some solutions for special applications like space may include far more detail [40].

A typical data trace details each access by recording information such as address, data value, transfer size and whether it is a read or write. Peripheral units may also provide status information such as interface bandwidth or communication error statistics, but this is currently not conventional. Status information not requiring real-time recording is normally read from memory-mapped registers using the run-control interface.

4.1 Trace compression

Conventional processors are sequential machines and are most efficient when executing programs that have only a small proportion of control instructions. Some program-tracing strategies generate a fixed width compressed message for every processing cycle [17, 39], which is straightforward to implement, but can lack flexibility and may not provide a high level of compression. An alternative strategy as defined by the NEXUS standard [33] and used by some SoC vendors [16] is to have messages that vary in length and only send messages when necessary. Using different lengths of messages can lead to better compression with certain algorithms, as the most likely messages can be kept short at the expense of making the least likely longer. This approach is well known and is used by some generic data compression algorithms like Huffman codes [41]. NEXUS adopts a slight variation where messages are only sent for events requiring synchronisation like branches or traps. Instead of dedicated messages for sequential instructions, the number of sequential instructions executed since the previous message is counted and included with the subsequent message. This reduces the trace size, but at the expense of a further increase in complexity, which consumes on-chip resources.

One widely adopted algorithm for program and data access trace is differential compression [32], as it takes advantage of the locality of reference between data successive addresses. When stripped of leading repeated bits, the difference between successive data addresses is generally smaller than the full value. However, this is seldom the case for data values [42], which can vary significantly between adjacent locations; structured data-like floating point numbers further antagonise the problem.

The most efficient method of reducing the trace volume is to avoid generating and recording it, which makes trace qualification extremely important as it helps reduce traces down to only the required information. In its most basic form, trace qualification allows developers to turn off the trace from certain units or disable parts of the trace such as program or data tracing. Qualifying data traces down to only reads or writes is also a good way of reducing trace volume without increasing complexity. An improvement is to include trigger and detection units, as discussed previously, to provide a finer grain of qualification such as an address range [43] or specific task. Task/context qualified tracing is especially important in multiprogramming systems, like those with an operating system; it is also very useful in hard real-time systems, as it identifies interrupts and supports profiling. Symmetric multiprocessing clusters are starting to appear within SoC devices, introducing the possibility that tasks are not processor-resident. A universal approach would allow context identification no matter which processor a task runs on.

Context-based trigger and trace requires the processor to have a context identification register that is absent on most processor designs, even those targeted at hard real-time automotive systems [37, 39]. Close integration with the scheduler is also essential, as it forms part of the multiprogramming mechanism. Without these features, debugging in the presence of an operating system becomes very challenging. Interrupt-driven systems are slightly less dependant on these features, as most interrupts start from the same instruction address. Therefore detecting an interrupt and sending its full start address provides implicit identification, which is a useful half-way solution.

4.2 Trace management

The generation of trace messages is not the only factor when developing trace systems for complex SoC devices, management and storage is also important. Traces are managed either as individual source-centric traces or as a combined system-centric trace. The trace can be controlled manually via the external interface; however, control using triggers is more effective, as they allow developers to focus on the interactions before or after an event [32, 43]. Tracing 'from' an event is useful for capturing behaviour when the developer suspects part of the system, whereas tracing 'to' a trigger event is useful when the developer is unsure about the cause of a bug and wants to see the behaviour building up to an error's measurable effect. The most basic tracing approach for multiple cores is where the trace infrastructure is switched between each trace source [37], but this approach is not system-centric and is severely limited where cores interact.

A true system-centric combined trace must provide a coherent view of the system's interactions. The method for combining trace messages from each source is an important factor in maximising the trace's effectiveness. A shared trace alone provides an economy of scale, as separate traces replicate much of the infrastructure. Also, one large single on-chip trace memory is denser than two small memories. When the trace messages are combined in their correct temporal order, the developer is provided with a trustworthy 'picture' of what is happening within the SoC as a whole. Such a trace allows the developer to observe concurrency-related bugs, including accesses to shared variables. However, in SoC devices with heterogeneous processor cores, there is often no choice but to include multiple separate trace systems, as each processor family is bound to a specific debug implementation. Introduction of a generic standardised debug interface would resolve this system integration problem. An effective solution that combines trace from multiple sources in correct temporal order is currently the subject of active research, although the next step would be to combine system and debug traffic using an NoC [44]. Debug trace data are not depended on by the system; its protocol can be designed to maintain the correct message order in the presence of high latency and jitter, making it well suited as background traffic.

To support clear task identification, NEXUS class II and above define messages to identify both trace sources and any component trace threads [33]. Provided tasks are identified uniquely across multiple processors, a NEXUS trace can even manage tasks that are not locked to specific processors.

4.3 Trace interfaces

It is highly desirable to trace as much data as possible, but this is limited by bandwidth and storage. The trace can be stored externally via a trace port interface that provides ample storage capacity, but has limited bandwidth for real-time trace extraction. Such a solution is ideal for single processor systems with an internal clock frequency close to the

maximum external pin frequency, as the bandwidth requirement is not too high for a port. Some solutions use a proprietary trace port, [17, 32, 39, 43] whereas others use a standardised interface, such as the NEXUS auxiliary port [33].

An alternative approach is to store the trace in a dedicated or shared on-chip memory, supporting a higher bandwidth than an external interface. This allows sudden burst transfers from multiple sources, but the amount of storage must be kept small to avoid consuming production chip resources. As transistor geometries shrink, large-scale on-chip memories become increasingly feasible for tracing and calibration, as the required chip area reduces, making them economically acceptable, as demonstrated by increased commercial use [16].

The duration of a trace is directly proportional to the capacity of the trace memories, for example, if a processor generates 1 B of trace data per cycle then a 4 kbit memory will hold only 512 cycles of trace. For a single processor running at 400 MHz, this is only sufficient storage for a 1.28 μs long trace. The duration would be further reduced when tracing multiple cores. Restricted trace duration highlights the importance of an effective trigger solution, trace compression and qualification algorithms; as SoC devices become more integrated, these features will have to be further developed to keep pace. Combining on-chip buffers with an external port helps extend the buffer capacity, but will become less effective as the mismatch between internal frequency and pin frequency extends.

Both trace ports and on-chip trace memories are expensive to implement and have restrictions that compromise development, especially when searching for sporadic faults. For low-volume devices, it is practical to integrate extensive debug support resources, because the increase in manufacturing cost is small compared with the development time saved. For mass manufactured chips, the cost of development is comparatively small to the total increase in device cost.

Demand for lower cost production parts with comprehensive debug support has lead to the development of a new construction technique called emulation extension [45]. With emulation extension, the high-volume production SoC is manufactured with only the essential tightly coupled debug support components such as run control. A low-volume development part is then produced, which contains system circuits identical to the production SoC, but also more comprehensive debug support circuits to supplement those already on the production SoC. Furthermore, very large circuits such as trace buffers and calibration overlay memories can now be included without impacting the manufacturing cost. This removes the pressure placed on developers to use SoC devices that are compact, but have inadequate debug support for complex systems. Reducing the overall pressure allows developers to concentrate on the real challenges of reducing defects and meeting time to market demands.

One method of realising emulation extension is to flip-chip bond a second integrated circuit containing the additional development resources, including overlay memories onto the SoC, creating a multichip module [12]. As the SoC acts as the main chip, the assembly automatically provides near-perfect emulation of the production SoC, while maintaining the same footprint. It is also possible to create a carrier chip that incorporates the debug support circuits and bonding pads to attach the production SoC. Both solutions have similar advantages, but also the disadvantage of requiring a dedicated mask set for the debug support chip. The design effort and manufacturing costs can be reduced by re-using the emulation extension chip across an entire SoC family. An alternative strategy for including the emulation extension circuits for development-specific parts is the side booster concept [12].

In the side booster chip, area outside of the main pad ring is used to hold additional development circuits as shown in Fig. 6. The side booster concept has been realised for a commercial SoC, the TC1796 SoC as the TC1796ED emulation device that adds significant debug support resources [45]. The TC1796ED's side booster contains a 512 kbit static RAM, a USB controller peripheral for use as a development interface and even a processor core to service requests by the development resources or tools [45]. The side booster not only allows comprehensive debug support, but it is also economically well founded, as it avoids placing significant extra resources in the production SoC. It uses the same manufacturing process and incorporates the exact SoC layout, including its pad ring, as a hard macro, preserving the behaviour of both digital and analogue circuits. The main disadvantage of the side booster architecture is that it again requires a separate low-volume production run.

5 Debug support recommendations

For complex SoCs, developers require system-centric features which consider the multiple processor cores and active peripherals that are key parts of low power, high performance embedded systems. Deep processor pipelines, multiple clock domains, real-time demands and increased complexity have shown the limitations of traditional post-mortem debugging techniques. Debugging mainstream systems and software using silicon debug and test infrastructure, such as scan chains, to debug the overall system and software behaviour is not a viable technique. This is mainly due to the fact that the test infrastructure focuses almost exclusively on testing-related activities. Therefore research of new methods that are aligned to all aspects of debugging and testing would benefit SoCs, although the solution would have to be excellent to displace established test methodologies.

The familiar features of hardware breakpoints remain useful in some scenarios and are expected by developers. Sharing these trigger units with tracing features is an appropriate method of reducing their resource cost. Some multiple core solutions exist in industry, but there is still scope for extending breakpoints further to provide more system-centric support of multiple core devices when implemented in deep sub-micron processes. Software-based debug is well understood and will continue to be a valuable tool. However, for effectiveness, it is unwise for a complex SoC design to rely heavily on software, unless the application has a special reason for doing so. SoCs need a dedicated non-intrusive run-control interface, although for some highly resource constrained applications, debugging performance may have to be traded for

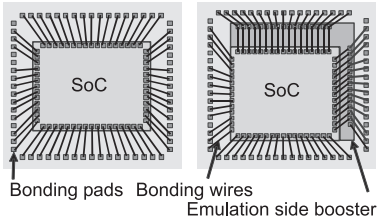

Fig. 6 *Side booster chip*

re-using the JTAG interface or similar. Once the system is stable, there is motivation to use an appropriate user interface like CAN or USB for high-level tuning and calibration, but the required functionality must be designed into the system architecture for it to work effectively. Low-level optimisation and calibration without system support is still reliant on debug support hardware. In general, full system-wide visibility of all critical nodes such as the registers and memory is required at runtime; SoC vendors should all implement this.

For real-time systems, tracing is a minimum requirement and should be boosted by strong trace qualification and hardware triggers. The rising mismatch between internal frequencies and device pin frequency is rapidly making on-chip and in-package storage the only effective solution for tracing, unless a very high-speed interface can be designed. Low production volume and the need for good electromagnetic immunity makes the trace interface a potentially good development bed for chip-level optical interconnects, once issues like temperature stability are addressed. Multicore SoCs require infrastructure to combine each core's trace, while maintaining a true temporal order of their messages with identification of the originating source. So far, only basic solutions have been proposed; hence, new methodologies need to be researched to support future generations of more complex SoCs. Area-efficient multiple source trace compression is a further aspect that could be improved significantly. Improved trace qualification is also required for future systems that will have increasingly more processor cores. One area that will provide clear benefits is a context-based trigger; however, this requires integration with the system's scheduler. Component and software designers need to understand that they have to work closely in terms of task identification to achieve practical results.

In an IP block-oriented world, it is important that the development support becomes a re-usable block of its own, detached from vendor-specific platforms and processors by standardised interfaces like those defined by previous academic work [46].

6 Conclusion

SoC has evolved to highly complex devices, but the development support necessary to make them less challenging to program has not kept pace. Rises in complexity, time to market pressure and demand for high levels of quality and dependability have made development support a decisive factor. With the right tools, these challenges can be overcome and project success achieved. For the advancement of system's integration, it is essential that SoC development support ceases to focus purely on low cost and less effective, core centric circuits. The future of debug support is without doubt with a system-centric solution. Some progress has been made, but many SoC designers are slow to adopt these new methods which the end user badly needs. This review has highlighted areas where there is opportunity for further research. It is clear that each application has its own requirements and that as technology advances so must debug support.

7 Acknowledgments

The kind assistance of the anonymous reviewers and Pieter Sartain is gratefully acknowledged. This research is funded through EPSRC grants GR/S13361/02 and EP/C005686/1.

8 References

1 De Michell, G., and Gupta, R.K.: 'Hardware/software co-design', *Proc. IEEE*, 1977, **85**, (3), pp. 349–365
2 Clement, B., Hersemeule, R., Lantreibecq, E., Ramanadin, B., Coulomb, P., and Pogodalla, F.: 'Fast prototyping: a system design flow applied to a complex system-on-chip multiprocessor design'. DAC, 21–25 June 1999, pp. 420–424
3 Hoffmann, A., Kogel, T., and Meyr, H.: 'A framework for fast hardware–software co-simulation'. DATE, 13–16 March 2001, pp. 760–764
4 Marantz, J.: 'Enhanced visibility and performance in functional verification by reconstruction'. DAC, 1998, pp. 164–169
5 Roychoudhury, A., Mitra, T., and Karri, S.R.: 'Using formal techniques to debug the AMBA system-on-chip bus protocol'. DATE, 3–7 March 2003, pp. 828–833
6 Pateras, S.: 'Embedded diagnosis IP'. DATE, 2002
7 Zorian, Y., Jan Marinissen, E., and Dey, S.: 'Testing embedded-core-based system chips', *Computer*, 1999, **32**, (6), pp. 52–60
8 Bommireddy, A., Khare, J., and Shaikh, S.: 'Test and debug of networking SoCs – a case study'. 18th IEEE VLSI Test Symp., 30 April 2000, pp. 121–126
9 Liu, J., Zhu, M., and Bian, J.: 'A debug sub-system for embedded-system co-verification'. 4th Int. Conf. on ASIC, Xue Hongxi, 23–25 October 2001, pp. 77–780
10 Semiconductor Industry Association (SIA): 'International Technology Roadmap for Semiconductors', 2003 edition, http://public.itrs.net/Files/2003ITRS/Home2003.htm.
11 US Department of Commerce 'The economic impacts of inadequate infrastructure for software testing'', Technical Report, RTI-7007.011US, National Institute of Standards and Technology (US), 2002
12 Mayer, A., and McDonald-Maier, K.D.: 'Debug support, calibration and emulation for multiple processor and powertrain control SoCs'. DATE, Munich (DE), 7–11 March 2005
13 Scottow, R.G., and McDonald-Maier, K.D.: 'Measuring determinism in real-time embedded systems using cached processors'. Proc. ESA, Las Vegas, 7–11 March 2005
14 Huang, I.-J., and Kao, C.-F.: 'Exploration of multiple ICEs for embedded microprocessor cores in an SoC chip'. 2nd Int. Asia Pacific Conf. on ASIC, 2000
15 Huang, I.-J., and Lu, T.-A.: 'ICEBERG: an embedded in-circuit emulator synthesizer for microcontrollers'. DAC, 1999, p. 580
16 Motorola Inc. (now Freescale): 'MPC565/MPC566 user's manual', MPC565UM/D REV 2, 2002, www.motorola.com/semiconductors.
17 Infineon Technologies AG: 'Tricore 1 architecture manual', ver. 1.3.3, 2002, www.infineon.com
18 Mayer, A., Siebert, H., Kolof, A., and el Baradie, S.: 'Debug support for complex system-on-chips'. Embedded Systems Conf., San Francisco, 'CMP media LLC', April 2003
19 Scottow, R.G., and McDonald-Maier, K.D.: 'How to manage determinism and caches in embedded system'. ESS, Birmingham, 2005
20 Golshan, F.: 'Test and on-line debug capabilities of IEEE Std 1149.1 in UltraSPARCTM-III microprocessor'. Proc. Int. Test Conf., October 2000, pp. 141–150
21 Vermeulen, B., and Goel, S.K.: 'Design for debug: catching design errors in digital chips', *IEEE Des. Test Comput.*, 2002, **19**, (3), pp. 37–45
22 van Rootselaar, G.J., and Vermeulen, B.: 'Silicon debug: scan chains alone are not enough'. Int. Test Conf., Atlantic City, USA, September 1999, pp. 892–902
23 Jung, D.-J., Kwak, S.-H., and Lee, M.-K.: 'Reusable embedded debugger for 32 bit RISC processor using the JTAG boundary scan architecture'. Proc. 2002 IEEE Asia-Pacific Conf. on ASIC, 6–8 August 2002, pp. 209–212
24 Goel, S.K., and Vermeulen, B.: 'Hierarchical data invalidation analysis for scan-based debug on multiple-clock system chips'. Int. Test Conf. (ITC02), Baltimore, USA, 7–10 October 2002, pp. 1103–1110
25 Maier, K.D.: 'On-chip debug support for embedded systems-on-chip'. ISCAS, Bangkok, Thailand, 25–28 May 2003, pp. 565–568
26 Melear, C.: 'Using background modes for testing, debugging and emulation of microcontrollers'. Conf. Proc. of Wescon'97, 'IEEE', 1997, pp. 90–97
27 IEEE JTAG 1149.1-2001 Std.: 'IEEE standard test access port and boundary-scan architecture', IEEE Computer Society, 2001
28 Whetsel, L.: 'An IEEE 1149.1 based test access architecture for ICs with embedded cores'. Proc. Int. Test Conf., 1997, pp. 69–78
29 Vermeulen, B., Waayers, T., and Bakker, S.: 'IEEE 1149.1-compliant access architecture for multiple core debug on digital system chips'. Int. Test Conf., 7–10 October 2002, pp. 55–63

30 Oakland, S.F.: 'Position statement: TAPs all over my chips'. IEEE Int. Test Conf., 7–10 October 2002, p. 1192
31 McLaurin, T.L.: 'Position statement: TAPs all over my chips'. Int. Test Conf., 7–10 October 2002, p. 1193
32 MIPS Technologies: 'EJTAG trace control block specification', MD00148 Rev. 1.04, 2002, www.mips.com
33 IEEE-ISTO 5001™-1999 Std. 'Standard for a global embedded processor debug interface', The Nexus 5001 Forum™, 1999, http://www.nexus5001.org.
34 Debug Innovations: 'J-Link system overview', 2004, http://www.debuginnovations.com/
35 ARM: 'How CoreSight technology gets higher performance, more reliable product to market quicker', 2004, www.arm.com
36 ARM: 'Embedded cross trigger technical reference manual', issue A, revision r0p0, 2003, www.arm.com.
37 Infineon Technologies AG: 'Infineon TC1920 system units user's guide v1.2', 2003, www.infineon.com.
38 Dally, W.J., and Towles, B.: 'Route packets, not wires: on-chip interconnection networks'. DAC, Las Vegas, USA, 18–22 June 2001
39 Renesas Technology Corp. 'Hitachi SuperH RISC engine SH7144 Series hardware manual', ADE-602-254A, Rev. 2.0, 2002
40 Gaisler Research: 'GRLib', 2004, http://www.gaisler.com
41 Huffman, D.A.: 'A method for the construction of minimum redundancy codes', *Proc. IRE*, 1952, **40**, pp. 1098–1101
42 Hopkins, A.B.T., and McDonald-Maier, K.D.: 'Generic data trace unit and trace compression for system-on-chip'. IEE/ACM postgraduate Seminar on SoC Design, Test and Technology, 15 September 2004
43 ARM: 'Embedded trace macrocell architecture specification', ARM IHI 0014H, 2002, www.arm.com.
44 Benini, L., and De Micheli, G.: 'Networks on chips: a new SoC paradigm', *Computer*, 2002, **35**, pp. 70–78
45 Mayer, A., Siebert, H., Leteinturier, P., and Qual, A.: 'Embedded system tool to support debugging, calibration, fast prototyping and emulation'. SAE World Congress and Exhibition, 'SAE International', March 2004
46 Hopkins, A.B.T., and McDonald-Maier, K.D.: 'Debug support strategy for systems-on-chips with multiple processor cores', *IEEE Trans. Comput.*, 2006, **55**, (2), pp. 174–184