

DesignCon 2004

Multi-Core Embedded Debug for Structured ASIC Systems

Dr. Neal Stollon, First Silicon Solutions
neals@fs2.com

Rick Leatherman, First Silicon Solutions
rickl@fs2.com

Bruce Ableidinger, First Silicon Solutions
brucea@fs2.com

Ernie Edgar, First Silicon Solutions
ernie@fs2.com

4000 SW Kruse Way Place, Bldg. 3, Suite 210
Lake Oswego, OR 97035
(503) 489-0311 x103
Fax (503) 489-0315

Abstract

Providing in depth analysis and debug of applications that utilize multiple embedded processors and buses is a critical piece of system on a chip (SoC) design processes. While debug of a single embedded core is challenging, it is a relatively well understood problem, whereas MultiCore architectures add new considerations and complexity that must be factored into the debug solution. As core implementations become more complex, they expand both in subsystem functionality (as an example, moving from a processor core to processor + caches + bus interfaces + dedicated peripherals as a pre-integrated IP block) and in interface complexity (multiple bus interfaces to shared resources). The debug questions for embedded systems design move beyond those of “is this core working correctly” to “how do I get my application code operating more efficiently” and “how well is this part of the system interacting with other subsystems”. These are problems that require a system debug focus rather than just analysis of a single core

In this paper, a system level debug approach and architecture called MultiCore Embedded Debug (MED™) is presented. MED architecture supports structured ASIC integration and diagnostics by creating a distributed subsystem of on-chip instrumentation (OCI®) blocks, customized to support diverse processors, embedded logic blocks, and embedded buses. This approach provides a debug backplane to address dense and complex multi-core systems analysis. By using instrumentation blocks as resources for embedded intelligent debug operations, analysis features such as system-wide error recognition and filtering, and cross triggering and performance analysis between different subsystems of a complex architecture are supported, which are not achievable with other currently available debug strategies.

Author(s) Biography

Neal Stollon is systems engineer with First Silicon Solutions. He has over 20 years digital design and processor development experience at Texas Instruments, LSI Logic, Alcatel, and others. Dr. Stollon has a Ph.D in EE, is a Professional Engineer, has written over 25 papers (including 4 papers for DesignCon in 1999-2002) and holds 7 patents.

Rick Leatherman is President & CEO of First Silicon Solutions (FS2). He has over 20 years experience in development tools at Intel and Microtek, and has championed the concept of On-chip Instrumentation (OCI®) in FS2 beginning in late 1998. Rick has an EE from Virginia Tech and an MBA from the Tuck School at Dartmouth College. He has written numerous articles on development tools and currently is listed as "co-inventor" on a number of pending patents

Bruce Ableidinger is Director of Business Development at First Silicon Solutions (FS2). Bruce has worked in the instrumentation and development tools business for over 25 years including time at Intel and Tektronix. He has a MSCS from Oregon State University and a BSEE from Washington State University. He is co-inventor of 3 patents.

Ernie Edgar is cofounder and Vice-President of Engineering at First Silicon Solutions (FS2) and is the principal architect for MED. He has worked in the development tools business for over 15 years. He holds a BSCS and MSEE from MIT and has 3 patents pending.

Introduction

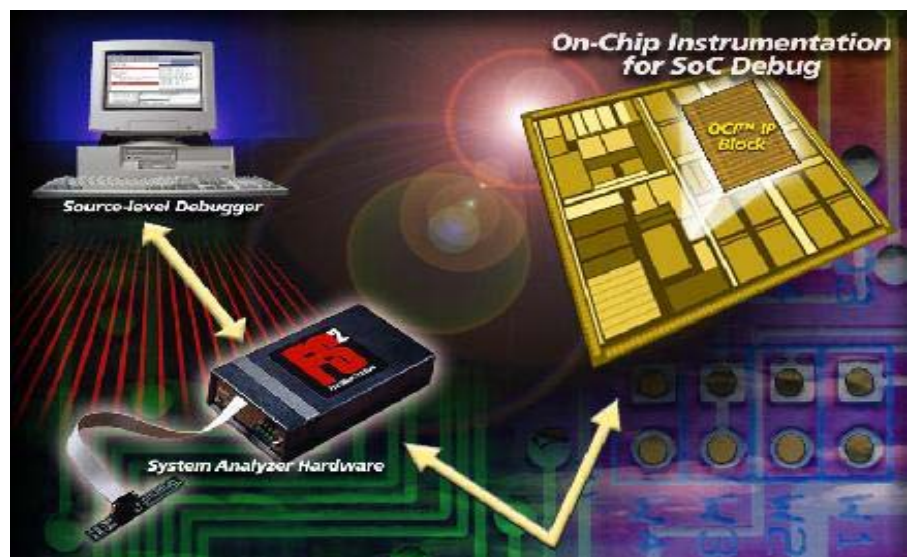
With each new generation of ASIC technology, the level of integration, functionality, and complexity provided on a single chip increases significantly. The problem that goes along with this increased amount of integration and functionality is that tasks and difficulty associated with getting a design working and integrated increases at least proportionally to the size and complexity of the chip. Over a range on modern silicon implementations, ranging from system FPGAs to ASIC System-on-Chip (SoC) platforms, there is a common need for better debug solutions.

As more processing elements, features and functions are simultaneously embedded into the silicon, the emerging level of embedded complexity outstrips the capability of standalone logic analyzer, debugger and emulator based diagnostic tools. While these tools allow the capture of data off the system data bus, they work only as long as every access (read and/or write) occurs over the external data bus. This points to an increasing gap in effectively being able to provide the necessary controllability and, in particular, the visibility of the internal operations of a complex system.

The need for improved methods of observing and analyzing embedded processor and System on Chip (SoC) operation has increased at least at a proportional pace to the explosive growth in SoC designs and new Intellectual property (IP) cores. This forces the analysis side of the SoC world into a constant process of catch up to the designer's ability to add cores and integrate new resources on chip. With an ever-shortening development cycle, and often several generations of products being produced in parallel or rapid succession, standardized embedded tools and capabilities that enable quick analysis and debug of the embedded IP is a critical factor in keeping SoC verification a manageable part of the process.

In formal testability terms, multi core embedded systems present an asymmetrical functional test problem. Their controllability is high, since the systems are dominated by programmable processor cores. The observability is low however, both in terms of critical signals are directly available, and in terms of the amount of embedded logic and internal signals as a ratio of to the available IO in which to observe them. The addition of dedicated resources and structures that support functional analysis is needed to increase the system observability. This requires a hierarchical focus to the issue of system analysis, starting at individual core level of debug instrumentation and resources and increasing to a more system centric diagnostic capability to facilitate increased observability. While embedded debug instrumentation approaches are becoming increasingly common at the core level, system level diagnostics and analysis at the MultiCore level has been a largely under-addressed and unresolved area of focus in complex embedded systems.

Figure 1 – On Chip Instrumentation Based Debug



Based on the shortfalls in applying current debug approaches to complex SoCs, the debug of structured ASICs and related single chip systems containing many embedded processors cores requires new systems level instrumentation approaches. The integration and debug of multiple cores, combined with an increasing ratio of overall gates vs. package IO, makes an increasingly dominant amount of a system design "“deeply embedded”, such that only minimal amount of data needed for analysis can be made available real time at the chips pins. These deeply embedded systems introduce new analysis problems, due to the interaction and communications of multiple cores, in addition to the more traditional debug issues associated with single processor systems. The Multi-core Debug requirement implicit for Structured ASIC requires new capabilities that exceed what can be addressed by traditional in-circuit emulation and logic analyzer capabilities, and JTAG and BDM resources used in single processor architectures. Whereas a JTAG or BDM can provide a snapshot of a piece of the system, the dynamics and interaction of multiple processors requires a more dynamic and robust means of providing diagnostic information necessary to the designer and integrator.

The availability of gates and on chip resources of modern ASICs allows for more innovative approaches to systems debug and embedded logic analysis by allowing dedicated debug subsystems to be created, with a minimal or even negligible impact on the overall chip size. Dedicated debug subsystems would effectively extract and analyze signals and operations within and between deeply embedded processor subsystems of a complex design. The main focus in this paper is on systems debug approach based on On-Chip Instrumentation (OCI) and an encapsulation architecture called MultiCore Embedded Debug (MED) that provides observability and analysis features needed to address the deeply embedded multi processor debug problem. MED leverages Structured ASIC resources to create a distributed subsystem of instrumentation blocks, customized to support diverse processors, embedded logic blocks, and embedded buses. This approach provides a "debug backplane" to address dense and complex multi-core systems analysis.

1. The MultiCore embedded debug problem

In depth analysis and debug of applications that utilize multiple embedded processors and buses is a critical piece of the system-on-chip design process. While debug of a single embedded core is challenging, it is a relatively well understood problem, whereas MultiCore architectures add new considerations and complexity that must be addressed in a debug solution. As core implementations become more complex, they expand both in subsystem functionality (moving from a processor core to processor + caches + bus interfaces + dedicated peripherals as a pre-integrated IP block) and in interface complexity (multiple bus interfaces to shared resources). The debug questions for embedded systems design move beyond those of “is this core working correctly” to “how do I get my application code operating more efficiently “ and “how harmoniously is this part of the system interacting with other subsystems”. These are problems that require a larger debug focus than just a single core"

Providing real-time analysis and debug of applications that utilize multiple embedded processors and buses presents a challenging verification problem. [2] Assuming debug logic and JTAG interfaces are present in all cores, JTAG can be configured in some combination of serial and parallel operations as seen in Figure 2. The tradeoffs in the serial and parallel JTAG implementation are

1. To have separate JTAG interfaces for each core, in which case each core may be debugged simultaneously but independently, with the cost of an additional JTAG port for each core. (as shown with cores 1-3) or
2. Have multiple core on a single JTAG chain, which limits JTAG information access to one core at a time (this is shown in cores 4-6)

Neither solution has proved optimal in practice. An alternative approach implemented as part of MED is a new encapsulating architecture for JTAG that combines the advantages of serial and parallel JTAG and provides sufficient bandwidth, flexibility and integration to address the debug interface requirements for large MultiCore architectures.

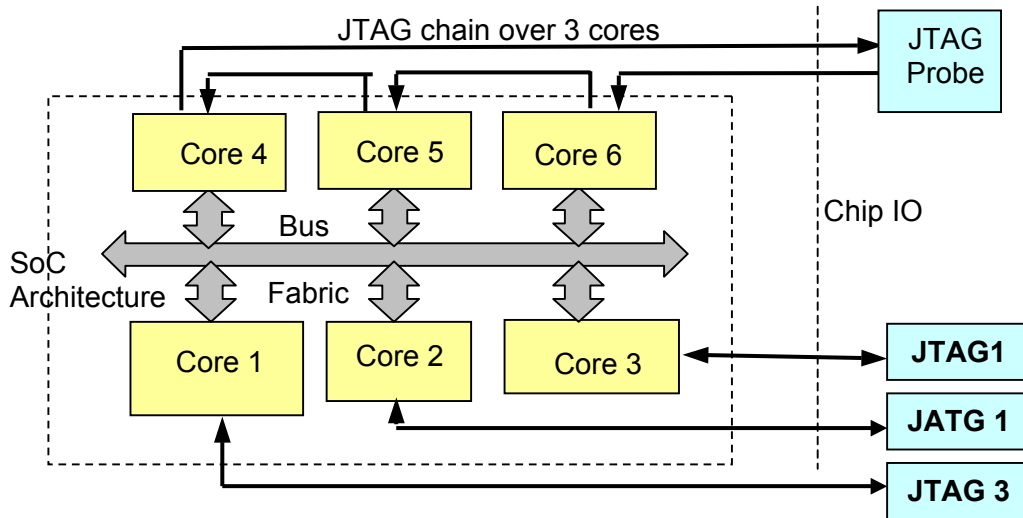


Figure 2 – Common Alternatives for JTAG

The amount of debug information that must be potentially considered from both different cores and their common infrastructure (embedded buses, shared peripherals, etc) is much larger than that required for single core debug. The related challenges of MultiCore debug are therefore more wide-ranging and diverse than debug requirements for simpler systems. Since subsystems are interacting on chip, implementing debug controls for a single core (halt, single step, etc.) can be ineffective and sometimes misleading. Additionally, different cores, from different IP sources, have different types of debug support, provided by different debug tool vendors and with varying levels of inherent debug capabilities. Current JTAG approaches allow only one core in a chain to be integrated at a time, whereas the information required for system debug may require synchronized access to information from more than one core. If separate JTAG interfaces are implemented to allow concurrency in debug, the IO requirements for different cores, each having its own debug interface can be excessive and presents additional challenges of synchronization and coherent integration.

Verification of internal operations and software execution data from processor sub-systems and debug of user defined logic cannot be adequately addressed using in-circuit emulators or logic analyzers without some assistance to make the embedded information of interest observable. Consequently, On-Chip Instrumentation (“OCI”) is rapidly becoming a preferred method for verification/debug/integration of embedded processor sub-systems and system application code. A key advantage of instrumentation is the ability to support real-time debug and tracing of code execution and operational interfaces of embedded processors and buses, that are “buried in the silicon”. As such, OCI approaches provide a powerful general solution to embedded processor and on-chip bus debug, which forms the basis of the MultiCore Embedded Debug approaches under discussion.

On Chip Instrumentation was developed as an approach that addresses signal visibility issues of debugging a highly integrated embedded processor and SoC. OCI is a next emerging stage of debug approaches and capabilities that have evolved along with the state of embedded processor design. Historically, this evolution started with ICE, which was developed to support debug of processor-based parts with minimal amounts of additional integration, but by emulating operations, rather than executing them normally. ICE based approaches have been largely subsumed by the adoption of JTAG, which

popularized the use of a debug port that allows low overhead access and provides embedded developers with a range of static capabilities for debugging. JTAG's limited bandwidth however, was never designed to support real time analysis. On Chip Instrumentation complements the debug port philosophy of JTAG by extending the debug bandwidth to address one or multiple cores, internal buses, complex internal peripherals and high speed data traffic found at SoC levels of complexity.

By using instrumentation blocks as resources for "embedded intelligent" debug operations, analysis features such as system wide error recognition and filtering, and cross triggering and performance analysis between different subsystems of a complex architecture are supported, which are not achievable with other debug strategies. At a minimum a MED environment must address three major facets of the multi-core debug problem

- **MultiCore Debug Concurrency:** The need to concurrently access debug and JTAG ports for all the cores in a system. To analyze problems and optimize performance in multi-core operations, the designer should be able to exercise any and all core debug features and interfaces through them. This capability is not supported in current JTAG debug architectures. New debug schemes need to address concurrent JTAG or debug port communications with (native) debug tools that support them.
- **System Debug Integration:** Triggering and trace needs to be addressed at a system rather than core specific level. System level instrumentation for a debug, trace, and triggering environment need to support multiple on-core and inter-core conditions (breakpoints, tracepoints, other specific control or status conditions) and send global actions to all or a subset of the cores (halt being the most obvious example). Most systems communicate over a range of buses. MultiCore Embedded Debug must be able to monitor signals on the embedded bus backplane, support the specifics of widely used on-chip bus schemes, and to trigger on and trace bus operations based on specific conditions.
- **Debug Synchronization:** Systems debug must address blocks running over several time and clock domains. Robust approaches to global timestamp synchronization are needed to support coherency between multiple debug environments that may be involved for different cores on a single chip.

2. What are Structured ASICs

Structured ASIC is an emerging class of systems that in particular benefit from MED. Structured ASIC concepts are an umbrella of definitions for a range of lower cost and time to market alternatives to more traditional ASIC approaches. Structured ASIC vendors differentiate on a number of physical level features in the ASIC process in their structured ASIC physical implementation. Of more interest from an architectural point of view, is a common thread in many structured ASICs of the combination of fixed resources for user-defined logic and connectivity on a single chip. This infrastructure standardization brings ASIC cost down by allowing a vendor to reduce and reuse many of the steps involved in ASIC physical design.

These structured ASIC fixed resources may include high performance IO, memory resources, and in many cases, embedded IP blocks and cores. In many cases a priori interconnect or bus architecture may be defined to simplify or optimize connectivity of these on chip resources. In a debug system context, we prefer to stress the architectural structure aspect in Structure ASIC, rather than the physical design innovations.

By this architectural definition, Structured ASIC can also include a range of system level FPGA products that either integrate instances of processor subsystems and high performance IO or allow automated creation of blocks and interfaces that become user-defined subsystems .

The inclusion of increased amounts of dedicated resources for core and system design appears to be a logical direction for emerging architectures. As mainstream geometries shrink, we see two trends logically emerging for all high-end ASIC designs, structured and otherwise.

1. Barring revolutionary changes in IO design and package design, the physical requirements for packaging of ASICs will make all designs pad and IO limited. As a result, in many cases, gates and memory for system debug applications will be “free” in that they have no impact on die size and can be implemented in what would otherwise be unused die area.
2. Barring revolutionary changes in design methodology, the gate availability of ASIC designs will outstrip the ability to effectively utilize them in near term architectures. This point is shown in Figure 3, which projects available vs. used gates, based on current process roadmaps and design methodologies. This reinforces the economic viability of allocating larger amounts of the overall design to system debug to facilitate the silicon verification and integration.

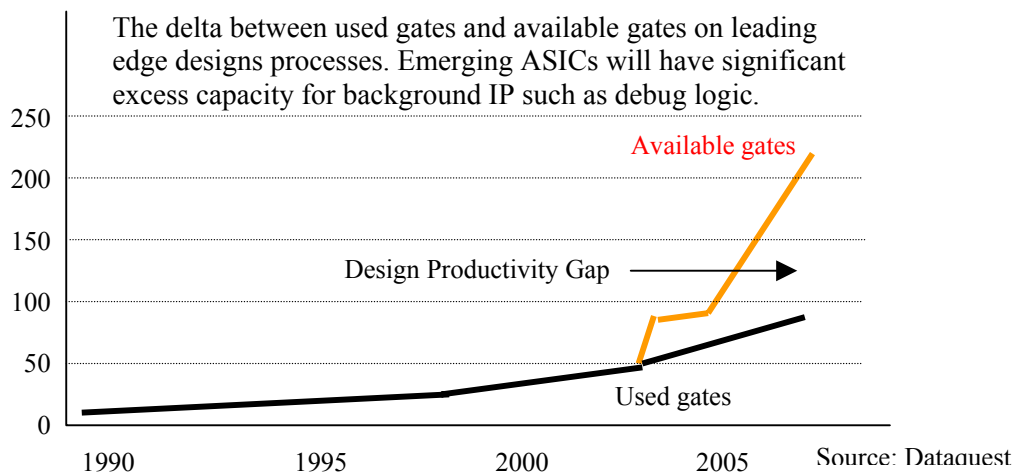


Figure 3 – The ASIC Drivers for Debug

In cases where a paradigm shift to a purely IP based design methodology is used to increase the amount of useable logic that can be inserted in a design, the resulting architectures will have an increased need for debug tools to bring up and integrate the resulting architectures.

3. MED Systems Architecture

MED as a system solution, is a hierarchical distributed architecture that leverages predefined core specific debug interfaces and instrumentation IP and builds on the integration of these blocks to provide the desired amount of system debug capabilities. Consider a typical MultiCore system consisting of 3 major types of embedded elements; processor cores, other customer (user defined) IP, and on chip buses as shown in Figure 4.

Robust debug approaches exist for most of the key components in an SoC solution (cores, user IP, embedded buses) [1] Integrating these into a comprehensive system debug architecture is not as well understood a solution. Developing a system debug solution involves both the application and integration of processor and core specific on-chip instrumentation and an in-system debug architecture to allow the individual debug blocks to communicate and interact. The system infrastructure is the key to turning a collection of point debug interfaces (shown in Figure 5) into a coherent debug solution.

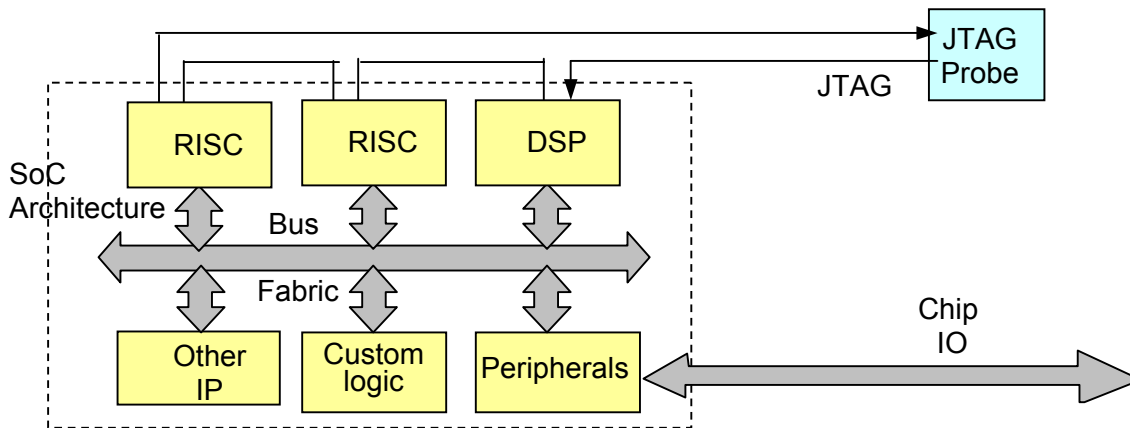


Figure 4 – Baseline Structured ASIC

Simply adding independent OCI blocks for debug support for each of the major components, as seen in Figure 5, in a systems does not result in a desirable solution, from a IO, internal wiring or integration point of view. The debug infrastructure should address at least 4 major concerns.

1. Reduce the IO requirement concurrent debug interfaces for each of the major blocks of a design.
2. Provide synchronization and interactive debug communication for each part of the design to be able to contribute into event recognition of systems events and be able to trigger debug actions in any other part of the architecture.
3. Be scalable to allow for a minimal increase in debug internal wiring and IO as the size and complexity of an architecture increases.
4. Also be scalable to address core, core subsystem, multi-core, and multi-core debug subsystems at different levels of hierarchy and abstraction with a similar set of debug resources.

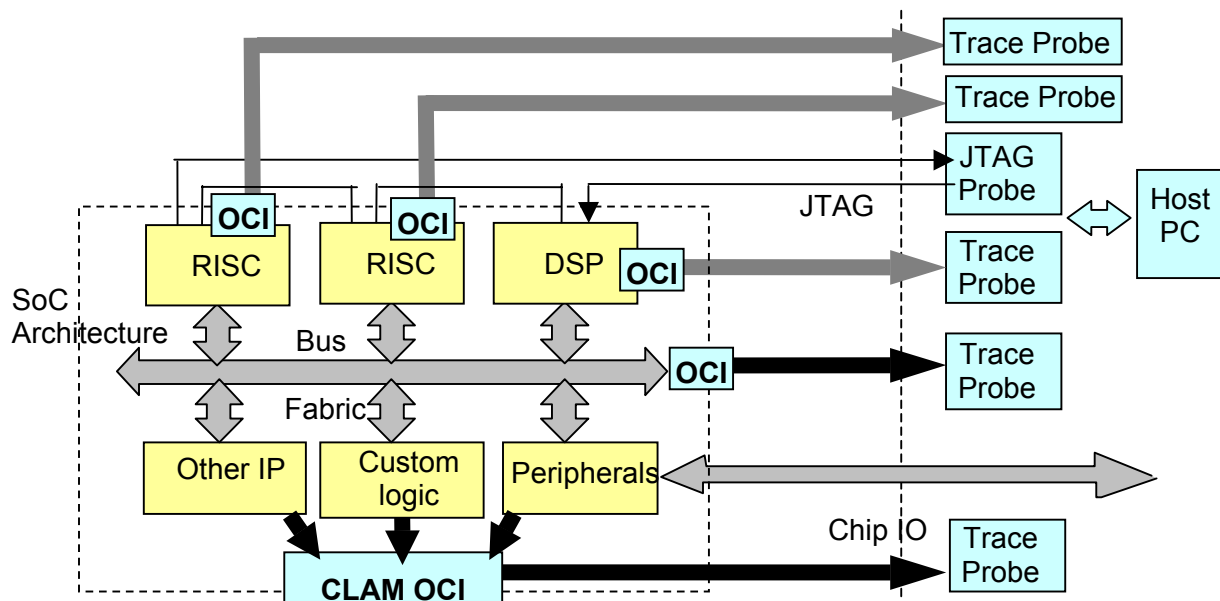


Figure 5 – Adding Independent OCI for Debug

For the first issue of reducing the IO requirement, an on-chip debug collection and concentration and concatenation block is needed to reduce the larger number of debug interfaces into a single debug port. In general, each JTAG chain can only communicate with the JTAG interface for one core. The number of cores that may be concurrently debugged is directly proportional to the number of JTAG chains on the chip. While some processor debug solutions allow simultaneous access to multiple cores on one chain. (ARM MultiICE as an example), these solutions are limited to debug of a single probe type, whereas the key ability for system level analysis is to simultaneously debug different types of cores.

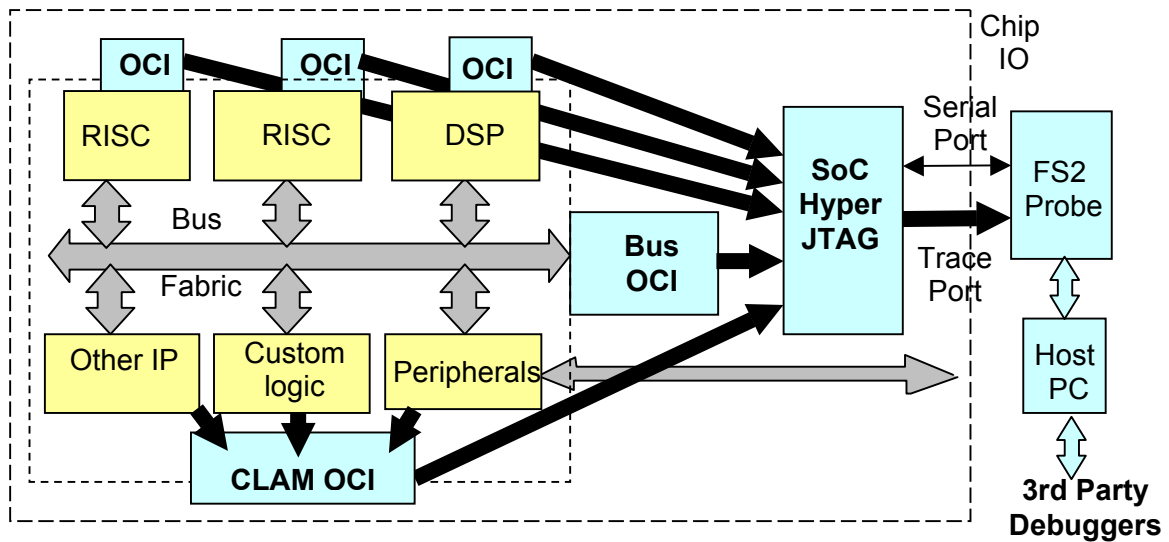


Figure 6 - Funneling OCI through HyperJTAG

While having multiple JTAG chains on a chip is a small overhead for the ability to simultaneously debug multiple cores, having separate JTAG pin IO for each chain is often unacceptable in a world of pin/IO limited chips. Ideally the information from each JTAG interface should be concentrated to can be run over a smaller number of external pins. We refer to this debug information collection / concentration block which supports simultaneous debug of different cores in an architecture as HyperJTAG™. As shown in Figure 6. HyperJTAG communicates with all cores in an architecture and formats the debug information from each core as a superset of JTAG. HyperJTAG leverages many of the common JTAG attributes and signal logic widely used in current debug tools. The specifics of the MED HyperJTAG are discussed in the next section

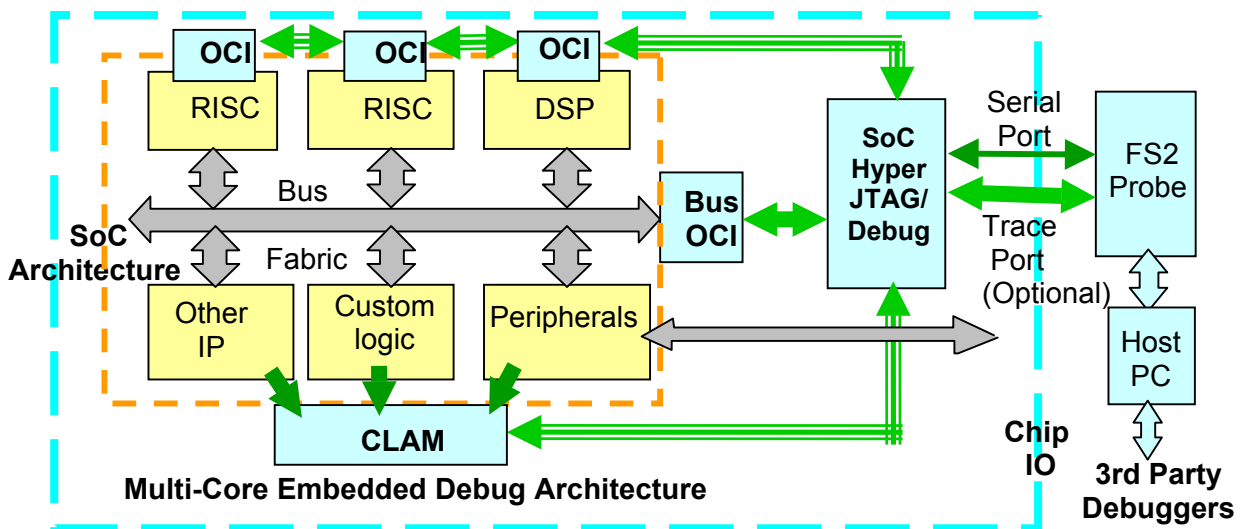


Figure 7 – Serial OCI integration with HyperJTAG/HyperDebug

To address the second issue - enabling synchronization and interactive debugging of the architecture, a distributed debug architecture called HyperDebug™ provides system wide event recognition, global or selective triggering, and synchronization of MultiCore debug activities. In most architectures, the primary cores in the design will have some level of diagnostic resources including JTAG and in many cases, instrumentation blocks to support debug and trace. HyperDebug leverages these resources by providing a layer of encapsulation and integration to core specific instrumentation blocks in a design to allow them to address system level debug operations. The specifics of the MED HyperDebug are discussed in the next section.

The HyperJTAG interface reduces the debug IO allocation and makes the debug interface more compact and manageable, does not address the issues of having a potentially very large number of wires communicating between the HyperJTAG and the core debug OCI and JTAG blocks. While this is a manageable problem for small numbers of cores, as the size of architecture scales up to many cores, the wire routing and synchronization issues increases in complexity proportionally to the number of cores.

An approach taken in MED is to have all of the cores be connected in a set of serial JTAG rings that allows for shorter and more routable core to core serial connections. This approach is used in both the HyperJTAG and HyperDebug components of MED as shown in Figure 7. This architecture takes advantage of the core/IP/bus trace information having been buffered locally in the OCI so that bandwidth requirements for real time trace operations are minimized.

Supporting the on chip architecture, MED (as a system analyzer environment as shown in Figure 8) also consists of external probe hardware to provide off chip HyperDebug interfaces and physical interfaces to core specific and third party debug tools. It also contains a software environment that provides the GUI for user setup and for HyperDebug and HyperJTAG control, display of generic trace information such as on chip buses and applications layer communication with other debug software components. This is discussed in more detail in section 6.

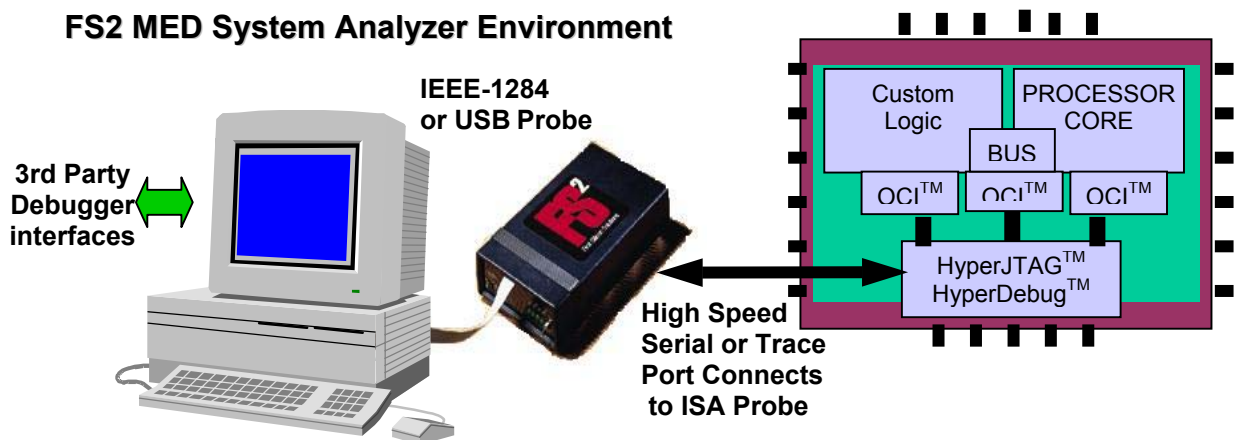


Figure 8 – MED Environment Overview

4. MED On-Chip Instrumentation Components

In this section, we will examine some of the individual components that make up the MED architecture. MED can be considered as a hierarchical debug architecture, consisting of

1. Different types of On-chip Instrumentation (OCI) blocks that are associated and locally integrated with the individual component cores, IP and custom logic blocks, and buses that makes up an architecture

2. An encapsulating debug infrastructure, primarily consisting of HyperJTAG and HyperDebug OCI, that tie together the system level debug communications and provide a system level debug capability
3. Supported external probes and software, both developed by FS2 to directly support MED operation and 3rd party point solutions for debugger interfaces to specific processors

The individual debug requirements for processor cores, IP and custom logic blocks, and buses differ and should be considered individually to help put a system debug solution in context.

4.1 Embedded Processor Instrumentation

Processor core debug instrumentation is primarily concerned with a limited and, for a given processor, well-defined set of signals. These typically include instructions and data along with their respective address buses, internal registers such as program counters, status registers and execution history. Most processor cores have JTAG interfaces that allow a controller to halt the processor and read a range of register values. Halting a processor to extract internal information raises its own issues when problems being analyzed are a result of external conditions such as signals from another processor. In addition, since instructions are typically interdependent on prior instructions and register information, the halt and read approach of JTAG provides a very limited snapshot of sequential information, which is often needed to examine processor operations in context. To address this limitation, processors have started integrating debug trigger and trace capabilities into and adjacent to the cores to allow for collection of sequential data (frames) without having to halt the core. As an example consider MIPS TCB and ARM ETM, - trace blocks for both cores typically consist of an instrumentation block integrated with a processor core that selects, collects, and streams real time trace data.

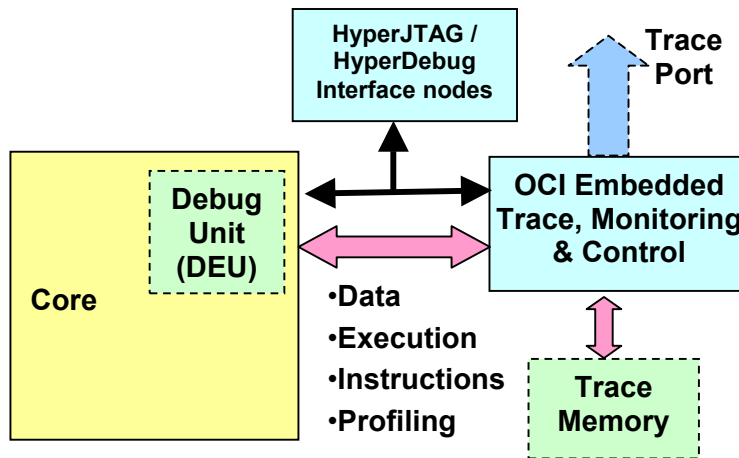


Figure 9 - Processor OCI Example

Processor trace instrumentation consists of a JTAG based controller and configurable trace processing logic that allow captures of processor bus trace, execution history, and/or other real-time information from the core. This information is converted to a HyperJTAG format at the HyperJTAG interface node associated with the core as shown in Figure 9. In many cases, event recognizers and trigger logic are also included in the instrumentation. The event triggers can control core operations such as enable or disable of breakpoints and triggering of trace collection actions. In MED, the event triggers for each core are cascaded using a HyperDebug architecture to allow MultiCore and system level event logic and trigger actions.

Trace instrumentation can be implemented as either internal (on-chip trace buffering) or external (off-chip trace storage), depending on trace depth requirement and resource availability (Internal uses on-chip trace memory and logic, external uses less resources, but more pins). MED, in order to maintain a low pin requirement, focuses primarily on internal trace. The processor instrumentation concept can

interface to a variety of debug blocks for specific processor cores, allowing instrumentation interface controls of start/stop execution, single-step, breakpoints, and register/memory access in the processor. Typical choices include execution trace, data cycle trace, and profiling trace as shown in Figure 10. Processor trace collection may also be enabled and disabled by hardware breakpoint registers set to generate trace actions. Processor trace instrumentation can also be configured to collect profiling data for performance analysis (as discussed in the next section).

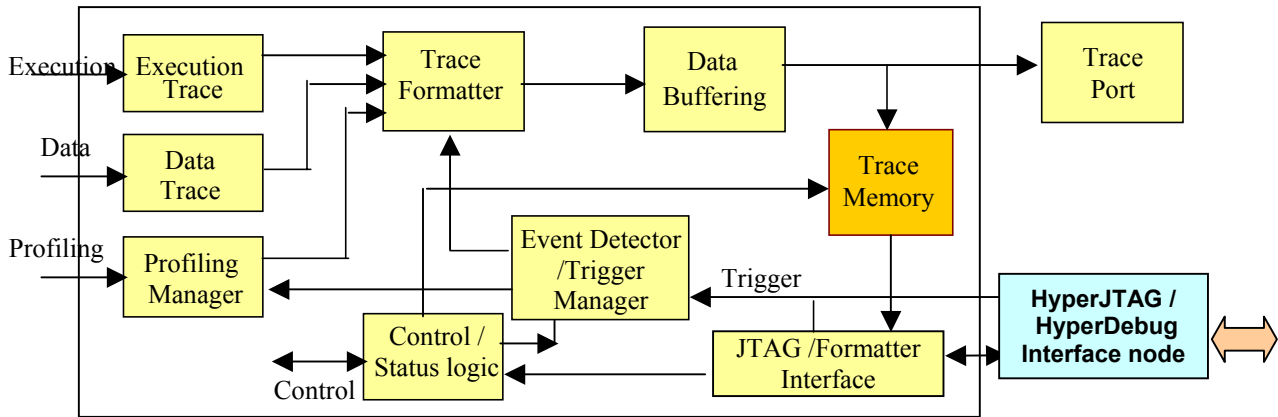


Figure 10 – Processor (Trace /Control) OCI Block Diagram

4.2 Custom IP Instrumentation – Configurable Logic Analysis Modules (CLAM)

Since processors have a regular structure and well-defined architecture, a fixed debug block architecture can be implemented as a general solution that provides analysis to key processor operations (PC, Instruction address, and load and store data) that provide information for analysis of the core operation in most circumstances. For user defined IP and custom logic, the debug requirement is often very different. Debug points need to be assigned at the IP, which requires a much more general logic analyzer solution. Embedded configurable logical analysis modules (CLAM) have seen wide acceptance in FPGA design, but a much more limited use in the ASIC community. A typical CLAM block is shown in Figure 11 and consists of configurable event triggering logic and a trace buffer for local storage of traced signals. The size of the buffer is configurable based on amount of memory resources allocated by the designer. CLAM architectures can support trace depths of 4K trace samples and trace widths in excess of 256 signals per cycle.

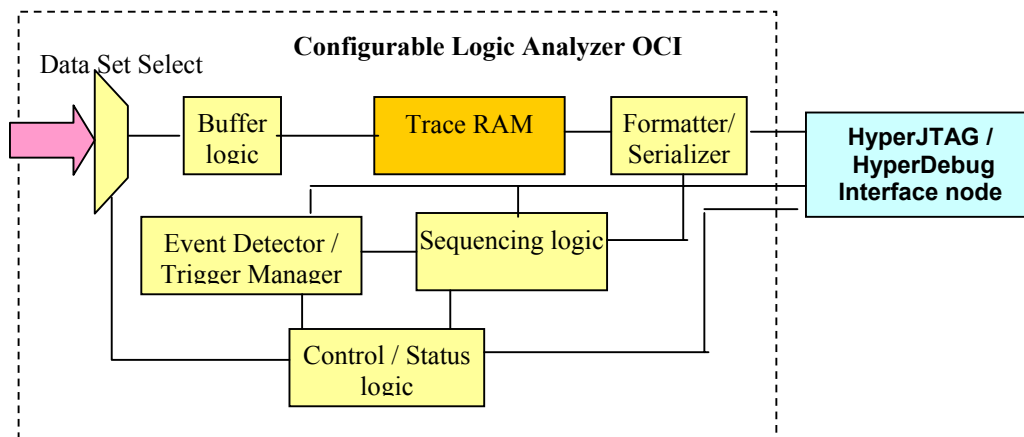


Figure 11 – Configurable Logic Analyzer OCI Block Diagram

Being able to trigger from OCI data allows for dynamic interactions with the target system and improved capture of information of interest. FS2 analyzers nominally support up to 4 triggers with up to 4 states per trigger. Trigger conditions can be created as application specific combinations of three components:

- Raw or processed data (filtered or aligned) compared to logic or edge events on each signal,
- Counter or times values matching a preprogrammed value
- Trigger state (what trigger related operations have occurred previously)

4.3 Embedded Bus Instrumentation – Bus Monitors

In complex MultiCore systems, the bus interfaces are an increasingly important factor in the overall functionality and performance of the system. Understanding real time bus operation is critical to being able to diagnose and analyze the overall system. Deeply embedded busses (buses with no direct interface to the system IO) present special challenges in typical embedded system debug since their observability is low and their operations are observed indirectly. In addition, most embedded bus architectures (AMBA AHB, CoreConnect, etc.) are based on multiplexed bus implementations with dedicated unidirectional read and write logic and bus components.[4] This is shown in Figure 12 for the case of an AMBA AHB bus Monitor. Multiplexed based buses double the number of data signals that need to be considered for debug compared to tristate bus implementations typical on external pin busses such as PCI. Typically Bus Monitor OCI integrates some decision logic to ensure that time and memory is not wasted in tracing invalid bus information.

As is the case with the CLAM IP, a bus monitor can be configured as either on or off chip trace; although for MED, only on-chip implementations are addressed due to our focus on minimizing trace specific IO. For a given bus architecture, the bus signals being monitored for trace are relatively fixed by the bus type, and includes address, data, and control.

The key tradeoff of a bus analyzer is how to most effectively trace a large number of interrelated and synchronized signals. As an example, even though the read and write data are separate wires in most embedded bus systems, a read or write operation may occur, but not both simultaneously. A bus monitor can leverage this, to choose either read or write data to trace during a clock cycle but not both. This embedded inline processing of bus data can significantly reduce on amount of useful data that is traced and managed.

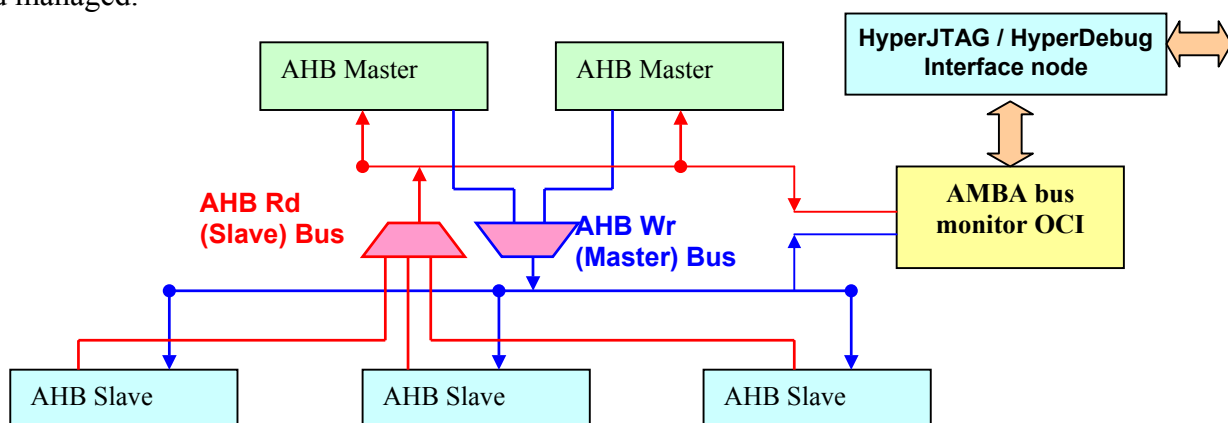


Fig. 12 AMBA AHB example of an bus Monitor OCI

Bus OCI should support a range of single-master and multiple-master bus architectures. It should be independent of a given arbitration scheme or specific signaling of a bus architecture. It should allow additional signals to be hooked up to any nodes in the SoC, such as interrupt requests, core and IO status, and CPU control signals which can be used to compare and recognize and trigger on specific on-

chip activity outside the bus. For real time OCI debug processing, the bus monitor should allow probing of data in different modes. As an example, bus data for multi-cycle transactions can be aligned for differing analysis

- Bus Cycle Mode - captures all address/control and data signals exactly as they occur per clock on the bus.
- Bus Transfer Mode - in which the master transfer and slave response, which occur on different clock cycles are aligned onto the same clock cycle, allows the combination of address-data-control event triggering.
- Filtering Mode - the trace triggering configured to filter out Idle and Busy and not Ready cycles where no active data is being transferred.

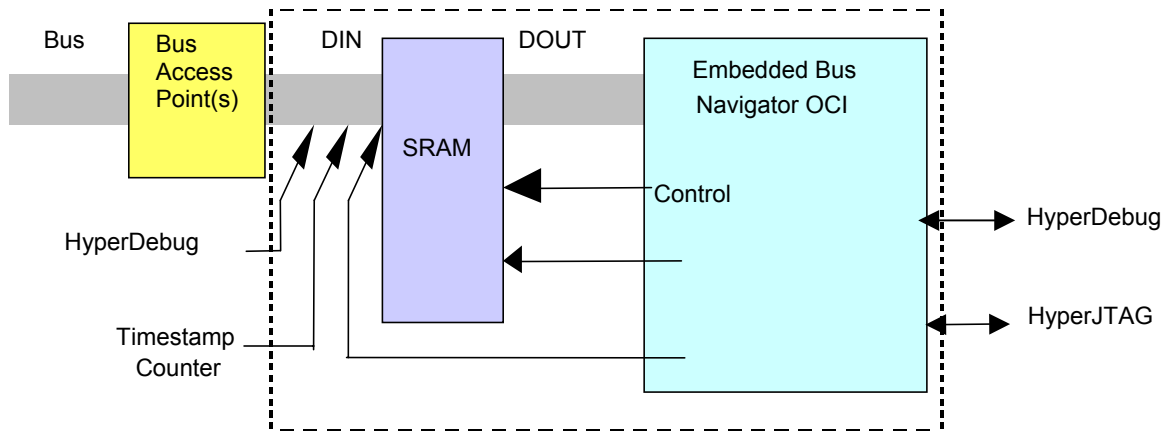


Figure 13 – Bus Monitor OCI Integration

Another useful feature for bus monitoring is the ability to align bus cycles. In many on chip bus architectures, the bus operation is pipelined such that address and control information from a master is sent in prior bus cycle to data transfer. The pipeline operation allows for faster bus operations but also spreads the information needed for analysis over 2 or more cycles. This latency between the address and data cycles may depend on specifics of a system and types of transactions involved. Debug monitors can align these address and data cycles to improve areas of understanding and reduce memory overhead.

5. MED Infrastructure

5.1 MultiCore Concurrent JTAG – HyperJTAG

For initial MED implementations, the HyperJTAG is designed to support 4 serial chains, each of which may be connected to any JTAG compliant core in the system. As such, HyperJTAG allows concurrent communications between 4 cores and their corresponding debug probes and SW tools (vendor probes). As shown in Figure 14, HyperJTAG consists of a HyperJTAG OCI block and a HyperJTAG probe. Operationally the information flow is bi-directional. Information from each vendor probe interface is connected to the probe which multiplexes and compresses the data to pass between the probe and the HyperJTAG OCI interface pin over an 8 pin interface (with an additional 16 pins being optional information) as shown in Figure 14. The HyperJTAG OCI decompresses the information and routes it to the core over one of the HyperJTAG chains to a HyperJTAG Interface Node (IN) that adjacent or part of a processor OCI and which forms a Debug interface (DI) to the JTAG interface for the core. Information from the core to the vendor probe travels in the opposite direction from JTAG DI to core OCI/HyperJTAG IN to Hyper JTAG OCI to Probe to vendor probe and debugger tools.

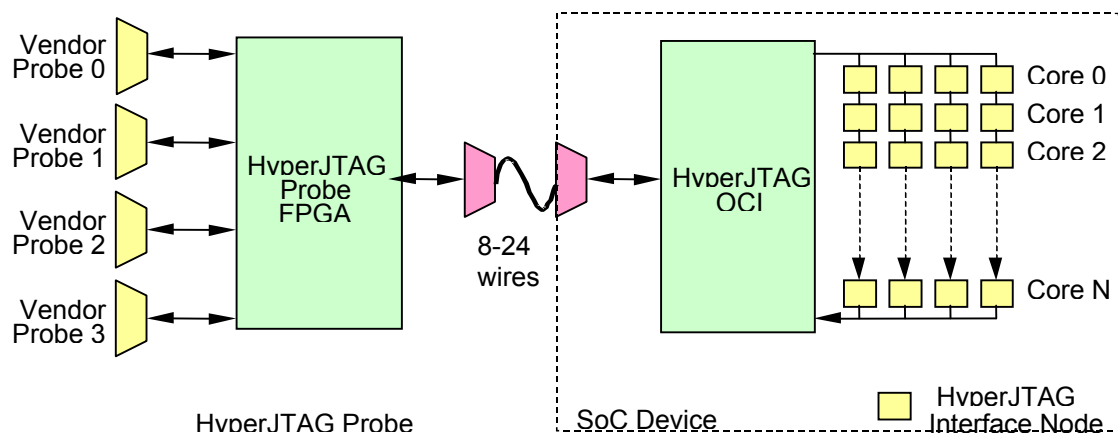


Figure 14 - HyperJTAG BUS Integration

Additional detail of the on-chip component of the HyperJTAG is shown in Figure 15. This shows the signaling of one loop running between the HyperJTAG OCI and a series of HyperJTAG Interface Nodes associated with individual cores. The internal HyperJTAG bus to each of the HyperJTAG interface nodes consists of 8 signals, 4 of which are standard JTAG (TCK, TMS, TDI, TDO). The remaining 4 signals provide HyperJTAG specific signaling to the Interface Nodes. The HyperDebug Interface Nodes local to each core are loosely integrated to one of the HyperDebug chains to provide configuration and setup of the HyperDebug node for specific triggering operations.

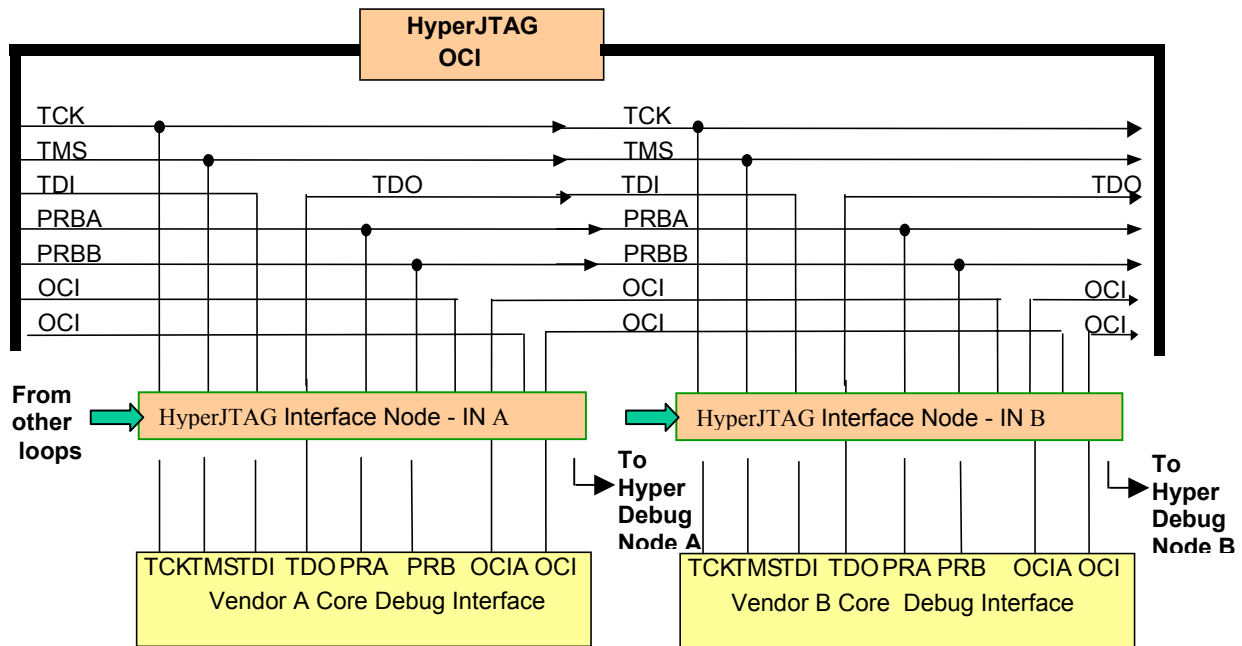


Figure 15 - HyperJTAG Node Integration

5.2 HyperDebug - MultiCore Synchronization Triggering and Global Actions

The amount of information in a MultiCore SoC is large enough that global event recognition is often needed to identify and isolate events occurring throughout the system. Event recognition is widely used in conjunction with trace to capture information on events and operations in the SoC. Trace data values are monitored and compared to event sequences to provide real time triggers in the OCI block(s). These

triggers in turn can be used to control event actions such as breakpoints and trace collection. MED OCI event recognizers can simultaneously look for bus address, data, and/or control values and be programmed to trigger on specific values or sequences such as address regions and data read or write cycle types.

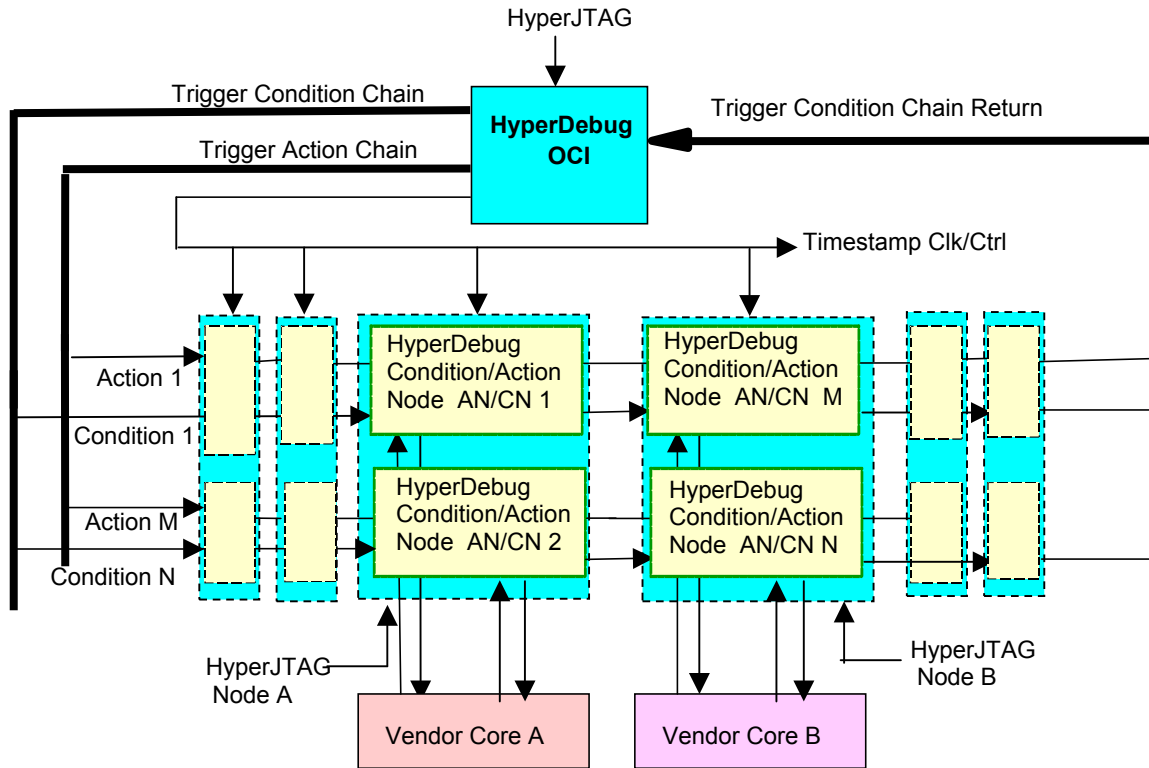


Figure 16 - HyperDebug System Integration

MED HyperDebug consists of 3 distributed types of components:

- HyperDebug OCI, which initiates trigger condition and action operations and maintains the overall HyperDebug control and status.
- HyperDebug Condition Nodes (CN)– which modify the trigger conditions based on local conditions in the core, OCI, and other CNs connected to the core. Typically a number of CN blocks are implemented related to trigger conditions monitored in a given core.
- HyperDebug Action Nodes (AN) – which initiate logical actions such as setting of registers in the core or OCI. AN operations are local to specific codes or may be global to all cores in the SoC. (halting or resetting of the core is one such example). Typically the number of AN blocks is related to the number of actions that would be required for the debug logic to control cores operations.

The OCI accepts a configurable number of Condition inputs and generates a configurable number of Action outputs. Condition inputs may come from a chain of CN's or may come from external pins fed from the HyperJTAG probe. Action outputs may go to a chain of AN's or may go to external pins leading to the HyperJTAG probe. The number of Conditions need not necessarily match the number of Actions.

Condition inputs are synchronized and stretched to match the clock period of the HyperDebug OCI. HyperDebug trigger conditions are an AND combination of one or more of the following:

- Condition input from a CN chain.
- Condition input from an external pin.

- A 32-bit event counter matches a pre-programmed value.
- HyperDebug Sequencer state.

When a condition is indicated, the HyperDebug OCI may be programmed to perform one or more actions:

- Assert, negate, or pulse one or more action outputs to an AN chain.
- Assert, negate, or pulse one or more action outputs to an external pin.
- Start, Stop, Increment, or Clear operations on a 32-bit event counter.
- Change to another HyperDebug Sequencer state.

Typical uses of HyperDebug are:

- Periodic trigger signal to insert synchronization messages in each core’s trace buffer.
- Assert a logical break signal to all cores under debug when any one core hits a breakpoint.
- Insert a trace message in each core’s trace buffer when one core reaches a trigger point.

The HyperDebug block also sources a reference clock signal for timestamping of data at each OCI block. Given that cores in a system may operate over range of frequencies, including asynchronously to each other, a master timestamp provides a means of synchronizing the time of a core operation in relation to other cores in the system

6. MED Systems Integration

Any comprehensive debug system needs to have the supporting infrastructure that facilitates both ease of use and provides sufficient power and flexibility to address large and complex problems. The discussion in this paper focuses on the MED on chip architecture. MED also includes an off chip architecture, consisting of both hardware and software, that is required to support MED applications. The off chip components for MED are referred to as HyperJTAG Probe and System Analyzer Software.

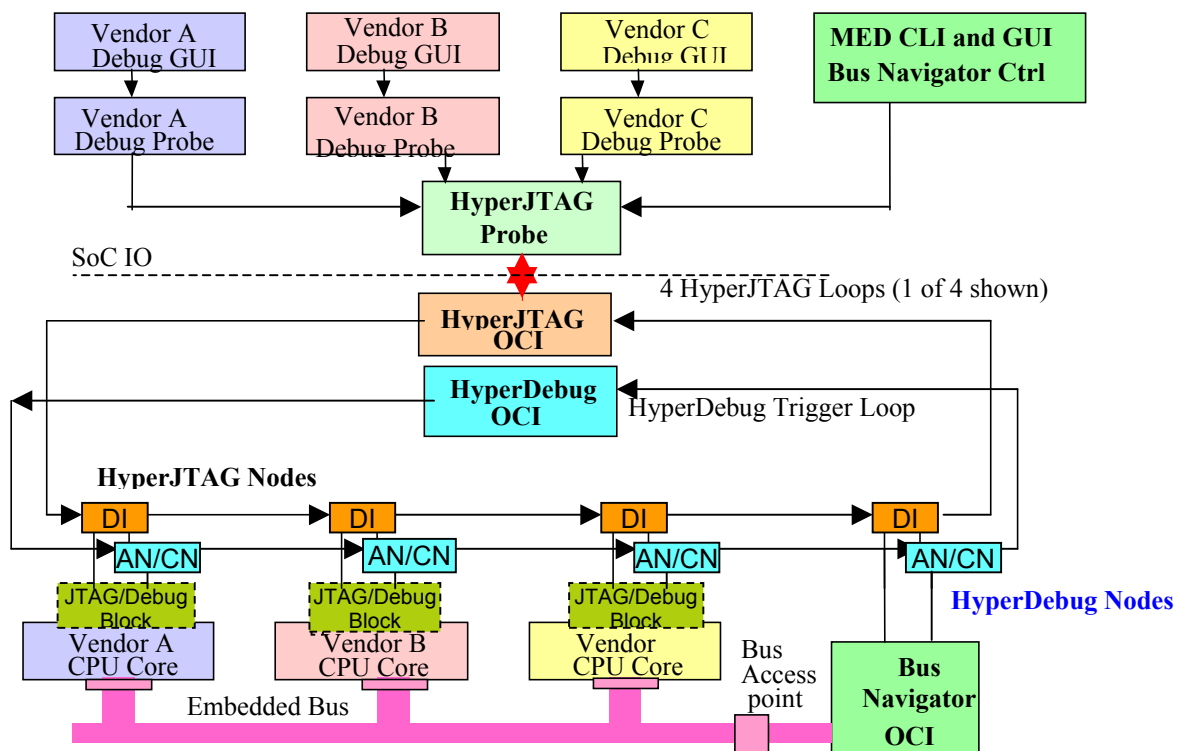


Figure 17 – MED System Integration

The HyperJTAG Probe supports the HyperJTAG communications as discussed in section 5. The probe provides a physical interface between the MED OCI system ASIC and other debug tools as shown in Figure 17. It supports both the routing and communication between the processor OCI and their corresponding debug tools and the interface of MED software tools to the on-chip HyperJTAG OCI.

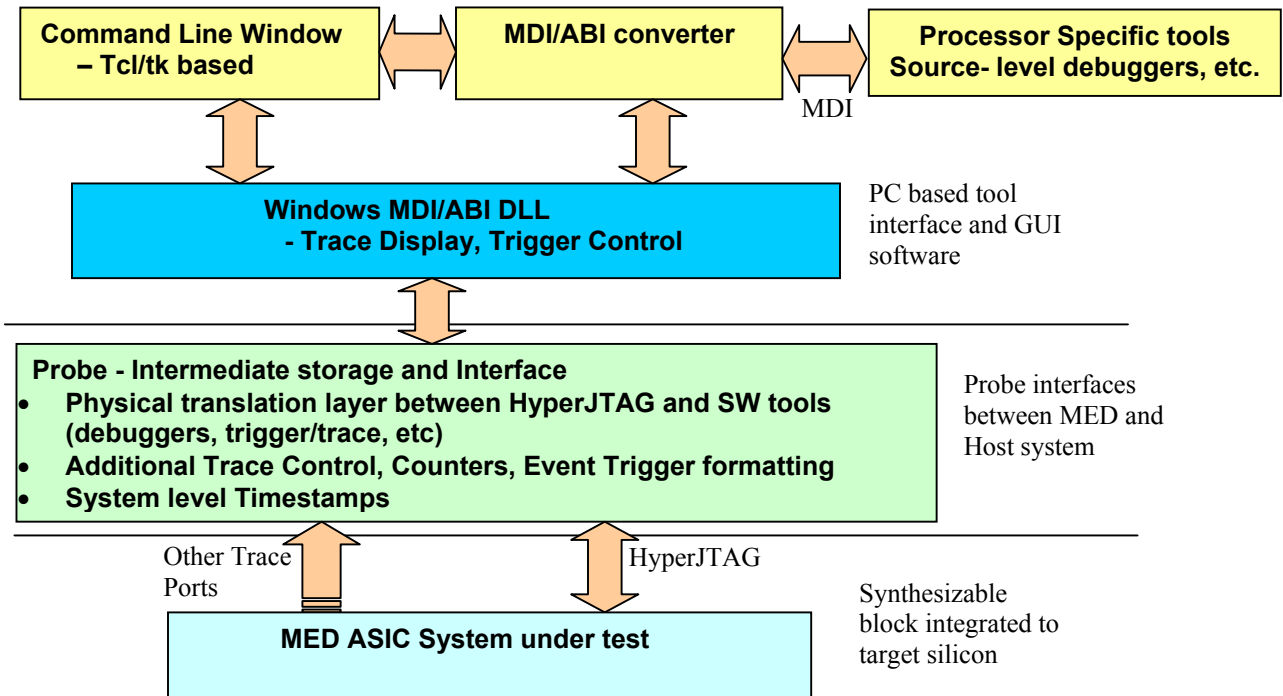


Fig. 18 MED System Environment

MED Software Environment

A wide variety of software tools such as JTAG debuggers from multiple vendors may be used in various phases of the SoC design and integration. Processor specific tools may include dis-assemblers and debuggers along with RTOS, tool chains, and host development platforms required for processor operations. Despite progress in tool plug and play capabilities, integration of multi tool environments is still challenging.

As industry standards for sharing JTAG hardware between different applications using standard software API's progress, very few debugger or probe vendors support such a standard, generic interface. To work with today's debug tools, MED must be able to accommodate several independent real-time JTAG channels and disparate on-chip status and control signals through HyperDebug. To simplify tool integration, HyperJTAG provides a virtual connection between any vendor's probe and control software and the corresponding core in the SoC.

The MED HyperJTAG approach supports any vendor's software, probe, and core without modification. The GUI for MED must address a diverse range of operations required to control and display information provided by the MED OCI. Though MED is not directly involved with processor debug, MED software does provide the ability to set up the HyperJTAG virtual connections, HyperDebug triggering system, and the on-chip bus monitor and trigger system.

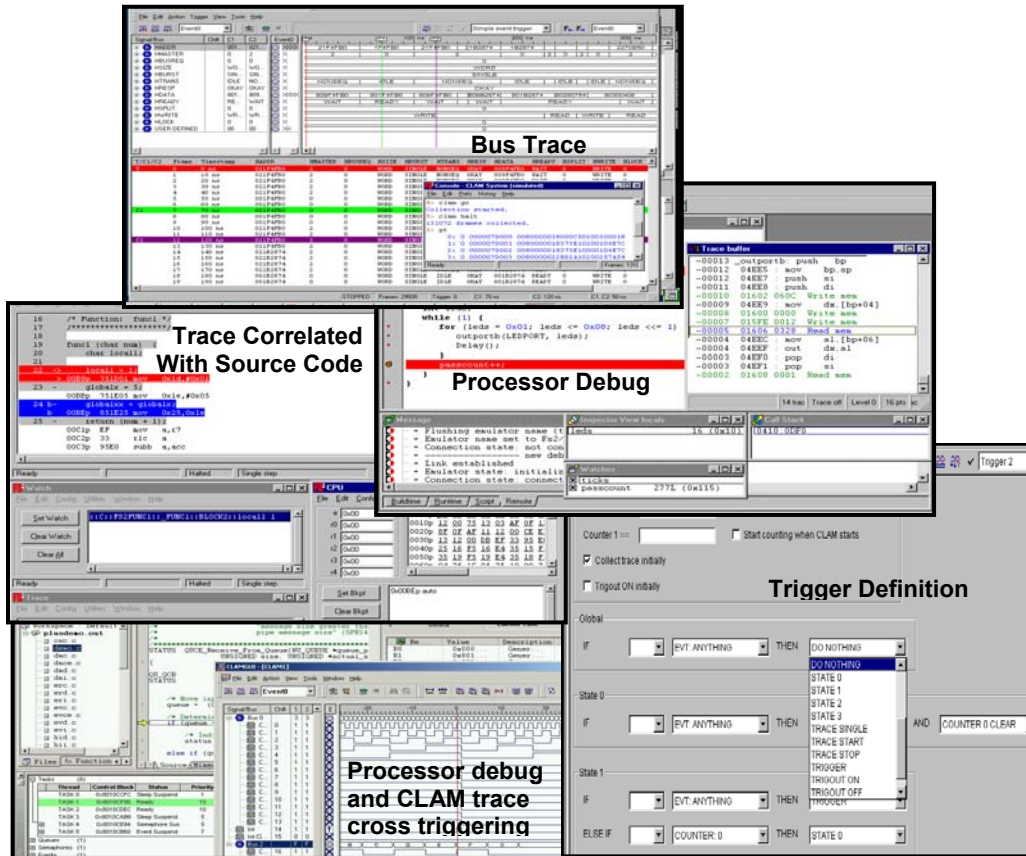


Figure 19 – GUI Windows for MED Software Control and

MED control software to set up the virtual JTAG connections and other on-chip resources includes a Command Line Interface (CLI) based on Tcl/Tk, a command language widely used in validation and manufacturing test. Tcl/Tk’s wide acceptance allows common interface and command primitives and Tcl procedures to be used across a range of tool platforms. Many vendors customize Tcl for use as a control and communication mechanism for their tools. As examples - commands are included for system configuration, emulation control, memory access including an assembler and disassembler, register access, trace and trigger access, file download, FLASH programming, and status indication.

Graphical interfaces simplify the set up process for first-time and occasional users while a Tcl-based command-line window supports the advanced user through scripting and customized windows. MED software is also responsible for connecting a GUI developed for the CLAM to the on-chip bus monitor for trigger set up and trace display.

8. Emerging Applications - Performance Analysis Instrumentation

Performance Analysis (PA) as an all-encompassing term refers to many types of on chip measurements that provide information on how a particular core is being utilized, both in context of other parts of the system and with regards to specific algorithms. Integrating OCI to allow processor characterization, software performance, and system performance metrics provides valuable and concise information, which is more simply gathered on chip than off. Lack of useful signal visibility at the IO can make embedded processor performance testing challenging. Direct performance analysis is often limited or obscured by the layers of system buses, peripherals, and limited IO access between an embedded processor and the external test environment.

Some common types of testing for SoC system performance analysis that OCI can enable are:

- Processor Optimization
- Software Performance Analysis
- System Performance Characterization

8.1 Processor Optimization

For processor characterization, the common method of performance monitoring is use counter instrumentation and a selectable set of processor and bus events to count. Typically counter reaching a programmed terminal count can generate an interrupt allowing the processor or the instrumentation to read and reset counters and log the information.

In an embedded processor, performance monitoring counters and hardware triggers for start and stop measurements are set up and read out via the Processor instrumentation or JTAG port so that the measurements do not interfere with processor execution. While a single point-to-point measurement can help in determining performance problems, capturing many occurrences of these event counts, in real time, is more useful. A range of measurements can be made with performance monitoring OCI both to determine the overall performance of a processor and its operations on critical executing algorithms.

- overall I or D cache hit/miss ratios
- cache hit/miss ratios in a specific section of code
- instructions executed per clock
- number of executed instructions between two points in the program, and how it varies
- memory access utilization of on-chip or off-chip memory blocks
- number of processor stalls versus total number of instructions executed
- number of clocks during which processor is stalled relative to total clock cycles
- internal bus utilization; i.e. ratio of bus busy to total clocks
- number of branches taken relative to total number of branch instructions executed
- mis-predicted branches relative to number of branches taken
- number of clock cycles between location A and B in the program (A-B timing)

8.2 Software Performance Analysis

Software performance analysis helps to identify areas and reasons for code not executing as expected. OCI methods allow software performance analysis without perturbing the target execution. Approaches that are widely used include hot-spot profiling, where a processor's program counter (PC) is periodically sampled and binned to provide information on where the processor is spending most of its execution time and tracing instruction type information - in particular call or return instruction or ISR qualified trace. When used along with timestamping software performance analysis can be used to report information on a range of operations.

8.3 Multi-core Optimization Analysis

Performance Analysis options can range from system level characterization down to very detailed application performance. For multiprocessor designs sharing bus resources, system level measurements can include bus utilization, contention, latency and other system performance metrics. For debugging and tweaking system software where there is interaction between multiple cores, instrumented code can be distributed at the important locations in each core. A bus trace tool can be set up to qualify on writes to a block of shared memory and provide a history of locations executed. With timestamping, the duration of various core events can be displayed, illuminating the parallelism of the code running on the different cores and how software synchronization occurs between them.

Other types of measurements can include:

- Software throughput of the critical system functions - i.e. how much work per unit time the processor is doing. Examples would include network packets processed per unit time, streaming blocks of data (such as audio or video data) converted and forwarded to another stage of processing, interrupts processed, graphical data converted to pixels, etc.
- Measure loop times in critical algorithms - determine maximum duration, which may be more critical than the average duration
- ISR (interrupt service routine) min, max, and average execution times
- Placing markers around code that disables then enables interrupts providing measurement of worst-case interrupt latency
- Thread loop times, defined as the time from when the thread is activated by a semaphore synchronization until it is put to sleep waiting for the semaphore again

7. Putting It All Together – Implementing MED

The initial implementation of MED is being released for a leading custom ASIC platform that is designed to be extensible to support up to 64 cores. The MED environment would initially allow up to 4 chains of cores out of the possible 64 to communicate via each HyperJTAG probe at any given time. The HyperJTAG connections between probes and cores can be changed dynamically.

The MED ASIC architecture includes a fully configurable HyperDebug support architecture and a customized bus monitor interface that supports monitoring of one or more instances of bus structures. MED OCI platform integration currently supports a wide range of leading RISC, DSP and controller cores.

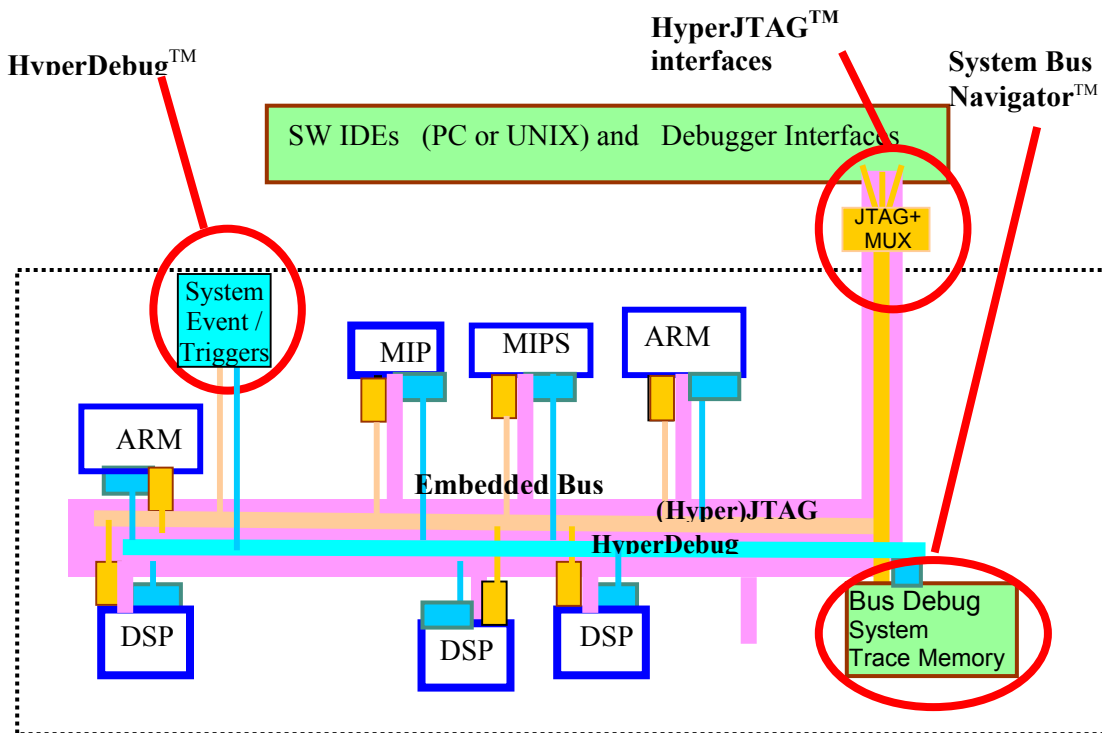


Figure 20 – MED SoC Implementation

MED has and will have increasing value in system level FPGA as they evolve to overlap more traditional ASIC capabilities. FPGAs currently have the capability to include several complex IP blocks

and processing cores, which lead to the same needs for diagnostic and analysis as in an ASIC. MED prototypes have been implemented on leading FPGA architectures and MED variants specific to providing MultiCore embedded debug for system level FPGA's will be released in 2004.

From a verification point of view, MED will integrate in a range of verification strategies. While the focus of much of the verification world has been on simulation based verification technologies, MED provides a counterpoint that focuses on the physical hardware. A common MED environment can be implemented that provides a contiguous extension from co-simulation and system level models of SoC systems, through FPGA and emulation based prototyping, and that can be finally used for diagnostics and analysis of the final hardware. In this area, MED provides the architectural infrastructure for unification of the often fragmented verification and debug capabilities currently available to ASIC and SoC designers.

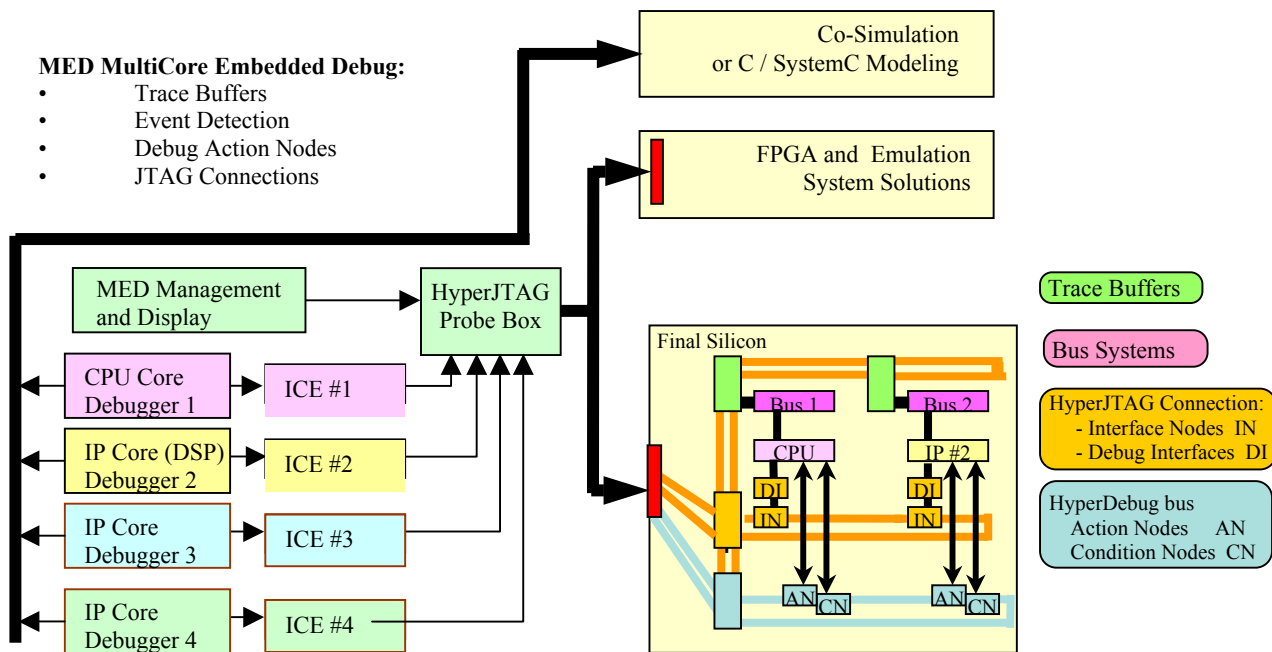


Figure 21 – MED integration in a verification flow

Summary

The need for more advanced debug solutions is arriving rapidly, if not already upon leading edge system developers. While many point solutions for debugging of individual processors or IP subsystems have been developed, the focus is now moving to the system level. As this debug scope increases, the responsibility for debug solutions moves for the processor developer – interested foremost in the debug of their own core to the system developer – with a core and architecture independent focus on diagnostics and analysis of the whole system and all its components.

In this paper we have presented a systems debug approach and architecture called MultiCore Embedded Debug (MED). MED leverages Structured ASIC resources to create a distributed debug subsystem of instrumentation blocks, customized to processors, embedded logic blocks, and embedded buses. This approach provides a "debug backplane" to address dense and complex multi-core systems analysis. By using instrumentation blocks as resources for "embedded intelligent" debug operations, analysis features such as system wide error recognition and filtering, and cross triggering and performance analysis between different subsystems of a complex architecture are supported, which are not achievable with

other debug strategies. The baseline MED environment addresses three major facets of the multi-core debug problem:

1. The need to concurrently access debug and JTAG ports for all the cores in a system. To analyze problems and optimize performance in multi-core operations, the designer should be able to exercise any and all core debug features and interfaces through them. This capability is not supported in current JTAG and debug architectures. Hyper-JTAG is presented as an interface that transparently interfaces to the JTAG or debug ports of four cores and enables (native) debug tools to concurrently communicate with the cores.
2. To address system level triggering, HyperDebug is presented as a debug, trace, and triggering environment supporting multiple on-core and inter-core conditions) and global actions to all or a subset of the cores on an SoC.
3. Since Complex systems communicate over a range of buses, it is important to be able to monitor signals on the embedded buses, and to trigger on and trace bus operations based on specific conditions. This system bus monitoring is discussed as Bus Navigator™, a system trace that supports widely used on-chip bus schemes.

On-chip intelligent instrumentation, especially as implemented in system level architectures and platforms is needed to address features and tradeoffs needed for MultiCore Embedded Debug architectures. MED is an initial step towards this new class of diagnostics and analysis solutions. *For more information see www.fs2.com*

[1] “In-Circuit Emulation A powerful hardware tool for software debugging” By Robert R. Collins

[2] “Processor and System Bus On Chip Instrumentation” Proceedings of 2003 Embedded Systems Conference, April 2003

[3] “A Reconfigurable Bus IP For Modular Function Integration”, Proceedings of 2003 International Signal Processing Conference (ISPC), April, 2003

[4] “ Multi-Core SoC Platform Integration using AMBA”, Proceedings of DesignCon 2001, Feb. 2001

MED (MultiCore Embedded Debug), HyperJTAG, HyperDebug, OCI, and Bus Navigator are trademarks of First Silicon Solutions, Inc.

All other trademarks referenced are the property of their respective companies.