

ESL Driven Instrumentation Interfaces

Dr. Neal Stollon
neals@hddynamics.com

Dr. Mark Burton
mark@greensocs.com

Abstract:

ESL modeling and system analysis are being used at the conceptual front end of SoC design to improve functional and performance analysis, and allow parallel hardware and software development. Similarly, increasingly amounts of instrumentation IP and logic-based analysis is being used to facilitate system verification and performance analysis of physical (FPGA and ASIC) implementations. This paper discusses commonalities in requirements, implementation approaches, design flows, and analysis tools at both the ESL and instrumentation based analysis stages of a design. We present several approaches enabling more common analysis tools and interfaces to allow better integrated ESL and instrumentation applications for complex chip level systems analysis and discuss standardization efforts from OCP-IP and Nexus 5001 industry groups that support these goals.

Author(s) Biography

Neal Stollon is Principal Engineer at HDL Dynamics, an instrumentation consulting and IP development group, focused on SoC on-chip instrumentation and analysis solutions. He has over 25 years technical and business experience in digital design, processor architectures, and tools development at MIPS Technologies, Texas Instruments, LSI Logic, Alcatel, and others. Dr. Stollon earned a Ph.D in EE from Southern Methodist University and is a Texas Professional Engineer. He has written over 30 technical papers and holds 10 patents. He can be reached at 973 816 7384.

Mark Burton is the founder of GreenSocs, the open source ESL infrastructure project. He is the current chair of the OCP-IP SLD working group, and has previously chaired the OSCI TLM working group.

Introduction

Most engineers involved in leading edge design will agree that verification and validation of system-on-chip (SoC) level architectures has become a critical path stumbling block in development and release of new devices. This is now equally true of the software components of those systems as the hardware. There is an ongoing need for more and better ways to address the verification and analysis of complex SoC designs, which is being addressed with a corresponding evolution of new methodologies, tools and capabilities to get the job done at different phases of the design and development cycle. This paper discusses integration of embedded instrumentation as part of the system verification and analysis process and discussed both approaches and methods for complementary integration of EDA, and in particular at the ESL (electronic systems level) tools to add value and closure to the SoC verification and analysis problem.

The central thesis of this paper is that we require better approaches to address the problem of integrating pre-silicon and in-silicon analysis and verification components. We believe this will address the rapidly advancing inequality between complex and sophisticated new silicon systems (including both hardware and software components) versus the tools available to support their construction, at a viable cost and within a reasonable time to market.

It is appropriate to define the scope of some of the terms being used; ESL and instrumentation are both applied to a range of technologies, with varying relevance to the title of this paper. ESL (Electronic System Level) in our context refers to the ability and tools used to model, analyze, and debug systems containing both hardware implementations (including but not limited to processor subsystems) and the software components used with them, together forming an electronic system.

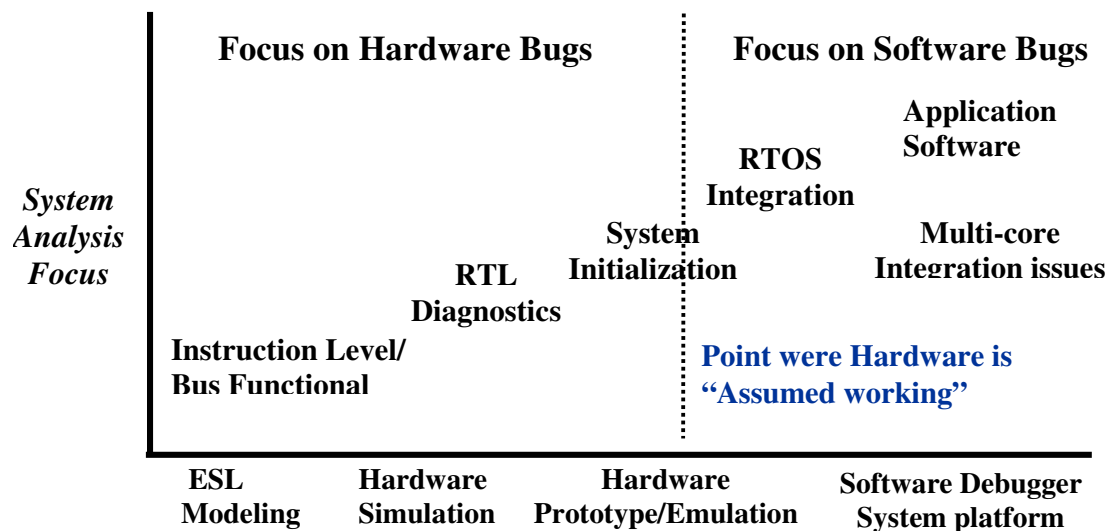


Figure 1: Verification Abstraction vs. Debug Tasks

The challenge and importance of ESL has risen in recent years, driven by the size and complexity of the software component in many systems and the increase in multi-core architectures that increase the complexity of multi-processing software and stress the limits of existing software tools automation and flows. Arguably, software development capability has fallen behind hardware capabilities in the ability to design multi-core

systems; there are still limitations on viable general compiler based methodologies that supports multi-core design today. With a less robust compiler technology for these complex systems, the need for validation expands compared to previous generations of systems and the requirement for instrumentation to support this analysis increases. Various ESL languages are in use, but arguably the most dominantly emerging ESL language in the electronics community is SystemC [1]. In many cases in this paper, assumptions regarding ESL can also apply to RTL oriented behavioral languages such as VHD and (System)Verilog. In other cases, specifically where debug of software based systems is required, SystemC has advantages due to it's implementation as a C++ class library, rather than as an independent language. Among many methods proposed and used for ESL, approaches based on SystemC and related methodologies that extend the software development analysis world into the hardware domain have developed more traction than hardware analysis methods being applied to the software domain. In addition to the software development component, a significant effort today is focusing on direct synthesis from SystemC ESL models, bypassing in some cases use of RTL representations. This represents a paradigm shift in the development of digital electronics, in having a common language and tools platform that concurrently support digital systems, processor cores and software, but as with any emerging technology, the true costs and benefits will be seen in results in reducing development efforts and improving quality. However, while the hardware centric industry is just now beginning to come to grips with design using languages such as C++, the software world is moving on to higher level notations such as UML, in order to increase their productivity. This further "leap" is still a matter of research for hardware developers.

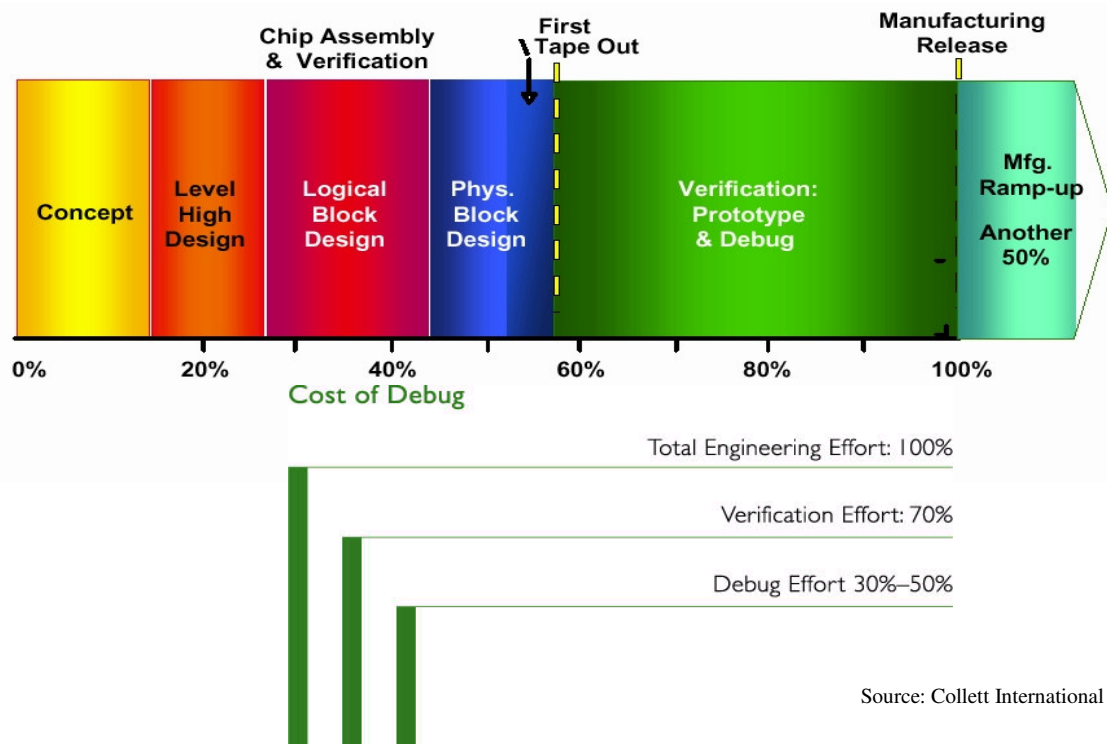


Figure 2 : Cost of Debug in the design process

None the less, by implementing better linkages between the tools and languages for ESL design and those of the hardware and software implementation, we see an opportunity to

have more unified flows between various levels of debug and verification. One of the primary methods used today in verifying hardware systems is the addition of instrumentation blocks to increase the controllability and observability of a hardware based product. Instrumentation in this case refers to a toolbox of methods, IP, and tools to support analysis of hardware based systems at varying levels of abstraction and application. Instrumentation at the ESL level includes combinations of software tool methods (which can be thought of as “print” statements and breakpoints) and EDA methods (monitoring of events using assertion based approaches and capture of sequential information in EDA formats). There are several design challenges: It is often hard to choose instrumentations that make sense, and are available at different levels of abstraction, hence the ability to reuse these approaches consistently as the design moves through different levels of abstractions and different levels of a design process is often limited. None the less, designers seek to provide such information. This often limits their choice of instrumentations, and therefore the possible debug, analysis and verification advantages some levels of abstraction could offer.

This paper is fundamentally about questioning those design decisions, and suggesting that instrumentation should be added more liberally, and more appropriately at all levels of abstraction to better serve the needs of system debug and verification. If instrumentation that can be implemented easily at a high level of abstraction proves vital for software debug, it may well be worth considering the extra gate cost of including the same instrumentation at the RTL level.

The question of cost of debug in the development of complex SoC systems has not received the level of analysis that other parts of the methodology, such as EDA tools use, but is pragmatically known to be a significant portion of the overall cost of releasing new systems (Fig. 2). EDA tools and flows have focused on evolving a variety of solutions to address verification and debug for *pre-silicon* design, with diverse simulation based methodologies that leverage High level verification languages and formal and assertion based methods to verify at increasingly levels of system level abstraction. These approaches are in large part compatible with both ESL based design and to a more limited extent with instrumentation based debug.

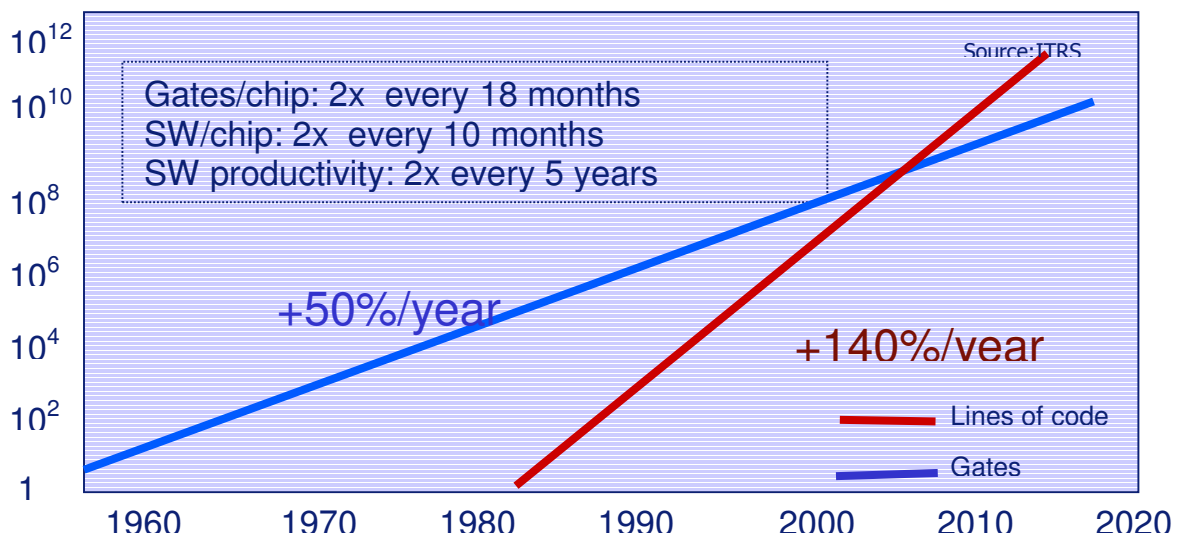


Figure 3 : Growth of Hardware vs. Software Complexity

Progress however is still limited by the level of integration and reuse between the pre-silicon and in-silicon part of the verification process. In particular, two rapidly increasing dominating factors are the increase in chip size, which is mutually driven by increasing numbers of processors and other subsystems to achieve System on Chip status, and a corresponding increase in the size and complexity of software, often being developed for more complex multi-core, multi-threaded, and multi-functional processing architectures. Figure 3 shows one analysis of the potential impact of this to next generation requirement for system debug. While development flows obviously vary significantly among companies, the problem of extending and uniting verification flows between pre-silicon and in-silicon activities significantly impacts the engineering effort associated with verifying software content in the design. Current solutions to provide some level of unification have largely been ad hoc and specific to a particular company, program, and tool suite.

All too often, verification flow focuses under an assumption that the verification effort is largely done when the design files are handed off to the foundry, while engineers who are involved in the *in-silicon* debug cycle, loosely defined as everything that must be verified and integrated from the time that silicon is received back from the foundry to the point of being ready for a production release, know that this is far from the case. While accepting that improved tools and rigor in pre-silicon verification are key to better quality of design and discovery of fundamental problems and errors, as the rates of functional first pass silicon have increased, much of the problem solving and validation shifts to more subtle, environmentally constraints analysis and bug-fixing, including resolving hardware/software issues that can not be addressed at speed other by an in-silicon hardware platform itself.

This is a point in the flow where instrumentation comes into play as an important analysis and system validation method, assuming that it has been properly considered and integrated into the silicon hardware (and, or the ESL level models). The value of instrumentation is, directly and indirectly, a function of several factors, which include the resources inserted on-chip or the simulation speed cost of the instrumented code, the overall applicability of the instrumentation and level of software and tools support that is available to make use of the instrumentation. In looking at the differing types of on chip instrumentation, they roughly break out into 4 major types of functions.

- Core Debug – most processor IP includes some debug blocks that simplify run control (go, halt, single step,...) and optionally provide instruction and data trace (MIPS EJTAG and ARM ETM examples). The core level integrated debug blocks and debugger features can differ significantly from processor to processor, where there are few industry standards, Nexus 5001 being a notable example we discuss further on.
- Logic Debug – For more generic control and trace, IP that essentially allows the embedding of a logic analyzer on the chip provides major flexibility and visibility into the IP operation by enabling trace of deeply embedded signals.
- Bus Debug – embedded bus fabrics such as AMBA and OCP present additional challenges for system debug due to complex interactions and the sheer amounts of data transferred over bus channels, requiring instrumentation that supports specialized filtering and complex triggering to allow focus on only the data of interest.
- System Cross Triggering – for multi-core systems, a comprehensive multi-core view and the ability to monitor events from different cores and to send control actions to different cores is required to synchronize and manage the complexity of multi-core debug. Cross triggering instrumentation provides one flexible means of controlling and coordinating the concurrent operations of several cores and IP.

The number of specialized and customized instrumentation blocks to address analysis such as system or core performance analysis is even larger. As important as the instrumentation function, is its integration and communication with other tools and user interfaces. Many instrumentation systems utilize JTAG as a primary debug interface. Others use more specialized and higher performance debug access ports, such as Nexus. The ability to seamlessly interface different instrumentation blocks to different debug tools requires a sophisticated hardware (probe) and instrumentation software environment that supports the requirement to service diverse and concurrent debug requests.

Evolution of Debug and EDA capabilities has progressed along parallel, but separate, paths for much of the history of electronic systems. Historically looking at the major inflection points for EDA verification, debug tools, and silicon complexity in Figure 4, it is interesting to note the interwoven relationship between these differing but inter-related technologies that are central to the progress of many aspects of the evolution of leading edge electronics technology. While the interrelationships and actual cause and effect between these domains is worthy of a paper in itself, the emergence of new EDA tools can be consistently seen as both a driver and as a result of new and increasingly complex levels of analysis of systems architectures.

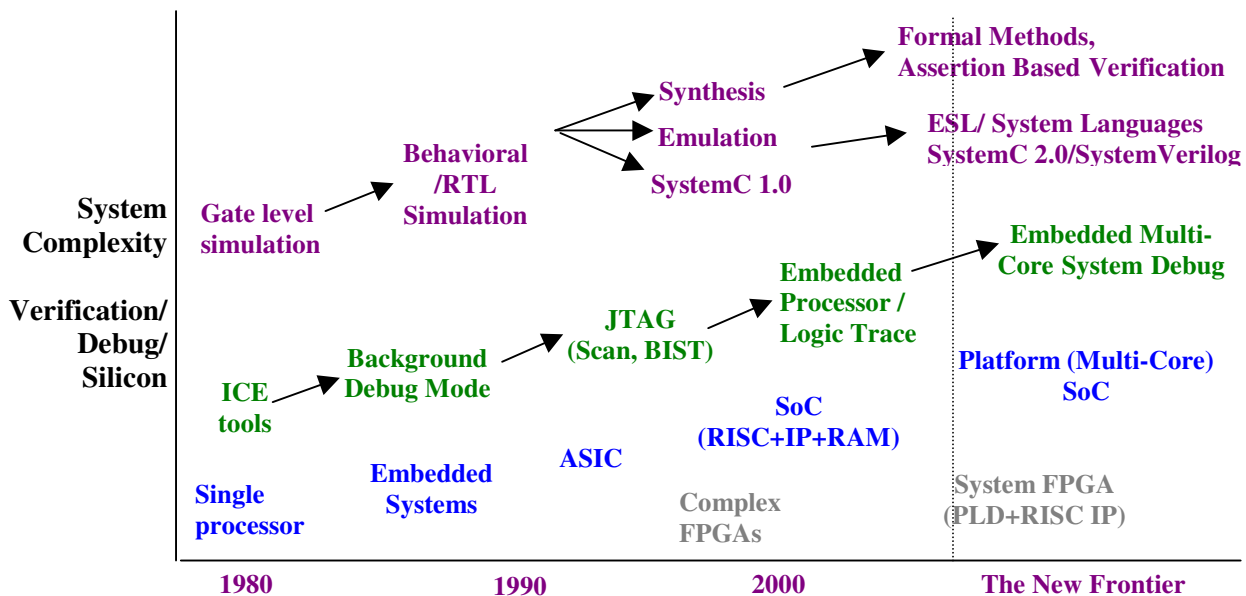


Figure 4 : Inter-related Evolution of Analysis Tools

Similarly, emerging new complexities in architectures have spurred the requirement for new methodologies and capabilities to address the analysis and instrumentation needs of these architectures. We are arguably in the middle or moving towards a new inflection point of requiring a sea change in debug assumptions, based on changing design methodologies that widely embrace multi-processor architectures and their associated software development and integration issues, dramatically increased gate count availability, and increased complexity in all the diverse interfaces and peripherals making up a emerging SoC device.

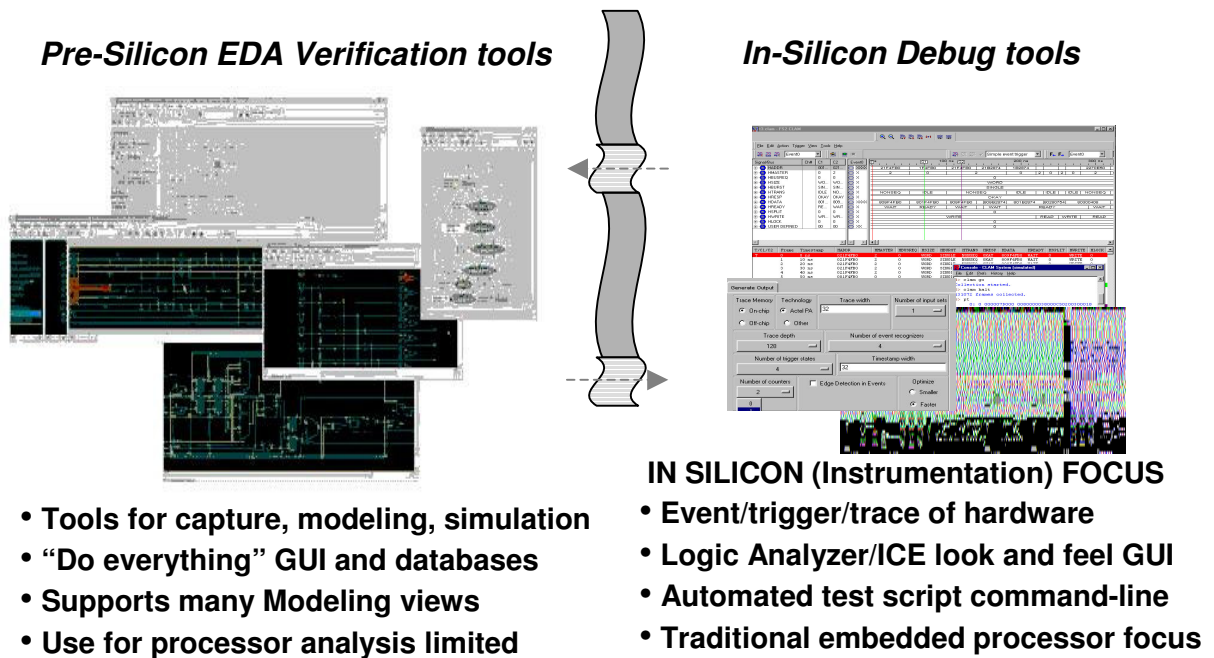


Figure 5 : EDA Verification vs. In-Silicon Debug tools

With an evolving focus in the EDA industry to new tools and methodologies to address this new inflection point in debug complexity, emerging EDA tools include (but are not limited to) System Level Design via a new focus on Electronic System Level (ESL) languages, and the adoption and migration of formal and assertion based methods of verification, to compliment more traditional verification activities. These potential improvements are primarily focused on the pre-silicon design and verification process. New approaches to debug have evolved to address silicon complexity as well. These include the recognition of more widespread adoption of system level debug instrumentation with features such as cross-triggering, time-stamping, more robust trace, etc. to aid and reduce the product analysis and validation. With a much deeper level of available gates on a device, older concerns about the costs of on-chip resources required for instrumentation are dissipating, when compared to the concern with the issues of how to address the necessary debug visibility and controllability associated with the analysis of the silicon.

So, a hardware team, in addition to having to address the issues of verification continuity and model reuse at different stages of the design flow, typically must also now develop (largely separate) debug and analysis flows to address both hardware prototypes and in-silicon verification and on-chip debug needs of their colleagues in the software teams that will be critical for getting electronic systems to market. (Figure 5) There is an opportunity to better integrate these efforts; to mitigate, especially for larger and more complex designs, the penalties of this discontinuity between hardware and software verification and debug and to leverage existing standards for next generation solutions.

SoC Debug Requirements

Analysis, at all levels of implementation, relies on methods of configuration, control, and data capture. Control refers to the manipulation of a system, outside of its normal execution, for the purpose of debug, analysis and verification. The control can be

influenced at any time during the execution of the system, so it is “real time” in this respect. A simple control example might be to execute a single instruction, but more typically it may involve execution of a range or duration of operations either tied to some initializing event and concluding event. Configuration is actually nothing more than a special case of control, referring to the initial set up of a system to a known state. In some cases, this configuration may be part of the normal execution of a system (for instance the default settings used after reset). Examples include setting of mode or configuration bits for cores, arbitration states for buses, loading of data into specific locations in the system to (re)produce a system state for a particular operation or sequence of interest. Data capture refers to the export and storing of some system information occurring at a user-defined time. A simple example would be capture of a register value occurring at some triggering event in the system. Both ESL and instrumentation tools have similar requirements at a SoC level in how to address these 3 tasks, and ideally would rely on standardized mechanisms for implementing them.

Most SoC included some programmable processor, and in many cases, multiple processors as the core functionality. They also consist of infrastructure, either in terms of dedicated co-processors or other logic and a communications infrastructure to allow both inter-core and chip to outside world communications. The analysis of processors and the rest of a complex chip follow different paths and have traditionally relied on different approaches to verification and debug. Digital hardware design, on the other hand, typically relies on a synthesizable RTL (Register Transfer Level) models that assume implicit clock-cycle timing during simulation. RTL has been the primary debug tool for configuration, control, and data capture of dedicated logic based portions of the architecture, with hardware support based on either on or off chip logic analysis, although with the advent of synthesizable ESL language subsets and methodologies, these functions may be absorbed into the ESL level of design flow. In either case, merging simulation and synthesis approaches have been proven over countless designs over the last 15 years, since logic based functions are typically able to be analyzed over the range of less than a million clocks cycles, which are manageable for both simulation and logic analysis. Processor architectures conversely, while relying on synthesis for implementation, are less successful in using RTL and logic analysis approaches, due to length of time required for execution for complex algorithms and complexities of hardware /software interactions that are not amenable to RTL simulation and related approaches.

Simulation is always an important part of the development flow, however just as important is the ability to analyze hardware during prototyping and system verification and, increasingly, on the final products themselves. While the focus of much of the verification world has been on simulation based verification technologies, instrumentation provides a counterpoint that focuses on the physical hardware. The problem in analyzing embedded information (on chip buses being a typical example) in hardware in many cases devolves to a visibility problem— *it is difficult to fix problems that you cannot see*. The test community traditionally has referred to this problem in terms of levels of controllability and observability of a design. It is important to note that this analysis visibility cannot be adequately addressed by traditional on-chip test methods such as JTAG scan, and that the analysis and instrumentation problem, whole overlapping with test issues and techniques in many cases, is fundamentally different.

One method of working around the analysis bottlenecks in simulation is to build hardware emulators or prototypes of (usually field programmable) hardware implementations of the digital and possibly the analog portions of a system. These hardware systems will run

orders of magnitude faster than simulations, making running of software applications feasible, but are still typically at least an order of magnitude slower than the final silicon system, which results in both false positive (errors in the emulator that are not in silicon, due to differences in timing paths, synchronization of subsystems, etc.) and negative problems (found in the silicon that are not seen in the hardware, due to lower speed) while still in many cases not being able to run the system application at a speed compatible with the final system requirements.

Modern silicon systems level implementation typically proceeds through a design life cycle of increasing detail and refinement that must include modeling, verification, and analysis of hardware and software components. Software development has typically relied on analysis with a hardware target using ISS (Instrument Set simulation) models where timing is abstracted or non-existent. These ISS models can vary significantly from vendor to vendor, which inhibits general methods for model compatibility both between different core models and their integration with RTL simulation. While RTL is synthesized into gate level implementations that map into hardware and becomes a deliverable product, along with software that is either embedded as part of the product, or added at a later stage by customers. More complex modeling is complicated in modern devices by several factors.

- Preferred software development environments may vary significantly for different processors. While hardware development tools have developed in parallel but largely independently of different ways to implement a design (programmable logic, ASIC and ASSPs and their related IPs) a limited number of common representations (gate level, RTL, hardware itself) allow for straightforward integration, software development tools and models are developed by and in conjunction with processor and software IP vendors and have more limited commonality, for modeling and verification of multiple processors or even different processor targets running a common algorithm. The problem is even more acute for debug related tasks, where different debuggers having different features. More commonality is found in use of GNU Debuggers (GDB), versions of which have been developed by and for many processor architectures. GDB and variants are commonly used as a user interface for configuration, control, and data capture of software architectures during ISS, emulation, and in-silicon debug and will be discussed as a vendor neutral debugger approach later in this paper.
- For multicore devices, different suppliers often provide subsystems, both in terms of hardware blocks, each developed with their own assumptions and incompatibilities in ISS modeling. Due to a lack of standardized sequential timing models in software languages used to develop ISS models (C, C++), new modeling approaches that include understanding of sequential and concurrent operations is needed for modeling of systems that include multiple processors (running under differing clocking and instruction flows) and processors and their supporting blocks (buses, peripherals, interfaces, etc.) that are typically modeled at RTL or other hardware methods. The dominant language with native abilities to concurrently model both processor architectures and software and supporting hardware blocks is SystemC, which combines compatibility with C++ as a class library with a set of corresponding modeling and simulations features to those used in RTL.
- Real device speeds are higher (typically by orders of magnitude) than that achievable by simulation. As a result, system modeling relies on abstractions and simplifications increase simulation performance to a point where it is feasible to run software applications over the multi-millions of cycles needed to verify operation. Complexity

and performance are further impacted if different subsystems are asynchronous or have other analysis intensive incompatibilities. The lowest risk and often simplest solution to real time analysis is to use the actual hardware, however, even with added instrumentation, there are significant limitations in observability and controllability of a design as previously discussed, so while hardware is a good verification platform, it is limited as an analysis platform. Simulation does not have the same limitations, since all signals are visible. One of the more important simulation efforts of SystemC is related to tradeoffs between speed and visibility with abstracted TLMs (Transaction Level Models) that by abstracting away non-critical functionality or timing, can simulate orders of magnitude faster than cycle timed RTL models while being able to be integrated with RTL level models. By adjusting the level of integration between TLM vs. RTL blocks in a simulation, any level of resolution of signal analysis can potentially be supported at the expense of increased simulation timing.

- Complicating simulation analysis further is the modeling of the complex environments that the device must operate in. These can include the need for modeling of complex stimulus with both signal and noise characteristics, human interfaces, and analog subsystems that have their own modeling and analysis complexities, which are incompatible with large systems digital analysis and have their own traditional (frequency domain based) analysis methods. The effective integration of mixed analog and digital systems remains an open area of refinement in EDA analysis methods and in hardware based debug and analysis, test features within ESL tools include the ability to model many analog and systems characteristics as part of verification blocks (test benches) as well as the ability to integrate models from verification level languages (Specman, Vera, Testbuilder etc.) that have been developed and which are being incorporated into new revisions of RTL languages (ex. SystemVerilog).

Choosing between many design tradeoffs efforts is a tiered approach of modeling refinement and migration from ESL, to more detailed models, to hard platforms, to final silicon. As the modeling and analysis moves from simulation to hardware, another factor to consider is an accompanying loss of visibility and access into the internal signal operation. In simulation, all signals, variables, and modeling parameters are available for viewing and in most cases, for direct modification, providing a rich analysis environment, regardless of other limitations. Hardware, whether in emulation or more pronounced, in final silicon, has a limitation of the amount of visibility and control of embedded signals that are available at the system IO pins (Figure 6). In this case, instrumentation takes on a new role and importance, since the amount of real time visibility and control of the design is directly related to the effort involved in adding analysis blocks to a design, either to provide trace, to directly configure, take direct control of, or inject some stimulus into a subsystem.

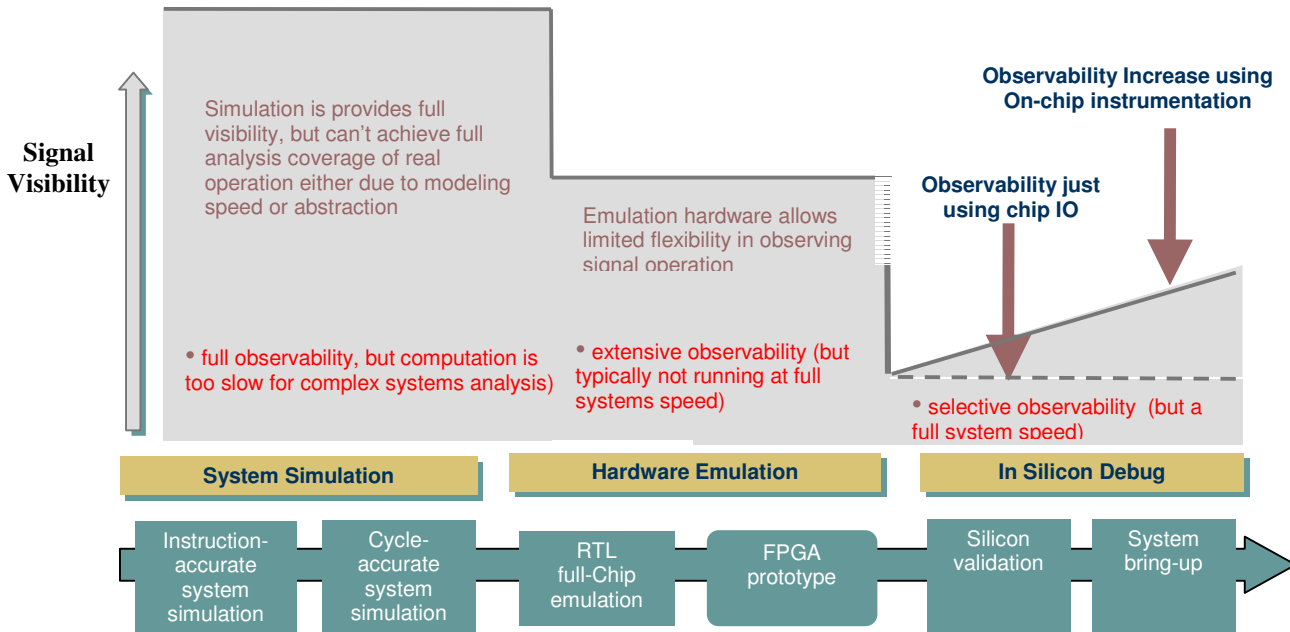


Figure 6 : Observability during design flow stages

In recent years, this level of instrumentation design has taken on a specialization of its own, referred to in different contexts as On-chip Instrumentation, Design for Debug (DfD), etc. A flow of debug and analysis tasks that can be provided using good instrumentation are shown in Figures 6 and 7, consists of several diverse independent and interdependent activities required to address different aspects of verifying an in-silicon product. Design for Debug methodologies are still emerging areas of investigation. DfD differs from DFT and related approaches in the level of customization required to support specific debug requirements of an architecture or system. With many devices often consisting of both processor and fixed IP, along with related software and firmware, the verification concern is not only *operating as designed*, but is *also performing as required* in its natural environment. For many products, this may include exercising and verifying operational scenarios that were not foreseen or not feasible to include during the pre-silicon verification cycle.

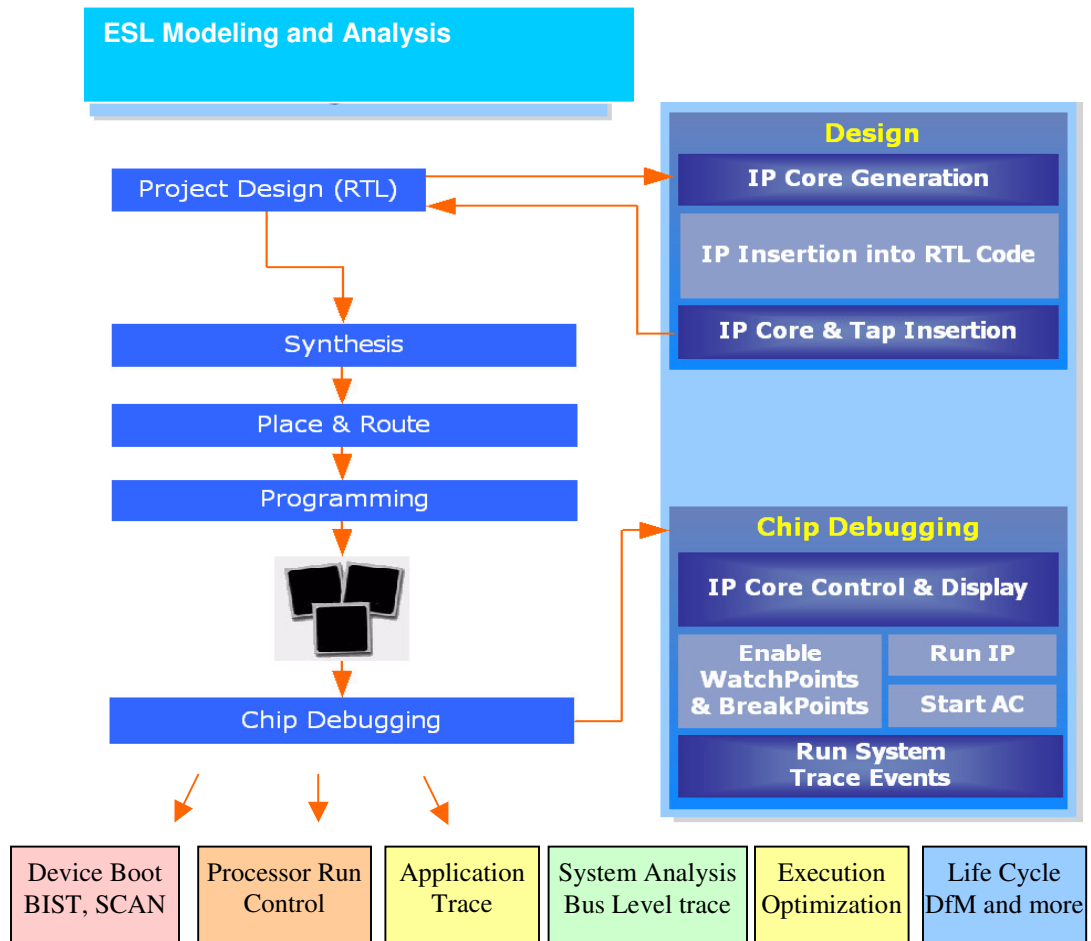


Figure 7 : Debug Activities in the In-Silicon Verification Flow

Ongoing Efforts in Debug

In this section, we discuss some of the ongoing work in developing standards based ESL driven instrumentation solutions. The world of ESL development, in particular work in SystemC is developing in a range of areas, and the reader is advised to visit several websites that provide information on SystemC and other ESL tools, and methodology [2,3]. A variety of system level debug issues applicable in both ESL and instrumentation are being addressed through both commercial activities and standards groups. There are two activities have the most direct focus on this problem, (though there are other industry and standards working groups focusing on other aspects of this problem [4,5]). The activities discussed below are complementary, with the OCP-IP Debug working group focused on the embedded interfaces and integration required for debug of embedded IP and core components and the Nexus/IEEE 5001 activity focused on the chip level interface and tools integration and communication. GreenSocs (www.greensocs.com) has expressed an interest in providing equivalent interfaces for ESL models. Fig. 8 shows this relationship as well as the technical positioning of some of the other industry related on-chip instrumentation and debug tools work being done concurrently.

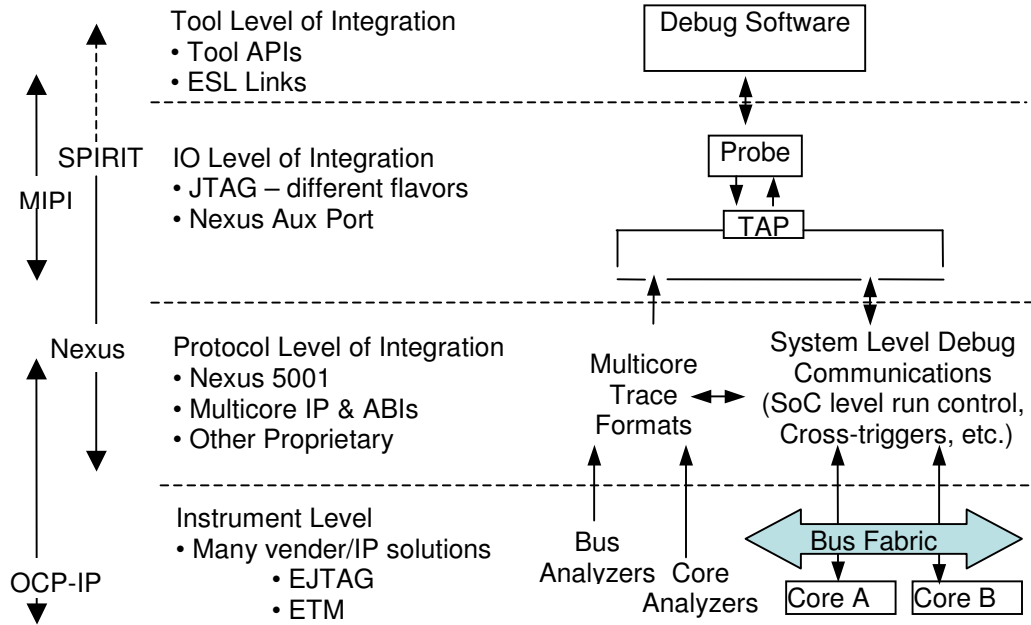


Figure 8 : Debug Standards Group Efforts

OCP-IP Debug Activities

The Open Core Protocol International Partnership (OCP-IP) is an industry based standard group defining vender-neutral socket interfaces for interconnecting cores and other on-chip components. [6]. The OCP-IP socket based integration strategy has been proven out in a multitude of designs over the last 5+ years. OCP-IP's strategic focus differs from other socket-based interfaces as it address to a wide range of related topics including System Level Design, Debug and others.

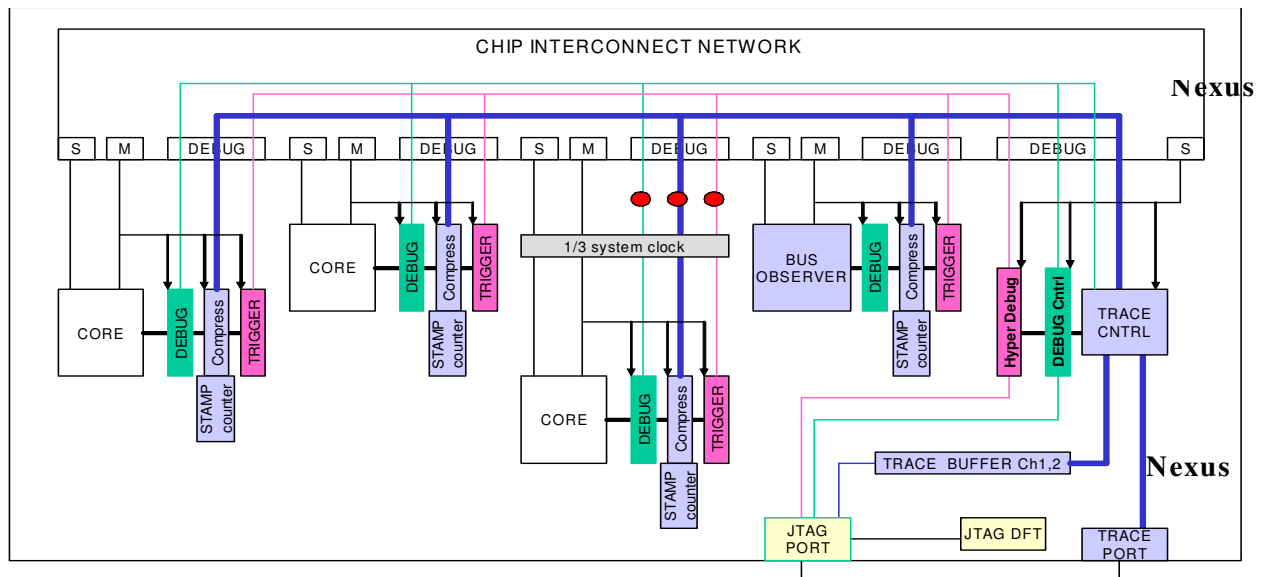
With increasingly complex multicore System on Chip (SoC) silicon and applications being developed by OCP-IP members, one area of OCP-IP standardization is in both interfaces and processes for on-chip instruments and their integration. To address these requirements, OCP-IP formed a Debug Working Group in late 2005, to investigate and propose on-chip analysis solutions. The membership of this group has included debug IP providers, ESL tools developers, and SoC OEMS.

Historically, the competitive nature of different IP providers have limited their collaboration. The commonality of interfaces and methods for analysis needed for complex SoCs have been correspondingly limited. There has been a lack of open SoC debug solutions that could support diverse architectures. The problem comes to a head with more complex SoCs with heterogeneous (often from diverse venders) IP, and increased analysis and optimization requirements for

- increased 1st pass design success by improved prototyping,
- reducing post silicon verification and time to market,
- optimizing hardware/software interfaces and operations,
- complementing and validating documentation and
- validation of correct HW+SW functionality by observation of internal chip events

The Debug WG has initially focused on 3 areas generally considered to be critical to having an in-silicon debug solution, the summary of which is currently available [7];

1. Defining a critical set of debug functions that could be implemented on-chip to include global run control signals, as well as trace, triggering, timestamping, and other on-chip analysis functions supporting modern SoCs that incorporate asynchronous domains, diverse voltage islands, various power saving schemes, varying levels of embedded security, etc.; all of which add to the complexity of a debug solution. The on-chip debug functions may be supported using either IEEE 1149.1 JTAG serial interfaces, dedicated debug access (parallel) ports such as IEEE 5001 Nexus interfaces, or as custom memory mapped functions that could be supported by embedded processors. (Figure 9).



By defining a standard flow and IP for extracting signal information from hardware, corresponding models and methods in the ESL domain can be developed to simplify viewing of common signal formats and sequences. This has the dual advantage of extending reuse of test methods used in ESL to hardware analysis and allowing cross-verification of the ESL golden model and the final hardware implementation.

2. Defining both critical and optional sets of debug signals that would allow different IP blocks to be able to communicate and coordinate their specific debug requirements and features. The critical signals are those typically common to all the IP blocks and would be supported by a common JTAG chain or debug port. Optional signals are those supporting functions specific to particular blocks or asynchronous/secured/powered subsystems. A common schema for debug signals and blocks would allow simpler development of models of hardware and supporting schemas for APIs that could support both ESL and instrumentation data extractions and processing.
3. Defining common tools and system level interfaces and schemas that support more commonality and integration between pre-silicon and on-chip analysis. Working on coordination with the OCP-IP System Modeling working group, there is a common focus on common ESL and Instrumentation hardware requirements such as triggering and

breakpoints, and common software, API, and tool database requirements between different vendor solutions (Figure 10). For processor based architectures, triggering and breakpoints, one common method of control and configuration could be using GDB, which is a widely used, public debugger tool, supporting many (most) processor interfaces and interfacing with many tools environments.

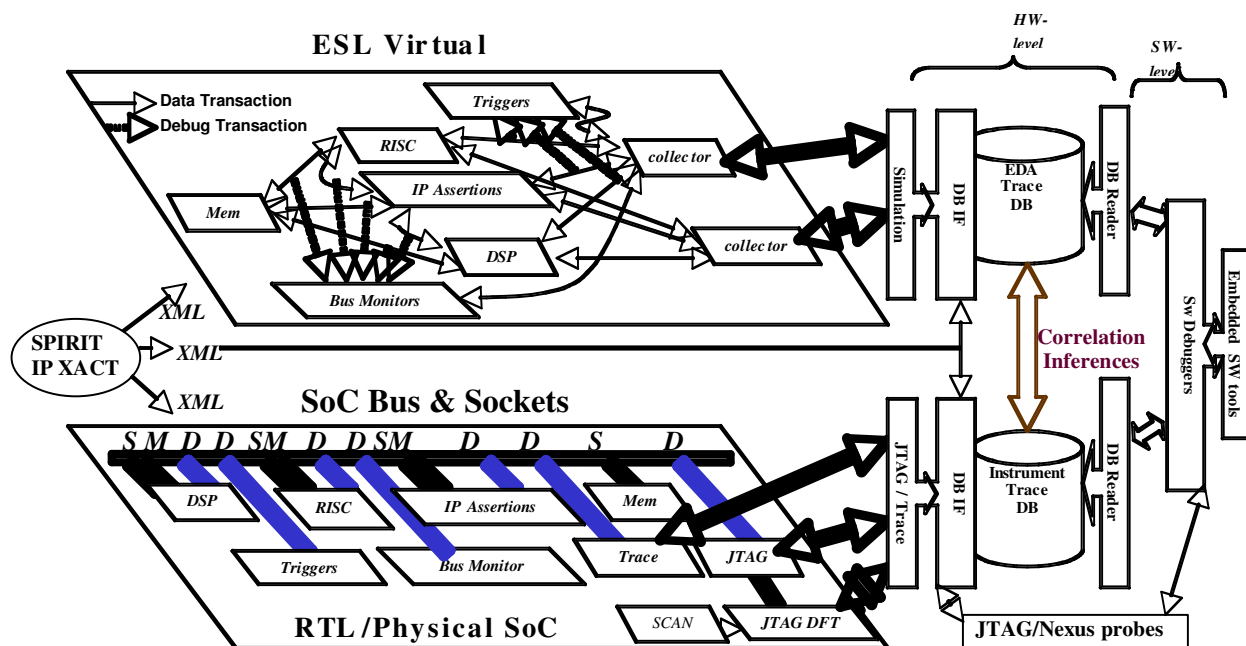


Fig. 10- SoC ESL and Physical Instrumentation Based Debug Architecture

OCP Debug Working Group has coordinated with a sister OCP-IP SLD working group, which has been a leader in developing System Level models and methodologies for simulation based analysis of SoC architectures (with assumption of, but not limited to, integration over an OCP architecture). OCP-IP provides access (to its members, and for research purposes) to one of the more comprehensive public SystemC TLM models of bus architecture components, which can be used with freely available SystemC simulators [8]. Also provided to OCP-IP members are “Corecreator” modeling tools that supports building of OCP interfaces and bus functional models (BFMs). These modeling can include different instrumentation blocks that are used for SoC event monitoring, triggering, signal injection, trace, etc.

The combined effort of the OCP Debug and SLD working groups provides much of the infrastructure needed to implement system analysis and is available for adoption and integration. Ongoing work is starting to address the interfaces and methods to allow ESL analysis of debug functions and signal interfaces needed for OCP design, which include deeply embedded bus level signals, defined at the core and bus fabric and interface levels, whose specific functional and performance information critical to debug is difficult to access. The interfaces between these embedded subsystems and the outside world is being defined with an emphasis on currently utilized and standardized interfaces.

Nexus System Activities

Nexus 5001 Forum is an industry based standards group that manages the IEEE 5001 (Nexus) Debug Specification [9]. The Nexus 5001 activity was initiated in 1999 as an extended and inclusive specification based on the Global Embedded Processor Interface Forum work to address a standardized interface for on-silicon instrumentation and debug tools providing expanded features and higher performance. Nexus infrastructure supports multicore development and multi featured trace and configuration/control. Nexus at its simplest level is compatible with JTAG, but recognizes that the limitations in JTAG bandwidth are not realistic for the debug requirements for complex or multicore environments. The current version of the specification was released in 2003. Since specification release, versions of Nexus architectures have been used extensively in US Automotive applications and more chips have been produced incorporating Nexus ports than any other non-proprietary, debug specific interface.

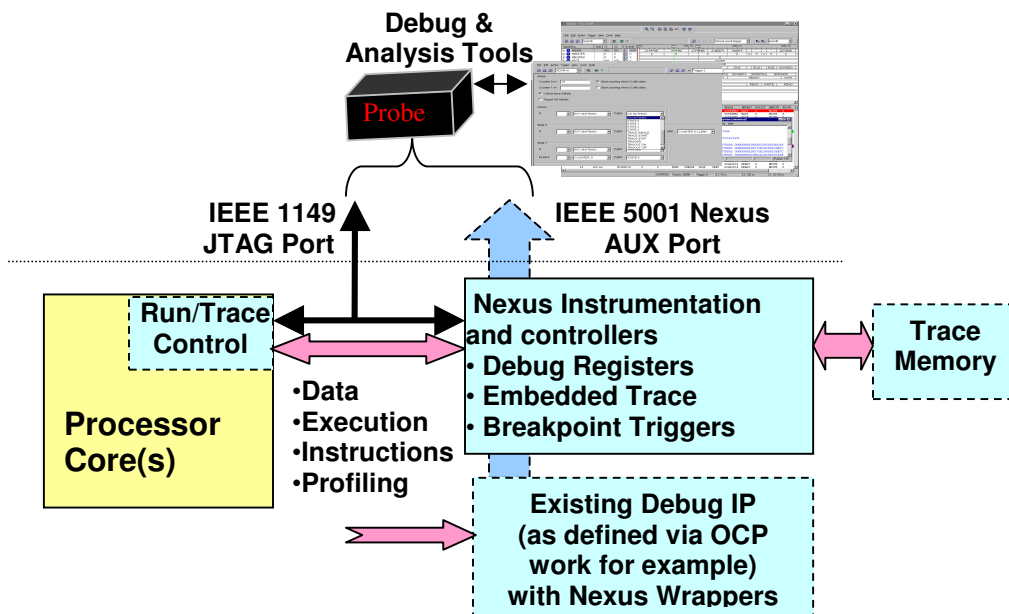


Figure 11 : Nexus Interfaces

The Nexus specification defines a vendor-neutral IO signal interface and communications protocol that supports parallel debug and instrumentation support (Figure 11). The Nexus interface defines a small set of control signals and auxiliary (AUX) data ports that may be used in conjunction with JTAG or as a self-contained port. The additional data pins provided by the AUX interfaces are scalable for matching of the debug requirement, allow much higher read/write throughput between the target and debug and analysis tools compared to JTAG.

The Nexus architecture differs from other in-silicon debug schemes by openly defining high performance data interface, protocol, and register infrastructure that can be used to implement a variety of trace and control instrumentation.

The AUX interfaces are uni-directional (either Data In or Data Out), with each AUX port having its own clock. The Data Out pins of an AUX interface is typically used for trace, and the Data In mode is typically used for configuration or calibration of an IC. AUX Data In

and Out ports may be operated concurrently. Nexus also specifies how a JTAG interface can be used in conjunction with the AUX ports. JTAG interface operations in Nexus may be used both for configuration and control of the on-silicon instrumentation and for embedding Nexus protocol and data into a JTAG message. Both AUX and JTAG interfaces are controlled by FSM based controllers allowing a variety of transfer operations.

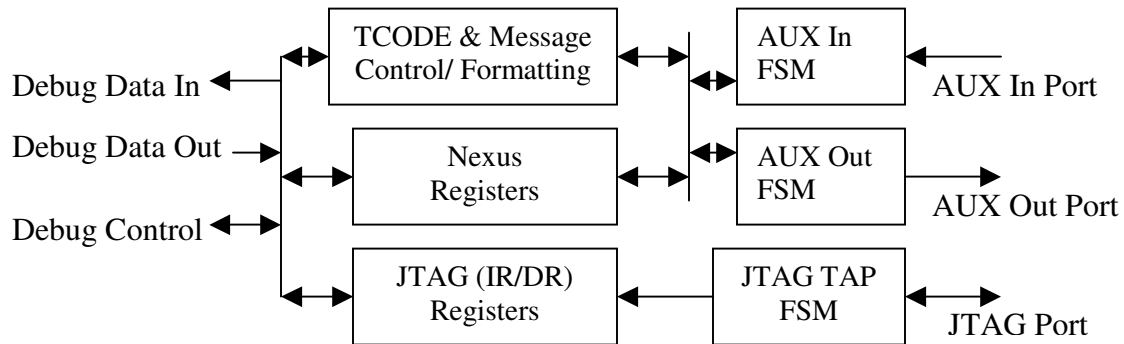


Figure 12: Nexus Internal Architecture

Nexus architecture was designed based on a packet based messaging scheme, which supports debugging complex multicore systems and control of multicore debug processes using a transaction protocol (TCODE) that allows data to be sent in packets, using a packet header to provide information on the source and assumed destination of the data on-chip components as well as information on the subsequent data packets containing trace or other information. This simplifies interleaving of multiple trace sources and concurrent communication with multiple Nexus instruments. The Nexus specification defines a standard set of TCODEs for common identification and trace operations; the TCODE protocol is also extensible to user defined debug commands (Figure 12)

Nexus also defines a standard set of debug related on-chip registers, which facilitate the identification and interface to different cores and sub-systems and multicore control and debug operations. A standard register set allows simpler integration and control of the instrumentations with embedded debuggers and related tools.

The Nexus 5001 Forum has ongoing work on collaboration with other industry debug related efforts, including OCP-IP and MIPI and is in process of extension of the IEEE 5001 specification to support emerging debug interfaces such as SERDES and 2-wire JTAG (1149.7) ports to address diverse debug requirements.

Approaches to an ESL and Instrumentation Debug Infrastructure

In this section elements of a debug environment common to both ESL modeling and hardware instrumentation are discussed. We have developed a Debug Reference model, based on [10] where we can see that in a properly designed debug environment, many of the upper layers (from level 4 and up) can be common between ESL and instrumentation based debug, with the implication that it is feasible implement common analysis environments and frameworks regardless of where the analysis data is created from ESL tools or frameworks. For some cases, such as visualization, this is already in place.

Open Debug Interconnect model

Implementation Layer	Typical Tasks	Location
1. Physical Port Layer	JTAG/Nexus TAP IO, chain and debug block wires JTAG/Nexus TAP FSM (schematic level connection)	Target
2. Data Control Layer	Low level JTAG/Nexus commands and registers, Extended debug instructions, Optional Debug block registers	Target
3. Debug Driver Layer	Debugger Protocol, clocking (probe specific API)	Probe
4. Data Transport Layer	APIs debug command sets, run control API	Host PC
5. Session Control Layer	Device connection setup & parameters, Remote debug server ex. GDBserver,	Host/PC
6. Debug GUI Layer	Debugger UI, GDB commands, trace viewers (ex. VCD) Set/observe breakpoints, watchpoints & event triggers, Run control go/halt/single step	Host/PC
7. Application Layer	Eclipse, other IDE, global (Multi-tool) data management	Host/PC

Applications have differing debug requirements, Nexus defines debugger functionality and compatibility over 4 classes of operation. This paradigm appears equally applicable to ESL level analysis methods. Device instrumentation and tools are defined as class 1-4 compliant if they support all of the features defined for a class. Class 1 starts with basic debug functions over a JTAG port, with higher classes that involve more instrument access and system complexity utilizing the AUX port to progressively increase the debug capabilities, such as addition to more complex trace and emulation analysis of processor operations.

- Class 1 provides run control debug features that are common with most processor implementations, including core identification, single stepping, breakpoints and watchpoints and static memory and I/O access..
- Class 2 enables processor execution trace related features including real-time monitoring of process ownership and instruction tracing, along with complex watchpoints and branch tracking flag indirect branches and eliminate redundant addressing information.
- Class 3 supports Data tracing and memory and I/O read and writes while the processor continues to run.
- Class 4 allows direct user control of a processor to execute programs from Nexus port (memory substitution), plus additional features for remapping memory and I/O ports and starting trace upon watchpoint occurrence.

A methodology for ESL Driven Debug

As systems complexity increases, the amount of data that must be analyzed increases at a highly non-linear rate. At many levels, analysis using both ESL and instrumentation approaches rely on methods of configuration, control, and data capture. Ideally these tasks are implemented by standards based methods and interfaces, which facilitate integration and shared innovation, academic research, and application diversity. ESL and instrumentation are both areas where a variety of standards based approaches and pieces

coexist with commercial and proprietary solutions.

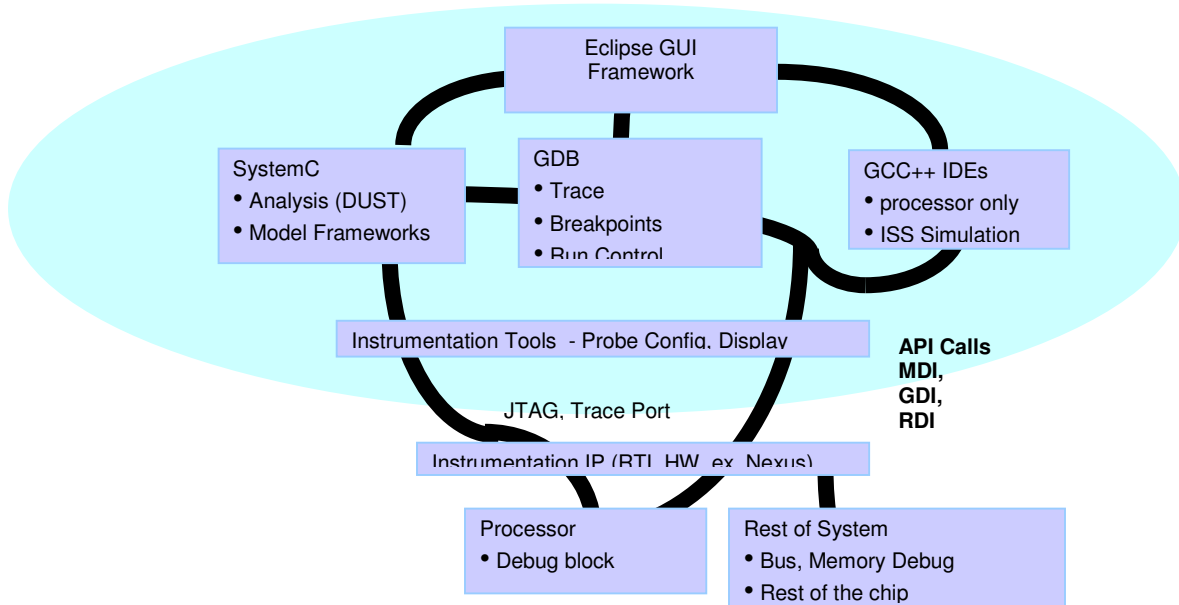


Figure 13 : ESL Driven Instrumentation Interfaces

SystemC, by itself, can lack a rich debug infrastructure; while as a simulation based environment, it provides comprehensive access to signal information, it lacks native tools to address tasks of analysis of modeling structures. Several non-commercial tools have been developed to provide a more robust debug and analysis framework [11,12], in addition to implementations supported by commercial EDA vendors. In the interest of vender neutrality, discussion in this section will focus on open standards based approaches, while recognizing that commercial tools with proprietary interfaces include many innovative and sometimes more targeted solutions. Figures 13 and 16 show flows for an instrumentation and analysis platform that is based on a variety of ESL interfaces.

Assertions – Assertion based analysis is based on the idea of incorporating constructs that monitor for specific events. Assertions are typically structured as true/false statements that check for the compliance to certain conditions and trigger only when exceptions to a condition state or other error state occurs. Assertion capabilities have been available for many software and hardware languages and VHDL models have included assertion statements from its inception and have been a key component in use of Hardware Verification Languages such as SpecMan and Vera. More general adoption of assertion based verification has been facilitated by the standardization of Property Specification Language (PSL) as an IEEE 1850 Standard which support incorporation of common assertions in both RTL (VHDL, Verilog) and ESL (System, SystemVerilog) languages. In the general case, software debug constructs such as breakpoints and watchpoints can be implemented as a special class of assertions. A subset of assertions can be synthesized into the hardware itself, providing a powerful mechanism for event monitoring as triggering. This approach has been implemented in several instrumentation architectures today [13] and provides the advantage of closed loop feedback of assertion effectiveness and reuse for triggering applications (Figure 14).

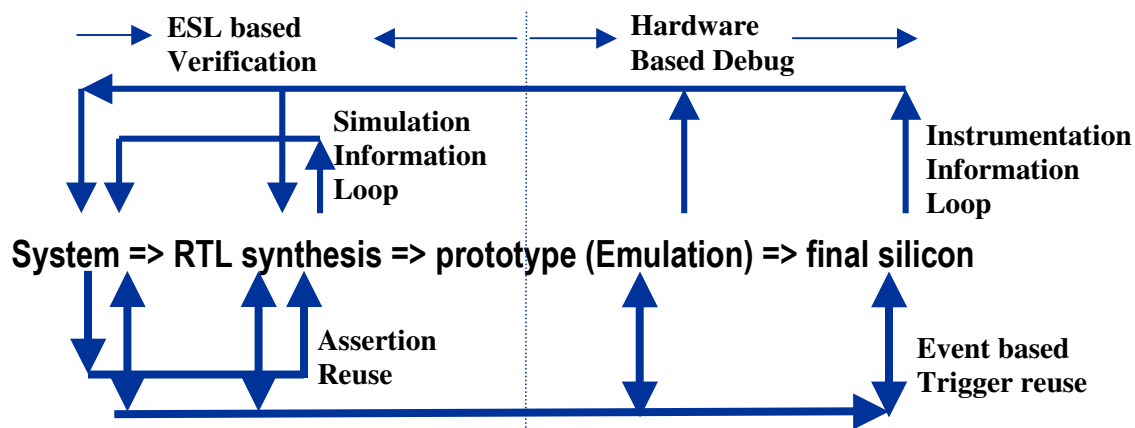


Figure 14 : An ESL / Instrumentation Debug Platform

Run Control – GDB is widely used in both processor ISS and hardware debug environments for a range of debug related control, configuration, and data capture operations including run control, setting and monitoring of breakpoints, watchpoints, and tracepoints, and managing the collection and storage of trace data. While primarily used for processor debug level, these capabilities are also applicable to non-processor logic monitoring and debug. GDB integration is not supported as part of the standard SystemC release but has been integrated in custom environments [11]. GDB is also used in many hardware debug environments for setup and operations of real time run control and trace.

API support – One of the limitations to standardization of debug environments has been the lack of standard APIs to facilitate common interfaces between different tools environments. Having common and open APIs that support both software and hardware tools interfaces would simplify many of the tasks involved in ESL and instrumentation tools development and integration. A wide number of tool vendor specific API's exist today that support debug of different processors, but only under single vendor environments. Several debug tools related APIs have significant tool vander independent adoption, but are tied to specific processor types, examples being RDI (ARM), GDI (PowerPC), MDI (MIPS). etc.

Visualization of debug and analysis information - in particular timing and signal information is a critical part of any analysis process. For most digital systems designers, regardless of whether the signal information is from simulation or from hardware trace, the preferred visualization is in a waveform view. While EDA and debugger tools vander has developed a variety of value added visualization approaches, the most widely supported waveform view is via VCD (Vector Change Dump) files VCD files are widely supported in simulation environments (including virtually all commercial digital systems simulation tools) and a variety of open-source viewers [12]. Many hardware trace tools support VCD as a method of viewing trace information in a common format with simulation data. VCD signal logging (sc_trace) is one of the native debug related features supported in SystemC as a write to file option for saving simulation run data and related analysis.

Database support – The amount of data that must be managed in SoC debug can be extensive, especially when multiple (simulation and instrumentation) sources and alternate implementations are considered. While different tools typically have associated databases for managing data access, adoption of open database structure and languages such as

SQL, used in DUST, an Open Source SystemC project [14], are needed to facilitate more comprehensive data access and mining for debug and verification. Development of debug databases is only one part of a multifaceted problem, with integrated solutions in all of the above areas being important for user-friendly solutions.

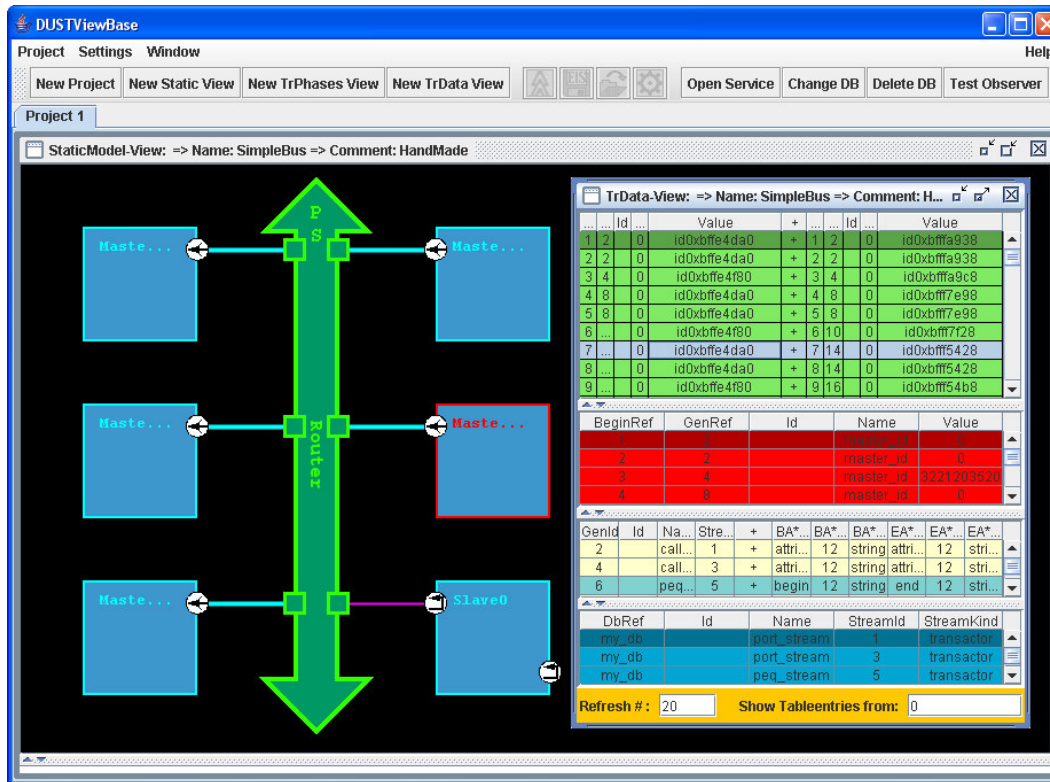


Figure 15 : DUST ESL Debug Visualization Platform

In Summary

System-on-chip (SoC) designs and architectures have and will drive new verification and analysis methodologies; in particular two emerging areas of analysis and debug of SoC architectures and design flows have been the adoption of

- ESL based (including but not limited to SystemC, SystemVerilog, and other SoC oriented languages) modeling and analysis at the conceptual front end of the design and
- Integration of instrumentation IP and hardware based debug tools to facilitate functional analysis of the final SoC (FPGA/ASIC) physical implementations.

Both these analysis approaches allow, in different ways, for improved optimization and verification of architectures, providing, integration, performance advantages not easily addressed using more traditional standalone RTL or ISS simulation and analysis. ESL and instrumentation independently fill important niches, and have historically been utilized separately and for independent tasks in design flows. This scenario appears to be changing drive by both the need for end-to-end consistency in analysis of complex SoC systems, the benefits of increased reuse in verification and analysis tasks in the design process and the emergence of direct and automated synthesis of ESL models, allowing a tighter linkage between the ESL modeling flow and in-silicon instrumentation (possibly

even eliminating the RTL design stages). In this paper, we discuss how ESL and Instrumentation based analysis share similar system level concerns, including, debug of their corresponding system configurations, need for integration and interoperability of IP at both levels, and the their respective roles in enabling of experimenting and performance analysis with different HW/SW architectures and implementations.

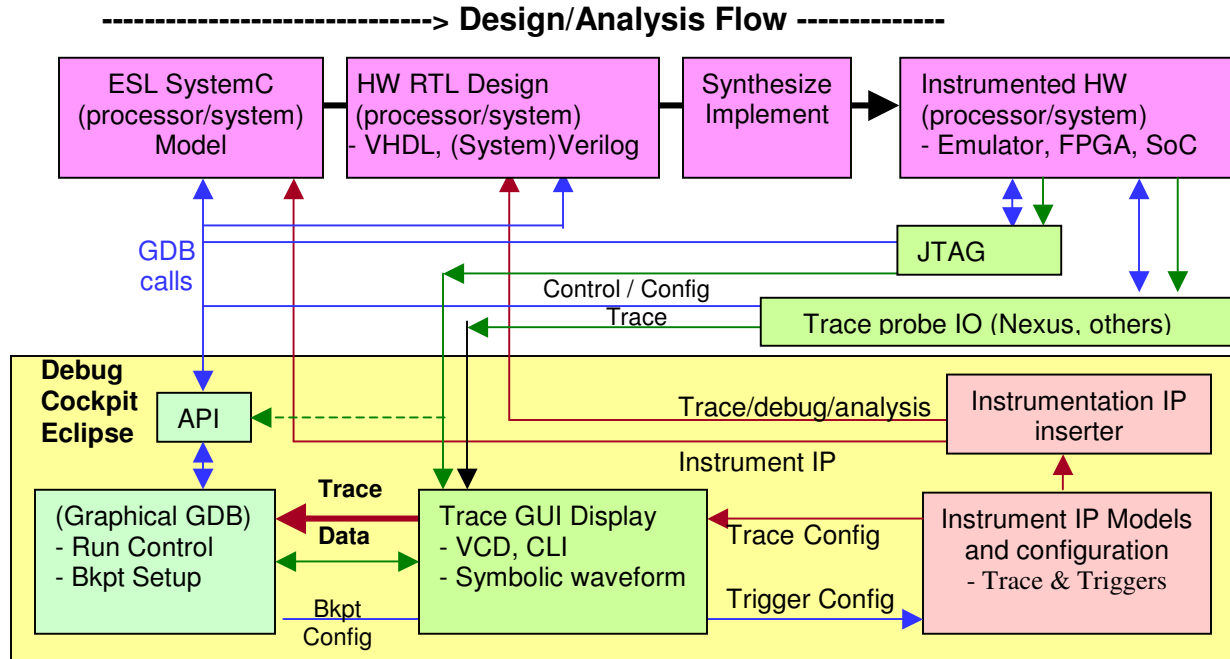


Figure 16 : ESL SoC Debug Platform Elements

A significant level of commonality exists between ESL and Instrumentation and leveraging common if independent capabilities allows better integration, merging of the strengths in these two analysis areas, providing significant value on to both activities. We present some of the common tools, interfaces, and methodologies tools that allow common analysis and integration of ESL and instrumentation based analysis and point out common scheme for allowing ESL tools and (Nexus) instrumentation tools to be integrated into a more cohesive system level analysis environment.

This approach enables two important activities. First to provide a mechanism for leveraging the strengths and limitations of ESL simulation (visibility, abstraction) and captured instrumentation based data (real time operation, platform analysis under real operation) which complement each other in helping to verify and quantify more abstract transaction level ESL models against the physical design implementation and leveraging effectively the more limited instrumentation based trace and data capture with more robust information captured during ESL simulation and analysis.

The second advantage is ESL simulation and on-chip instruments jointly provide a common platform for reuse in configuration, developing common breakpoints, triggers, run time conditions, etc.. This more comprehensive amount of reuse between different (front to back) stages in the design flow allows more extensive links and a common framework ranging between the early stage ESL modeling and analysis and final stage physical hardware/software testing and verification. Inferred are additional links and integration into the framework of intermediate RTL level design, development and analysis, that currently

link the ESL an instrumentation areas The paper presents common, standards based design flows and tools that support overlap of basic ESL analysis and instrumentation based testing, including bi-directional passing of trace between the ESL and instrumentation activities.

All other trademarks referred to herein are the property of their respective owners.

- [1] SystemC specification [_www.systemc.org](http://www.systemc.org)
- [2] http://www.eslx.com/Library/open_area/sysc.html
- [3] www.greensocs.com
- [4] www.spiritconsortium.org
- [5] [http://www.mipi.org/docs/MIPI TDWG whitepaper v3 2.pdf](http://www.mipi.org/docs/MIPI_TDWG_whitepaper_v3_2.pdf)
- [6] <http://www.ocpip.org/membership/information/wheel/debug/>
- [7] "Defining standard Debug Interface Socket requirements for OCP-Compliant Multicore SoCs" Neal Stollon, Bob Uvacek, Gilbert Laurenti, www.embedded.com/showArticle.jhtml?articleID=201000620
- [8] OCP SystemC models <http://www.ocpip.org/socket/systemc/> , [http://www.ocpip.org/data/ocpip wp SystemC Communication Modeling 2002.pdf](http://www.ocpip.org/data/ocpip_wp_SystemC_Communication_Modeling_2002.pdf)
- [9] IEEE 5001 Nexus specification www.nexus5001.org/
- [10] "Using an open debug interconnect model to simplify embedded systems design", Tom Cunningham, http://www.embedded.com/design/opensource/201802792?_requestid=503349
- [11] "Visualized SystemC Debugging" Christian Genz, Frank Rogin, Rolf Drechsler, Steffen Rülke Date 2007 http://www.informatik.uni-bremen.de/agra/doc/work/workshops/date07_uni-booth.pdf
- [12] SystemC_Win http://www.geocities.com/systemc_win/frame_home.html
- [13] "Assertions: Too good to be reserved for verification only" Brian Bailey <http://www.temento.com/data/whitepaper/AssertionsForDesignandVerification2.pdf>
- [14] *DUST : A non-intrusive analysis framework for SystemC* Wolfgang Klingauf <http://www.greensocs.com/Dust>