Nexus Based Multi-Core Debug

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

Rich Collins, Freescale Semiconductor
rich.collins@freescale.com

Abstract:

Nexus is an increasingly widely used IEEE standard for debug of processor and digital systems architectures. It provides for a range of debug features and is supported by many debug tool vendors. This paper discusses the Nexus standard and its capabilities and presents approaches to architectural implementation and integration of Nexus in several types of systems, including single processor, multiple processor, and systems debug of multiple processors and buses for integrated system level debug of complex architectures. The paper also discusses specifics of Nexus IP and architectures that have been developed by Freescale.

Author(s) Biography

Neal Stollon is a systems engineer and Director of Technical Marketing 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 from SMU, is a Texas Professional Engineer, has written over 25 technical papers (including 5 papers for DesignCon) and holds 7 patents.

Rich Collins is a Debug Architect within the Networking and Computing Systems Group of Freescale Semiconductor. He has over 13 years of digital design experience mainly focused on core peripheral and debug related IP development. He is currently serving as Technical Co-Chair of the IEEE-ISTO 5001 (Nexus) Consortium. Rich holds a BS in EE and BA degrees in Computer Science and Spanish from Duke University and has several debug related patents and patent disclosures.

1. The Need for Debug

Among the engineers doing complex designs, there is never any argument on the need for more and better tools to address the verification and analysis of complex SoC designs. Any engineer who has been through the effort of bringing up a new device, has always found times when having more visibility into the internal operations of the design would have significantly improved the time and effort involved in debugging problems. This paper addresses embedded debug based on Nexus 5001, as an approach to adding debug capabilities to SoC verification and analysis toolkit.

Debug methodologies serve as a compliment to EDA flows, which have evolved a variety of solutions to address verification needs for *pre-silicon* design, from diverse simulation based methodologies to emerging formal and assertion based methods and increasing levels of system level abstraction. This verification flow largely works under the assumption that the verification effort is largely done when the design files are handed off to the foundry for fabrication. Anyone who has been 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, knows that this is far from the case. Much as EDA based design flows and their use have benefited from standardization in tools and interfaces, debug methodologies can similarly benefit from standardization in implementation and capabilities. Nexus 5001 is one such approach to debug standardization.

Debug as an issue will increase with complexities, and recent data has indicated that while design verification times have decreased per given silicon complexity due to the use of 3rd party IP and advanced EDA, the relative time of in silicon debug has trended to increase, signaling for the need for better debug tools.

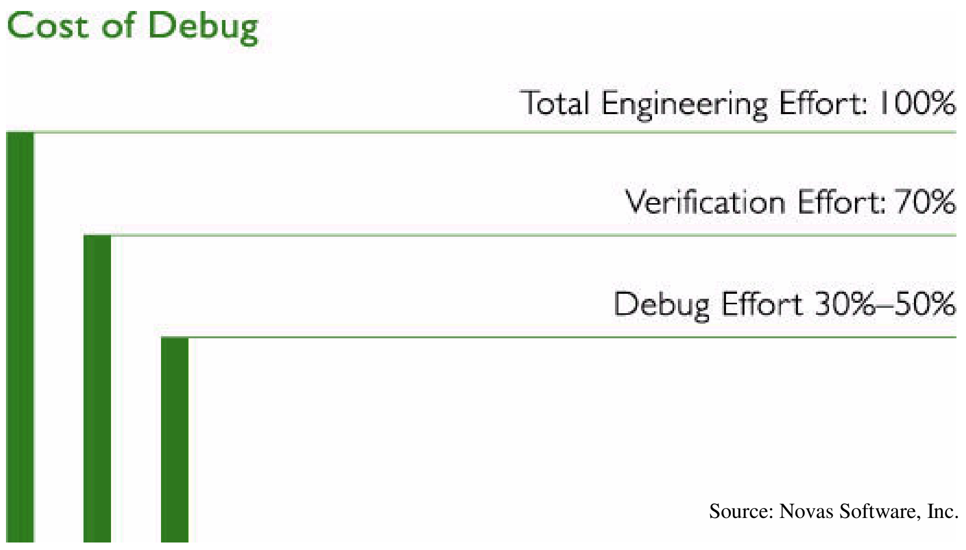

Figure 1 : Cost of Debug in the design process

The Nexus 5001 is a debug initiative that is based on the IEEE ISTO 5001 debug specification. Nexus 5001 was defined in 1999 and its development and proliferation is managed by the Nexus 5001 Forum™, which evolved as a successor to the Global Embedded Processor Debug Interface Standard Consortium (GEPDISC), formed to develop an embedded processor debug interface standard for embedded control applications. The latest version of the Nexus standard was released in 2003.

2. Nexus 5001 – A Brief Overview

The goal of Nexus 5001 is a general-purpose specification that addresses the diverse challenges for embedded processor and digital systems debug interfaces. An ever-increasing range of applications (data communications, automotive powertrain, computer peripherals, wireless systems and other control applications) have constantly increasing complexities that require more comprehensive debug features and benefit from more standardized interfaces. As advances in semiconductor and system design continue, these types of embedded applications are using higher-performance embedded processors. Efficient use of these embedded processors requires software and hardware development tools that can easily access critical processor functionality. The lack of a unifying standard among the various embedded processors on the market has impeded this accessibility, preventing tool vendors from creating standard tools with consistent functionality across a broad range of processors. Nexus 5001 addresses this issue, by providing a consistent set of auxiliary pin functions, message based transfer protocols and standard development features to facilitate debug implementations. The standard itself is open and processor-independent, but the implementations will be user-specific. The rest of this section provides a basic overview of key Nexus features; discussing the Nexus signals, messaging resources, registers, and concluding with a discussion of the different classes of Nexus implementations. Some descriptions are abbreviated in maintaining the flow of information, however the full release of the Nexus 5001 specification [1] is freely available for download from the Nexus website. http://www.nexus5001.org/

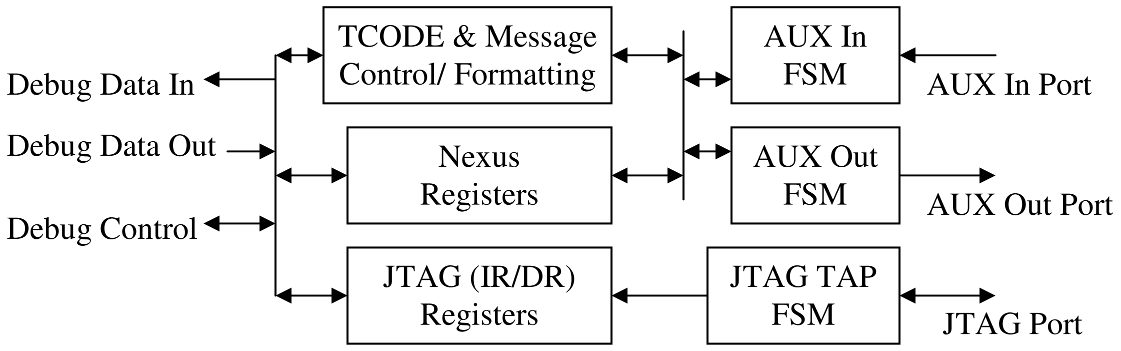

Figure 2 : Key features in Nexus debug block

2.1 Nexus Signal IO

Nexus 5001 leverages the IEEE 1149.1 standard, which is widely accepted as a test and debug pin interface. The Nexus standard defines an extensible auxiliary port (AUX) that may either be used with the IEEE 1149.1(JTAG) port or as a stand-alone development port. The Nexus standard defines the auxiliary pin functions, transfer protocols and standard development features to support both 1149.1 and AUX usage. The auxiliary port provides a wider, higher-bandwidth data transfer conduit and can define both AUX input and output ports. Auxiliary Out ports are used primarily to provide additional pins in the port for higher trace throughput. It also uses the JTAG port in Nexus-specific ways to implement various classes of services such as allowing Nexus trace output to be embedded into JTAG messages.

AUX IO	**Description of Auxiliary Pins**
MCKO	*Message Clockout (MCKO) is a free-running output clock to tools for timing MDO and MSEO pin functions. MCKO can be independent of the embedded processor's system clock or an embedded processor's clock pin may be used as a functional equivalent for MCKO.*
MDO[M:0]	*Message Data Out (MDO[M:0]) are output pin(s) used for sending messages such as trace export and other read operations, memory substitution accesses, etc. Depending upon output bandwidth requirements, one, two, four, eight, or more pins may be implemented.*
MSEO[1:0]	*Message Start/End Out (MSEO [1:0]) are output pins that indicate when a message on the MDO pins has started, when a variable-length packet has ended, and when the message has ended. Only one MSEO pin is required, but two pins provide for more efficient transfers.*
EVTO	*Event Out (EVTO) is an optional output pin to development tools indicating exact timing for a single breakpoint status indication. Upon a breakpoint occurrence of the programmed breakpoint source, EVTO is asserted for a minimum of one clock period of MCKO.*
MCKI	*Message Clockin (MCKI) is a free-running input clock from development tools for timing MDI and MSEI pin functions. MCKI can be independent of the embedded processor's system clock.*
MDI[N:0]	*Message Data In (MDI[N:0]) are inputs used for downloading configuration data, writing to on chip resources, etc Depending upon input bandwidth requirements, multiple pins may be implemented.*
MSEI[1:0]	*Message Start/End In (MSEI [1:0]) are inputs that indicate when a message on the MDI pins has started, when a variable-length packet has ended, and when the message has ended. Only one MSEI pin is required, but two pin implementations provide more efficient transfers.*
EVTI	*Event In (EVTI) is an input pin allowing off chip control such as processor halts (breakpoints) or synchronized Program/Data Messages*
RSTI	*Reset In (RSTI) is a pin for resetting the Nexus port resources.*

JTAG Pins	**Description of JTAG (IEEE 1149.1) Pins**
TDI	*Test Data Input (TDI) provides for serial movement of data into the JTAG port.*
TDO	*Test Data Output (TDO) provides for serial movement of data out of the JTAG port. All target accesses initiated via the JTAG port should be transmitted by the target via TDO (not via AUX OUT).*
TCK	*Test Clock (TCK) is an input pin that provides the clock for the JTAG port.*
TMS	*Test Mode Select (TMS) input provides access to the JTAG TAP state machine.*
TRST	*Test Reset (TRST) input optionally provides for asynchronous initialization of the JTAG IEEE 1149.1 controller.*
RDY	*Ready (RDY) output is optionally used to accelerate data accesses through the JTAG port.*

For a full-duplex AUX with IEEE 1149.1 pins, a minimum of two auxiliary pins are required for compliance [Message Data Out (MDO) and Message Start/End Out (MSEO)], assuming a system clockout pin can be used for MCKO. EVTI is also recommended for tool-initiated synchronization. The performance classification, however, would also be minimal and may meet the transfer bandwidth requirements for only low-end applications or lower compliance classifications.

Nexus implementations may have one or two Messaging Start/End-Out (MSEI/MSEO) pins, depending on complexity of the input and output state machines. Two bit messaging pins allow back-to-back data transfers, speeding delivery of memory data or trace information.

2.2 Nexus Messaging

Nexus Messages consist of a 6-bit TCODE (Transfer Code), which are Nexus specific instructions followed by a variable number of packets (the number of packets for each TCODE is defined in the standard). Messages can be Sync or Non-sync. Sync messages include full address and Non-sync only include relative address changes. Each message also contains a SRC field (source ID) to help development tools identify the source of a particular Nexus message in a multi-processing SoC sharing a single debug port. Packet types supported include:

Variable: a variable-size packet means the message must contain the packet, but that the packet's size may vary from a minimum of 1 bit. When messages are transferred via the AUX, variable-size packets must end on a port boundary.

Vendor-Fixed: These are used to allow Nexus packets to match characteristics of a vendor's device. Vendor-fixed packets may be of zero length (not implemented).

Vendor-Variable: These are used to allow Nexus packets to match characteristics of a vendor's device. Vendor-variable packets may be of zero length (not implemented). When messages are transferred via the AUX, vendor-variable packets must end on a port boundary. Variable-size packets may have different lengths in messages of the same type, so MSE signaling protocols are used to determine the end of packet boundaries. Typically vendor-variable packets are target processor dependent and have a variable size determined by the processor vendor. These packets are normally reserved for the end of a Public Message where the vendor may implement additional fields.

TCODES can be either Public (defined in the Nexus standard) or User defined. Public TCODES defined in the Nexus standard (IEEE-ISTO 5001-2003) include a range of trace options as well as other Nexus operations. These include:

- Program Trace:
  - Direct Branch
  - Indirect Branch
  - Indirect Branch With History
  - Synchronization
  - Resource Full
  - Repeat Branch

    - Repeat Instruction
    - Correlation
- Data Trace:
    - Data Write
    - Data Read
- Ownership Trace
- Data Acquisition
- Read/Write Access
- Memory Substitution
- Port Replacement
- Watchpoint
- Status

User Defined TCODES can be defined by silicon or IP developers for debug features not covered in the standard, similarly to User Defined instruction features in JTAG.

2.3 Nexus Registers

Nexus defines a number of recommended registers, which facilitate the integration of debug support to different cores. Of particular interest for multicore designs, each core or element on a device may be assigned a different ID in a Device identification (DID) register to allow discrimination and selection of control and debug operations associated with a given block or subsystem.

Nexus also defines other recommended registers for debug purposes. These include:

- Client Select Register (CSC)
- Development Control Register (DC)
- Development Status Register (DS)
- User Base Address Register (UBA)
- Read/Write Access Registers (RWA / RWD / RWCS)
- Watchpoint Trigger Registers (WT)
- Data Trace Attribute Registers (minimum of 2) (DTSA / DTEA / DTC)
- Breakpoint/Watchpoint Control Registers (minimum of 2) (BWC)
- Breakpoint/Watchpoint Address/Data Registers (minimum of 2) (BWA/BWD)

2.4 Nexus Implementation Classes

Nexus implementations are divided into four classes, so that given designs can select features of importance and not be burdened with more advanced features that are not applicable or efficient to their debug needs. This allows a variety of debug features to be supported, while at the same time keeping the number and types of different Nexus implementations that need to be tracked and supported to a manageable number. All Nexus classes by definition include all of the features in (i.e. are a superset of) the prior class.

The key features of the different implementation classes are summarized in the following table.

NEXUS 5001 IMPLEMENTATION CLASSES

Nexus	Services	Features
Class 1 Basic run control	Static debugging Breakpoints	Single step Set breakpoints and watchpoints Two breakpoints minimum Device identification Static memory and I/O access
Class 2 Instruction Trace Watchpoints	Watchpoints Ownership Trace Program Trace	All Class 1 features Monitor process ownership in real time Real-Time program tracing
Class 3 Data Trace Read/write Access	Data Trace Real-time read/write Transfers	All Class 2 features Access memory and I/O in real time Real-Time data tracing
Class 4 Memory and Port Substitution	Memory Substitution Port Replacement	All Class 3 features Start traces on watchpoint occurrence Program execution from Nexus port

The most basic, Class 1, provides features similar to standard JTAG implementations. However, it sets certain minimum requirements, such as the need for at least two hardware breakpoints. Debugging halts the chip like normal JTAG products.

Class 2 contains more complex debugging features, with real-time monitoring. It also adds instruction tracing and more sophisticated watchpoints. Class 2 program trace feature allows indirect branches to be flagged, making it easier to differentiate indirect branches from exception handling operations. Additional messages are included for improved branch tracking. The format of the trace data allows eliminating redundant addressing information, which thereby increases throughput.

Class 3 allows Data tracing services and also includes the ability to read and write memory and I/O while the processor continues to run. This makes the system design more complex but significantly improves the debugging capabilities.

Finally, Class 4 delivers features found in many in-circuit emulators (ICEs), like the ability to remap memory and I/O ports. This is especially useful when simulating peripherals. It can also be used to provide other applications running on the testing system with access to shared memory.

In addition to the four classes, Nexus defines a number of optional features. These include starting memory substitution upon watchpoint occurrence, monitoring data reads while the processor runs in real-time, port replacement and port sharing, and the ability to transmit data values for acquisition.

3. Processor System Debug

Debug features for embedded processors have been recognized from the earliest days of embedded processing as being an important requirement for processor verification. Since detailed simulation of processor operations for many applications has historically not been feasible due to the large number of cycles required for many applications, processor analysis via emulation and trace of processor operations has been required for verification and hardware/software integration. Most licensable embedded processors include some instrumentation features to support debug. While the specifics vary with each processor type, debug for processor cores typically provide similar debug features:

1. Processor specific run control (start, stop, software and hardware breakpoints, and single-step run control)
2. Monitoring of hardware and software breakpoints for triggering,
3. Real-time Trace that can include execution (instruction) and/or data trace. Trace operations can be triggered from conditions such as instruction execution, memory, or IO operations, address range, or opcode value.

Most processor debug environments can be made Nexus complaint by adding Nexus wrapper layer around the existing debug blocks. The value of Nexus for processor debug is that it allows a consistent environment for different processor types to be integrated using a consistent methodology.

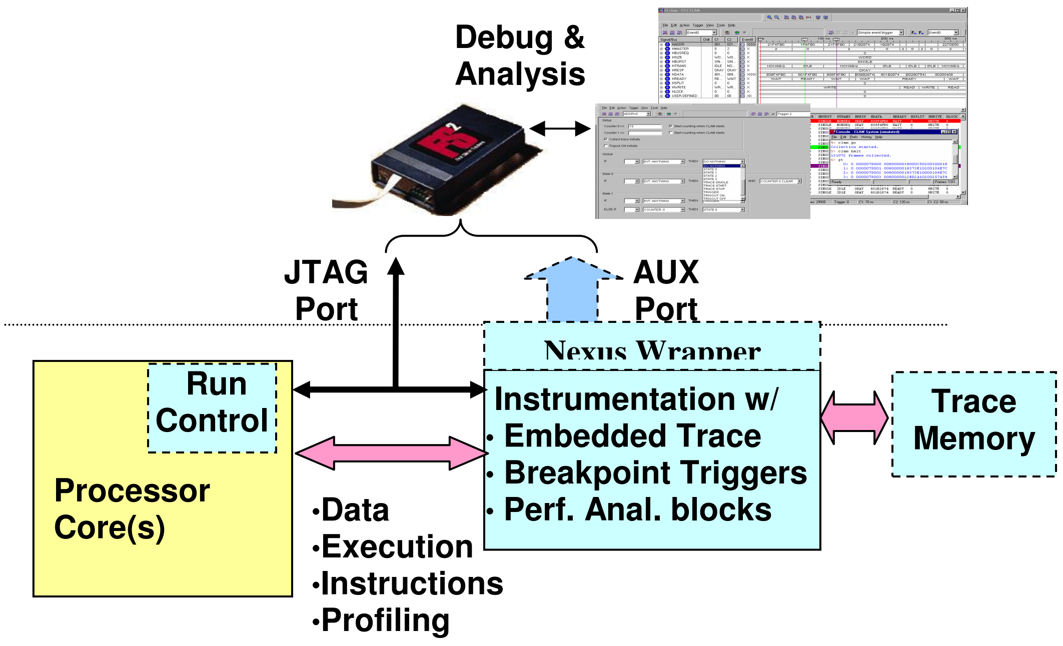

Figure 3 : Processor Trigger & Trace Instrumentation

Among the most valuable processor debug features for analyzing operational performance is execution trace. Trace in general, is a complex debug technology since it requires either a large buffer or high bandwidth in order to export trace information. Nexus defines a method of trace compression that takes advantage of the properties relating to execution of instructions being pre-defined during the programming and unlike many other types of trace operations, is largely deterministic. With the exceptions of branching and other instruction that are conditional on data, the sequence of instructions through a processor is pre-defined during software development.

To make efficient use of memory resources during execution trace, Nexus utilizes a processor instruction compression technique called Branch Trace Messaging, which reduces the trace memory required by focusing only on tracing instruction flow discontinuities (typically branches). Since branches and conditional operations typically are a small percentage of an overall instruction execution, this can greatly expand the trace RAM utilization. Trace information can be tightly integrated with debugger software tools chains, to allow correlation analysis to the source code. Nexus also supports relative addressing to reduce the number of required address bits transmitted for normal messages. Certain initialization and exception cases (defined within the standard) will cause normal trace messages to be "upgraded" to sync type messages in which the entire address is transmitted. Execution trace can be compressed and later expanded for integration with code debugger tools. This feature allows debug blocks storing instruction trace to leverage assumptions in instruction flow in order to conserve trace bandwidth and increase the number of instructions that can be stored in trace buffers or exported real time.

For data trace operations, other than the use of relative address transmission (as in program trace), there is typically no such determinism that can be leveraged for the data itself to extend the use of trace resources, and as such data trace may require either larger trace memories for a given trace size or alternate methods of storing trace information.

Even with compression, the time needed for trace export can be significant when relying only on JTAG TDO for transmitting data. This problem increases proportionally for multicore designs, where each processor and other block have their own debug information. The need for improving trace throughput is one of the reasons for implementing a Nexus Aux port as described previously.

3.1 Other System Debug Considerations

In most designs, processors are integrated with several other subsystems that also may be included in systems analysis, such as trace operations. Logic blocks included in many designs include co-processors for specific applications, memory controllers, peripherals and a host of other functions. Debug of these types of blocks can be supported by on chip logic analyzers that allow triggering and trace of logic operations, which is often done in tandem with processor debug operations. [2,3]. One variant of logic analysis important for many systems is bus level debug. Bus analysis typically takes one of two forms – signals of interest are traced at the bus interface (as example, an AMBA AHB port or OCP Socket interface), or from within the selected debug points in the bus fabric [4].

Just as buses operate in conjunction with processors and other IP, bus analysis must interface to other debug blocks. Typically this is done using cross trigger interfaces to the other debug blocks for low latency triggering of the processor debug operations based on status in another core. Likewise processor output signals can be used to allow triggering of other trace operations to start and stop based on processor operations. These cross-triggering resources, combined with more global resources, such as time-stamping of trace information to improve synchronization and alignment of debug data being brought off chip, allow a more systems oriented focus on debug process, by allowing debug of subsystems operating in differing clock domains.

3.2 Multicore Nexus Debug Approaches

Nexus implementations can support the concurrent debug of both processor and bus operations. While each processor or logic/bus element in a design may have a native debug environment, debug information can be reformatted using Nexus interface wrappers, that packetize debug information into Nexus messages. These Nexus messages can then be merged at a Nexus port control level, to allow packets from many debug sources to be communicated over a common Nexus port. Since each debug block can be assigned an independent identification (DID) value, debug information can be redirected once off chip, at the probe interface or as a software operation.

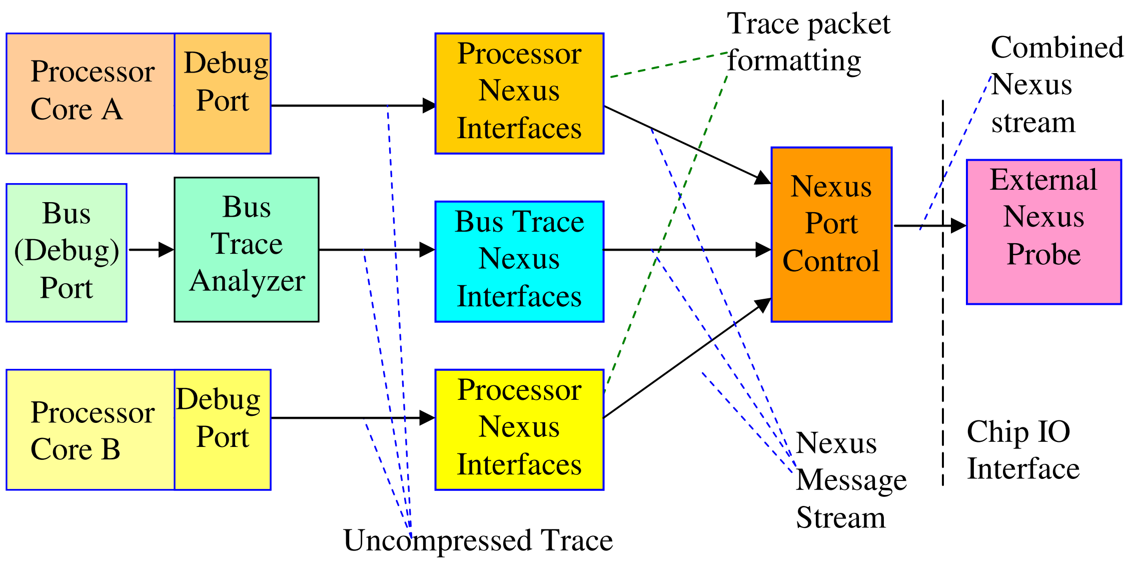

Figure 4 – Basic Nexus Multicore Debug flow

Figure 4 shows this debug data flow, supporting a multicore architecture consisting of 2 processor (or other) cores and a bus port or other bus level debug interface. All blocks have some native debug or analyzer blocks. The debug information is made into Nexus compliant messages, including any additional compression; by in line Nexus interface blocks with the different independent message streams consolidated into a single combined Nexus stream at the port interface.

One of the issues in debug of multiple core systems is that even with debug information from different blocks being combined into a single Nexus stream, the control and synchronization of debug over many different core or subsystems remains largely independent. Having better control and synchronization of different debug resources can significantly improve debug efficiency. MED (Multicore Embedded Debugger) is, as its name suggests, a debug architecture for multicore systems [5,6]. In addition to the Nexus interfaces for each of the on chip debug resources, it also includes cross triggering and system wide timestamping resources to help synchronize and cross-reference debug operations occurring at different parts of the architecture, allowing different off chip debugger environments to better comprehend the context and operations occurring in other parts of a design.

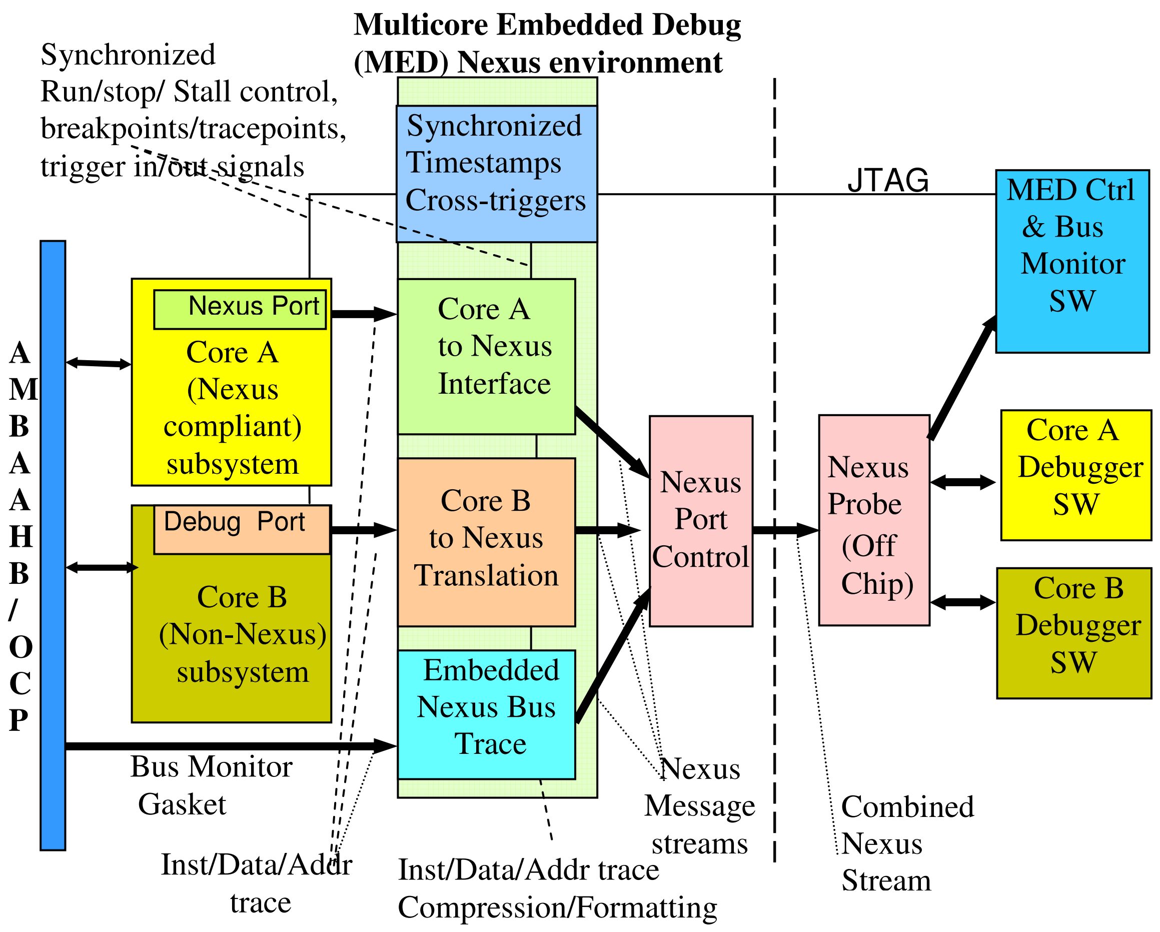
Multicore Embedded Debug (MED) Nexus environment
Synchronized Run/stop/ Stall control, breakpoints/tracepoints, trigger in/out signals
Synchronized Timestamps Cross-triggers
JTAG
MED Ctrl & Bus Monitor SW
AMBA AHB / OCP
Nexus Port
Core A (Nexus compliant) subsystem
Core A to Nexus Interface
Debug Port
Core B (Non-Nexus) subsystem
Core B to Nexus Translation
Nexus Port Control
Nexus Probe (Off Chip)
Core A Debugger SW
Core B Debugger SW
Embedded Nexus Bus Trace
Bus Monitor Gasket
Inst/Data/Addr trace
Inst/Data/Addr trace Compression/Formatting
Nexus Message streams
Combined Nexus Stream

Figure 5 – A Nexus compliant MED environment

Debug subsystems like Nexus introduce important changes in methodologies in the concept of analyzing the debug requirements during the architectural design phases of a project. Like many other supporting technologies, analyzing chip needs and debug strategies needs to be comprehended at early stages in a project. It is much more difficult to add debug at late stages of a design. Considering the debug resources after everything else is designed often severely limits the capability and quality of the debug solution. Different generic debug instrumentation IP is available, but the architecture and interface intensive nature of hardware debug often requires some customization. Nexus provides a toolbox and an approach to implementing a debug architecture, which can be customized to properly address differing architectures and unique analysis considerations. Properly implemented, a comprehensive debug solution can measurably improve the level of testability, maintainability and analysis capabilities throughout the lifecycle of a chip design, however implementing the right on chip debug solutions also requires an engineering investment in understanding of how debug tools will be used as well as the considerations of all the tradeoffs for integrating debug solutions into a design.

4. Nexus Product Implementations

Freescale Semiconductor has architected and implemented Nexus based debug on several SoCs. These SoCs have serviced many industry-wide markets including automotive, wireless and networking. Two example SoCs are discussed in this section.

One family of SoCs, initially offered for the automotive powertrain market utilizes the multi-processing features of Nexus to provide debug visibility to the processor core – a PowerPC e200z6, the enhanced timer processor units (ETPU), as well as the secondary peripheral bus.

The MPC5500 family of SoCs support various debug facilities. There are five major architectural blocks that provide the debug functionality:

- PowerPC e200z6 Nexus1 Module (OnCE) – Class1 compliant debug of the CPU
- PowerPC e200z6 Nexus3 Module – Class3 compliant trace of the CPU
- DMA Nexus Module – Data Trace support for DMA data accesses
- ETPU Nexus – Class3 compliant trace of Enhanced Timer Processor Units
- Nexus Port Controller – Arbitration for Nexus I/O port

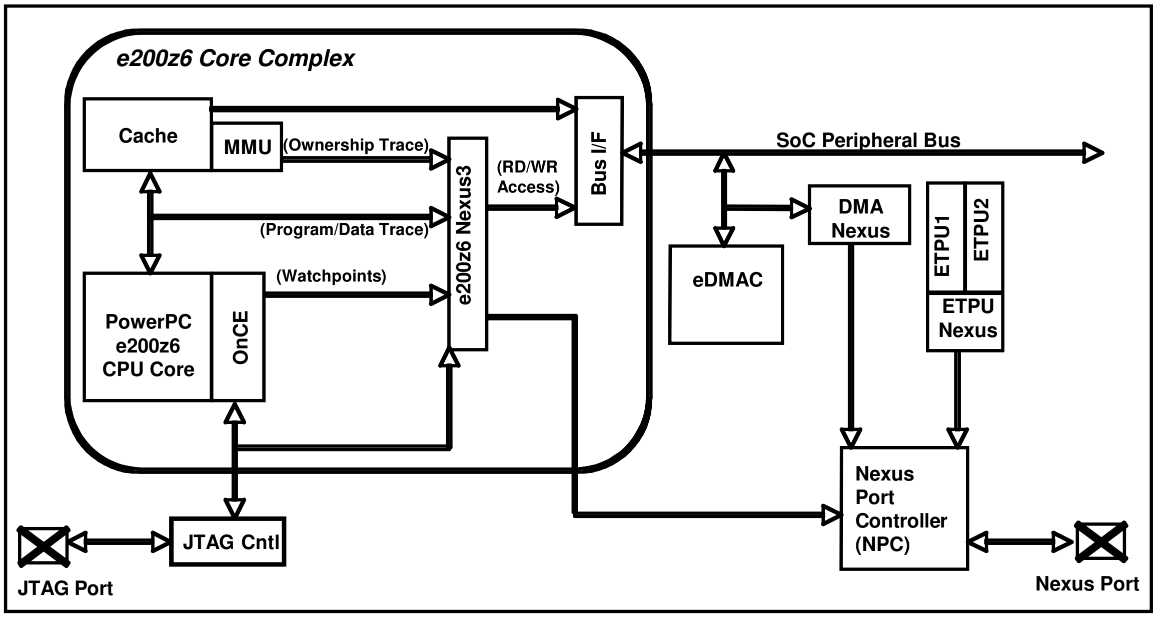

Figure 6 : Freescale MPC5500 Multi-Nexus Implementation

The PowerPC e200z6 Nexus modules support all required features as defined in Nexus Class1 and Class3 as well as the optional features of watchpoint trigger enable of program/data tracing and burst capability on Nexus initiated read/write accesses.

Class1 features such as breakpoint generation, single stepping, and internal resource access (CPU halted) are handled within the CPU's JTAG-based static debug OnCE (On Chip Emulation) block. Watchpoints for Nexus3 are also generated within the OnCE module. These eight watchpoints (for various programming events) can be used to trigger trace enable/disable, generate Watchpoint Messages and drive an optional EVTO output pin.

The DMA Nexus Module supports tracing of data reads and writes on the peripheral bus.

The Nexus Port Controller (NPC) module arbitrates between the various debug modules for the shared port and controls the port settings (MCKO divide ratio, port-width option).

The second example is from a family of wireless processors nicknamed MXC. The first generation of these SoCs combines a StarCore SC1400 DSP with an ARM11xx core and various mixes of peripherals and memory configurations.

The DSP subsystem supports a slightly more enhanced set of debug facilities. The major architectural blocks consist of:

- SC1400 Nexus1 Module (EOnCE) - Class1 compliant debug of the DSP
- SC1400 Nexus3 Module – Class3 compliant trace of the DSP
- AHB Nexus Module – Data Trace support for AHB data accesses
- Nexus Trace Buffer – Shared internal memory for dumping Nexus trace data
- Nexus Port Controller – Arbitration for Nexus I/O port and Timestamp generator

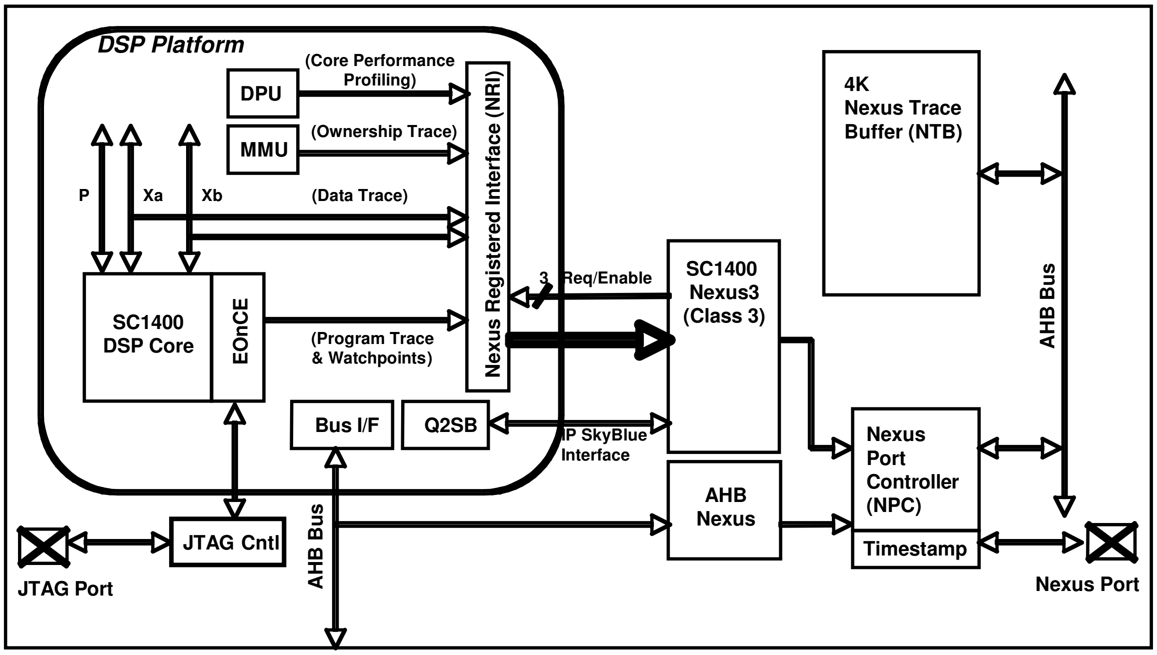

Figure 7 : Freescale MXC DSP subsystem and Multi-Nexus Implementation

The SC1400 Nexus modules support all required features as defined in Nexus Class1 and Class3 as well as the optional features of watchpoint trigger enable of program/data tracing, and data acquisition messaging for data logging. In addition, the Nexus3 module supports vendor-defined triggering of program/data tracing using the process ID, and specific messages for reporting core performance profiling information from the SC1400 Debug and Profiling Unit (DPU).

Class1 features such as breakpoint generation, single stepping, and internal resource access (CPU halted) are handled within the CPU's JTAG-based static debug block - EOnCE (Enhanced On Chip Emulation). Watchpoints for Nexus3 are also generated within the EOnCE module. These seven watchpoints (for various programming events), can be used to trigger trace enable/disable, generate Watchpoint Messages and can be connected to a cross triggering module for triggering events in other portions of the SoC. They also drive an optional EVTO output pin.

The AHB Nexus Module supports tracing of data reads and writes on the peripheral bus and can generate additional watchpoints based on AHB address and/or data values. These watchpoints can also be used by a cross triggering module within the SoC. Additional AHB Nexus modules support data trace on the application side (ARM11) of the processor as well.

Similar to the MPC5500 family, the Nexus Port Controller (NPC) module arbitrates between the various debug modules for the shared port. In addition to the arbitration and port control, the MXC NPC module provides timestamping capability for the debug system by maintaining an "absolute" timestamp value that the individual Nexus modules can use within their messages, or for generating their own "relative" timestamp to reduce bandwidth penalty.

The MXC SoCs also support internal storage of Nexus messages to an internal Nexus Trace Buffer (NTB) for retrieval at a later time. These messages are sent to AHB memory within the SoC, which has allocated a secondary function for the storage of trace information. This information can be read out through the JTAG port (or other memory access mechanisms) when real-time visibility is not as critical. This allows more trace data to be stored by reducing bandwidth restrictions associated with sending data off chip.

Summary:

Nexus has been evolving as an IEEE standard for several years and is seeing increased use as a debug solution in many different architectures and markets. Using Nexus provides several advantages to designers, in providing a widely supported infrastructure and providing a framework for customized solutions. As an "architecture agnostic" interface, Nexus also provides advantages to tool vendors by reducing development costs and time to market. Freescale has been an industry leader in developing Nexus based solutions to support a range of processor cores, and configurations. The Technical Committee within the IEEE-ISTO 5001 Consortium is continually working to add feature enhancements to the standard and support for wider range of SoC architectures.

In this paper, we have presented a technical overview of Nexus architecture and the IP and integration activities required to integrate a Nexus solution into a on chip systems architecture, including Nexus based debug components and interfaces for debugging the cores, and subsystem, which include debug of embedded buses. Nexus features can be added to most debug blocks to allow their integration into a Nexus debug environment. Nexus environments also allow support of advanced debug features such as complex triggering, performance analysis, and debug control that are needed for emerging multicore architectures. Nexus integration of debug resources provides a multi-core debug environment that allows port level sharing for the debug of multiple cores. Implementing system level debug features such as cross triggers and timestamping and the ability to merge Nexus information using a Nexus Port Controller as a single port interface enables integration of debug information, communications, and arbitration of multiple Nexus debug blocks, which can have significant benefits in debug of system level silicon.

Additional information on Nexus, including membership in the Nexus 5001 Forum, is available at http://www.nexus5001.org/

References:

[1] IEEE-ISTO 5001™-2003, The Nexus 5001™ Forum Standard for a Global Embedded Processor Debug Interface. http://www.nexus5001.org/standard.html

[2] “Logic Navigator – A vender neutral logic analyzer for FPGA debug” FPGA Journal,

[3] "Processor and System Bus On Chip Instrumentation" Embedded Systems Conference Spring 2003

[4] “Trace Instrumentation and Architectures for OCP buses” DATE 2005

[5] "Multi-core Embedded Debug Techniques" ARM Developer's Conference Oct. 2004

[6] “Multi-core Embedded Debug for Structured ASIC Systems” DesignCon 2004

References 2-6 are available for download at http://www.fs2.com/news.htm