

Evaluation of a New Emulation Port Using an M-CORE Architecture System

A new standard for real-time emulation is being developed by a consortium of companies including Motorola, Siemens, Hitachi, STM, Etas and Hewlett Packard. Officially named the Global Embedded Processor Debug Interface Standard Consortium, this group of technical and business professionals is addressing the rigorous challenges encountered daily by design engineers who develop complex real-time systems.

The goal of this consortium is to define a general-purpose specification which describes a common set of microcontroller on-chip debug features, protocols, pins and interfaces to external tools which may be used by real-time embedded control application developers. The consortium was started in late 1997 and has met on several occasions as well as conducted numerous conference calls to develop the current proposed specification for standard with intentions to pass this specification to the Institute of Electrical and Electronic Engineers (IEEE) for development as an IEEE standard. The project has been code named Nexus and will be referenced as such in this paper.

INTRODUCTION

A new standard for real-time emulation is being developed by a consortium of companies including Motorola, Siemens, Hitachi, STM, Etas and Hewlett Packard. Officially named the Global Embedded Processor Debug Interface Standard Consortium, this group of technical and business professionals is addressing the rigorous challenges encountered daily by design engineers who develop complex real-time systems.

The goal of this consortium is to define a general-purpose specification which describes a common set of microcontroller on-chip debug features, protocols, pins and interfaces to external tools which may be used by real-time embedded control application developers. The consortium was started in late 1997 and has met on several occasions as well as conducted numerous conference calls to develop the current proposed specification for standard with intentions to pass this specification to the Institute of Electrical and Electronic Engineers (IEEE) for development as an IEEE standard. The project has been code named Nexus and will be referenced as such in this paper.

Initial studies show that 9 of the 13 major microcontroller vendors have developed dedicated circuits and pins which are used for new product development. The protocol of choice has been the IEEE 1149.1 Joint Test Action Group (JTAG) 4 wire serial interface. Although the interface is targeted for testing chip interconnect integrity on high density printed circuit cards, it has become widely used for chip verification as well as accessing vendor defined functions such as on-chip debug blocks.

Dedicated on-chip debug blocks normally provide a means for communicating with a target processor while in the application environment in a static form. They play an important role in the product develop-

ment cycle but their usefulness diminishes as the product matures. Therefore a key requirement in providing these special on-chip debug blocks is to minimize their chip die area, power and the coupling to the overall system.

To provide the best opportunity for achieving a worldwide development interface standard at an accelerated pace without significantly impacting existing tool infrastructures, the Nexus standard defines a scalable set of features whereby existing debug blocks may be used. The definition of an extensible auxiliary port which may either be used with a JTAG port, or as a stand-alone debug port, is defined in the proposed standard. The features associated with this new auxiliary port focus on real-time transfer of information to and from the embedded microcontroller.

To properly justify implementing new real-time debug features on a commercial chip, a thorough analysis is being conducted for throughput capabilities as well as correctness of the proposed standard. The vehicle selected to perform this evaluation is a Motorola M...CORE based micro-RISC architecture designed for automotive as well as telecommunications applications. The purpose of this paper is to discuss the progress of this consortium, provide a technical overview of the features, protocol, and pins of the Revision 1.0 release of the proposed standard and discuss the results of a proof of concept behavioral model conducted on the M...CORE based microcontroller.

STATIC VERSUS DYNAMIC DEBUG REQUIREMENTS

Debug requirements increase proportionately with respect to a system's development phase. Initial system start-up problems such as exit from reset, verifying integrity of a system hookup, evaluating processor and code behavior are considered fundamental control

DEBUGGING & TESTING

features. State machine behavior, program tuning, and closed loop feedback are more complex behavior which require special evaluation techniques.

Two basic categories are defined as the base line for debugging systems, i.e., static and dynamic. Static debug refers to the classical method of halting the processor, loading a program and presetting programmer model resources such as registers and/or memory. If breakpoints are required, they are set and the processor is put in a real-time mode to execute code until a breakpoint condition is met. If a breakpoint condition occurs the processor is halted and the user model resources are interrogated to observe program behavior. This technique is repeated until the application system has been properly evaluated for obvious faults and the system functions somewhat predictably in its environment. Static debug does not require high speed throughput other than to load large programs in relatively short periods of time, thus serial debug interfaces are sufficiently adequate.

Dynamic debug refers to non-intrusive access to program model resources while the system operates in real-time. Program data values dictate program flow behavior so observation of data blocks may be necessary to fine tune a control algorithm. Dynamic observation of program behavior requires a high speed interface which monitors program addresses and reports any change of flow.

The ability to access program model resources while the processor executes code in real-time is increasingly more difficult when the target processor is running at speeds greater than 40 megahertz. High density surface mount packages as well as low pin count microcontrollers with on-chip memory systems complicate the task. Observation of real-time program behavior within these constraints requires a new mind set.

The classical method of providing a logic analyzer hookup to processor external memory interface pins is

rapidly becoming an obsolete debug method. Dynamic debug features require a different set of circuits and tools and are typically more complicated and expensive.

CLASSES AND LEVELS OF COMPLIANCE

In order to address various levels of development needs, the Nexus consortium has categorized static and dynamic debug features according to class levels. These class levels provide a means for implementing a scalable debug block which addresses different market segment requirements. Also, it should be noted that when a product is in development, it is desirable to have as many debug features available because of time to market constraints. Once a device is put into production, it may not be necessary to have all the development features and pins. A cost savings may be realized by implementing a scalable debug port which meets only requirements needed for specific stages of the product life cycle.

One major objective of the consortium is to help development tool vendors more easily provide a standard set of tools which may be used on a number of embedded microcontrollers as illustrated in Figure 1. In the spirit of reusability, it has been recognized that many semiconductor vendors currently have debug ports and tool sets that sufficiently address the static debug requirements of their architectures. Providing a cost effective yet powerful migration path to a standard set of dynamic debug features is a goal of the consortium.

Table 1 describes 4 classes of compliance based on features in the proposed global development standard. Each class level increases in complexity from the classical static debug capabilities to complex dynamic debug and each higher numbered class is inclusive of its respective lower numbered class. Another unit of measure for classification is throughput performance

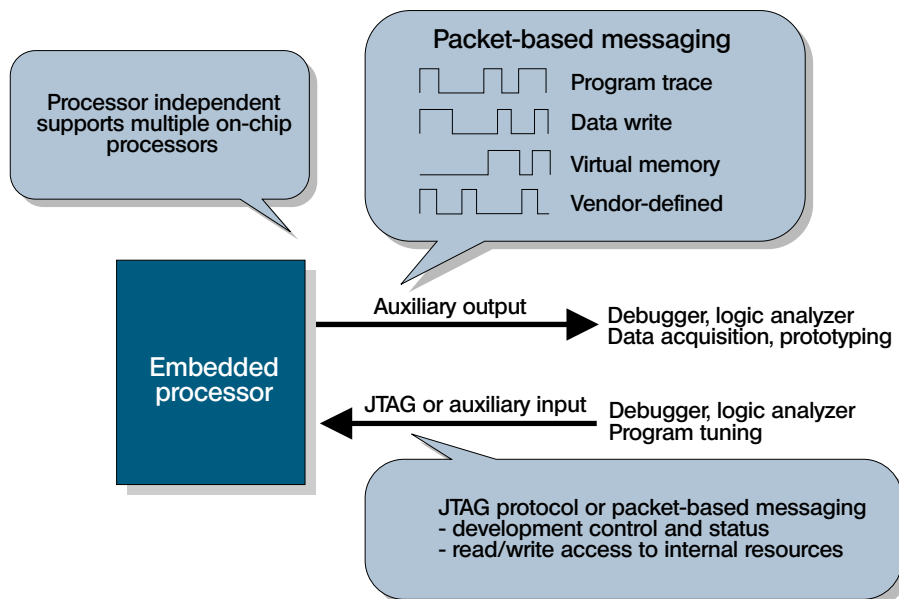


Figure 1. Illustration of JTAG/Nexus development interface.

Development Feature	Class 1	Class 2	Class 3	Class 4	Nexus Feature
STATIC DEVELOPMENT FEATURES					
Stop program execution on software breakpoint and enter debug mode	V	V	V	V	Currently not defined in Nexus
Read/write user registers in debug mode	V	V	V	V	Currently not defined in Nexus
Read/write user memory in debug mode	V	V	V	N	Read/Write Access
Enter a debug mode from reset	V	V	V	N	Development Control
Enter a debug mode from user mode (either pin or bit sequence)	V	V	V	N	Development Control
Exit a debug mode to user mode	V	V	V	N	Development Control
Single step instruction in user mode and re-enter debug mode	V	V	V	N	Development Control
Stop program execution on instruction/data breakpoint and enter debug mode (min. 2 breakpoints)	V	V	V	N	Breakpoint Control
DYNAMIC DEVELOPMENT FEATURES					
Ability to set breakpoint or watchpoint	V	V	V	N	Breakpoint/Watchpoint Control
Device Identification	V	N	N	N	Device ID
Ability to send out an event occurrence when watchpoint matches	V	N	N	N	Watchpoint Message
Monitor process ownership while processor runs in real-time		N	N	N	Ownership Trace
Monitor program flow while processor runs in real-time (logical address)		N	N	N	Program Trace
Monitor data reads/writes while processor runs in real-time			N	N	Data Trace
Read/write memory locations while program runs in real-time			N	N	Read/Write Access
Program execution (instruction/data) from development port				N	Memory Substitution
Ability to start ownership, program/data trace and memory substitution upon watchpoint occurrence				N	Watchpoint Trigger
Low speed I/O port replacement and High speed I/O port sharing		O	O	O	Port
Replacement/Sharing Transmit data values for acquisition by tool		O	O	O	Data Acquisition Message

Table 1. Nexus Classes of Compliance based on Features.

of the debug port. Half duplex operation is limited to Class 1 while all other classes require full duplex communication.

Each class level has increasing feature complexity. Class 1 encompasses vendor defined static debug features that exist today and would not require redesign of the respective debug blocks provided they implement the baseline set of features described. Class 2 addresses fundamental real-time debug needs for program flow behavior. Class 3 adds real-time data flow behavior and real-time read/write access to programmer model memory. Class 4 addresses both static and dynamic debug requirements and adds virtual memory access capabilities. It should be noted that Class 4 is intended for new integrated circuit designs where the debug block is imple-

mented without legacy constraints.

Each class of compliance will require specific protocol packets to transfer information to and from an external host computer. Also, there is a minimum number of Nexus debug registers at each class level as well as pins to meet throughput requirements. In Table 1 V = vendor defined method, N = Required for Nexus Compliance, O = optional but not required for Nexus Compliance.

APPLYING REAL-TIME FEATURES USING PUBLIC MESSAGES

A set of protocol packets, commonly referred to as Public Messages, has been defined for transferring information associated with accomplishing each of the

DEBUGGING & TESTING

Feature	Class Level	Public Messages used to communicate Feature
Ownership Trace	2,3,4	Process Ownership Trace Message
Program Trace	2,3,4	Direct Branch Message, Indirect Branch Message, Synchronization Message, Error Message
Watchpoint Occurrence	2,3,4	Watchpoint Message
Data Trace	3,4	Data Write Message, Data Read Message, Data Write Message with Sync, Data Read Message with Sync, Error Message
Control and Status & Read/Write Access	3,4 4	Auxiliary Access messages - Device Ready for Upload/Download, Upload Request (Tool Requests Information), Download Request (Tool Provides Information), Upload/Download Information (Device/Tool Provides Information)
Memory Substitution	4	Virtual Memory Access (from Device), Virtual Memory Transfer (from Tool), Virtual Memory Complete (from Tool)
Port Replacement	4	(optional) Port Replacement Output Message, Port Replacement Input Message

Table 2. Usage of Public Messages.

major debugging features. Public Messages consist of a transfer code or TCODE, source processor identification number, and the data associated with the particular feature being accomplished. A key requirement in the definition of the Public Messages is efficiency, thus packets may be variable in length depending on the TCODE.

For example, one particular feature may require 5 different Public Messages to transfer the information needed. Table 2 describes the various development features, which compliant class level they meet and the Public Messages used to accomplish them.

OWNERSHIP, WATCHPOINT AND PROGRAM TRACE MESSAGES

Ownership Trace Messaging is a means for a real-time kernel to provide a developer with which process currently has ownership of the supervisor mode of opera-

tion. This message is normally generated by a kernel write to a special address in target memory called the Ownership Trace Register (OTR). During configuration of the Nexus debug block, ownership trace messaging is enabled and a Nexus register is loaded with a User Base Address which tells the debug port when to capture data writes. The value written onto the data bus is considered the ownership tag and is then sent to the debug port.

Watchpoint Messaging is a means for reporting an event occurrence based on specific preset qualifiers. Rather than halt processor execution, a watchpoint occurrence based on breakpoint comparator qualifiers initiates a watchpoint message.

Program Trace Messaging is a means for observing real-time program change of flow. A complete embedded software program may be followed by only providing messages that indicate change of flow of the

Public Message	Examples events which cause message generation
Synchronization	<ol style="list-style-type: none"> Sequential instruction counter overflow (periodic synchronization). Exit from one mode to another mode (ex: normal, reset, debug, low power, etc.) and sequential instruction. Assertion of EVTI pin
Ownership Trace	<ol style="list-style-type: none"> Target software write to process ID register.
Watchpoint Message	<ol style="list-style-type: none"> Target hardware qualifiers are met which produce a watchpoint occurrence to be messaged.
Direct Branch	<ol style="list-style-type: none"> Target opcode execution which causes program counter change of flow via direct addressing.
Indirect Branch	<ol style="list-style-type: none"> Target opcode execution which causes program counter change of flow via indirect addressing. Exception or interrupt
Direct Branch with Sync	<ol style="list-style-type: none"> Sequential instruction counter overflow. Program Trace Message FIFO Overrun error
Indirect Branch with Sync	<ol style="list-style-type: none"> Exit from one mode to another mode (ex: normal, reset, debug, low power, etc.) and target opcode execution which causes program counter change of flow via indirect addressing Program Trace Message FIFO Overrun error

Table 3. Program Visibility using Watchpoint, Ownership and Program Trace Messages.

Public Message	Examples events which cause message generation
Data Write	<ol style="list-style-type: none"> 1. Writes to all data locations when data messaging enabled (Class 3) 2. Writes to all data locations after start qualifier in watchpoint trigger register, and breakpoint registers is met. Messages stop when stop qualifier in watchpoint trigger register and breakpoint registers is met. (Class 4)
Data Read	<ol style="list-style-type: none"> 1. Reads of all data locations when data messaging enabled (Class 3) 2. Reads of all data locations after start qualifier in watchpoint trigger register, and breakpoint registers is met. Messages stop when stop qualifier in watchpoint trigger register and breakpoint registers is met. (Class 4)
Data Write with Sync	<ol style="list-style-type: none"> 1. Write to a data location when data messaging enabled and sequential instruction counter overflow has occurred. (Class 3) 2. Write to a data location when enabled and exit from one mode to another mode has occurred. (example modes: normal, reset, debug, low power, etc.). (Class 3) 3. Sequential instruction counter overflows and a write to a data location after start qualifier in watchpoint trigger register, and breakpoint registers is met. Messages stop when stop qualifier in watchpoint trigger register and breakpoint registers is met. (Class 4) 4. Exit from one mode to another mode has occurred and a write to a data location after start qualifier in watchpoint trigger register, and breakpoint registers is met. Messages stop when stop qualifier in watchpoint trigger register and breakpoint registers is met. (Class 4) 5. Overrun error
Data Read with Sync	<ol style="list-style-type: none"> 1. Read of a data location when data messaging enabled and sequential instruction counter overflows. (Class 3) 2. Read of a data location when enabled and exit from one mode to another mode (ex: normal, reset, debug, low power, etc.). (Class 3) 3. Sequential instruction counter overflows and a read of a data location after start qualifier in watchpoint trigger register, and breakpoint registers is met. Messages stop when stop qualifier in watchpoint trigger register and breakpoint registers is met. (Class 4) 4. Exit from one mode to another mode has occurred and a read of a data location after start qualifier in watchpoint trigger register, and breakpoint registers is met. Messages stop when stop qualifier in watchpoint trigger register and breakpoint registers is met. (Class 4) 5. Overrun error

Table 4. Target Data Bus Visibility Using Data Trace Messages.

current program instruction counter. An initial synchronization or reference address must be provided. Subsequent changes of program instruction flow messages will be based on the type of change of flow that occurs.

Direct branch messages are shorter since only the last count of sequential instructions executed since the last reference address is required to reconstruct the current program counter value. Indirect branch messages will provide where you came from and where you are going. If a specific count of sequential instructions expire, for example 256 sequential instructions, then a synchronization message is generated to keep a cur-

rent program counter address on hand at the development tool monitoring the Nexus auxiliary port. Table 3 below describes what events generate Ownership and Program Trace Messages.

DATA TRACE MESSAGES

Data Trace messages provide a means for reporting real-time data accesses to memory locations. This feature is quite useful for monitoring specific parameters stored in data memory or values being accessed by a memory mapped peripheral port. Data trace qualifiers include the access type, i.e., read/write or either, as well as a start and stop address range. If the data address

Public Message	Examples events which cause message generation
Device Ready	<ol style="list-style-type: none"> 1. Internal target signal indicating target memory is ready to receive or transmit a value. This signal produces the tcode to be transmitted by target.
Download Request	<ol style="list-style-type: none"> 1. External controller sends this TCODE to initialize the target registers for downloading data to the target.
Upload Request	<ol style="list-style-type: none"> 1. External controller sends this TCODE to initialize the target registers for uploading data from the target.
Upload/Download Information	<ol style="list-style-type: none"> 1. For reading data from the target, the target receives an ENABLE UPLOAD TCODE, presents the data to the RWDR and initiates a DEVICE READY TCODE. The external controller will then initiate a UPLOAD/DOWNLOAD TCODE, read the data and wait for a DEVICE READY TCODE. 2. For writing data to the target, the target receives an ENABLE DOWNLOAD TCODE from the external controller, and initiates a DEVICE READY TCODE when its memory is ready. The external controller will then initiate a UPLOAD/DOWNLOAD TCODE, write a new data and wait for a DEVICE READY TCODE.

Table 5. Accessing Nexus Registers Using Auxiliary Port Messages.

Public Message	Examples events which cause message generation
Access	1. Initiated by target to request an instruction or data starting from a unique address.
Transfer	1. Initiated by external controller to transfer the instruction or data to the target.
Transfer Complete	1. Initiated by external controller to tell the target that the instruction / data stream has completed.

Table 6. Program ROM Patching Using Memory Substitution Messages.

and access type qualifiers are met, a data messages are generated and sent to the debug port.

Output bandwidth requirements for the debug port are reduced by only sending the unique portion of the data address instead of the complete address. Consequently a data trace message is reconstructed relative to each prior message using a synchronization message as a reference address to begin with.

All data trace information arrives from the target and provides information on data accesses, their values and locations. Table 4 describes what events generate Data Trace Messages.

AUXILIARY ACCESS MESSAGES

Auxiliary Access Messages are used to read and write data to the auxiliary control and status registers through the auxiliary port. It provide a means for transferring information to and from the target system at relatively high speed. Configuration of debug port registers is a classical use of the auxiliary access messages. For example, to enable program trace messaging, a write to the auxiliary port development control register (DCR) must be initiated using an auxiliary access message.

Auxiliary port messages may be not be required for Class 2 or 3 compliant devices if the JTAG port is used to access the required debug registers. Table 5 describes the usage of the 4 auxiliary port messages.

MEMORY SUBSTITUTION MESSAGES

Memory Substitution Messages are used to emulate a bus where opcodes and data may be accessed through the auxiliary port. Currently the Nexus standard only requires reading of data and/or fetching instructions via the auxiliary port. Discussion is currently underway for definition of a full memory emulation capability for the next specification release.

Memory Substitution Messages support run-time patching for portions of internal ROM memory, with the patch provided via the auxiliary port. A classical example of this is to begin reading instructions upon occurrence of a watchpoint qualifier. Once activated, memory substitution accesses continue until an external development tool disables memory substitution. Table 6 describes the application of the Memory Substitution

Messages.

PORT REPLACEMENT MESSAGES

Port Replacement Messages are used to emulate a low speed port which is being used by the auxiliary port. In embedded processor applications the use of every pin is scrutinized by embedded processor developers. Inevitably there are never enough pins available on the embedded processor to meet the application needs.

Pins which are designated for product development are often reduced or removed to make way for other pin functions directly used in the application. Port replacement provides a mechanism for low speed I/O pin functions to be replaced using messages via the auxiliary port. Table 7 describes the application of the port replacement messages.

NEXUS REGISTERS, COMMUNICATION PROTOCOL AND INTERFACE

The proposed Nexus development standard supports development for 1 to 32 clients on an embedded processor. Each client on embedded processors complying to class 1 shall provide development tool access to control and status according to the proposed standard via the IEEE 1149.1 JTAG interface. Each client on embedded processors complying to class 2, 3 or 4 shall provide development tool access to control and status according to the standard via either the auxiliary port or the IEEE 1149.1 JTAG interface.

To facilitate development tool vendors, Nexus development registers which provide control and status of features are mapped incrementally to a base offset. This base offset provides a standard means for decoding access opcodes which a Nexus development port controller uses to connect the register to the auxiliary port. Table 8 describes the Nexus registers which provide access and control to their associated features. Not shown in Table 8 is a required Device Identification Register (DID) which is transmitted via the auxiliary output port upon exit of auxiliary port reset. This register provides manufacturer ID, product number, version number, Nexus standard version, compliance class, and port replacement capability. This con-

Public Message	Examples events which cause message generation
Output message	1. Initiated by the target to set up external logic on the target system to replace an output port value.
Input message	1. Initiated by the target to read external logic that is replacing an input port.

Table 7. Low Speed Port Using Port Replacement Messages.

***AD NATIONAL
INSTRUMENTS***

Control/Status Register	Compliance Class	Access Opcode	Read/Write	Comment
JTAG Public Opcodes	--	0-7	--	IEEE 1149.1 standard required registers
Top Level - Reserved	--	8-9	--	Reserved
Development Control	2,3,4	10	R/W	Enables/Disables all features
Reserved	--	11	--	Reserved
Development Status	4	12	R	Provides status for all features
User Base Address	2, 3, 4	13	R	Provides target memory mapped address which is used to transfer information between microcontroller core and Nexus development port.
Reserved	--	14	--	Reserved
Read/Write Access	3, 4	15	R/W	Provides DMA-like control to target memory mapped addresses.
Upload/Download Information	3, 4	16	R/W	Provides DMA-like access to target memory mapped data.
Reserved	--	17	--	Reserved
Watchpoint Trigger	4	18	R/W	Defines which breakpoint registers are to be used to start/stop watchpoint triggering.
Reserved	--	19	--	Reserved
Data Trace Attribute (2)	3, 4	20-21	R/W	Defines data trace address range and access type qualifiers for generating data trace messages.
Data Trace Attribute - Reserved	--	22-23	--	Reserved
Breakpoint/Watchpoint Control (2)	4	24-25	R/W	Provides control and address/data/access type qualifiers for generating breakpoints and watchpoints.
Breakpoint/Watchpoint Control - Reserved	--	26-31	--	Reserved
Reserved	--	32-47	--	Reserved
Vendor-Defined	--	48-255	--	Optional Vendor defined registers.

Table 8. Nexus Development Registers.

figuration is very important to the tool vendor for proper support of the Nexus proposed development port standard.

NEXUS AUXILIARY PORT AND INTERFACE

A physical interface which may include the current IEEE JTAG 1149.1 interface has been defined in revision 1.0 of the proposed Nexus standard. Microcontroller vendors who currently provide a JTAG interface may consider communicating with the Nexus registers via their respective JTAG port, thus reducing the number of additional debug port pins and logic. An auxiliary port which provides uni-directional pins for full duplex operation is proposed for the real-time transfers.

The objective in the definition of the auxiliary port is to be efficient and scalable. Real-time program and data trace information must be communicated at the microcontroller clock speed so that there will be no loss of Public Messages.

Relative address generation and address re-creation for program and data trace messaging must be done via an exclusive-or of the last reference address sent. Since information is transmitted least significant bit(s) first, this eliminates transmission of bit values of zero

above the last non-zero bit. The host computer will reconstruct the full address from these variable length packets using the same technique.

A minimum of 3 auxiliary port pins are required for compliance, i.e., MDO, MSEO and EVTO assuming a system clock output may be used for MCKO. Table 9 describes the auxiliary port pins.

Depending on the target system throughput requirements, MDI and MDO pin count will be of variable size with a maximum of 2 MSE pins. Figure 2 illustrates the transfer protocol for the indirect branch Public Message using 1 MSEO pin.

IMPLEMENTING NEXUS FEATURES ON AN M...CORE BASED MICROCONTROLLER

M...CORE based microcontrollers are ideal for automotive and hand-held portable applications. A key requirement for portable products is low pin count and low power consumption. All M...CORE based microcontrollers provide a OnCE

TM debug block for rapid new product development utilizing the JTAG protocol for communication. This debug block contains a superset of the features required for Nexus Class 1 compliance. It serves very well for static

Pin name	Direction	Full Duplex w/J TAG	Full Duplex	Half Duplex	Pin Function
Messaging Clock-In (MCKI)	In		R		Independent free running input clock for timing of MDI and MSEI pin functions.
Messaging Data-In (MDI)	In		R		Input pin(s) for writing to Nexus registers.
Messaging Start/End-In (MSEI)	In		R		Indicates when a message on MDI pins has started and when a variable length packet has ended.
Messaging Clock-Out (MCKO)	Out	R	R		Independent free running output clock for timing of MDO and MSEO pin functions.
Messaging Data-Out (MDO)	Out	R	R		Output pin(s) for reading from Nexus registers.
Messaging Start/End-Out (MSEO)	Out	R	R		Indicates when a message on MDO pins has started and when a variable length packet has ended.
Event-In (EVTI)	In	R	R	R	Input which causes processor breakpoint or initiates program and data sync. message.
Reset-In (RSTI)	In		R	R	Input for resetting Nexus port resources.

Table 9. Auxiliary Port Pins Description.

debug control and contains limited observation of real-time program flow. To enhance the M...CORE debug block with minimal impact to new designs, features for Nexus Class 2 compliance and Read/Write Access for Class 3 were modeled. Providing real-time program trace visibility and high speed DMA-like accesses to memory mapped resources would add significant capability with low power consumption penalty. Also, by using the OnCE controller to access 5 required Nexus registers, only a small amount of additional controller logic was needed.

Figure 3 illustrates an implementation of a Nexus Class 2+ port utilizing the JTAG/OnCE port for configuration of registers and for performing Read/Write Access features. This approach is a hybrid of a 6 wire JTAG/OnCE port with a 4 wire Nexus Auxiliary Port. All static debug features are conducted through the JTAG port including real-time access to memory. Utilizing the JTAG port removed the requirement of adding MDI and MSEI pins as well as a Nexus register decoder.

The data path for Ownership Trace Messages, Program Trace Messages and Watchpoint Messages

occur through a dedicated Nexus block consisting of 3 sub-blocks. The Bus Interface block snoops the core virtual bus so that program flow within a cache unit could be monitored. The Nexus DCR register in the OnCE interface enables messaging via the auxiliary port. Real-time program change of flow addresses are stored in a FIFO block so that no messages are lost. The FIFO block in turn sends message packets to the I/O block for transmission to the MDO and MSEO pins. A group of automotive/industrial algorithms written in C from the EDN Embedded Microprocessor Benchmark Consortium (EEMBC) were compiled using a Diab Data Compiler and loaded onto a verilog behavioral model of a M...CORE based architecture. The test bench instantiated the Nexus block so that it was evaluated for throughput capability. Using the protocol defined by the Nexus consortium, it was realized that coherent program flow could be accomplished using only 2 MDO pins, 1 MSEO pin and a 16 message deep FIFO.

The existing OnCE breakpoint logic was slightly modified to add watchpoint messaging capability. This

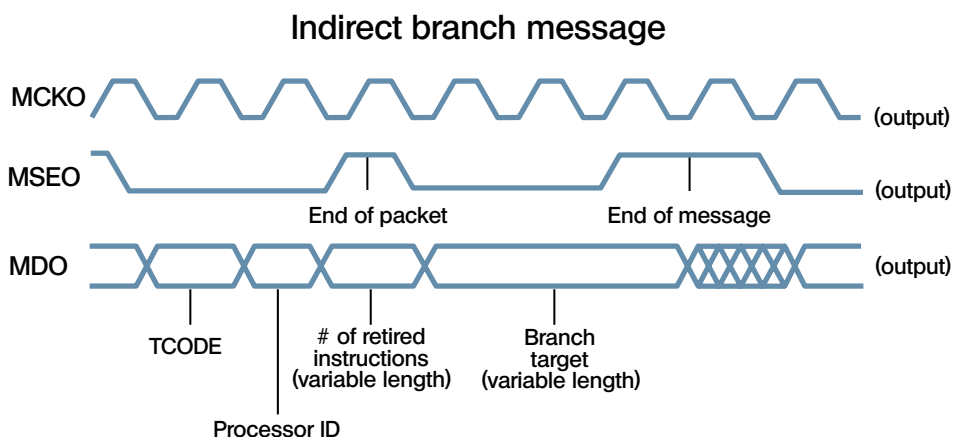


Figure 2. Example of Auxiliary Port Pins Hardware Protocol.

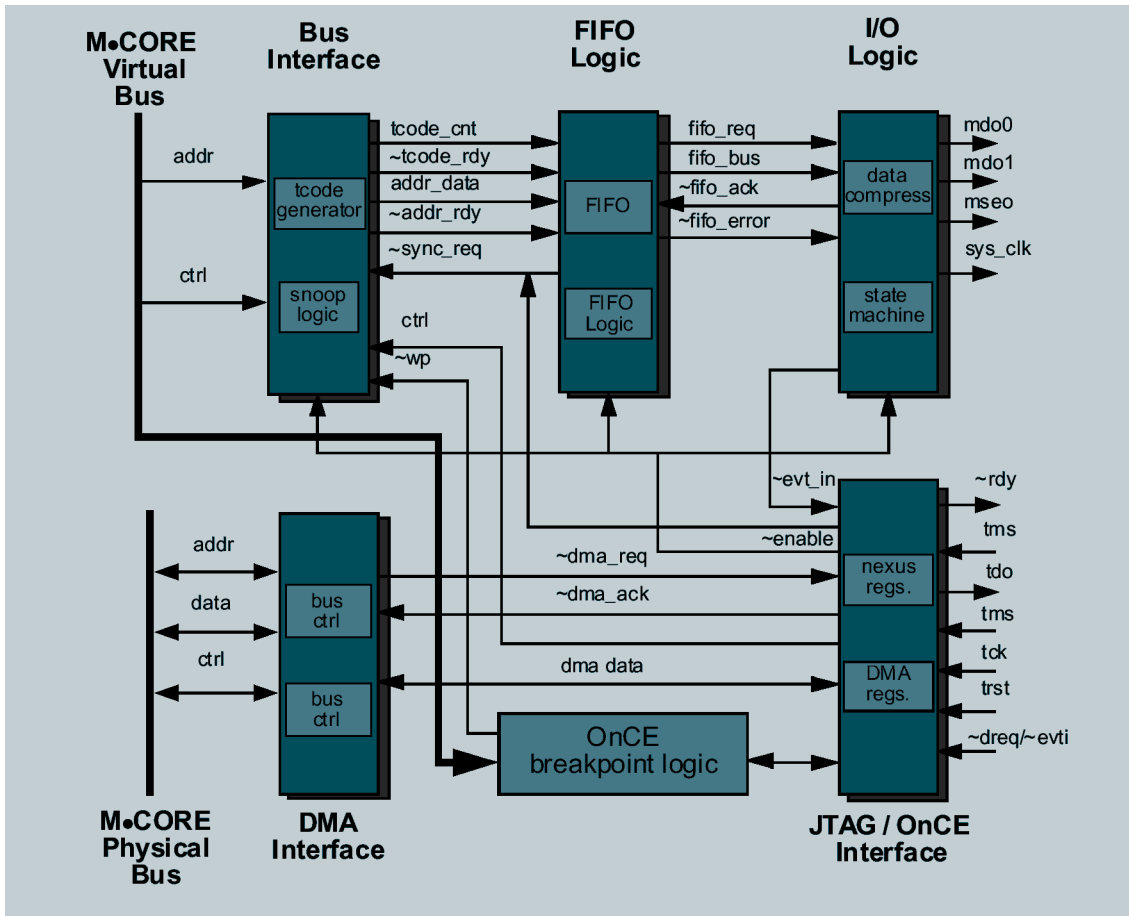


Figure 3. Nexus Compliant Class 2+ on M...CORE Based Architecture.

approach allowed for more sophisticated watchpoint capability as well as removed the task of adding any new comparators to qualify watchpoint messages. Read/Write Accesses are performed using the JTAG/OnCE serial data path. A Ready for Transfer pin (RDY) was added to increase DMA-like transfers. Calculations show that accesses to the required Upload/Download Information Register (UDI) allowed for a throughput of 1 megabyte per second on an M...CORE based microcontroller operating at 40mhz system clock.

Overall, 10 Public Messages were required to accomplish a Nexus Class 2 compliant interface. Without using the JTAG/OnCE port interface and logic, 4 additional Public Messages and 3 input pins would have been required.

SUMMARY

Significant effort is underway throughout the electronics industry to improve tools and methods for designing complex embedded systems. The Global Embedded Processor Debug Interface Standard Consortium, code named Nexus, is a testament to this and demonstrates that there is a dire need to standardize on a set of features, protocols, pins, interfaces and tools for rapid development of real-time microcontroller based products. The consortium has visited many customers to educate the design community and there is considerable enthusiasm from the feed-

back received. The main question asked by customers is "When will this become available on the products I design with?". As with all industry standards, this will take time to be accepted but the momentum, desire and energy is there to accomplish this effort.

Originally targeted to solve automotive engine controller design problems, the Nexus consortium has extended the scope of this effort to encompass telecommunications, industrial and portable hand-held products. The problems of real-time visibility to deeply embedded microcontrollers is very similar if not identical to most product types. Each will have special cases for solving specific design issues but the proposed global standard has addressed this by allowing for vendor defined blocks for special features, all addressed by a common protocol. The Nexus consortium frequently updates its progress, solicits feedback and comments and places the most current specification on internet web site www.nexus-standard.org ■

David Ruimy Gonzales is a Senior Member of Technical Staff at the Motorola Embedded Platform Solutions group. He has worked on microcontrollers, DSPs, and micro-RISC processors for 22 years at Motorola. He holds a BSCS from St. Edwards University in Austin, Texas, has published over 50 technical articles, and holds 2 patents in the area of real-time embedded systems development.