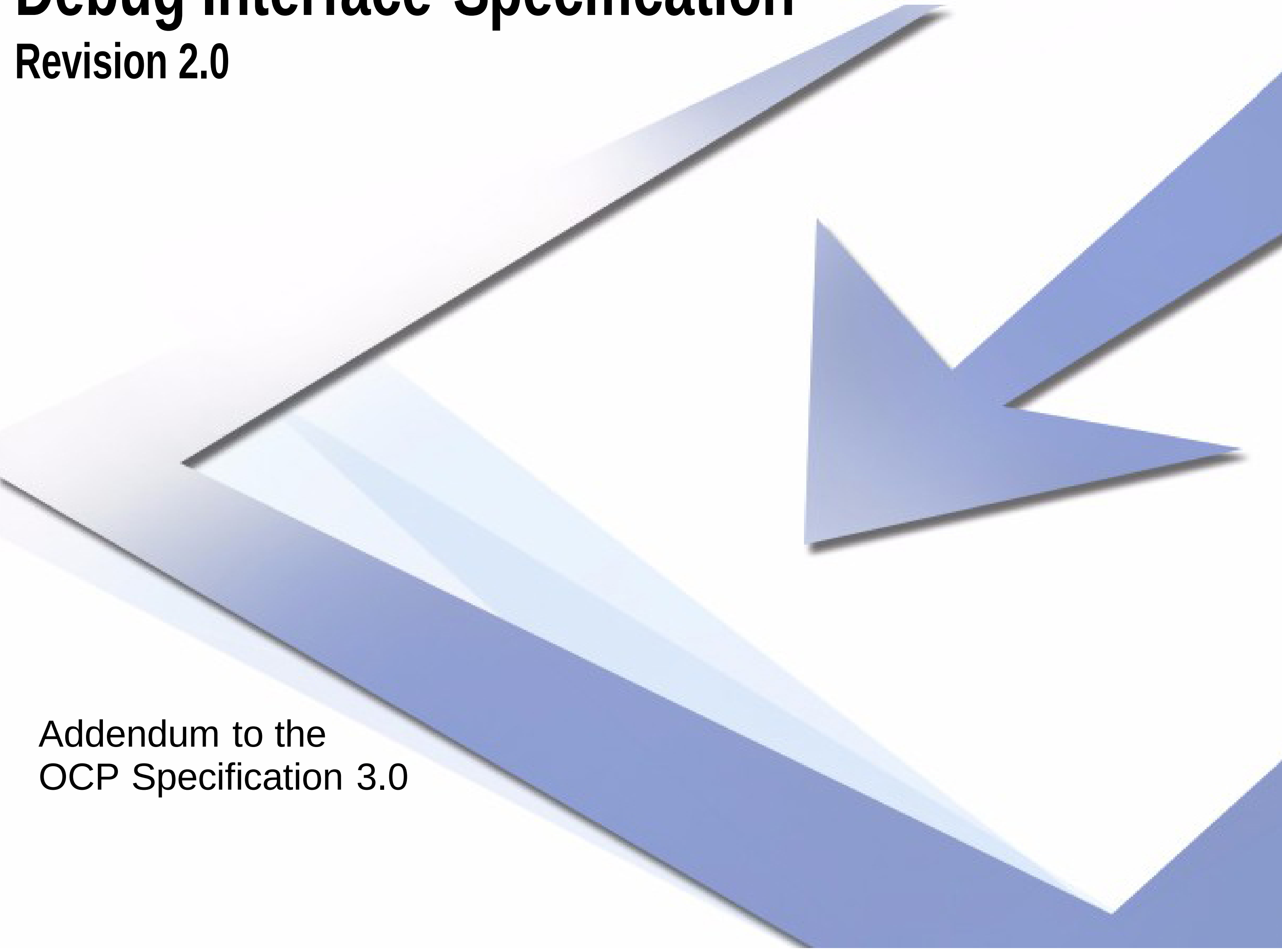
Open Core Protocol
Debug Interface Specification
Revision 2.0



Addendum to the
OCP Specification 3.0

Open Core Protocol Debug Interface Specification Document Revision – 2.0

In September 2013, Accellera Systems Initiative (Accellera) acquired certain assets of OCP-IP. These assets include the current OCP 3.0 standard and the supporting infrastructure. OCP 3.0 was released by Accellera in October 2013.

Notice

> Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. Accellera Systems Initiative is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patent Claims or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the Accellera Systems Initiative IP Rights Committee.

The trademarks, logos, and service marks displayed in this document are the registered and unregistered trademarks of Accellera, its members and its licensors. The following trademarks of Sonics, Inc. have been licensed to OCP-IP and subsequently to Accellera: FastForward, CoreCreator, SiliconBackplane, SiliconBackplane Agent, InitiatorAgent Module, TargetAgent Module, ServiceAgent Module, SOCCreator, and Open Core Protocol.

The copyright and trademarks owned by Accellera, whether registered or unregistered, may not be used in connection with any product or service that is not owned, approved or distributed by Accellera, and may not be used in any manner that is likely to cause customer confusion or that disparages Accellera. Nothing contained in this document should be construed as granting by implication, estoppel, or otherwise, any license or right to use any copyright without the express written consent of Accellera, its licensors or a third party owner of any such trademark.

Table of Contents

1. Introduction

Version 3.0 of the OCP Interface Standard defines two new classes of signals now part of the OCP Interface: Low Power signaling and Coherence signaling. The OCP Debug Interface Standard 1.0 is hereby extended to include those two signal groups and to clarify how to design for debug visibility into these new concepts.

The goal of the OCP Debug Standardization Working Group is to foster OCP compatible Debug IP blocks that can be reused in ASIC and FPGA designs.

In the past, we found existing Debug IP blocks that could be reused in new designs posed challenges if a substantial adaptation to a different class of processors was required. This adaptation alone could require over 6 months work by up to 3 people. Therefore a classification of processors and Debug-IP blocks is presented to measure the similarity and help estimate effort required if adaptation is desired.

At the time of this writing no OCP compatible debug IP block yet exists on the market, but several general debug IP blocks for Multi-Processor SoC exist that could be adapted to OCP with a wrapper.

This version of the OCP Debug Interface Standard offers a definition of logical levels and timing of the involved Debug Interface signals.

2. Abstract

The previous version of the OCP Debug Interface Standard contained 6 basic and 4 extended debug sockets. This latest version now offers two additional sockets for debug interfacing. Most complex processors can be debugged with exact visibility of traffic to or /from the OCP bus on all levels of transactions including all transfer states.

The focus of this specification is on heterogeneous multiprocessor systems that are most complex to debug, simpler systems are covered implicitly.

3. OCP Debug specification overview

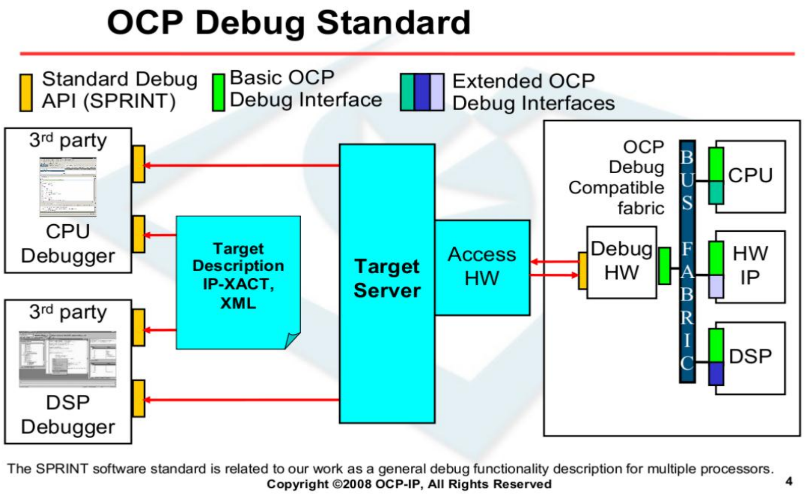
The OCP-IP Debug Working Group was formed in 2005 to address the definition of debug resources and integration to enable comprehensive debug of OCP based systems. Membership has included several OCP-IP companies (IP vendors, systems integrators, end customers) with interest in contribution to and need for debug solutions and/or debug of the OCP Infrastructure. The initial focus is on

a. Definition of debug signals for the OCP Socket and fabric levels, but leaves their specific implementation open to IP and tools vendors.
b. System level interfaces, in particular the merging of ESL, Debug tools and software.

The scope does not address in detail external debug interfaces between an OCP on chip Debug environment and external components (probes, debuggers, etc), which can be considered separate from the OCP Debug System and which are being addressed by other industry working groups (Nexus, MIPI, IJTAG, Multicore Association Debug Working Group, etc). The OCP Debug Infrastructure defines requirements for interfaces between the Debug and EDA infrastructure and features, but do not address specific debug to EDA interfaces and implementations.

3.1 OCP statement of need for debug standardization

With the evolution of heterogeneous MC-SoC (Multi-Core Systems On the Chip) the debug interconnection deserves special attention. A set of standardized signaling and definitions make the debug wiring core-independent to match the aim of the OCP standard. By doing so we stimulate development of predefined and verified debug IP blocks for quick and successful assembly of large MC-SoC including debug functionality like multi-core cross-triggering, MC tracing, bus event observation, debug bus masters, … that communicate with the next defined debug interface on the software side, the DAS API (Device Access Server) see figure 3.1. The existing core-specific debuggers for the hardware level debug and for the software level debug connect seamlessly and transparently to their hardware cores using this software DAS-API interface.


OCP Debug Standard
Standard Debug API (SPRINT)
Basic OCP Debug Interface
Extended OCP Debug Interfaces
3rd party CPU Debugger
3rd party DSP Debugger
Target Description IP-XACT, XML
Target Server
Access HW
OCP Debug Compatible fabric
Debug HW
BUS FABRIC
CPU
HW IP
DSP

The SPRINT software standard is related to our work as a general debug functionality description for multiple processors.

Figure 3.1 DAS concept with a Debug Target Server

OCP compatible IP blocks typically need debug interfaces that are not comprehended in OCP Socket specification. Additional debug signal interface definitions make OCP an “Even More Complete Socket” by addressing the visibility and control needed to better analyze the operation of OCP architectures and their design flows and providing a common set of debug options and consistent signal interfaces. These signal interfaces are not currently being addressed by the OCP 2.2 specification. Enabling a robust on-chip debug capability is being recognized as an important Design for Debug (DFD) capability for complex SoC and having DFD standardization makes OCP more attractive as a SoC platform.

The OCP Debug definitions addressed are defined using a common set of interfaces to allow improved incremental support from debug probe and tool vendors and EDA tool vendors to allow debug and verification convergence. In most cases, the interfaces and requirements are discussed, while the specifics and detailed design itself are treated as a black box which may be implemented in different ways by IP companies and other OCP members.

The ESL (Electronic System Level) abstraction of MC-SoC is often used to simplify chip system exploration complexity and reflects all hardware interfaces by SystemC and C++ models with signal true and port exact realizations. The debug interfaces, on the hardware side OCP and on the software side DAS-API, are excellent candidates for this abstraction and offer unprecedented debug experience in the SLD (System Level Design) project phase equal to the final hardware without any changes.

3.1.1 Other ongoing Debug activities

There are several working groups that are addressing aspects of DFD and related on chip debug problems. The work addressed by the OCP Debug Group complements these other efforts. Most notably, the Nexus (IEEE 5001) Forum has been focused on high performance trace related interfaces based on the IEEE 5001 standard and the MIPI Consortium has a debug working group effort that has focused on low pin count debug interfaces. The OCP Debug working group focus differs from Nexus and MIPI activities by focusing on debug interfaces and operation standardization at the socket level to the OCP interconnect, i.e. at the core and bus interface level, to address OCP compliant IP or subsystem debug control and visibility. These interfaces are interoperable with and do not overlap the IO level definitions that are the primary focus for other organizations such as Nexus and MIPI.

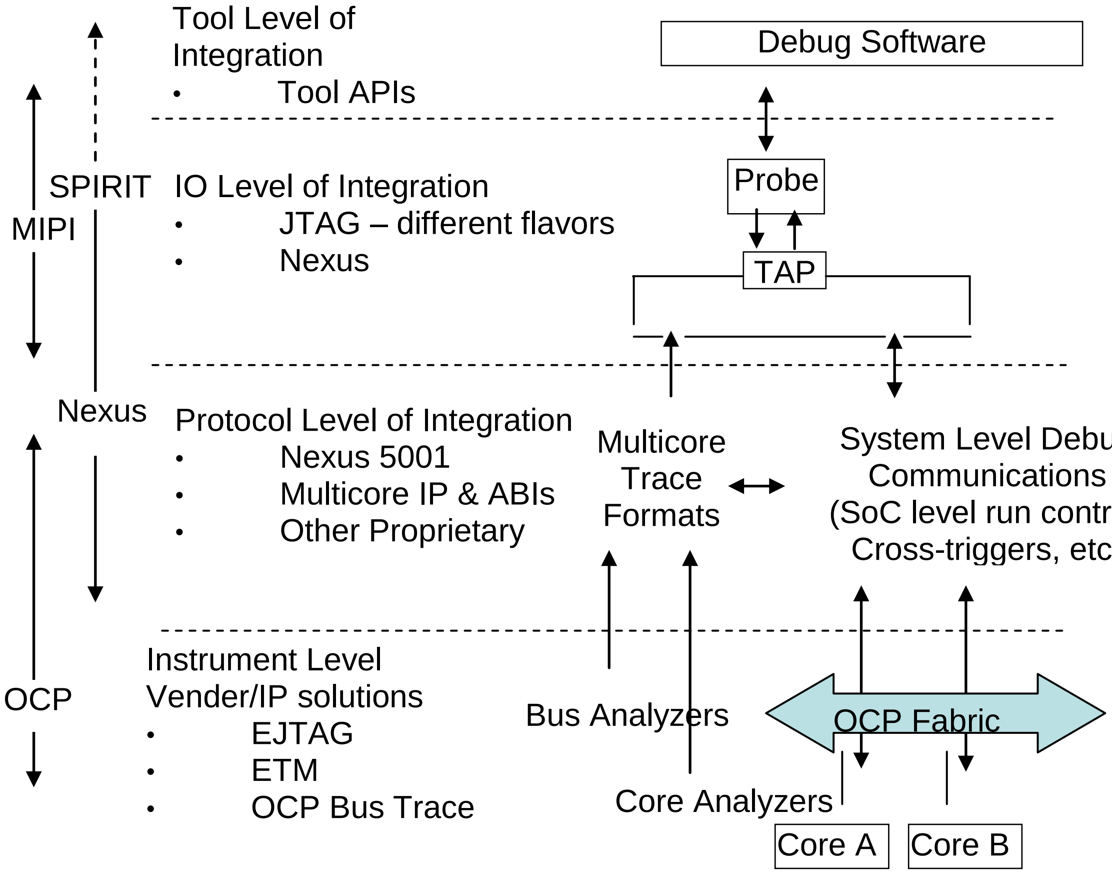


Figure 3.1 – Debug Infrastructure landscape

3.2 OCP Debug strategy

The initial goal of the working group is to document a common set of Debug Guidelines and signal models that address the range of simple to more complex debug of OCP based systems. Many approaches are based on contributed prior art. Others identify new capabilities that are needed for debug configurations and strategies that incorporate embedded analysis that comprehends multiple clock domains, power management domains, security domains, etc. required in modern SoC and embedded systems design.

Where possible we leverage signals and interfaces defined in other OCP specifications. As an example, JTAG signals as defined in OCP2.0 are the initial primary debug interface. Due to (largely bandwidth) limitations of JTAG for SoC debug, other options are presented where possible, to enable alternatives. As example, for debug control interface, we discuss memory mapped debug control options that use one of the embedded processors for debug configuration and control as alternative to JTAG. Similarly, we discuss trace port interfaces (in the general case, compatible with Nexus Interfaces as defined by IEEE 5001) as higher bandwidth parallel trace port alternatives to JTAG serial interfaces.

3.3 Common guidelines and infrastructure

There are two preferred methods of mapping the registers of the debug IP-blocks - such that all debug registers should be memory mapped to fit well into the usual programmer’s models and allow for standard and extended testability concepts in manufacturing:

a. memory space mapping. – On-chip processor core can operate the debug blocks
b. JTAG mapped register access - controlled by external software debuggers over JTAG can operate all debug IP-blocks

Comparative two-channel debugging with true time display of events is similar to the Logic Analyzer philosophy. The time aligned display of system bus traces of data events from different initiators on different buses is the main source of information. Setting of triggers on any signals or combination of events from different cores, IP-blocks and firing assertions is also basic to this idea. That is accomplished by the cross-trigger debug hardware block.

The standard debug method from instruction to instruction is possible but is enhanced by more hardware close concepts described below.

3.4 Scalability and configurability of debug resources

The OCP Debug Interface is following the general concept of master-slave request-response philosophy to assure easy mapping of existing signaling schemes in the contemporary debug interfaces to various cores and IP-blocks including assertions.

In general there will be two signal wrappers required on the hardware side: Between the core and the debug interface to the OCP interconnect and between the OCP interconnect and any existing debug infrastructure. Gradually debug infrastructures will be developed that connect natively to the OCP Debug Interface without a wrapper.

The main idea of presenting it as a Debug Socket to the SoC designers is a structural regularization to minimize errors in understanding of its functionality and to allow the development of automatic checkers for this well defined debug interface.

The OCP Debug Interface is part of the OCP Sideband signaling scheme. It is partitioned into Basic Signals and Extended Optional Signal groups as found similarly in the OCP data socket definitions. The Basic Signals assure run control for debuggers and Extended Signals deal with special situations like voltage islands, security islands and power down modes. Performance metering and assertions are part of the optional signals in the Debug Socket.

3.5 Comprehensive set of debug features

- Debugging in the real target system: No mechanical or electrical constraints
- Full visibility: Cycle-accurate trace of multi-processor, multi-bus SoCs
- No limitation for low-pin-count, high-frequency devices
- Complex triggering modes—for example, triggering on an event not happening—allowing you to minimize the amount of trace data you collect
- Support for code profiling and performance analysis through programmable event counters
- Portability: OCP Debug Interface is adaptable to any processor or bus architecture; software developers continue to use tools they are familiar with
- Low cost: No expensive hardware needed to access OCP Debug Systems
- Proven implementation: OCP Debug System prototype was executed successfully
- Non-intrusive debugging of embedded multi-processor systems
- Target system runs at full speed in application environment
- Access to internal buses
- Real-time, cycle-accurate tracing
- Trace capabilities for:
    - Processors: Process ID, program, data, status, watch point
    - Buses: Data, status, watch point
    - Signals: Status
- Complex trigger system including cross-target triggers
- Translates raw data into meaningful messages
- Compresses trace messages to save memory
- Trace memory can be configured as a circular buffer to collect trace messages either continuously or before and/or after a watch point occurs
- Implementation partitioned for easy adaptation to new cores
- Independent of physical interface between chip and debug host (DAS)
- Security: OCP Debug System is locked by default and can only be unlocked by system hardware

3.6 OCP debug Business Model

This Debug Specification is assumed to be openly available to all OCP-IP member companies. All implementations as discussed remain the properties of the contributing parties. The goal is to facilitate development of debug IP and tools for sale to the OCP community and this document shall in all cases, attempt to place no limitations on specific OCP debug architectures, either for internal use or for sale.

4. Technical approach

In this section, an overall debugging framework is described as a base of the OCP debug interface. In the same way like the OCP data socket was a superset for the different bus interfaces and data structures we seek to define an OCP debug socket that can be a superset of the debug solutions. Most concepts discussed are based on common denominators for the past and present debug concepts. The strategic goal of this document is to enable OCP members to develop standardized libraries of debug IP blocks for debug situations and purposes, including

- Signal level observation (bus and system trace) and control (triggering)
- Consistent (multiple) processor software debugger and bus traffic observation interfaces (GUI)
- Special debug features for security islands, voltage islands, gated clock islands etc.
- New classes of debug errors (which are different from system errors) analysis

The debug concepts addressed can be applied to single core debugging (without cross-triggers, trace, or time stamping) and it can be extended to more core and channels of debug for more complex systems.

For multi-core chips, there is implicit debugging requirement to observe activity of (at least) two cores out of many in order to enable comparative analysis of operations and communications. As a default, OCP debug interfaces should support multiple cores. We use a dual channel synchronous debug socket as an example. Dual channel debug is a minimum to enable comparison, and synchronous means that instructions and events must be displayed in correct temporal relation what is accomplished by time stamping during collection of trace information. The idea is similar to a dual channel logic analyser and when cores are not in debug mode then any two IP blocks can be observed or traced in temporal comparison with a common and extensible set of signal interfaces.

We must recognize that the purpose to debug in a chip can be very different and at least three variants want to be satisfied by a standard:

- Pure software debugging concentrates on minimum additions to proven hardware still providing a rich debug environment for development of software.
- Pure hardware debugging concentrates on simplest additions in hardware to expose chip internal signals on the pins (JTAG) to be used to prove correct functionality and correct design.

- System debugging concentrates on software debug and hardware observation. We avoid defining a separate debug bus to keep a simple modular IP structure on the chip.

4.1 Three Views of Debugging

Debug as a process can differ between companies, projects, and points in the design cycle.

Pure Software Debugging

Pure software debugging concentrates on minimum additions for instrumentation to proven hardware and IP while still providing a rich debug environment for software. The debugger connects to the processor that programs all debug-hardware over the system bus. Target system hardware is fully utilized for debugging. Assumption is that all hardware is correct. Special instructions and signals to let the processor prevail in locked situations are desirable and included in the basic OCP Debug Interface signals. This style of debugging is well documented on several chip architectures. Systems are built by connecting several proven chips together, therefore debugging with inter-chip cross-trigger is a second special requirement. To simplify a dual trace memory one trace buffer can be used in connection with “synchronous run” from optional debug signals. That will make the ordering of events in the trace buffer in correct temporal relation possible without time stamping.

Pure Hardware Debugging

Pure hardware debugging concentrates on simplest additions in hardware to expose chip internal signals on the pins (JTAG or other) or in registers to be used to prove correct internal functionality or correct design parameters in mission critical applications and warranty cases.
Most important for this concept is an independent clock from outside that is reliably working even if the system clock is stopped. Also triggering precise to one system-clock cycle, or local-clock cycle, is essential to let this debug hardware react exactly like assertions in a simulation. Often signals inside IP blocks are observed. No software debuggers need to be involved in the display of this information, however we believe there are analysis advantages to including the display of such extra information in common software debuggers.

System on Chip Debugging

System on Chip debugging concentrates on software tracing and hardware observation requirements common in initial SoC. Observation of the on chip hardware interaction is essential to complete the software application and verification. Comparative debugging of any two cores is equally important for multicore systems. The debug-system is independent of the target hardware and captures both “pre-reset” and “post-crash” events as well as bus traffic bottlenecks. Debugging must proceed even if the major parts of the system are in power down or a core is in sleep mode. The debug hardware may be shut down during normal chip operation, for security or power improvement. Another security demand is to make parts of the debug hardware inoperable in production chips by burning fuses. Such a concept of debugging is best suited to support ASIC designs. To simplify a dual trace memory one physical buffer can be used that holds two compressed trace streams with origin tags and time stamps.

Debug features need to support the system level verification and analysis of OCP based systems. Where possible, RTL or other (System-C) blocks should be available for EDA analysis for JTAG and DFT, BIST, and other debug structures, even when these are implemented as physical (post synthesis gate level insertion) macros. From a system point of view, debug blocks should support the same level of model abstractions used in other areas of a design, in order to support it with miscellaneous simulators and software debuggers and eliminate special hardware and software personnel involved in the debug process.

4.2 Technical scope of OCP Debug Interface

In the next section we define 4 groups of signals that are basic to an OCP dual channel synchronous debug interface (debug control, JTAG, debugger interface, cross-triggering interface) and additional (extended) groups that are optional based on specific debug and analysis requirements. The optional Extended Debug Signals in this interface are defined for optional debug features such as time stamps and performance analysis and to simplify definition of special “debugging aware” functionality in designs that have security domains or power management with voltage islands. The figure below has overview of data and debug sockets in OCP.

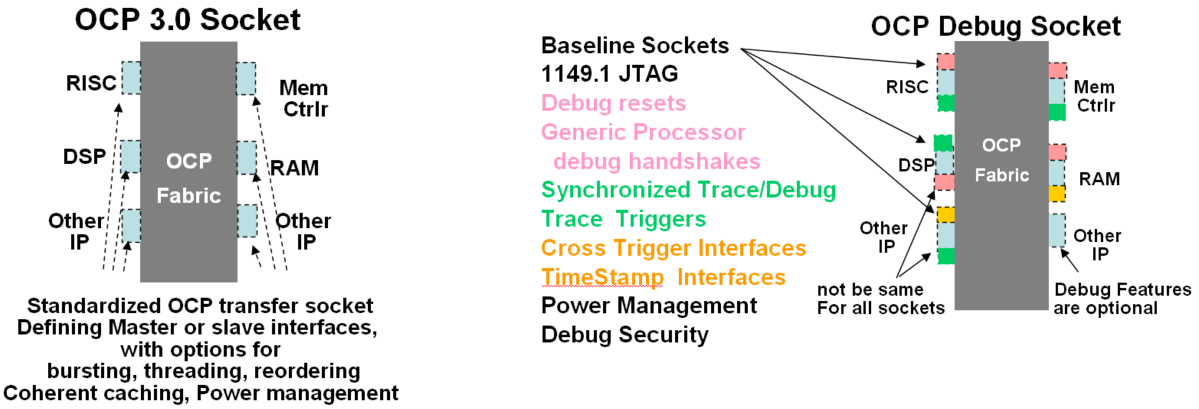


Figure 4.1 (a) OCP Sockets for various cores (b) OCP debug interface sockets

5. Debug Components and IP Interfaces

This section covers the basic signals and definitions for an OCP Debug Interface Socket. An optional OCP port, known as the Debug Interface port, is added to all cores and IP blocks that support or need debugging access. The OCP debug port may be implemented as an addition to the OCP Data (master and/or slave) port (in cases where debug blocks are memory mapped they may be controlled through the OCP data socket) or as an independent OCP port configuration.

Figure 5.1 shows a simple system where debug IP blocks deal with standardized signals integrated into a interconnect structure that is implemented as a modular OCP system created from library IP blocks around a bus fabric (as described in the SPIRIT XML conceptual framework). All debug wiring goes through the system bus and is contacted through the OCP debug port. OCP debug ports may be implemented at points in the OCP system where a Data port may not exist.

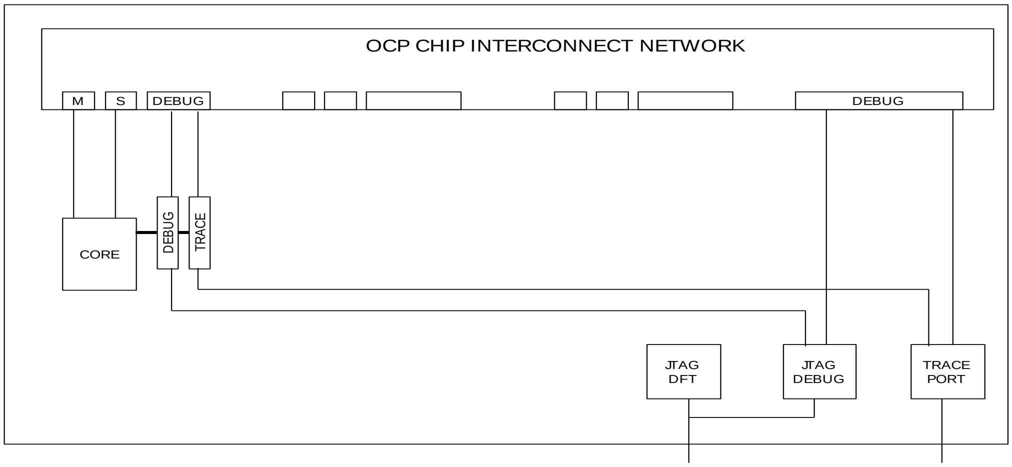


Figure 1- A One-Core Debug Situation

Figure 5.1 – Single Core Debug solution

5.1 Debug interface definitions

The programming of registers that contain either configuration or status information in the debug IP blocks may be JTAG-mapped or Memory-mapped. Either or both modes of control and access are acceptable, based on specific system requirements.

In the Memory-mapped case the master port of the main debug core provides the programming of the debug block registers that have an address in the main memory space. The master OCP data port is not part of the OCP debug port. This allows one core to be the main debug agent. The debugger sends instructions over JTAG to this core and the core accomplishes all actions through the main system bus. Then this one core needs special priority access to unlock stuck interfaces and locked transactions. Signals for this purpose like “Abort” and “Force” are part of the debug control interface.

In the more general and system independent JTAG-mapped register variant the JTAG is part of the OCP Debug Interface. Debugger sends instructions to the cores over JTAG and the debug registers are part of a JTAG-TAP controller. Optional “Abort” and “Force” signals are also JTAG controlled. (Time stamp in 4.6 and trace compression A.2 is explained later.)

For simplicity, debug ports discussed are limited to 1149.1 JTAG interfaces, as defined in OCP 2.0 specification (and many other documents). This restriction is initially done to simplify the interfaces initially addressed. The intent is not that implementation be necessarily limited to 1149.1 JTAG. With a separate interface layer debug interfaces discussed can be compatible with Nexus TAP and protocol and other debug interfaces – CJTAG, SerDes (Aurora), single/dual wire, etc, which are not discussed in this release of this document.

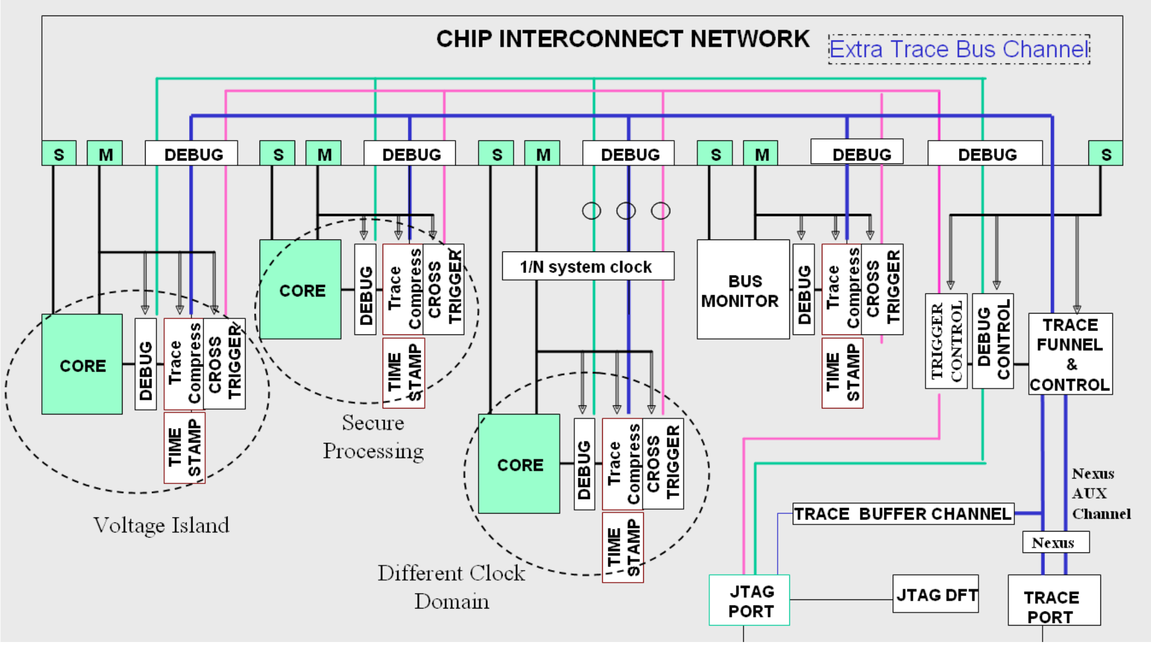


Figure 5.2 –Multi-core Synchronous Debug Implementation

5.2 Basic Socket Level Debug interfaces

Processor run control interfaces are typically implemented via the JTAG interface and by debug mode signals in the IP. The use of JTAG debug interfaces is supported via the OCP2.0 specification and is it is assumed that any JTAG signals are decoded at the core level JTAG TAP (Test Access Point) and are not addressed in this document. A JTAG only debug interface does place limitations on the ability to interface debug components on different cores and to set up and synchronize an OCP system into a debug mode.

Table 5.1 OCP Debug Clock, Reset Interface Socket

Debug Interfaces	*Description*	*Comment*
Debug_reset_n	Debug Clock source for instrumentation operation and Optional debug system reset	defined to be separate from system clk, reset so that debug can occur during systems reset operations
Debug_en	General enable for debug modes	system input

Ideally debug control signals are independent of the **target** system and have to duplicate many basic controls. The basic debug signals include an independent reset and independent clock signals for the debug system synchronized to the debugger interface. The reset and clock signals for time stamping counters are also part of this debug control interface. Often debug reset, debug clock, time stamping reset, as examples, may be common with system clk or reset. In chapter 12 signal timing and levels are described in more detail.

Table 5.2 OCP Debug JTAG Interface Socket

JTAG Interfaces	***Description***	***Comment***
Tck, Trst (optional)	JTAG TCK , JTAG Reset	JTAG input
tms	JTAG TMS	JTAG input
tdi	TDI from previous node in JTAG loop	JTAG input
tdo	TDO to next node in JTAG loop	JTAG output

The programming from debugger happens over JTAG so the 5 JTAG signals are included here. Even in the case of “memory mapped” debug blocks the processor control typically goes over this JTAG port. In chapter 12 signal timing and levels are described in more detail.

5.3 Core Debug Socket Interfaces

Table 5.3 defines a set of debug interfaces that address system level debug of run control and debugger tools interfaces. Debugger accesses can therefore be consistently controlled via the debug interface signals. Not all signals may be required for all cores or systems.

- Special signals that support unlocking of stuck situations and forcing completion of locked actions (**NoSResp, ForceResp, ForceAbort, ForceAbortAck**) are here.
- Debugger accesses are qualified through **MReqDebug**
- Processor acknowledges debug state entry through **MSuspend**
- An OCP target can be configured to be sensitive to **MSuspend** line
- A debug component is informed that Debugger is connected through the **DebugCon** signal
- A subsystem is informed that its TAP has been enabled by the application security software through the **TAPenable** signal. (Security is explained in chapter 4.4.)
- Depending on **DebugMode[1:0]** the debugger can initiate OCP transactions qualified as **MReqSecure.** (Security is explained in chapter 4.4.)

Table 5.3 OCP Debugger Interface Socket

Signal name	**Signal definition**	**Comment**
Minimum OCP debug signals set		
MReqDebug	Qualifies an OCP request initiated by the Debugger. MReqDebug may be a processor native feature.	If MReqDebug is derived from processor debug acknowledge, the OCP interface shall insure there is no outstanding application transactions when debug state is acknowledged. Write buffer shall be drained

Msuspend	Processor acknowledges the OCP initiator agent that is entering the debug state.	The OCP Initiator debug state acknowledge is routed to the OCP target – A debugger aware peripheral may (optionally) freeze a local HW process when the host enters the debug state.
DebugSerror	Out of band error	originated by debugger
DebugCon	Debugger is connected	Enables the on-chip debug hardware being active
NoSResp	Target not responding	Debugger status shall be managed for cases such as Sresp = FAIL, SCmdAccept failing. (Request phase or Response phase) and provide an indication that current transaction doesn't complete regardless of the root cause:
ForceResp	Debugger has programmed the subsystem to force a data independent response	No side effect to other threads
ForceAbort	Debugger has programmed the subsystem to solve the hang scenario	OCP interconnect handle abort without debugger intervention even in the case where the application SW has not enabled a time-out. The key requirement there is to complete the transaction in order to allow the processor entering the debug-state. This requires a Mabort input support in the OCP fabric to propagate the abort originated by the debugger to the initiator and OCP interconnect.
ForceAbortAck	Acknowledge sent to subsystem (or debugger?)	

Control signals in case of gated clock domains and voltage domains in that the continuity and functionality is not interrupted if any IP block on the bus switches off clocks or voltages. By proper definition of the idle levels a blocking of the debug system shall not happen if one core or IP block goes into sleep mode**.**
In chapter 12 signal timing and levels are described in more detail.

5.4 Cross Triggering Socket Interfaces

Information in MultiCore SoCs is complex and distributed such that global event cross triggering and system level control for multi-core debug and triggering is often needed to identify and isolate events occurring throughout the system. Event recognition and triggering is widely used in conjunction with trace to capture information on events and operations in the SoC. Conditions are monitored and compared to generate real time triggers in a Cross-trigger Manager. These triggers in turn can be used to control event actions such as configuration, breakpoints and trace collection. More complex implementations can be programmed to trigger on specific values or sequences such as address regions and data read or write cycle types.

The cross-trigger block may be distributed to all IP connections to the OCP bus. If wiring is in the OCP fabric then some pre-processing or wrappers (condition/action nodes) at each OCP interface can be used to simplify the cross-trigger information. Wrappers can be programmed via JTAG debugger (or native to a processor). Any block can send a trigger (edge or level) and receive a trigger. Debugger or processor can configure specific trigger lines for each IP to send a condition signal and from which trigger line it can receive a trigger/action).

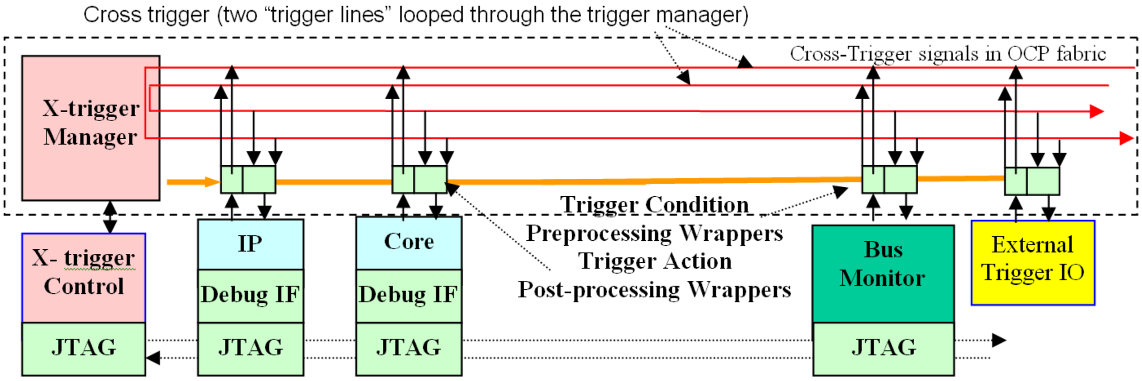


Figure 5.3: Cross-Trigger Block Diagram

Each trigger line consists of two uni-directional signals and one (optional) enable signal. A minimum dual channel concept consists of two independent trigger lines, but there is no upper limit on number of cross-triggers realized in a design. The trigger line in, out, enable may be the result of logic combination of several signals for a given core. Trigger lines can be connected directly to drive a bidirectional pin on the package and enable cross-triggering to continue between several chips. External (off chip) triggers will be supported with pulse width logic to interface external IO to the cross trigger manager.

Each debug channels needs one trigger line. The trigger logic grows linear with the number of cores or IP blocks that are debugged. No quadratic cross-trigger matrix is assumed necessary. In chapter 12 signal timing and levels are described in more detail.

Table 5.4 OCP Debug Trigger Interface Socket

Cross Trigger Interfaces	***Description***	***Comment***
Trigger_in_ condition[n:0]	Trigger input from other OCP subsystems	X-trigger input – shall support either High to low edge detection or level detection – during power down of subsystem, trigger_ in will not contribute to system cross trigger
Trigger_out_ Action[n:0]	Trigger output to other OCP subsystems	X-trigger output of either - Active low pulse or level–supports trace control or processor debug or interrupt request.

Trigger_out_enable [n:0]	Optional Trigger output enable to other OCP subsystems	X-trigger output
Ext_trig _clk	Optional Ext clock used for synchronizing trigger action	Ext. off chip input
Ext _condition[n:0]	Optional Ext condition (e.g. debug status, tracepoint)	Ext off chip input
Ext _action[n:0]	Options Ext action (e.g. debug request)	Ext output

5.4.1 OCP Cross Triggering – General Requirements

- Cross Triggering configuration shall be handled at subsystem level
- Subsystem can be programmed to:
  - Drive an OCP debug-trigger-out line
  - Be sensitive to an OCP debug-trigger-in line
- The OCP interconnect shall take care of the debug event triggers routing
  - Point to point [1 trigger-out and trigger-in]
  - Broadcast [1 x trigger-out, n trigger-in]
  - Sharing [n trigger-out, 1 trigger-in]
- The OCP-debug interconnect shall mimic a “tri-state” bus behavior through distributed combinatorial logic.
- An external device shall be able to contribute to cross-triggering

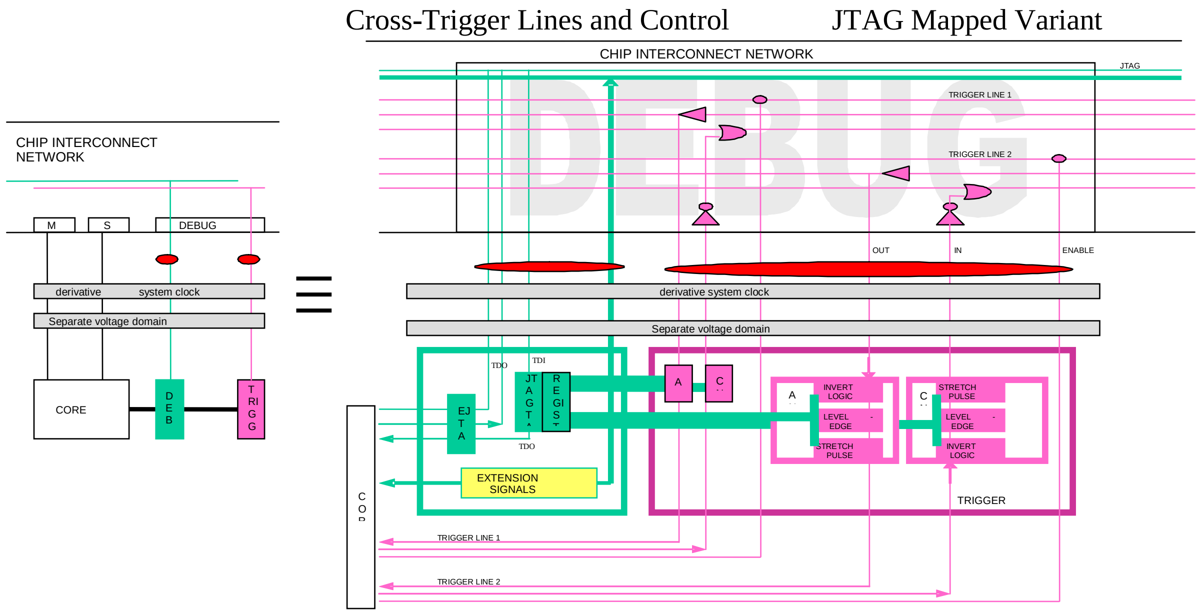


Figure 5.4 - Cross triggering interface over separate Voltage domain and Clock domain

5.4.2 OCP Cross Triggering – General Configurations

- Trigger-out (action) & Trigger-in (condition) routing for smaller implementations can be handled as sideband signals by the OCP interconnect.
- Trigger event may also be routed to Trace components
- Trigger event shall generate a user defined request. This is typically classified as either a debug request or interrupt request. These differ for different cores.
- The OCP cross-triggering shall be operational for any platform subsystems frequency operating point supported by the cross triggering configuration via level or pulse triggers
- The OCP cross-triggering supports independent clock domains for trigger-out and trigger-in pulse conversion. Level triggering is recommended for widely varing clock domains
- The OCP cross-triggering shall support external triggers. Triggers can connect to IO. Level or pulse triggers are supported with trigger pulse width modifiable to compatible with device I/O performance.

A subsystem in power down or where debug has not been enabled shall be configured not to contribute to cross triggering.

5.4.3 Fundamental Trigger Limitations

We must recognize that system observation using trace buffers and triggering on simultaneous events system-wide including cross-triggering between chips are concepts with limitations in time resolution and that translates into distance limitation as described in the first approach. To overcome limitation in space we can give up the precision in the feedback of the result as described in the second approach. To mimic a logic analyzer trigger we need to have delay-equalized star-configuration to the trigger controller that will behave the same like the second approach. Designers must decide which approach to take to create a consistent debug system.

5.4.4 Cycle-exact Trigger and Feedback

In this concept it is crucial that collection of all trigger conditions and distribution back to the origin happens within a fraction of the highest system clock cycle. Advantage is that sequencing of trigger conditions that are one cycle apart is possible even at the trigger sources. Difficulty is to close timing in such a design since the trigger path becomes the biggest bottleneck on the critical timing path.

The modern debug concept with assertions and assertion chains on-chip demands a cycle accurate trigger concept if the comfort in the simulation shall be synthesized into silicon with the same functionality. Therefore, it has to deal with increased difficulties in timing closure. Or it has to flag certain chains of assertions as not synthesizable at high system clock rates.

The proposed trigger logic in the OCP debug socket is based on a distributed model of a tri-state wire. The trigger events are collected with a chain of distributed AND-gates and the result is sent back over a second wire in a half-loop arrangement. The trigger controller connects to "the last OCP debug socket" at the end of the AND-gates and loops back the result to the second wire.

5.4.5 Cycle-exact Trigger with Relaxed Feedback

This concept accepts the feedback signal on the second wire to arrive in a later cycle to help with timing closure. Means that detection of a trigger equation has to happen within one cycle but distribution back to origin, for example to stop a trace buffer can extend over several cycles. To chain this delayed trigger result with distributed consecutive trigger decisions in assertions will only work for events that are several cycles apart.

Aligning debug information in the display to be cycle exact is by using local system-cycle-exact time stamping during collection of trace information. Then stopping the trace buffers few cycles after a trigger condition will still allow for exact time alignment in the display.

The trick is to equalize the arrival time at the trigger controller from any trigger source by inserting delay buffers before entering the AND-gate trigger line. Then it is possible to trigger on events that happened at the same time. Sequencing of triggers that happen one cycle apart is possible inside the trigger controller by using multiple arrival-time-equalized trigger lines.

Same like a logic analyzer with delay-equalized cable-probes can trigger on the acquired signals but does not supply trigger information back to the device under test, the OCP debug system with a relaxed feedback concept, does not demand to have delay equalized feedback connections back to the trigger sources.

For triggers coming from all corners of a big chip or for systems made of many chips this is a very good solution. It scales well to any size of a system and can have extra built-in arrival-delay of "several clock cycles" to accommodate triggers coming over external pins. The proposed OCP-debug cross-trigger concept can be used for this configuration. The fixed built-in target trigger arrival delay is independent of the highest clock in one chip or in multiple chips.

5.4.6 Exact triggering in a star configuration

Similar to a logic analyzer the cycle accurate trigger timing can be designed by delay-equalized trigger lines going to the trigger controller in a star configuration. This requires a separate trigger line from each possible trigger source. Then any sequence of trigger events can be realized cycle accurate inside the trigger controller. Yet, the feedback to the trigger sources, or to the assertion blocks, to allow them to do cycle accurate trigger sequencing is still not assured or possible. Means the stopping of tracing buffers still happens few clock cycles later.

This star topology concept can be made cycle accurate in any system at the expense of individual trigger lines with delay equalization. Clearly, this concept does not scale with large systems since wires grow proportionally to sources and not proportionally to trigger decisions. Star configuration is not part of this proposal because the arrival-time equalization with the proposed distributed AND-gate trigger line will work absolutely equally well.

5.5 OCP Synchronized Run Control

Sync Run Control allows a clock synchronized program execution of two cores that would run asynchronously in normal case. That makes it possible to time align the instruction streams to study interdependency. This is a simplified method to avoid the hardware for time stamping. In chapter 12 signal timing and levels are described in more detail.

Table 5.5 OCP Debug Run Control Synchronization Interface Socket

Synchronous	Description	Comment
SyncRun	Synchronous run	
SyncRunAck	Synchronous run acknowledge	

5.6 OCP Trace Interfaces

A trace trigger is a trigger signals that provides trace enable and control for OCP Bus trace and other OCP IP trace and performance and analysis interfaces. In the case of real time tracing to outside pins this trigger signal is included in the trace part of the OCP Debug Interface socket. Trace trigger is extracted from the information on the cross-trigger lines. In chapter 12 signal timing and levels are described in more detail.

Table 5.6 OCP Debug Trace Interface Socket

Trace	Description	Comment
TraceTrigger[x]	OCP system event generates a trace trigger	

OCP Traffic Monitoring and Trace – General Configuration

- The “OCP System monitoring” debug component shall allow monitoring the “OCP System” bus traffic
    - Focus on specific OCP transactions
    - User defined transactions filtering
    - Initiator, thread, address range, DMA logical channel
- An emulator shall be able to configure the “OCP System monitoring” component from the external [JTAG] interface through the “OCP Debug” bus.

The “OCP System monitoring” component shall have options to allow:

- Align the OCP transactions requests & responses
- Capture additional OCP transactions qualifiers
- Export the captured traffic data through the “OCP Debug” to a “Trace Export” component
- Support continuous System monitoring
- Preserve the OCP System bus behavior

- Be security aware

The “Trace Export” component shall:

- Implement an elastic buffer
- Optionally build trace packets for different (MIPI/Nexus) protocols.
- Support a trace export bandwidth compatible with OCP System traffic peaks
- Allow SW instrumentation interleaving

The “Trace Buffer” component shall:

- Provide flexibility to disable capture around a trigger
- Allow system trace data reads
  - From the JTAG-OCP component
  - From the application SW
- Allow interleaving several trace flows
- Allow multi-threaded data observation

6. New Interface for Low Power Signaling

The new class of OCP signaling enables and controls the power state of any IP block in a system and is achieved through the following set of signals in the OCP Standard 3.0.

6.1 Connection Signals

The OCP interface offers an optional connection protocol that enables the master to control the connection state of the interface based upon the input of both master and slave, which can be used to implement robust schemes for power management and is not limited to just power management. The OCP connection interface works as a state machine controlled by the master that has 3 states and a wait state:

0	M_OFF	interface to slave is off, power down
1	M_WAIT	master is waiting to side signals to end their activity or to slave that asserted SWait signal
2	M_DISC	interface is in disconnected state
3	M_CON	interface is in connected state

Normal sequence is to go from M_OFF to M_CON or from M_DISC to M_CON to enable the interface. Sometimes using M-WAIT if immediate transition cannot happen.
To disable the interface we go from M_CON to M_DISC and to M_OFF in case of power-down. Sometimes using M_WAIT if immediate transition cannot happen.

The state of the state machine is expressed by MConnect [1:0]. Master shall not change the state without staying at least for 2 cycles in any state. Slave can assert SWait to buy time to finish its sideband signals activity. The OCP control signals used are:

- MConnect, SConnect, SWait
- ConnectCap (disables CONNECTION interface if tied low; means interface is always on)

This requires visibility and optional controllability during debugging. The new interface for debug consists of the following signals:

- ID address of a subsystem

- Interface status signal using MConnect [1:0]

- Force low power status, register that can deassert SConnect for any slave

How this interface can be used in debugging is explained in the following text. The aim of the debug interactions is to be informed about the power down status of any subsystem during specified clock cycles or instructions and to alter the power down status if desired.

To observe the power down status we will use the signals

- ID address of subsystem

- Power status signal (MConnect)

To control the power down status we will use the signals

- ID address of subsystem (when MCmd is valid MAddress expresses the target slave)

- Force low power status, register that can override and deassert SConnect for any slave

MReq signals must be stable at rising edge of the OCP clock. SResp signals must be stable at rising edge of the OCP clock. Connect signals can be asserted asynchronously but master must change state synchronously to OCP clock.

The timing diagram is below:

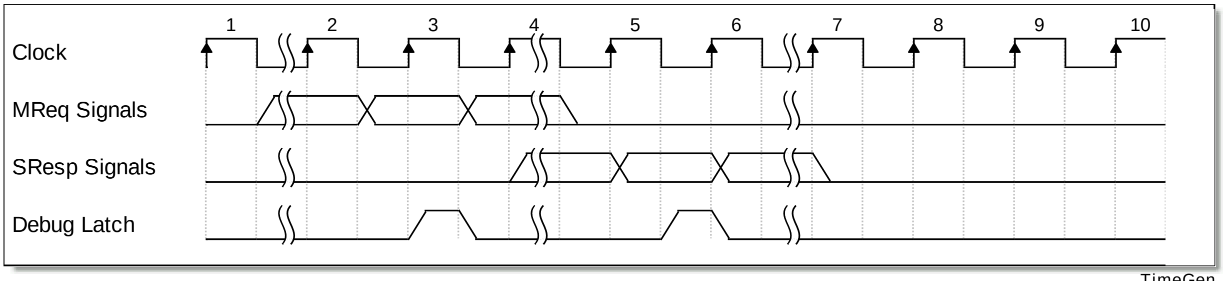


Figure 6.1 OCP Standard for general signaling between master and slave

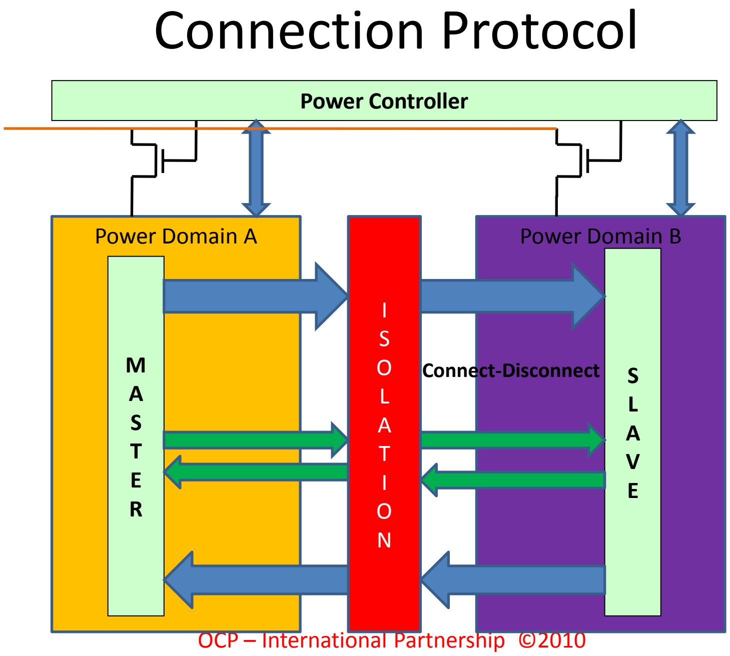


Figure 6.2 Debug interface for power control

7. New Interface for Cache Coherence Signaling

The new set of OCP signaling to enable the cache coherence in a multiprocessor design is composed of the following two parts: Extended OCPce port and a new intervention port OCPi.

The new signals in the backward compatible coherence extended interface OCPce are:
To enable coherence: *cohnc_enable; coh_enable; cohwrinv_enable*

New MCmd (5bit): CC_RDOW, CC_RDSH, CC_RDDS, CC_RDSA, CC_UPG, CC_WB, CC_CB, CC_CBI, CC_I, CC_WRI, CC_SYNC

New Coherence Group: *SCohState, MCohCmd, MCohID, SCohID, MCohFwdID, SCohFwdID*

The existing legacy OCP port has been augmented to an OCPce coherence extended port and an additional new intervention port OCPi has been created just for the exchange of coherence information. The part of the SoC system that shall be cache coherent must use both new interfaces on each processor and memory IP block.
The legacy OCP did not need identification of slaves since it was given by the data address. Coherence OCPce and OCPi ports need new identification ID for source and destination since different masters and slaves can be targeted in a command dealing with the same data address

in a cache. For debug purposes observing the OCPi intervention port will provide all information necessary. Quick overview of the new ports and connectivity is below.

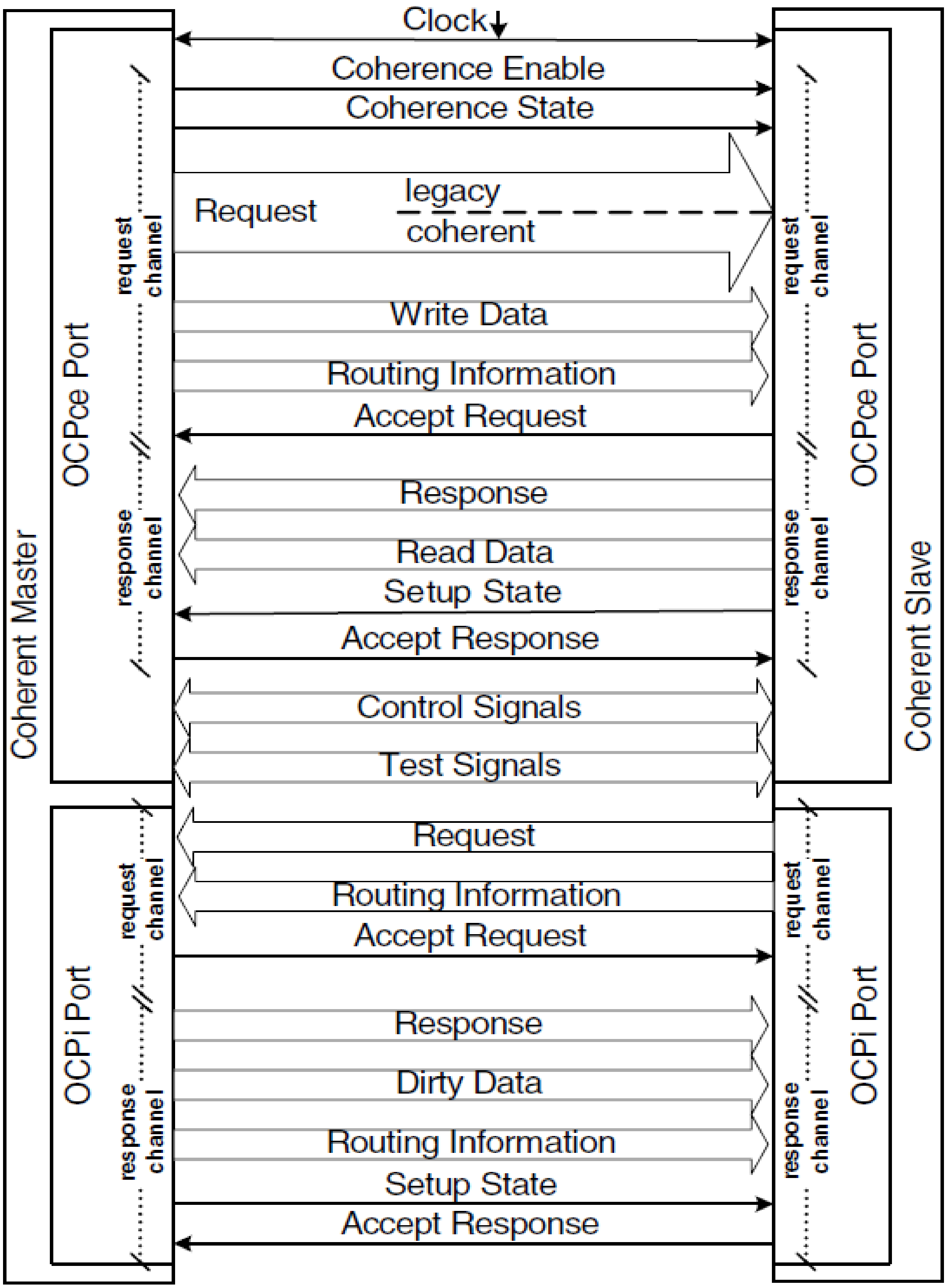


Figure 7.1 Old OCPce interface extended with OCPi interface for cache coherence

Signals in the new coherence intervention interface OCPi:

MCmd:	IDLE, I_RDOW, I_RDSH, I_RDDS, I_RDSA, I_UPG, I_WB, I_CB, I_CBI, I_I, I_WRI
SCohState:	I, S, M, E, O (Invalid, Shared, Modified, Exclusive, Owned)
MReqInfo:	e.g. , forwarding ID (0,1, or 2)
SRespInfo:	Optional, e.g., forwarding ID (0,1, or 2)
Clk: MAddr, MBurstLength, MBurstPrecise, MBurstSeq, SThreadBusy, SResp, SData, MThreadBusy	legacy signals

The concept of cache coherence is to refresh the data word in the cache of a processor that was copied from a memory. Refresh happens at the time when it is accessed and recognized as not up to date with the latest value held by another processor in its cache or in memory. That is a formidable task and if multiple processors change the same cached memory address constantly and one processor tries to work on it then a long delay may result. Processor is then stalled in that thread until the coherence update finishes. The updating works in the background non-blocking to other threads, transparent to the user, except for the "random" delays occurring in program execution to restore coherence. It takes 3 or 4 hops (cycles) to restore data coherence in one processor depending on implementation. See OCP 3.0 Spec, Chapter 5.3.2.

In the OCP Standard there are two examples how the signals in the standardized cache coherence interface could be used to build a cache coherent system with multiple processors. We rely on those two hypothetical examples to explain the debug visibility. Bus-snooping or directory based concepts are the two extreme schemas with many possible mixed flavors between. A very simple description can be found in literature LIT1.

To achieve visibility of coherence update activities in a debugger we describe the following levels of debug granularity with increasing hardware effort in the debug block:

- Was there a cache coherence update in progress during the instruction execution? (simple)
- Which processors were stalled because of coherence activity?
- Which processors caused a coherence stall and provided latest data from their cache?
- In how many cycles was there coherence activity? (complex)

The examples of a debug block to observe which processor is stalled because of which other processor that changed data and during which instruction it happened is like:

- Check if cache coherence data update is in progress during an instruction by using signals on the OCPi port:

  SRespData, SRespInfo, MReqSelf, intport_writedata=1 means update
  Every cache hit with data not owned and every cache miss needs a coherence request to coherence manager. The answer can be “use your cache” or “update cache with supplied data”. By observing the signal for self intervention we can tell that cache data must be updated if data is provided with it. Which instruction on which processor is being observed must be defined by the debugger.

- Check which processor cache is being updated by using signals on the OCPi port:

  SRespData, SRespInfo, SCohID, MCohID, SCohFwdID, MCohFwdID, MReqSelf
  All coherence cache data updates happen over the OCPi intervention port. By observing all self intervention requests on all processors if they come with or without data from the coherence manager we can tell the intensity and destination of coherence updates during an instruction. Normal cache line updates come from the memory.

- Check from which origin the data is fetched and deduce which processor changed it using signals on OCPi port:

  SRespData, SRespInfo, MReqInfo, SCohID, MCohID, SCohFwdID, MCohFwdID
  When broadcasting a coherence request in the snoop-bus concept one processor responds with latest data if it made updates to it. Observing the ID in MReqInfo and SRespInfo and who provided data with the response we can find the source of change. In directory concepts the directory knows who has the latest copy. By observing and decoding the intervention port of the directory slave we can see which processor was asked to provide the latest data.

- Count the cycles used in coherence transactions by using coherence signals on OCPce and OCPi :

  MCmd, MReqInfo, SRespData, SRespInfo, MCohCmd
  Counting means to implement a counter that is set and reset by appropriate events and that can be read by the debugger. The starting event for this statistics counting would be a specific instruction marked by the debugger and end would be a second instruction marked by the debugger. Those two triggers would come from the general debug block and the coherence activity counting will be based on observing the amount of commands on the OCPi intervention port of the coherence manager. Each command on the OCPi port means coherence activity.
  Each coherence command in the OCPce port must be accompanied by setting the MCohCmd signal high. Counting cycles when this signal is high would also give a clue on coherence traffic intensity.

The basic timing diagram is below. Detailed diagram is in chapter 12, Figure 12.13:

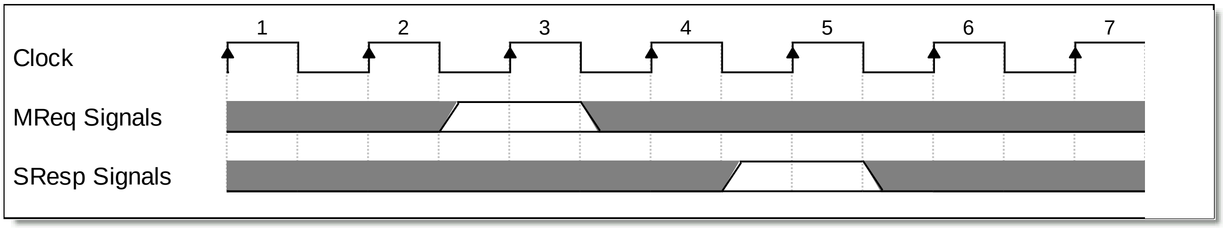
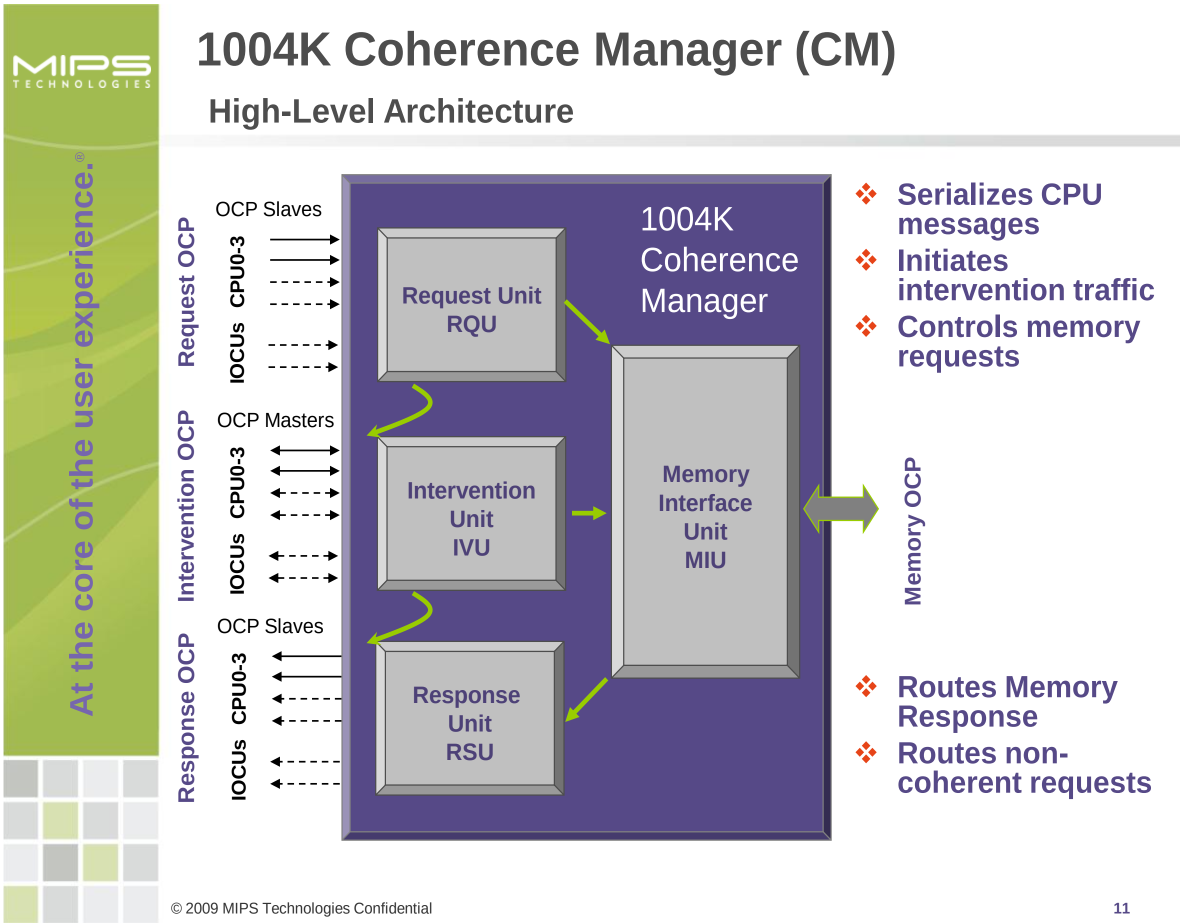


Figure 7.2 Master Slave signals in coherence debug interface

All master related signals are asserted together and are ready at the rising clock.
All slave related signals are asserted together at rising edge of the clock.
All debug related signals must read the information at the rising edge of the clock.

Example of a coherence manager, that is normally included in the memory management unit or in the bus interconnect unit, can be found in the MIPS1004K architecture. All signals for coherence debug can be found inside this block.



1004K Coherence Manager (CM)

High-Level Architecture

- Serializes CPU messages
- Initiates intervention traffic
- Controls memory requests
- Routes Memory Response
- Routes non-coherent requests

© 2009 MIPS Technologies Confidential

11

Figure 7.3 Example of a coherence manager that is always designed specifically for a system

The debugger can operate on two different types of systems:

1) Centralized coherent interconnect debug monitor (similar to what is shown in figure 7.3), where snoops are typically broadcast to all intervention ports and

2) Distributed coherent debug monitor. In this case, the snoops would be filtered by a snoop filter directory and only target a specific intervention port. In the distributed coherent debug monitor, the data at a single intervention port (related to reqinfo, and coherent state) are complete in describing the snoop transaction because the snoop is only targeting a specific port and not broadcast.

8. Extended (Optional) Debug Interfaces

Extended debug signals support application specific or optional functionality designed into the target system (Power Islands, Secure subsystems, others) or into the debug subsystem (Performance Monitoring, Time stamps, others). The extended signals support special functionality designed into the target system. In the OCP based case they follow the point-to-point communication protocol and expose the OCP signaling to the debugger.

RunControl allows a clock synchronized program execution of two cores that would run asynchronously in normal case. That makes it possible to time align the instruction streams to study interdependency. This is a simplified method to avoid the hardware for time stamping.

Security concepts require enabling debug of sensitive locations only during chip validation and disabling it in the production chips. OCP security signaling is here extended to the debug socket so debug IP blocks can implement such concepts.

Some deep submicron processes overheat easily so power management by switching off clock and even switching off power supply to certain IP blocks is very popular. Debugger shall not get locked or interrupted when dealing with such IP blocks if addressed accidentally in a debug session. Power-management debug signals help to avoid any confusion.

Performance monitoring enables observation of selected threads, initiators, and targets to identify data traffic and measure data bandwidth. In all cases, they follow the point-to-point communication protocol and expose the OCP signaling to the debugger via JTAG controlled or memory mapped registers.

8.1 Performance Monitoring

Performance monitoring enables observation of selected threads, initiators, and targets to identify data traffic and to measure data bandwidth.

Table 8.1 OCP Debug Performance Monitoring Interface Socket

Performance monitoring	***Description***	***Comment***
MConnID	Identifies the initiator. Routed to target	MThread-ID shall not be used to track cycles consumed by a specific initiator
MChannelID	Identifies the DMA channel initiator. Routed to target	MThread-ID shall not be used to track cycles consumed by a specific DMA channel initiator
MReqWatch[x]	Qualifies an OCP request	for which subsystem has detected a watchpoint hit
PMSampling	Periodic performance metrics sampling	Initiates a transfer of the performance metrics computed by the system interconnect to the trace export component. Periodic sampling strobe can potentially be generated within the OCP fabric.

OCP Performance Monitoring – General Configuration

- A built-in OCP debug component shall allow monitoring the OCP System bus bandwidth.
- An emulator shall be able to configure the “OCP Performance monitoring” component from the external [JTAG] interface through the “OCP Debug” bus.
- OCP Initiator's transactions to be monitored for a specific OCP target
- Monitoring window [Start & Stop triggers]
- System event latency

The “OCP Performance monitoring” component shall:

- Count within the [start, stop] window defined by triggers
- Effective cycles at the OCP target level
- Waiting cycles at OCP initiator level [latency; arbitration; shared link…..]
- Free cycles at the OCP target level
- Support continuous Performance monitoring [statistics]
- Export the computed performance statistics data through the “OCP Debug” to the “Trace Export” component
- Preserve the OCP System bus behavior
- Be DVFS aware (Dynamic Voltage and Frequency Scaling)

The “Trace Export” component shall:

- Implement an elastic buffer
- Optionally build trace packets for different (MIPI/Nexus) protocols
- Allows SW instrumentation interleaving

8.2 System Time-stamping

For distributed systems, a timestamp provides the means of temporally correlating different events that may be occurring in different systems or domains. There are many timestamp implementations – the simplest is a gated clock and reset that can be used to run timestamp counters at different blocks. In chapter 12 signal timing and levels are described in more detail.

Table 8.2 OCP Debug Timestamp Interface Socket

TimeStamp Interfaces	***Description***	***Comment***
ts_clk	Timestamp clock (gated version of clk) for global on-chip timestamping	global output
ts_reset	Timestamp reset	global output

8.2.1 Synchronous start of local time stamp counters

We rely on accurate distributed local time stamping to relax the trigger timing constraints. The synchronized start of all local time stamp counters is important for the correct display of debug events. Two simple concepts are proposed.

a. Make the counter reset arrive at the same time to all counters and the local clocks will increment them. If all the local clocks are time aligned and multiple of each other then this concept will work nicely. Else, this concept requires knowledge of the local clock frequencies and the skew between them at the time of reset-release to work properly. Alternatively, stamp counter reset can be de-asserted only at certain times when all clocks coincide with their rising edge.

b. Make the reset of stamp counters happen without tight temporal restrictions but one single clock goes to all counters at the same time (H-clock tree) and will be supplied only while tracing is active. If this one clock can be synchronous to or multiple of all local clocks then this concept is great. Otherwise, this concept requires fair amount of over clocking to resolve even smallest possible phase relationships between the asynchronous clocks.

A stamp clock and a stamp reset signal are both part of the basic OCP debug interface and designers must decide which one shall be base for the global debug synchrony. **In chapter 12 signal timing and levels are described in more detail.**

8.3 Power Management Monitoring

Some deep submicron processes overheat easily so power management by switching off clock and even switching off power supply to certain IP blocks is very popular. Debugger shall not get locked or interrupted when dealing with such IP blocks if addressed accidentally in a debug session. Power management debug signals help to avoid any confusion. In chapter 12 signal timing and levels are described in more detail.

Table 8.3 OCP Debug Power Monitoring Interface Socket

Power Management	Description	Comment
Sresp[2:0]	Additional error response codes signal a target not powered or not clocked	NULL, DVA, FAIL, ERR – new codes NOCLK, NOPWR
PWRDomainStatus	Indicates to target agent if Power Domain is active	These status signals contribute to error response generation
CLKDomainStatus	Indicates to target agent if Clock Domain is active	

General Requirements

The OCP platform Power Management module shall generate a trigger when:

- Switching off a domain
- Waking up a domain
- Switching frequency
- Switching operating voltage

The JTAG-OCP initiator shall support DMA transfers

- Separate thread

The CJTAG-OCP DMA engine shall move Power Management status to the Trace Export component through the OCP Debug interconnect.

- An emulator shall be able to configure the CJTAG-OCP component DMA engine
- PM status registers bank
- Trace export channel

The OCP Power Management monitoring shall:

- Support continuous Power Management monitoring
- Preserve the OCP System bus behavior
- Not require SW instrumentation

8.4 Security Debug Interface

Security concepts require enabling debug of sensitive locations only during chip validation and disabling it in the production chips. OCP security signaling is here extended to the debug socket so debug IP blocks can implement such concepts. In chapter 12 signal timing and levels are described in more detail.

Table 8.4 OCP Debug Security Interface Socket

Security	*Description*	*Comment*
MReqSecure	Qualifies an OCP request as secure transactions	The application security setup [HW & SW] may allow qualifying a debugger access as secure
DebugMode[1:0]	Debug operating mode	Debug can be disabled, restricted to public OCP transactions or allowed for both public & secure transactions.
TraceMode[1:0]	Trace operating mode	Trace can be disabled, restricted to public OCP transactions or allowed for both public & secure transactions.
TAPenable	Subsystem Test Access Port	enabled by application security software

9. Single Stepping and Virtual Single Stepping with Multi-core Chips

When we debug two cores at a time out of 5 heterogeneous cores with different clock speeds single-step SS needs a new definition.

This is not only a SS related problem, but also a problem of how to stop cores synchronously to events that are caused by a single core (e.g. breakpoint hit). The debugger reaction depends on the core interaction scheme, e.g. cores that are virtualized using SMP should be stopped synchronously by hardware within a few clock cycles. This is not a problem, since SMP cores are driven by the same clock domain. In isolated/loosely coupled multicore environments the core's stop-timing is usually less critical, thus achieving the required synchronization latency through separately issued TAP commands. Hardware synchronization would be also advantageous in case of higher latency requirements.

Restarting cores synchronously through the TAP is not difficult, since the TAP hardware has to be only slightly extended by a synchronization technique (e.g., run-test-idle pass can synchronize several cores in an ARM11 MP-Core).

Hardly any multicore architecture on the market implements SS by means of fetching and executing exactly one instruction on every single core. Neither the heterogeneous (e.g. RISC + DSP) nor the homogenous system (MultiCore PPC or ARM11MPCore) have cross-core SS hardware-implementation as of today.

→ One favorite solution for this problem is "Virtual Single Stepping".

1. System is halted. Debugger reads/has the full state of all cores and memories

2. Run ca. 100-1000 cycles and halt then synchronously.
Trace during this time frame IP and all data accesses. (No problem with MCDS, even with a small trace memory.)
3. With this information debugger can exactly reconstruct all states and data values between start and end point

This means the user can then virtually single step forth and back (!) in this time window.
The timing relationship between the cores is maintained as well. There is possibly only a slight impact at the start and the end of the period.

A similar solution is using an on-chip test platform with IEEE 1500 (developed by National Cheng Kung University, Taiwan). For each core there is a register to trigger the halt then use the 1500 scan chain to dump the core and memory. With the dump an ESL debugger can trace the bug. When thinking about how to halt all cores simultaneously, although we like to leverage test gates as much as possible, it seems additional debug gates are needed.

10. EDA and SW Tools Support

10.1 ESL Design and Design Checkers Support

The OCP Debug Interface, after selection of the debug signal groups, generates an XML description text file that allows for easy modeling in the ESL abstracted chip design. This XML file is the basic information for the automatic debug interface checkers.
The association of an XML interface file with a specific chip design is proposed to be an ID in the JTAG interface readable by software level debuggers and EDA hardware level debuggers that can access the XML file from a library maintained on the web. The ID must be at least 128 bits long or have a self-extensible format and size to be accepted by the world-wide chip industry.

10.2 Programmers Model

We do not intend to include a programmer’s model for debug interactions into the OCP standard. That is left to the vendors of the debug hardware blocks to satisfy the software DAS-API interface requirements in providing and covering the minimum set of control sequences down to the cores and IP-blocks. Just when connecting their debug hardware to the OCP system interconnect the OCP Debug Interface signals must be used.

By defining the two interfaces, on the electrical side the OCP Debug Signal Interface and on the software side the DAS-API Interface, we sufficiently pre-define and unify all the “Protocol Solutions” between them. To also prescribe one superset Debug Protocol is possible but unnecessarily restrictive.

Instead we propose optionally in front of the OCP Debug Signal Interface also an OCP Debug Register Interface with predefined locations of bits for predefined debug operations and debug data. Many existing debug solutions can be adapted to this register interface and natively matching debug solutions can emerge in the future.

11. Processor Classes and their Match to Debug IP Features

In order to estimate the work for adaptation of one debug IP block to a different processor we propose a similarity classification of processors. In fact there are several possible dimensions to classify and here are the most prominent features scales. If two processors are close or similar on these scales then adaptation of an existing debug IP block from one processor to the other will be simpler.

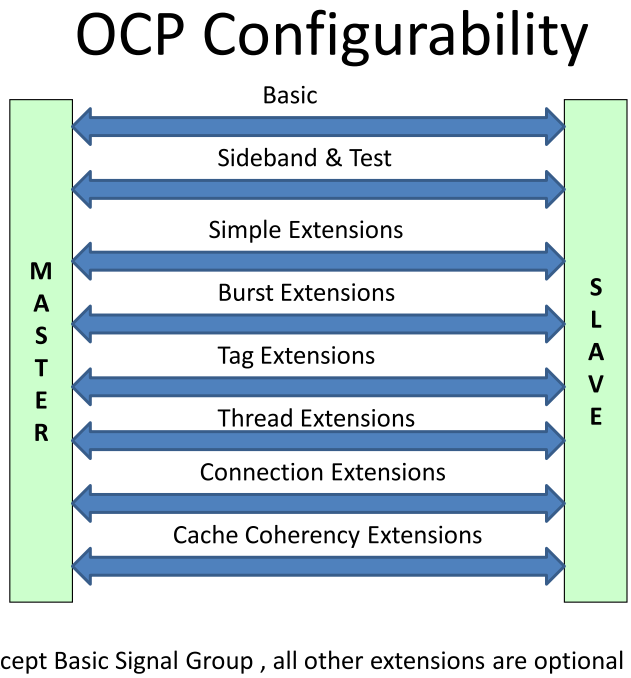


Figure 11.1 OCP Interface Standard with all sockets

Based on native OCP Interface Standard with growing complexity we have a first scale of similarity between processors. Using same **OCP extensions** in the system ports like in Figure 11.1 means similar behavior in debugging.

- JTAG Socket
- Reset Socket
- Control Socket (transactions, data)
- Run Control Socket
- Triggering Socket
- Trace Socket
- Time Stamp Socket
- Performance Monitoring Socket
- Power Monitoring Socket
- Security Socket
- Low Power Monitoring Socket
- Cache Coherence Monitoring Socket

The OCP Debug Standard defines classes of signals called **Sockets** listed above with increasing debug complexity when dealing with different processors. Comparing Sockets is a second scale of similarity. If two debug concepts need to use same sockets with different processors then they will have similar behavior in debugging.

Another scale of processor similarity is data transfer performance of the debug connection or **debug bandwidth**. If a trace buffer for debug data is on chip then this scale of debug bandwidth has less significance. To form processor classes based on real-time debug bandwidth we compare interface connections provided for a debug session:

Serial port	about 1 MByte/s
JTAG	1-20MHz clock, serial 1 bit connection
Single Wire	30kbit/s – 1Mbit/s
USB 1.0, 2.0, 3.0	12Mbit/s, 480Mbit/s, 5 Gbit/s
Ethernet 10/100/1G	10Mbit/s; 100Mbit/s; 1Gbit/s
PCIe 1.0, 2.0, 3.0	250MByte/s; 500MByte/s; 1GByte/s

Different debug-IP blocks can be compared for a match to a processor based on these three scales. Any mismatch in features means extra work in porting of a debug-IP block.

12. Definition of Debug Interface Signals

In general the Debug Interface signals have positive logic. Except in cases when short after reset the signals must remain in the same low state to assure smooth operation; then they are defined in negative logic. The following diagrams demonstrate this behavior. A basic OCP read and write cycle is presented first.

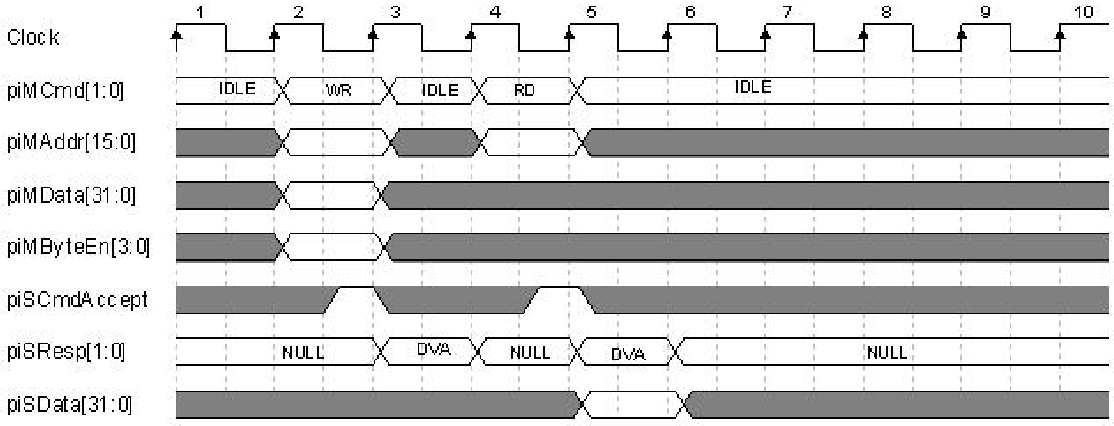
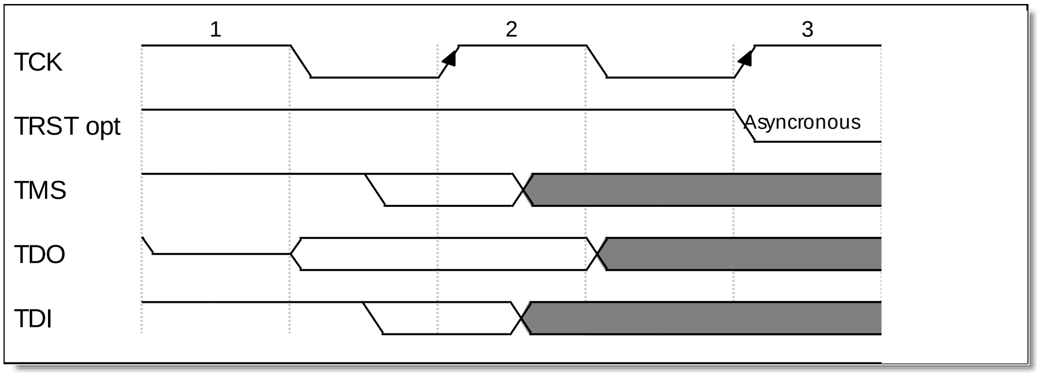


Figure 12.1 Example of Basic OCP interface signaling for read/write

Signals in basic OCP Debug Sockets:

Figure 12.2 JTAG Socket timing diagram



In a JTAG socket compliant to IEEE1149.1-2001, **TDI** and **TMS** are sampled on the rising edge of **TCK**, and **TDO** changes on the falling edge of **TCK**. To take advantage of these properties, master samples **TDO** on the rising edge of **TCK** and changes its **TDI** and **TMS** signals on the falling edge of **TCK**. Thus with a fully compliant JTAG socket, issues with minimum setup and hold times can always be resolved by decreasing the **TCK** frequency, because this increases the separation between signals changing and being sampled. **TRST** is optional JTAG asynchronous reset active low.

Figure 12.3 Debug Reset, Clock, Enable Socket

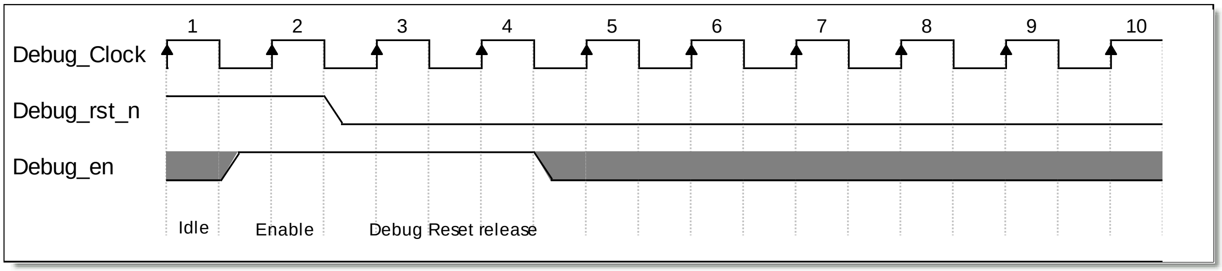


Figure 12.4 Control Socket (transactions, data)

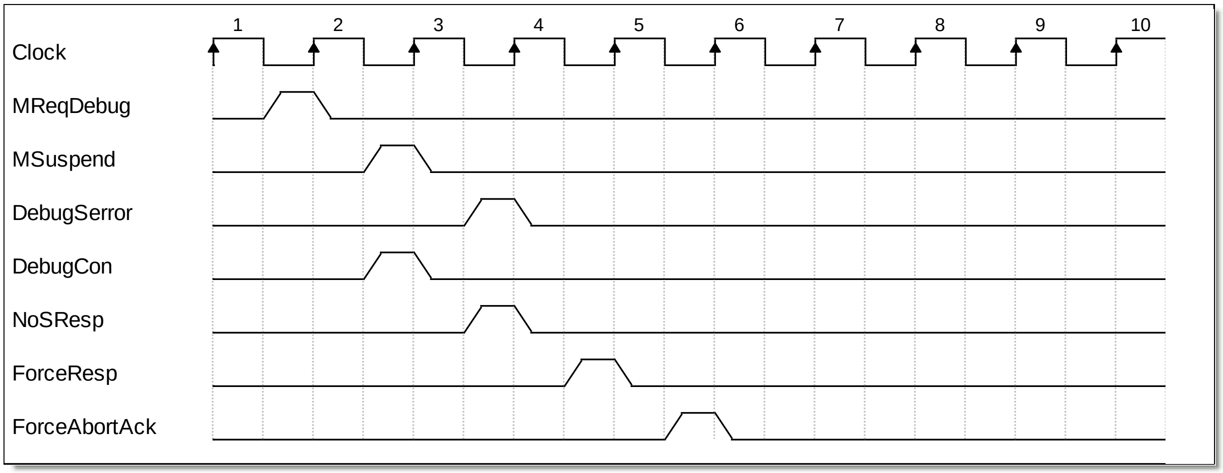


Figure 12.5 Run Control Socket

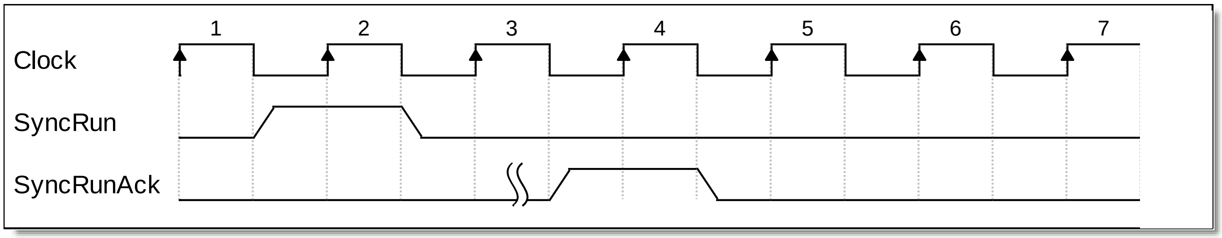


Figure 12.6 Triggering Socket

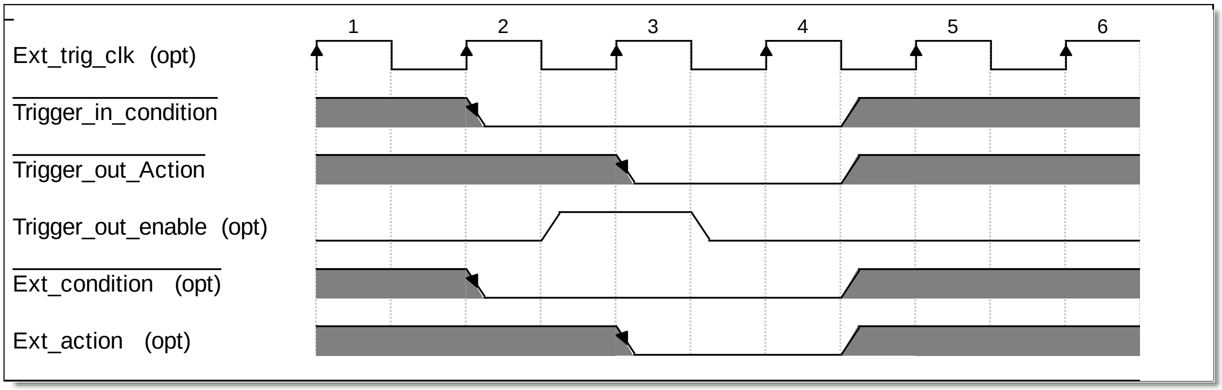
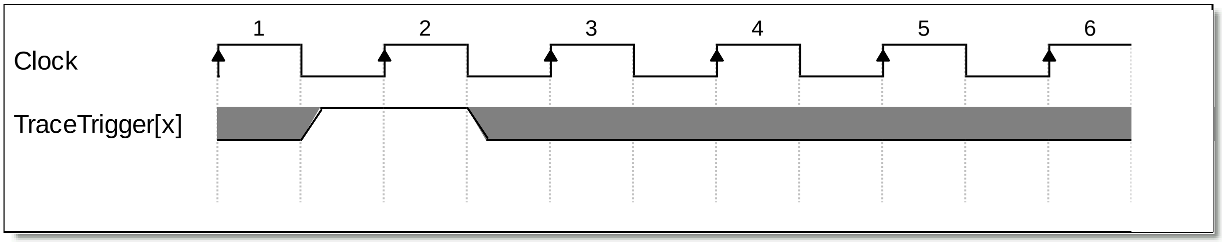


Figure 12.7 Trace Socket



Extended Sockets:

Figure 12.8 Time Stamp Socket

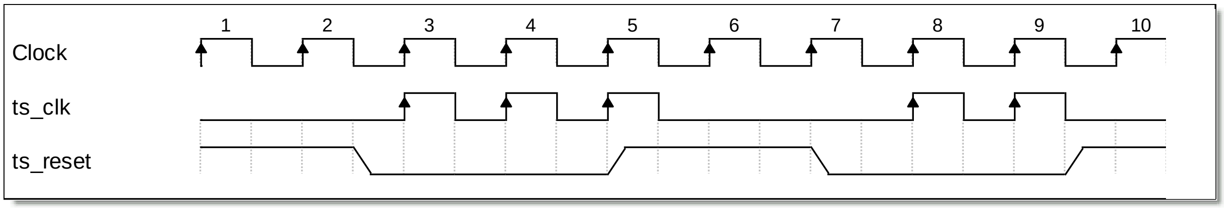


Figure 12.9 Performance Monitoring Socket

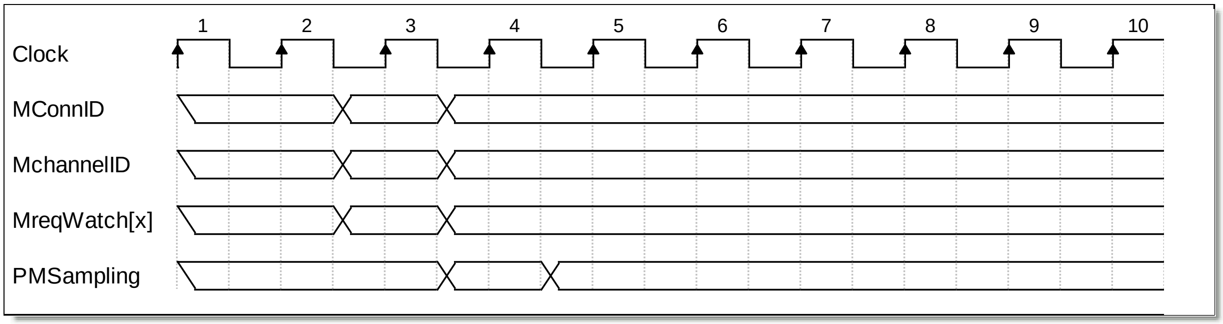


Figure 12.10 Power Monitoring Socket

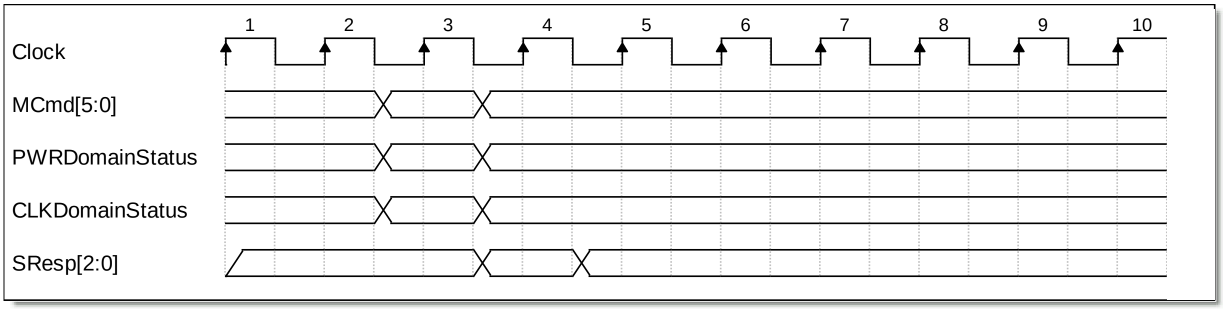
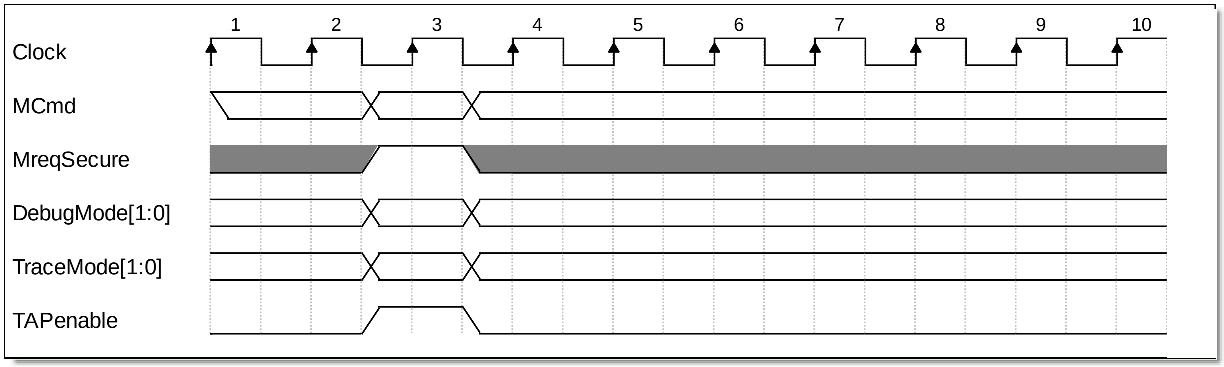


Figure 12.11 Security Socket



New Sockets:

Figure 12.12 Low Power Monitoring Socket

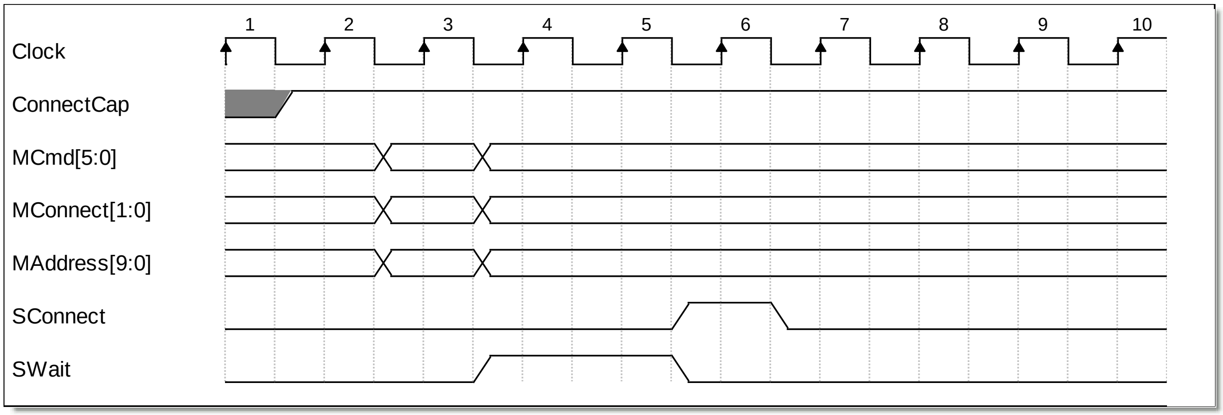
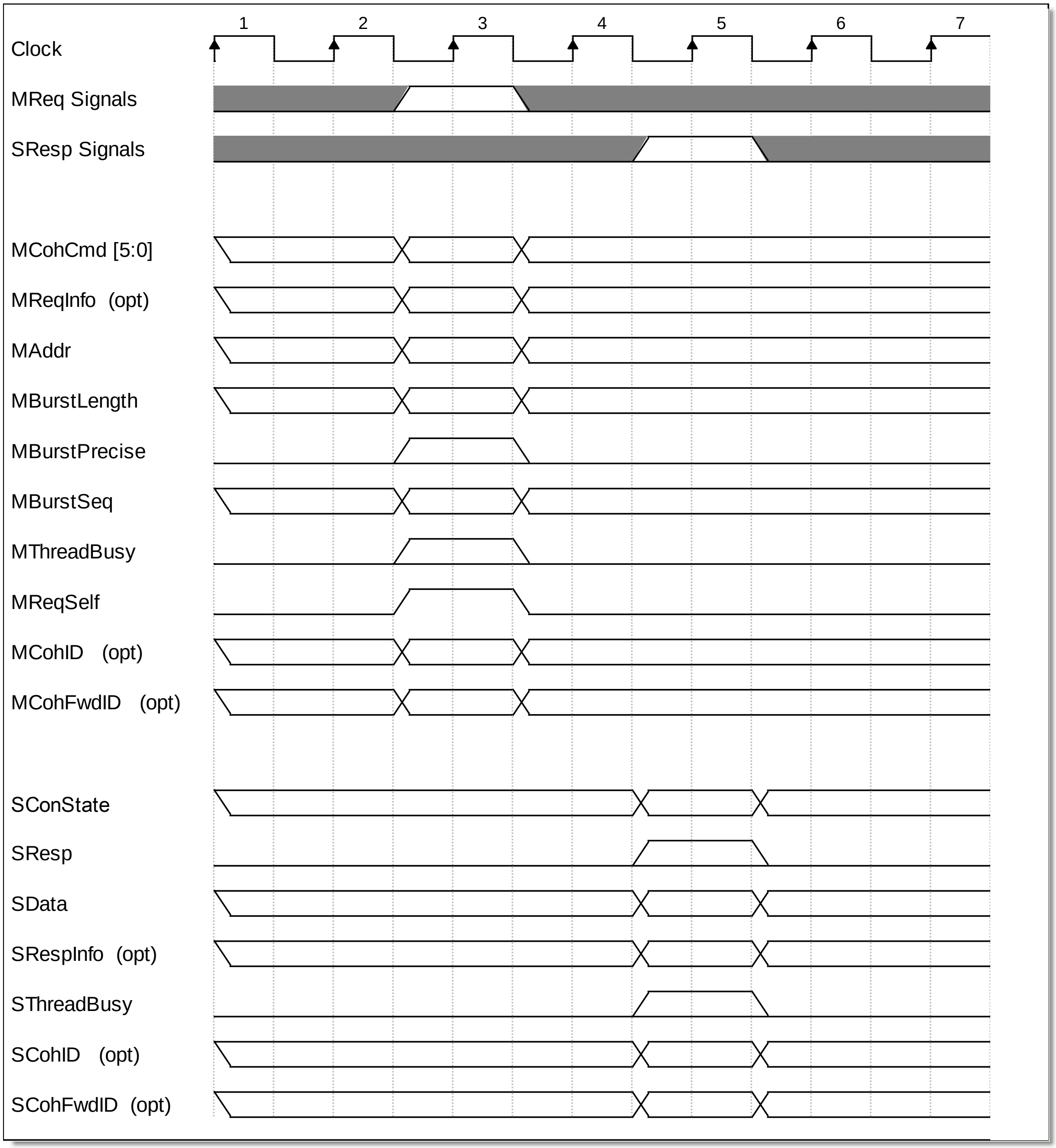


Figure 12.13 Cache Coherence Monitoring Socket



Appendix Implementation Examples

A. MCDS Example of OCP Debug Interface Implementation and Protocol

A.1 Trace Ports and Protocol, Complete Example with MCDS

By showing how OCP Debug Signals match and translate to a proven Multi-Core Debug Solution MCDS we satisfy the proof that the OCP Debug Socket is complete and functional.
(Other debug IP examples like CoreSight or Nexus could be adapted as well.) We choose MCDS because it comes close to the proposed OCP Debug HW implementation.
A matching framework for multiple debuggers running over one JTAG is the work of SPRINT or the framework of Eclipse with GDB. Both run a debug server that operates the interface to the chip JTAG/TRACE over USB or Ethernet link transparently for all software debuggers.

A.1.1 MCDS Trace Interfaces

Within this section the generic trace interface protocol of the Multi Core Debug Solution (MCDS) is described. The text is split into two distinct chapters:

- Protocol Definition: The concept of the protocol is described in an abstract way.
- Examples: Some instances of the protocol are shown to illustrate the concept.

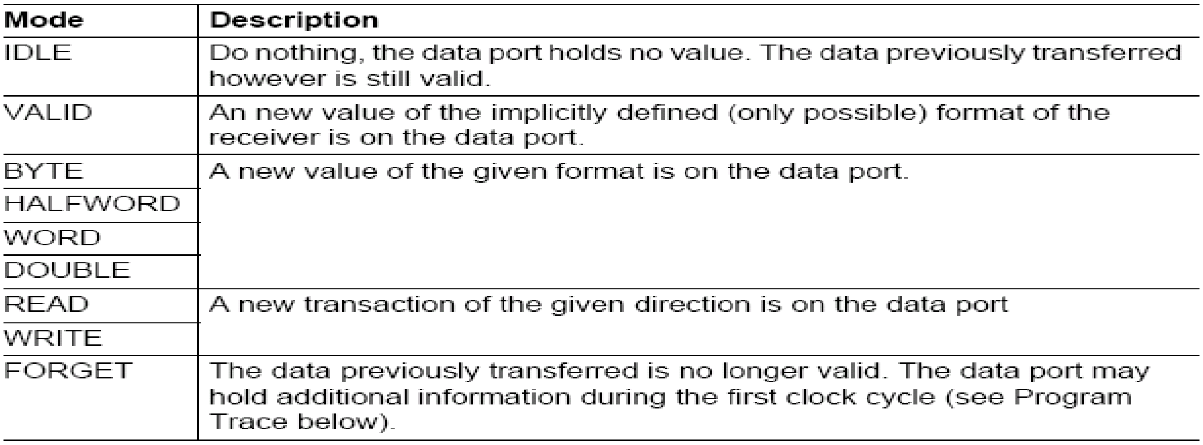
A.1.1.1 MCDS Trace Protocol Definition

The basic interface is a **synchronous tagged data protocol without handshake**. The sender places the data in well defined packets on the data port and indicates concurrently on the mode port which kind of packet is present.

Note: Merging the mode into the data packet is avoided to simplify the implementation.

The mode port must be able to express at least two different values: **IDLE** and **VALID**.
If different kinds of data are supported, VALID is replaced by as many different other values on the mode port.
Finally a standard way to invalidate the last packet is needed. This is done by the **FORGET** code. The MODE port is similar to the SResponse tag [3:0] in OCP and is realized in the OCP sideband interconnect. MODE port can be used to propagate SResponse to the debugger if it is extended from 3 to 5 bits. Usually the OCP bus observer block is between this trace interface and the system bus..

Table 1-1 Generic Interface Mode Encoding



Mode	Description
IDLE	Do nothing, the data port holds no value. The data previously transferred however is still valid.
VALID	An new value of the implicitly defined (only possible) format of the receiver is on the data port.
BYTE	A new value of the given format is on the data port.
HALFWORD	
WORD	
DOUBLE	
READ	A new transaction of the given direction is on the data port
WRITE	
FORGET	The data previously transferred is no longer valid. The data port may hold additional information during the first clock cycle (see Program Trace below).

Some fine points about the protocol implementation:

- Startup should assure FORGET on all mode outputs.
- If a time-out mechanism is provided, a synthetic FORGET is used.
- Protocol errors are also forwarded downstream as FORGET. As no back channel is provided, the offending message is dropped by the receiver.
- There is no way to slow down the sender; if the receiver is unable to accept a new message the protocol nevertheless has to be executed. The overflow processing is always done by the receiver.

A.1.1.2 Examples

A few typical applications of the protocol defined above are given here.

A.1.1.2.1 Program Trace

Great care was taken to cover all kinds of processor cores. The smallest common denominator is the interface described here, comprising of two ports:

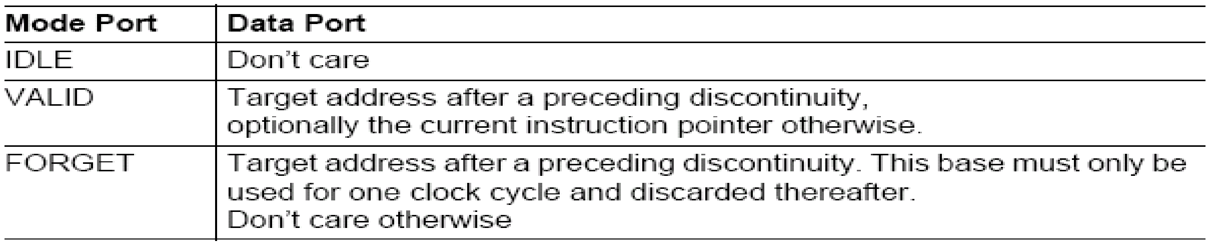
- **Base Address**: After power on and after each discontinuity of the program flow the trace logic needs to know the exact and complete value of the instruction pointer.
- **Instruction Pointer Increment**: Once the base is known, only the increments are required to keep the local copy in sync with the original instruction pointer. This split has the additional advantage of allowing arbitrary clock ratios between core and trace logic, as long as not more than one base address needs to be transferred per trace clock cycle: Increments can be pre-accumulated in the core at will without loss of information.

Discontinuity comes in two flavors:

- **Direct Branches** are caused by jump instructions in the executed program. The target address is a constant (label) in the source code and can be obtained from there by the decoder software. A branch of this kind is indicated by FORGET on the increment port.

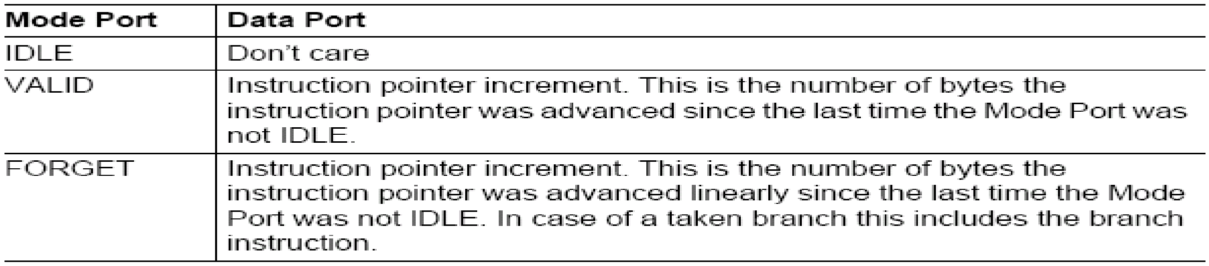
- **Indirect Branches** are caused by jump instructions with calculated target address (e.g. Return from subroutine 1)) or by exceptions (e.g. interrupts, traps). In these cases the target address must be contained in the trace memory. FORGET on the base port is used to indicate such a branch. Each FORGET received on any of the two ports invalidates the current base address. The exact protocol definition is given in the tables below.

Table 1-2 Base Address Protocol



Mode Port	Data Port
IDLE	Don't care
VALID	Target address after a preceding discontinuity, optionally the current instruction pointer otherwise.
FORGET	Target address after a preceding discontinuity. This base must only be used for one clock cycle and discarded thereafter. Don't care otherwise

Table 1-3 Instruction Pointer Increment Protocol



Mode Port	Data Port
IDLE	Don't care
VALID	Instruction pointer increment. This is the number of bytes the instruction pointer was advanced since the last time the Mode Port was not IDLE.
FORGET	Instruction pointer increment. This is the number of bytes the instruction pointer was advanced linearly since the last time the Mode Port was not IDLE. In case of a taken branch this includes the branch instruction.

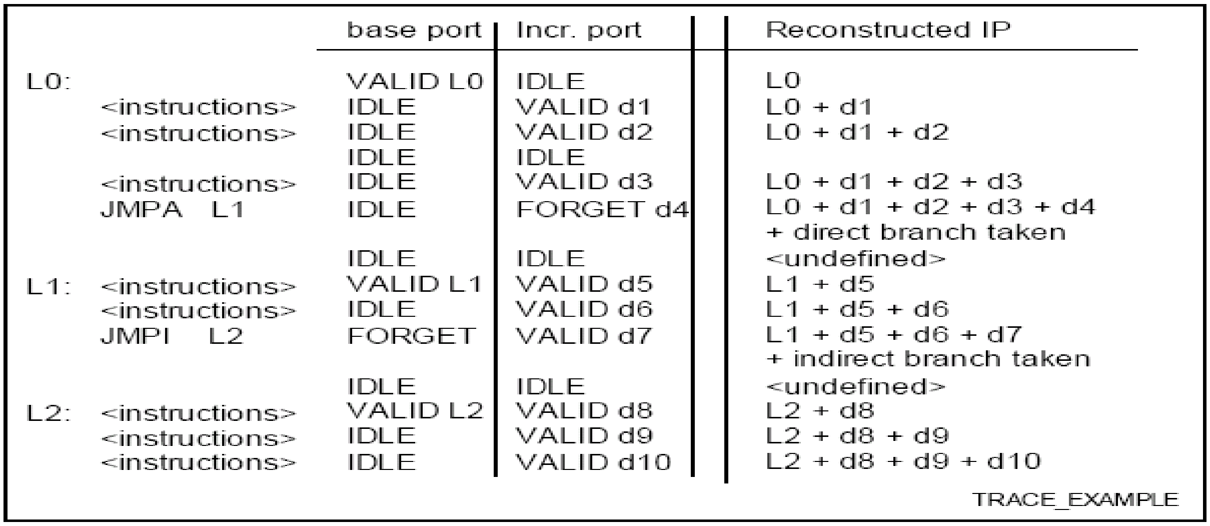
There is no need to serve both ports concurrently by the sender. The only requirement is that the base is sent prior to (or latest concurrently with) the next discontinuity. If this is not possible the sender may set both mode ports to FORGET. This is interpreted as .overrun. .

Note: This implies that the increments are accumulated starting from 0 after each discontinuity. A valid instruction pointer however, e.g. for comparison, is only available when the associated base was received.

A short examples may help to understand the deceptively simple interface protocol.

---

1) It is not recommended to treat "Return" instructions as direct branches: If the trace is turned on after the subroutine was called there is no way for the decoder to know the return address.



		base port	Incr. port	Reconstructed IP
L0:		VALID L0	IDLE	L0
	<instructions>	IDLE	VALID d1	L0 + d1
	<instructions>	IDLE	VALID d2	L0 + d1 + d2
		IDLE	IDLE	
	<instructions>	IDLE	VALID d3	L0 + d1 + d2 + d3
	JMPA L1	IDLE	FORGET d4	L0 + d1 + d2 + d3 + d4 + direct branch taken
		IDLE	IDLE	<undefined>
L1:	<instructions>	VALID L1	VALID d5	L1 + d5
	<instructions>	IDLE	VALID d6	L1 + d5 + d6
	JMPI L2	FORGET	VALID d7	L1 + d5 + d6 + d7 + indirect branch taken
		IDLE	IDLE	<undefined>
L2:	<instructions>	VALID L2	VALID d8	L2 + d8
	<instructions>	IDLE	VALID d9	L2 + d8 + d9
	<instructions>	IDLE	VALID d10	L2 + d8 + d9 + d10

TRACE_EXAMPLE

Figure 1-1 PTU Trace Example: Minimized Messages

A few points to consider:

- After each branch the instruction pointer is .unknown. to the trace logic until a new base address is received. The decoder software however may already know the address (e.g. L1 above) from the source code.
- For multi-scalar processors the last increment leading to a taken branch (e.g. d4 above) may include more than the branch itself. It is therefore not guaranteed that the branch instruction is stored at address L0+d1+d2+d3.
- In case of some exceptions (e.g. illegal target address) the target address must be analyzed to distinguish the exception from a taken branch. That’s why it is important to treat exceptions and interrupts as indirect branches.

A.1.1.2.2 Data Trace

Let the task be to trace transactions on an arbitrary bus system, consisting of address, data and control information, e.g.

- The effective address (byte granularity)
- The current data (size depending on transaction)
- Auxiliary information (.bus mode.) like mastership, privileges etc.

The mode of the third item is used to signal completeness: Whenever asserted READ or WRITE all information concurrently valid is considered belonging to the same transaction.
The protocol on the three input ports should be obvious from the tables below. FORGET is used to invalidate single items. If for example a read access fails the data port is invalidated and thus no data trace can be recorded - the offending address however is fully available.

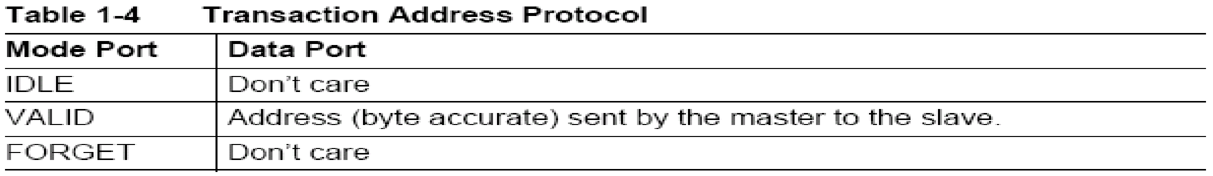


Table 1-4 Transaction Address Protocol

Mode Port	Data Port
IDLE	Don't care
VALID	Address (byte accurate) sent by the master to the slave.
FORGET	Don't care

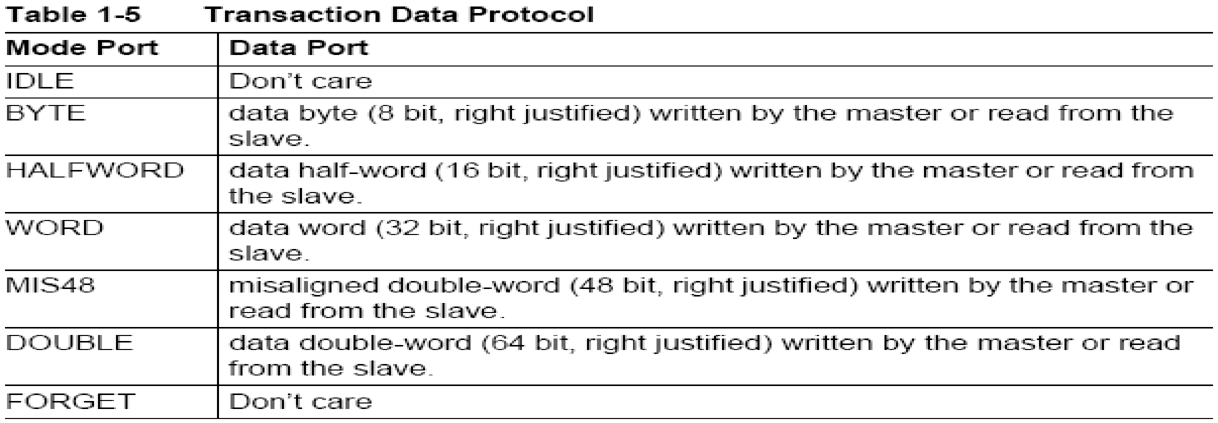


Table 1-5 Transaction Data Protocol

Mode Port	Data Port
IDLE	Don't care
BYTE	data byte (8 bit, right justified) written by the master or read from the slave.
HALFWORD	data half-word (16 bit, right justified) written by the master or read from the slave.
WORD	data word (32 bit, right justified) written by the master or read from the slave.
MIS48	misaligned double-word (48 bit, right justified) written by the master or read from the slave.
DOUBLE	data double-word (64 bit, right justified) written by the master or read from the slave.
FORGET	Don't care

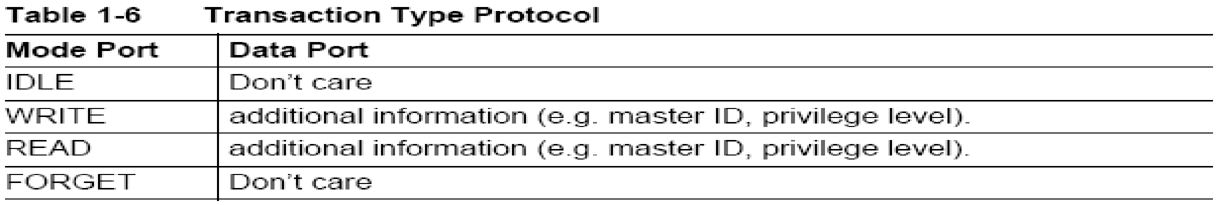


Table 1-6 Transaction Type Protocol

Mode Port	Data Port
IDLE	Don't care
WRITE	additional information (e.g. master ID, privilege level).
READ	additional information (e.g. master ID, privilege level).
FORGET	Don't care

Note that the sender has to take care that the mode port of Transaction Type is READ or WRITE for exactly one clock cycle for each transaction.

A.1.1.2.3 Ownership Trace

Ownership is a NEXUS term; it translates to .Task ID. or .Process ID. for most practical purposes. The generic OTU is able to process the ownership information of an arbitrary processor core if implemented in hardware.
As the rate of change for the process ID is rather low, it will be sent multiplexed over other trace interface signal lines of the core quite often. This is perfectly legal, provided a dedicated signal to drive the mode port is available. If the core is not doing any useful work (e.g. if no task is active), the process ID should be invalidated by FORGET.

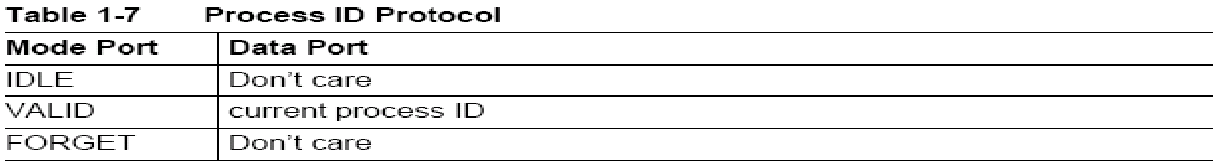


Table 1-7 Process ID Protocol

Mode Port	Data Port
IDLE	Don't care
VALID	current process ID
FORGET	Don't care

A.2 The 12 MCDS Trace Ports and corresponding OCP Debug Socket Signals

Instruction Trace (data port / mode port)
0 **base** address [31:0] / **base_mode** [2:0] (Instruction Base Address)
1 address **inc**rement [3:0] / **inc_mode** [2:0] (Instruction Address Increment)
(Data is usually [3:0], expecting no more than 16 bytes of code executed per emulation clock cycle.)

Data Trace (data port / mode port)
On a processor you usually have a write-only port at the end of the pipeline, and sometimes other ports (read only) on earlier stages.
2 **addr**ess [31:0] / **addr_mode** [2:0] (Transaction Address)
3 **data** [31:0] / **data_mode** [2:0] (Transaction Data)
4 **control** [31:0] / **control_mode** [2:0] (Transaction Type)
(This part of the trace interface is intended for multi-master bus traces.)
5 ownership **id** [7:0] / **id_mode** [2:0] (Process ID)
(Processor specific thread number OR task ID OR interrupt priority OR ..., mostly [7:0] or less.)
Example 1. Standardized MCDS debug port MODE definitions for 32 bit cores.

In bold is naming for the 12 ports of the 6 debug interfaces. The width of the ports 0,2,3 is 32 bits for a 32 bit RISC processor. Other cores might require 16 or 64 bits. Tracing is a one-way road from the cores to on-chip trace memory and to the debugger. Selecting just 2 cores out of N for tracing reduces trace information. The control data port 4 is used for multiple cores.

There is no acknowledge and no OCP-protocol-like handshake for trace. The observed core or bus INTENTIONALLY shall not know anything about being traced. That's the main paradigm of "non-intrusive" tracing.
You can implement a feedback path sometimes via the run control: The "buffer memory full" signal can be used to request a breakpoint or suspension in certain applications, but this is a safety hole (you can't "suspend" a turning crankshaft!) and needs great care in implementation.

Using 2 or 3 MODE bits allows for 4 to 8 TAGS in the MODE port that are partially listed and standardized/fixed in Table 1-1 to Table 1-7. The best standard offers a fixed basic framework but remains expandable by users.

Historically there was hard pressure to reduce the physical number of wires between the cores and the trace subsystem. So we settled for a dedicated enumeration type for each mode. This gives the **hardware compiler** (simultaneous synthesis of sender and receiver RTL) all the freedom to encode in the most efficient way (variable 1..3 bit wide **mode** ports, see example 2).

Of course this is not a good idea if we want to split the design of SoC and debug IP system at this point. In a standard we propose to have 3 wires for each mode port as in Example 1. Users can fill in the blank lines in Tables 1-1 to 1-7 that will have 8 possible cases each.

Example 2. Wire saving MCDS debug port MODE definitions:

```
 ---------------------------------------------------------------------------
 -- Type Declarations  for MODE ports
 ---------------------------------------------------------------------------
 -- Status ModeType is enum
 type tu_mode_2_t is
   (idle, valid);               -- 2**1

 -- Address Pointer Mode Type is enum
 type tu_mode_3_t is
   (idle, valid, forget, invalid); -- 2**2

 -- Transaction Type Control Mode Type is enum
 type tu_mode_4_t is
   (idle, write, read, forget);    -- 2**2

 -- Data Bus Mode Type is enum
 type tu_mode_6_t is
   (idle, byte, h_word, word, d_word, forget, none, invalid); -- 2**3
```

We need to change the meaning of acknowledge/handshake with debug signals. In debugging or tracing ACK shall mean that the connection between external debugger and chip is working. ACK tells back to the debugger that the debugger command arrived in the chip well. Also trace data shall send a parity bit in MODE after every word/burst to detect a bad/unplugged cable immediately. We will avoid any interaction with the core processor. Just the debug subsystem will do ACK management with the external debugger. The new MIPI/AJTAG has a master feedback signal back to the debugger that can serve as ACK or trigger out.

The MCDS trace port matches the OCP Trace Debug socket that exposes bus data of two cores to the debug IP block. How does the rest of OCP debug interface signals map into MCDS ports? More correspondence between the debug ports and OCP debug signals is in the next section.

Comments

This leaves compression (e.g. Nexus) to a separate module. We'll note however, that more information can help compression. You might want to consider more (optional) inc_mode variants for this, and add them to the standard. For example, in applications with a lot of calls and returns, a large portion of compressed trace is taken up by return PCs. These can often be compressed quite well if the compressor knows which branches are calls and which are returns. For calls/exceptions/interrupts, it seems we can usually expect that the PC past the branching instruction is the return PC, except perhaps with certain superscalar processors.

E.g., more optional base_modes that report the type of the trace event on the mode port. (E.g. function call, return, exception, interrupt, etc.)

It might be useful to provide information for a zero-overhead loop, i.e. where zero cycles are used to branch back to the top of the loop. (That's effectively implemented for zero-overhead loops with Nexus). Another example how to extend the base_mode port information.

One processor architecture has a literal load instruction that is typically never used to load anything other than literals, whose values are part of the program (constant) and thus need not be traced. Some information on the trace port inc_mode identifying such constant data is useful to filter it out, unless you design such filtering to occur before the trace port.

Some processors have 128 bit wide data paths. We use additional data_mode encoding for that.

Let’s stress again that we are interested in showing how the debug interfaces match between the debug block and the system bus/interconnect and not how the debug block internals have been designed in the MCDS implementation example. The match must be conceptual first and eventually going down to individual signals.

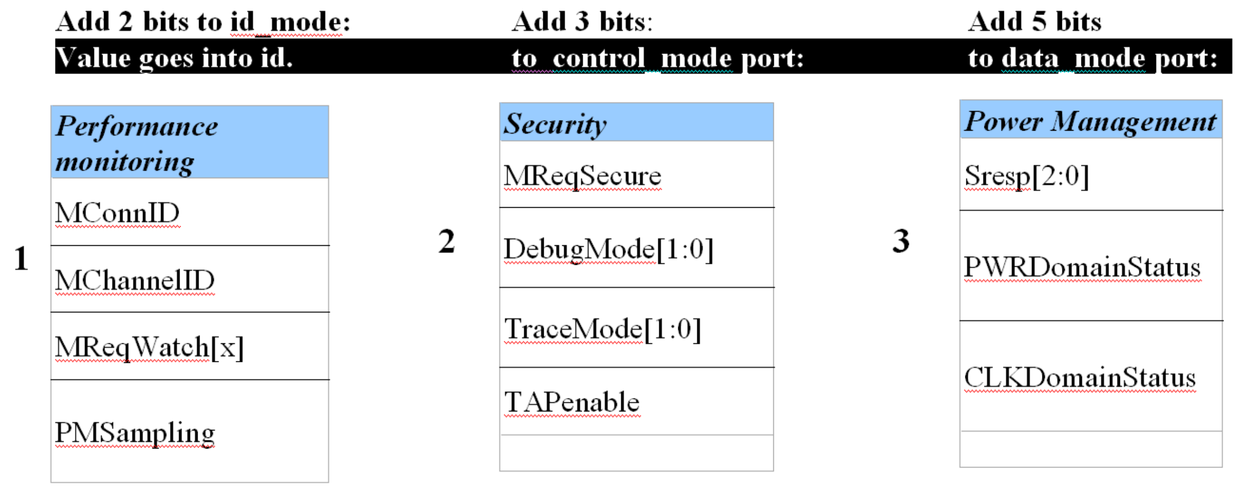
Debug wiring between any core and the debug-IP block shall be created as part of the OCP interconnect to maintain clean socket interfaces.

A.2.1 OCP Debug Signal Classes and their Mapping on the MCDS Interface

Instruction Trace (data port / mode port)
0 **base** address [31:0] / **base_mode** [2:0] = (idle, valid, forget, invalid);
1 address **inc**rement [3:0] / **inc_mode** [2:0] = (idle, valid, forget, invalid);

Data Trace (data port / mode port)
2 **addr**ess [31:0] / **addr_mode** [2:0] = (idle, valid, forget, invalid);
3 **data** [31:0] / **data_mode** [2:0]=(idle, byte, h_word, word, d_word, forget, none, invalid);
4 **control** [31:0] / **control_mode** [2:0] = (idle, write, read, forget);
5 ownership **id** [7:0] / **id_mode** [2:0] = (idle, valid, forget, invalid);

Status information and qualification of data trace goes into mode ports. Still it can be used by the run control CERBERUS transactor for orientation:
(The bold numbers are associated with the partial sockets to identify them in figures later.)



Add 2 bits to id_mode: **Value goes into id.**

1

Performance monitoring
MConnID
MChannelID
MReqWatch[x]
PMSampling

Add 3 bits: **to control_mode port:**

2

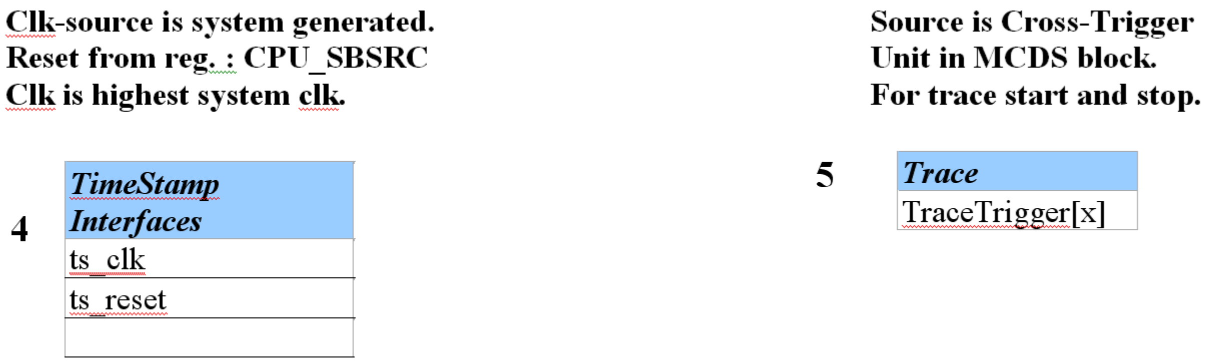
Security
MReqSecure
DebugMode[1:0]
TraceMode[1:0]
TAPenable

Add 5 bits **to data_mode port:**

3

Power Management
Sresp[2:0]
PWRDomainStatus
CLKDomainStatus

Debug Run Control in Cerberus Debug Bus Transactor and Cross-Trigger Unit in MCDS :



Clk-source is system generated.
Reset from reg. : CPU_SBSRC
Clk is highest system clk.

4

TimeStamp Interfaces
ts_clk
ts_reset

Source is Cross-Trigger
Unit in MCDS block.
For trace start and stop.

5

Trace
TraceTrigger[x]

6

Signal name
Minimum OCP debug signals set
MReqDebug
Msuspend
DebugSerror
DebugCon
NoSResp
ForceResp
ForceAbort
ForceAbortAck

These signals belong to the RUN-control that is implemented in the next chapter. They are found in the status register CBS_OSTATE in Cerberus/JTAG block.

Goes to Cross-Trigger MCDS:
n is number of debugged blocks.
Ext. cond/action is just one.

7

Cross Trigger Interfaces
Trigger_in_ condition[n:0]
Trigger_out_ Action[n:0]
Trigger_out_enable[n:0]
Ext_trig _clk
Ext _condition[n:0]
Ext _action[n:0]

Source is Cerberus register: CBS_SRC
ACK is in CPU DBGSR.

8

Synchronous
SyncRun
SyncRunAck

Many signals are flowing back to the debugger as feedback information or qualify/tag the data/instructions to explain the momentary context/status of the chip at the clock cycle moment of tracing.

Debug bus transactor “Cerberus” accesses all memory mapped register data.

Synchronous start of all cores is accomplished by pre-setting them before the start.

A.2.2 JTAG Pins

The 5 pin JTAG connects to the OCP TAP controller for control and status bypassing the bus, still all JTAG controlled registers are also memory mapped.

10

JTAG Interfaces
Tck, Trst (optional)
tms
tdi
tdo

9

Debug Interfaces
Debug_reset_n
Debug_en

Connects to JTAG reset.
Debug Enable connects to Cerberus OSCU register CBS_OEC.

A.3 Debug Transactor – RUN Control Bus Master

A basic bus transactor implementation would support only simple read and posted write data operations and may require stalling between operations to ensure synchronization of signaling operations (i.e. MData must occur concurrent with MDatavalid, SDataAccept occurs concurrent with Sdata). More advanced OCP operations such as bursting may require additional (dedicated) logic, to support full speed bursting. Signal timing is shown in the OCP Standard timing diagrams.

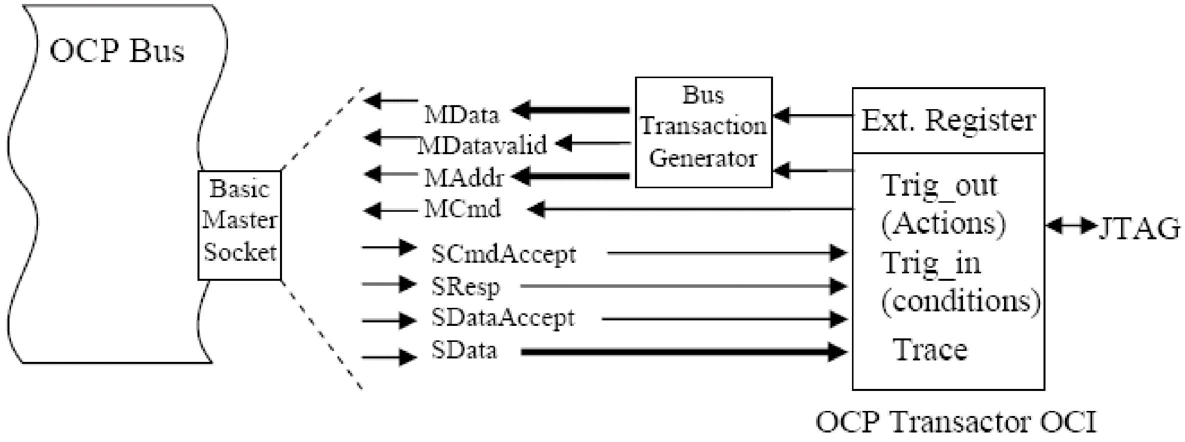


Figure A.3: An OCP Transactor for Bus Master Opertion

A transactor bus master operation can be initiated either from an external register load or from trigger output signals acting on specific bus monitoring operations. Address and data for individual bus transactions can also be written from the externally controlled registers, although this may be a slower manual process or require multiple cycles. Alternately, writing of regular (i.e. incrementing or other simple pattern) address and data can be controlled by counters or by logic enabled by trigger signals. Data and other OCP signals or performance data may also be traced (i.e. sequentially or periodically) with the bus master operations optionally stalled during JTAG data download phase, avoid loss of continuity. Additional trigger or state signals may be used for defining and controlling basic memory maps or domains.

A.4 OCDS/MCDS Complete Example of Run Control: On-Chip Debug Support

This chapter gives an overview of the debug features of the OCDS and MCDS. For detailed information about the On-Chip Debug Support (OCDS) and Multi Core Debug Solution (MCDS) please contact local Infineon representatives. This solution can be purchased as IP and proves that the signaling concept of the OCP Debug Socket is sufficient for trace and run control. In OCDS/MCDS we can find three levels of debug operation:

OCDS Level 1

The OCDS Level 1 is mainly assigned for real-time software debugging purposes which have a demand for low-cost standard debugger hardware.

The OCDS Level 1 is based on a JTAG interface that is used by the external debug hardware to communicate with the system. The on-chip Cerberus bus master module controls the interactions between the JTAG interface and the on-chip modules. The external debug hardware may become master of the internal buses, and read or write the on-chip register/memory resources. The Cerberus also makes it possible to define breakpoint and trigger conditions as well as to control user program execution (run/stop, break, single-step).

OCDS Level 2

The OCDS Level 2 will be gradually phased out and replaced by MCDS Level 3. Level 2 makes it possible to implement program tracing capabilities for enhanced debuggers by extending the OCDS Level 1 debug functionality with an additional 16-bit wide trace output port with trace clock. With the trace extension, trace capabilities are provided for several cores and IP-blocks with just one trace active at a time.

MCDS Level 3

The OCDS Level 3 is based on a Multi Core Debug Solution (MCDS) using a special emulation device that has additional features required for high-end emulation purposes. It does not use more interface signals but replicates the debug interface for many cores and provides two out of N simultaneous trace channels differentiated by the process ID port.

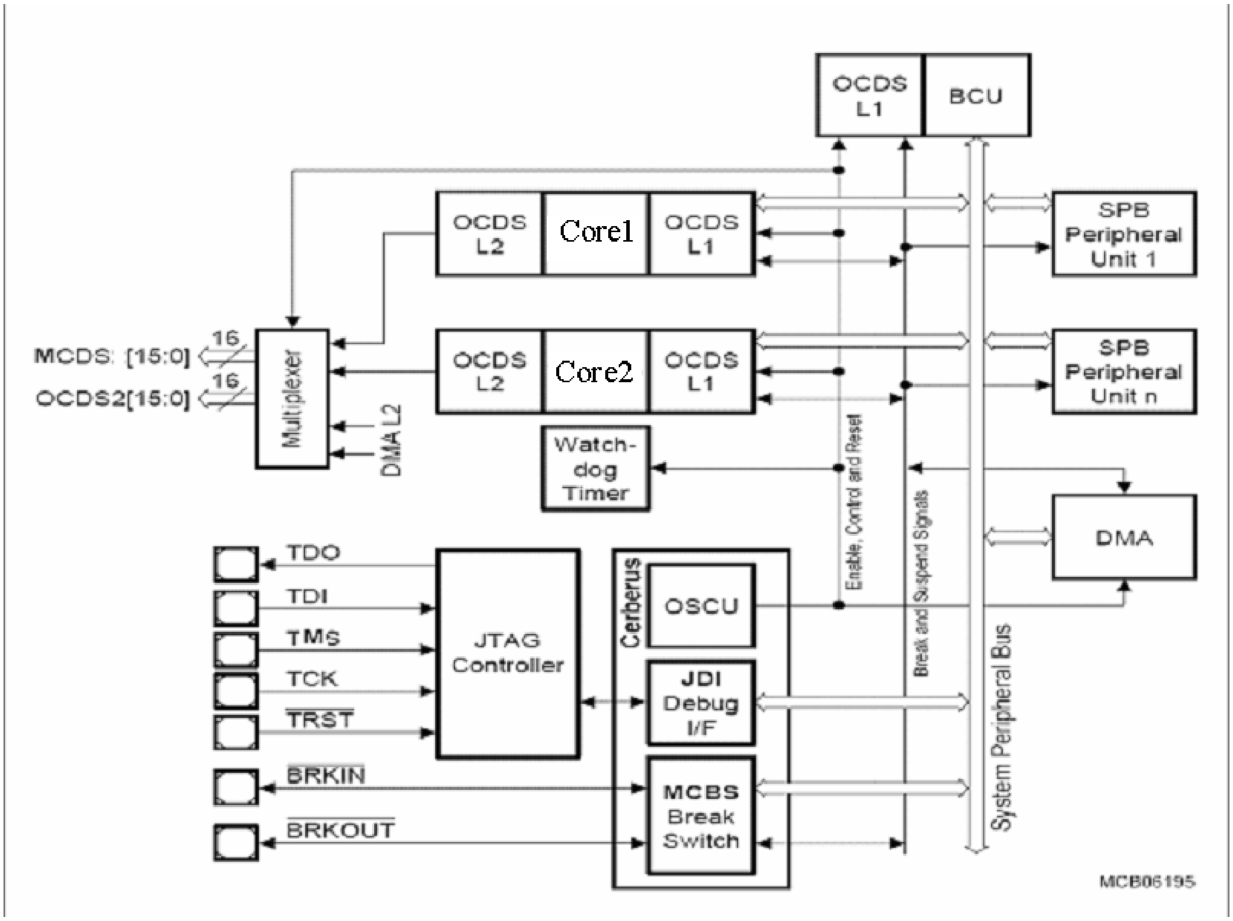


Figure A.4.1 - OCDS Based System Block Diagram

Components in Figure A.4.1

The OCDS consists of the following building blocks:

- Cerberus - OCDS System Control Unit (OSCU)
- Cerberus - Multi-Core Break Switch (MCBS) (Cross-Trigger unit with external extensions)
- Cerberus - JTAG Debug Interface (JDI)
- Suspend functionality of the peripherals (stop block activity for debug purposes)

- Several L1, L2 units for the cores and IP-blocks
- BCU allows cross-triggering by the system bus events

Cerberus is one example implementation of the OCP Debug Bus-Transactor.

A.4.1 OCDS Level 1

The main philosophy of the cores is that the complete architecture and the status of a target system are visible from the FPI Bus (memory-mapped debug concept). This means that every component of the system can be accessed through its mapping into the FPI address space, including on-chip memories, CPU core registers and register of the peripheral units.

A typical OCDS Level 1 debugging configuration is shown in **Figure A.4.2**. It includes two parts:

1. The debugger software, supporting a standard JTAG protocol via a PC port
2. The debugger hardware adapter, connecting the JTAG interface with the PC port (parallel, serial, Ethernet or USB)

This configuration makes it possible to realize a very simple debugging environment that permits comprehensive real-time debugging tasks to be performed.

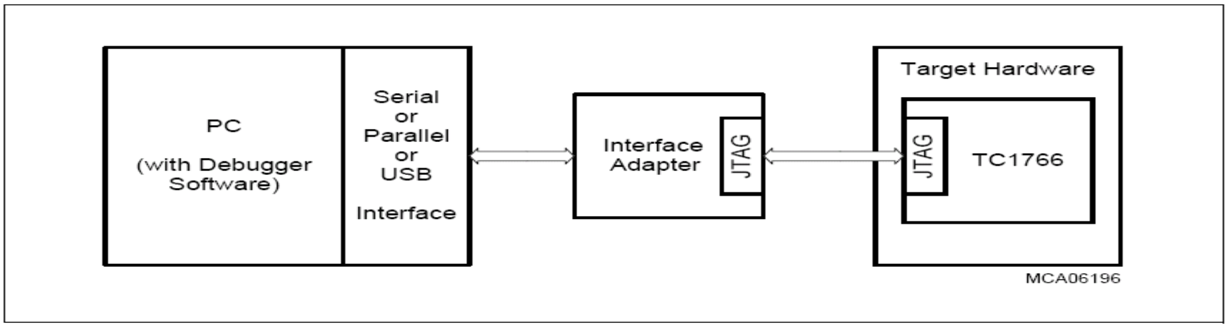


Figure A.4.2 - Typical OCDS Level 1 Hardware Connections

A.4.1.1 Basic Concept

The CPU core1 provides OCDS with the following two basic parts:

- Debug Event Trigger Generation
- Debug Event Trigger Processing

The first part controls the generation of debug events and the second part controls what actions are taken when a debug event is generated.

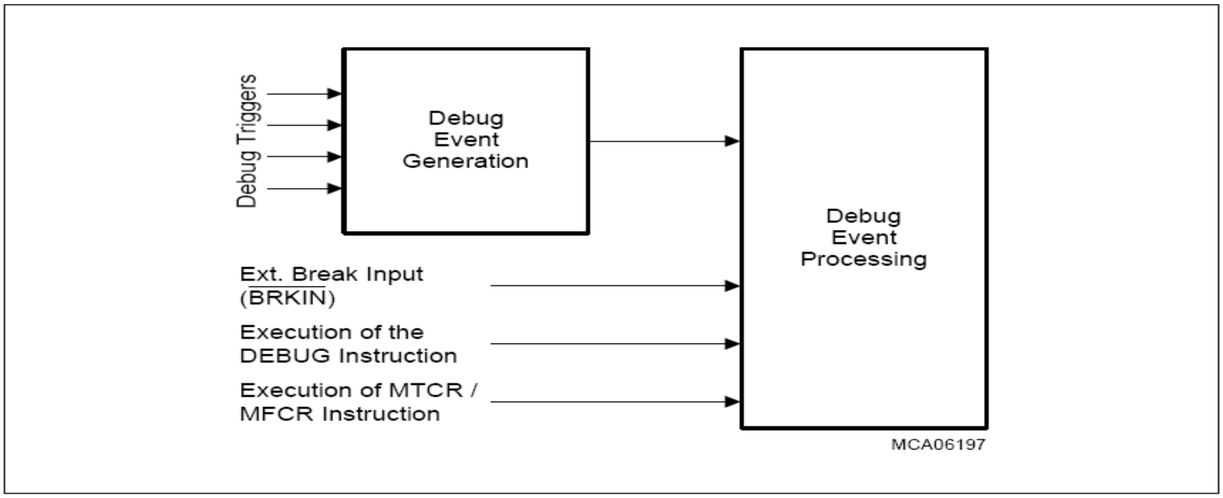


Figure A.4.3 – Core Debug Concept

A.4.1.2 Debug Event Generation

If debug mode is enabled, debug events can be generated by:

- Debug event generation from debug triggers
- Activation of the external break input pin BRKIN
- Execution of a DEBUG instruction
- Execution of an MTCR/MFCR instruction

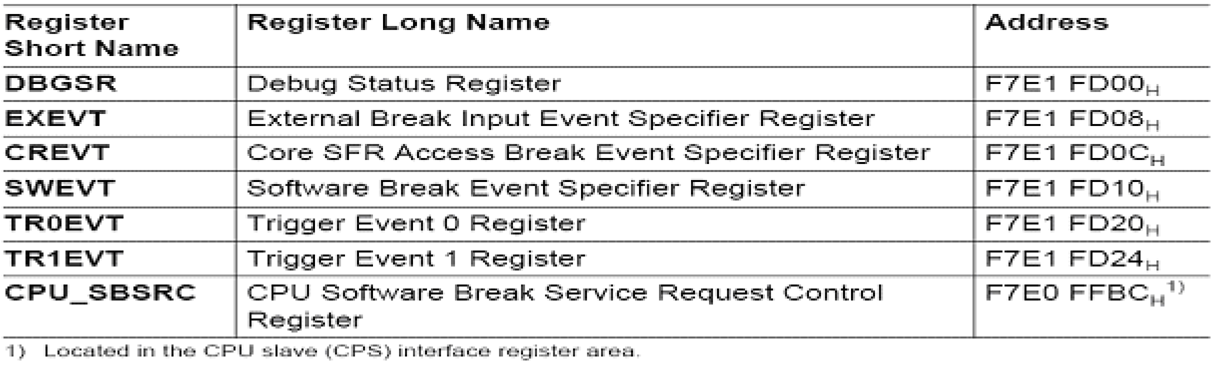
A.4.1.3 Debug Actions

Four types of debug actions are available:

- Assert BRKOUT signals by the MCBS unit
- Halt the CPU core
- Cause a breakpoint trap
- Generate an interrupt request

These debug actions are selected by programming the corresponding Event Specifier registers. Their contents determine which action shall be taken when the corresponding debug event occurs.

A.4.1.4 CPU Core-1 OCDS Registers (Many popular cores have similar registers)



Register Short Name	Register Long Name	Address
DBGSR	Debug Status Register	F7E1 FD00$_H$
EXEVT	External Break Input Event Specifier Register	F7E1 FD08$_H$
CREVT	Core SFR Access Break Event Specifier Register	F7E1 FD0C$_H$
SWEVT	Software Break Event Specifier Register	F7E1 FD10$_H$
TR0EVT	Trigger Event 0 Register	F7E1 FD20$_H$
TR1EVT	Trigger Event 1 Register	F7E1 FD24$_H$
CPU_SBSRC	CPU Software Break Service Request Control Register	F7E0 FFBC$_H^{1)}$

1) Located in the CPU slave (CPS) interface register area.

Table A.4 Core OCDS Registers

A.4.1.5 BCU OCDS Level 1 (OCP Bus-Observer unit on the system bus)

The BCU on the FPI bus supports OCDS Level 1 and offers very comfortable and powerful means for breakpoint generation. The BCU contains one comparator for
- the arbitration phase (look for specific bus master)
- the address phase (look for specific address or range)
- the data phase (look for read, write, supervisor mode, etc.)

The results can be combined to generate a break request signal, to be sent to the Break Switch (Cross Trigger Block).

A.4.2 CPU or PCP OCDS Level 2 Trace

Every trace clock cycle, 16 bits of CPU/PCP trace information are sent out, representing the current state of the CPU/PCP cores. The trace output lines are grouped into three parts:
- 5 bits of pipeline status information
- 8-bit indirect PC bus information
- 3 bits of breakpoint qualification information

With this information, an external emulator can reconstruct a cycle-by-cycle image of the instruction flow through the CPU or PCP. The trace information can be captured by the external debugger hardware and used to rebuild later on (off-line, using the source code) a cycle accurate disassembly of the code that has been executed. It is also possible to follow in real-time the current PC, facilitating advanced tools such as profilers, coverage analysis tools etc.

The trace output port is controlled by the OSCU. The trace data can be output at CPU clock speed (*f*TRCLK = *f*CPU). Trace clock can be higher if two cores are traced or a better compression of trace data of all cores can keep this trace clock low.

A.4.3 Concurrent Debugging in Level 3 MCDS (Two Channel Tracing)

A concurrent debugging is possible when the control port is used as second channel and ownership port is extended with process ID to differentiate between two sources that are traced. The debug setup must define which two cores or IP-blocks were selected for concurrent tracing.

A.4.4 Debug Interface (Cerberus) (Debug Bus-Transactor Module)

The Cerberus module is the on-chip unit that controls all OCDS Levels 1, 2 and 3 main debug functions. Generally, the Cerberus should not be used by any application software, since this could disturb the emulation tool behavior.
The Cerberus module is built up by three parts (see also **Figure A.4.1**):

- OCDS System Control Unit – OSCU (Debug Bus Master)
- JTAG Debug Interface - JDI
- Multi Core Break Switch – MCBS (OCP Cross-Trigger unit)

A standard JTAG interface is connected via the JTAG controller with the JDI. Two pins are available to handle an external break condition. An external debug hardware can access the Cerberus registers and arbitrary memory locations across the System Peripheral (FPI) Bus.

RW Mode and Communication Mode

As the name implies, the RW mode is used by a JTAG host to read or write arbitrary memory locations via the JTAG interface. The RW mode needs the FPI Bus master interface of the Cerberus to actively request data reads or data writes.

In Communication Mode, the Cerberus has no access to the FPI Bus and communication is established between the external JTAG host and a software monitor (embedded into the application program) via the Cerberus registers. The communication mode is the default mode after reset.

In Communication Mode, the external JTAG host is master of all transactions. He requests the monitor to write or read a value to/from the Cerberus register COMDATA. The difference to RW Mode is, that the read or write request is not actively executed by the Cerberus, but it sets request bits in the CPU accessible IOSR register to signal the monitor that the debugger wants to send (IO_WRITE_WORD) or receive (IO_READ_WORD) a value. The software monitor has to poll register IOSR. The IOADDR register is not used.

A.4.4.1 Multi Core Break Switch (OCP Cross-Trigger Unit)

In this example, there are two main processor units, the CPU core1 and the PCP2 core2. For debugging purposes, the OCDS run control of one processor unit can break (interrupt) the other processor unit or vice versa. This run control tasks are handled by the MCBS unit which is a part of the Cerberus. **Figure A.4.4** shows the break signal interfaces of this MCBS unit.

The MCBS unit supports the following features: (very similar to the OCP Debug Standard)

- Two independent break-out master units (Core1 and Core2)
- Six break-in sources (CPUCore, PCP, DMA, SBCU, MLI0, MLI1)
- Two port pins, BRKIN and BRKOUT
- Two independent break buses (two out of N)
- Suspend generation supports delayed suspend
- Break-to-suspend converter
- Create interrupt request with a break coming from a source
- Synchronous restart of the system

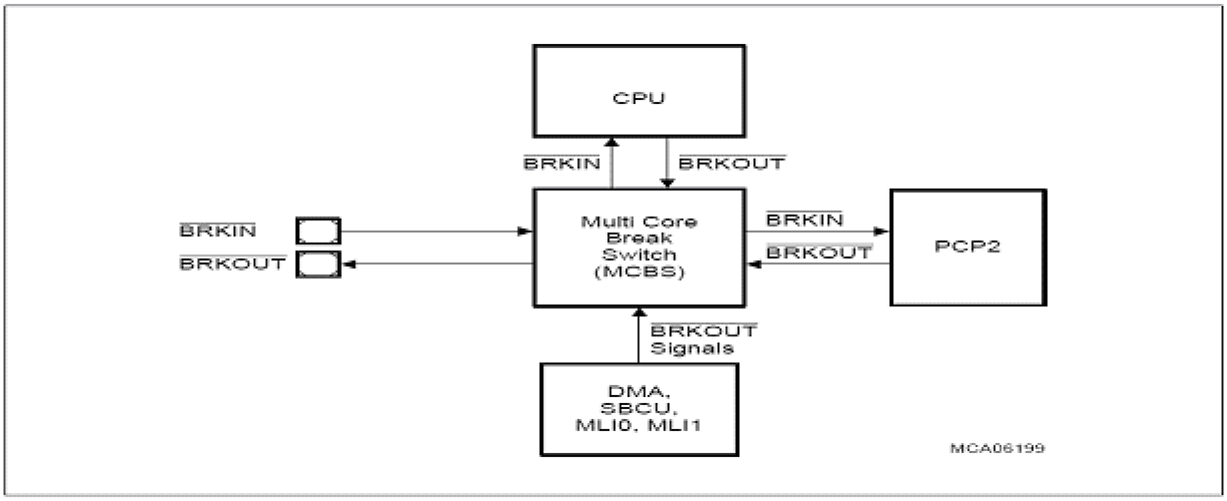


Figure A.4.4 Break Switch Interfaces

A.4.5 JTAG Interface

The JTAG interface is a standardized unit that is typically used for boundary scan and internal device tests. Because both of these applications are not active during normal device operation in a system, the JTAG port can be used during normal device operation as an ideal interface for debugging tasks.

On the other hand, the MCDS is designed to support complex multicore/debugging environments. The challenge here is that several debugger applications may have to share a single resource, i.e. the same JTAG interface. This becomes even more complicated because the JTAG module contains the IEEE 1149.1 JTAG state machine, which must be handled in the correct manner.

The solution to this problem is a JTAG driver with a defined API and a Debug Applications Server DAS. The server can connect to many debuggers and redirect the debug streams through one JTAG. It allows several debugger applications to share the same JTAG interface. In addition, the tool-specific PC interfaces like Ethernet, printer-port, or even USB can be hidden from the debugger software by the Debug Applications Server DAS. See Figure 3.1.

DAS enables the debugger vendor to ignore the complex task of understanding the JTAG module and supporting its functionality at low-level. All required information is provided as specifications and function references. The DAS API can be obtained from the SPRINT consortium.

A.4.6 Cerberus Bus Master Registers and JTAG

This section summarizes all Cerberus and JTAG registers as seen by the Debug Applications Server DAS for reference purposes.

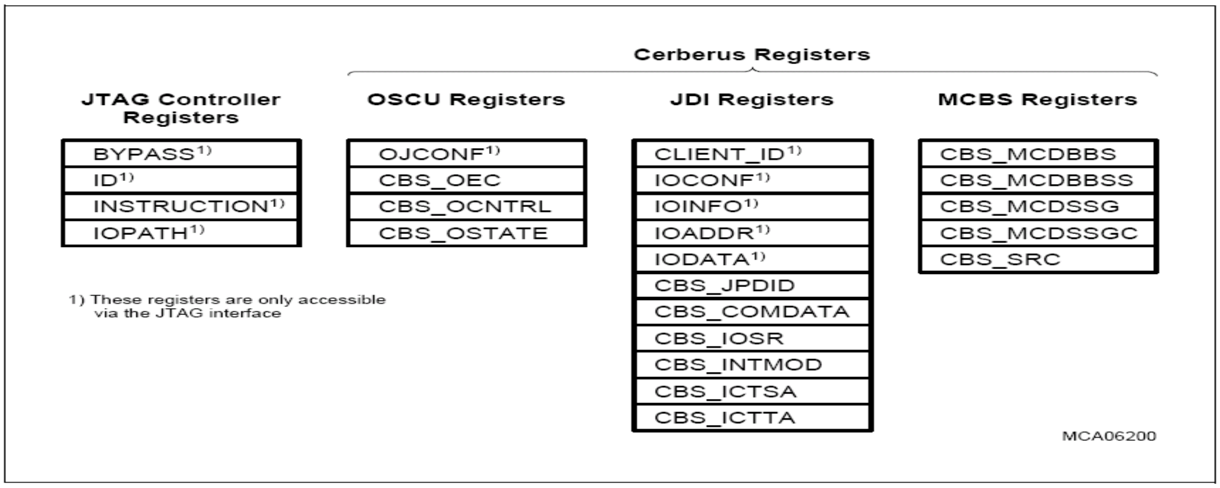
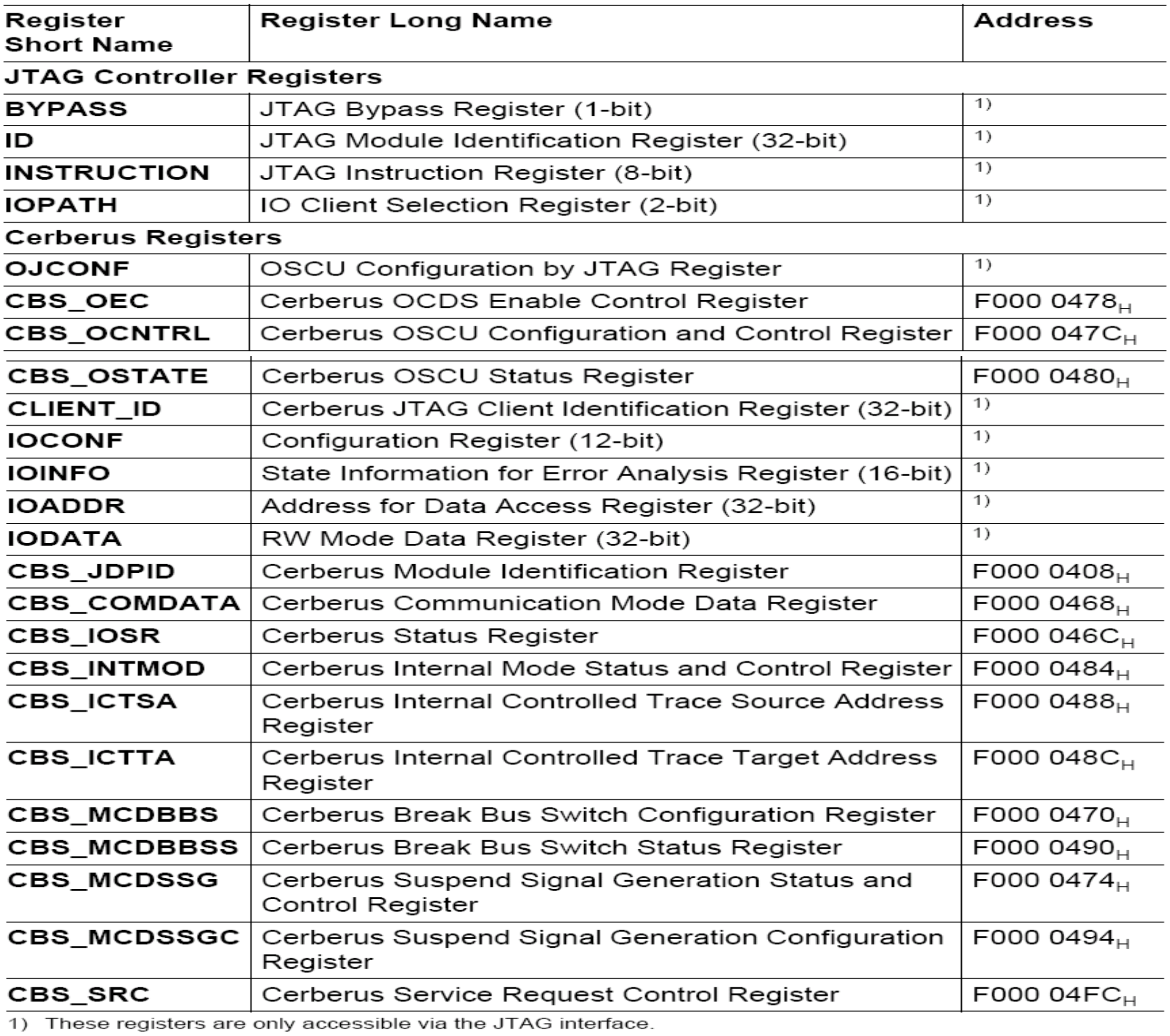


Figure A.4.5 JTAG/Cerberus Register Overview

Table A.4.2 JTAG / Cerberus Register Overview



Register Short Name	Register Long Name	Address
JTAG Controller Registers		
BYPASS	JTAG Bypass Register (1-bit)	[1]
ID	JTAG Module Identification Register (32-bit)	[1]
INSTRUCTION	JTAG Instruction Register (8-bit)	[1]
IOPATH	IO Client Selection Register (2-bit)	[1]
Cerberus Registers		
OJCONF	OSCU Configuration by JTAG Register	[1]
CBS_OEC	Cerberus OCDS Enable Control Register	F000 0478_H
CBS_OCNTRL	Cerberus OSCU Configuration and Control Register	F000 $047C_H$
CBS_OSTATE	Cerberus OSCU Status Register	F000 0480_H
CLIENT_ID	Cerberus JTAG Client Identification Register (32-bit)	[1]
IOCONF	Configuration Register (12-bit)	[1]
IOINFO	State Information for Error Analysis Register (16-bit)	[1]
IOADDR	Address for Data Access Register (32-bit)	[1]
IODATA	RW Mode Data Register (32-bit)	[1]
CBS_JDPID	Cerberus Module Identification Register	F000 0408_H
CBS_COMDATA	Cerberus Communication Mode Data Register	F000 0468_H
CBS_IOSR	Cerberus Status Register	F000 $046C_H$
CBS_INTMOD	Cerberus Internal Mode Status and Control Register	F000 0484_H
CBS_ICTSA	Cerberus Internal Controlled Trace Source Address Register	F000 0488_H
CBS_ICTTA	Cerberus Internal Controlled Trace Target Address Register	F000 $048C_H$
CBS_MCDBBS	Cerberus Break Bus Switch Configuration Register	F000 0470_H
CBS_MCDBBSS	Cerberus Break Bus Switch Status Register	F000 0490_H
CBS_MCDSSG	Cerberus Suspend Signal Generation Status and Control Register	F000 0474_H
CBS_MCDSSGC	Cerberus Suspend Signal Generation Configuration Register	F000 0494_H
CBS_SRC	Cerberus Service Request Control Register	F000 $04FC_H$

1) These registers are only accessible via the JTAG interface.

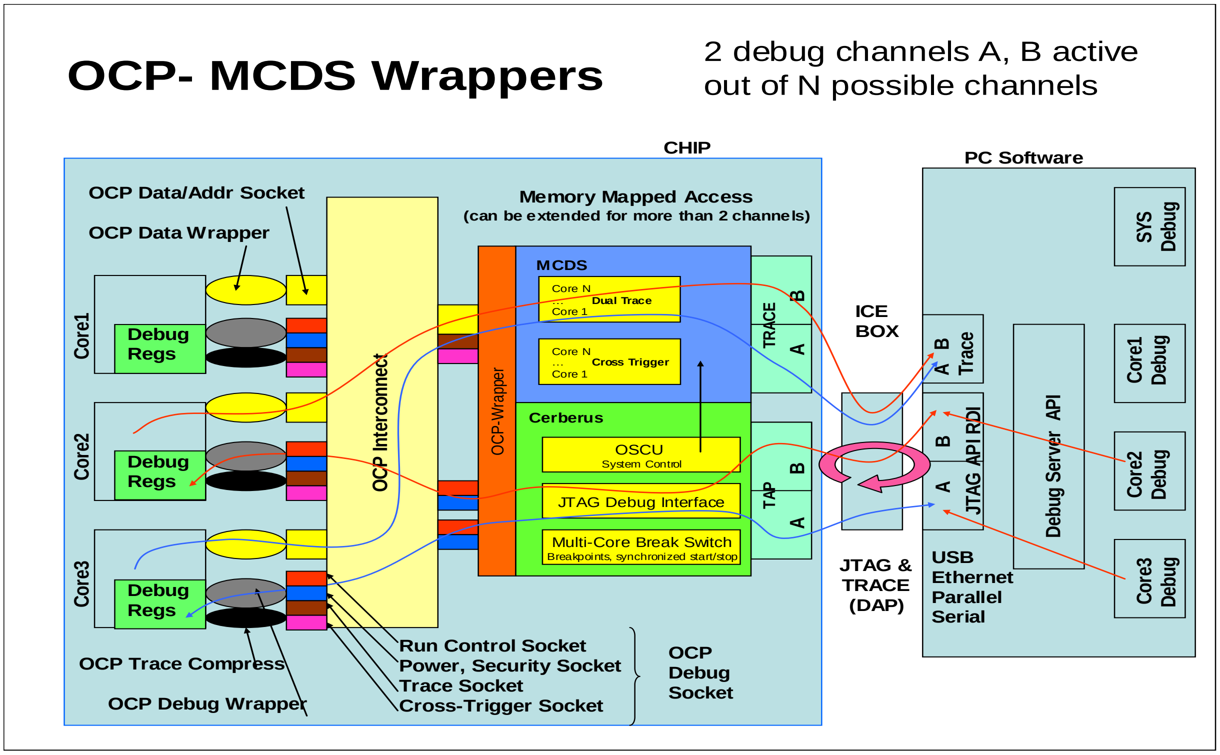
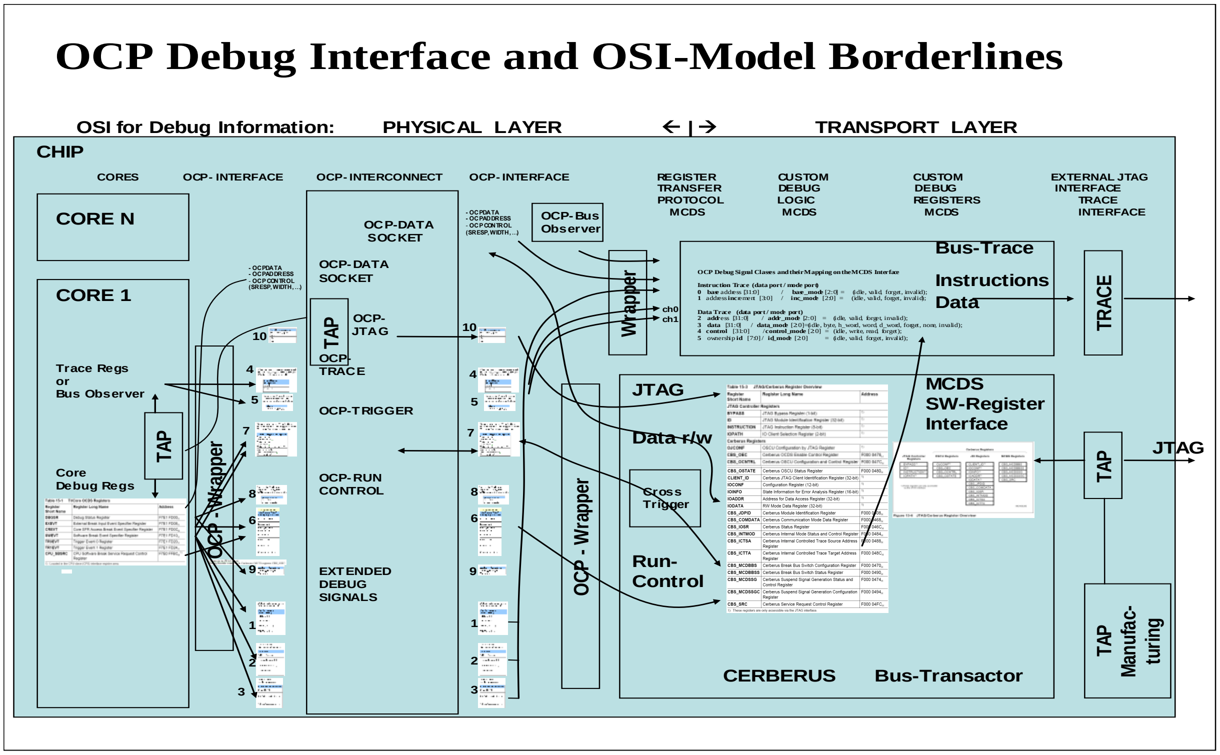


Figure A.5 : OCP/MCDS Wrappers



OCP Debug Signal Classes and their Mapping on the MCDS Interface

Instruction Trace (data port / mode port)
0 base address [31:0] / base_mode [2:0] = (idle, valid, forget, invalid);
1 address increment [3:0] / inc_mode [2:0] = (idle, valid, forget, invalid);

Data Trace (data port / mode port)
2 address [31:0] / addr_mode [2:0] = (idle, valid, forget, invalid);
3 data [31:0] / data_mode [2:0]=(idle, byte, h_word, word, d_word, forget, none, invalid);
4 control [31:0] /control_mode [2:0] = (idle, write, read, forget);
5 ownership id [7:0]/ id_mode [2:0] = (idle, valid, forget, invalid);

Figure A.6 : OCP Debug Interface and OSI Model Borderlines. See page 50, 51, 56, 59.
http://en.wikipedia.org/wiki/OSI_model

B. Cross-Triggering Subsystem Example

B.1 Functionality of Cross-Triggering in DSP Like Multichip Systems

Here we have a use case for the cross-trigger lines. Following are descriptions of functionality how the OCP Debug Interface signals can be mapped to existing implementations of debug systems.

We will explore the DSP like multi chip debug implementation similar to TI products where

- OCP Trigger-In is connected to nEMUI
- OCP Trigger-Out is connected to nEMUO
- OCP Trigger-Enable is connected to nEMUOE

B.2 Trigger Control Register

The register in Table 3 allows users to read or set each bit of the trigger lines. The register itself is part of the cross-trigger block. Since it is memory mapped it can be accessed for read/write by the core itself or by a debug transactor.

B.3 Emulation Trigger Functionality

Cross-trigger C-TRIG supports two emulation triggers: Trigger0, and Trigger1. For each trigger, there are three signals; nEMUxO is an output; nEMUxOZ is an output enable; nEMUxI is an input. These three signals are combined at the device level to create triggers to other cores on chip and to the external EMU pins. The trigger functionality ***shall*** be as shown in Table B.2.

Under certain configurations, Table B.2 indicates that EMUDBGREQx is asserted based upon nEMUxI. The EMUDBGREQx signal ***shall*** be gated by the security settings. Given that the security constraints are met, it may result in halting the CPU via its EDBGRQ signal.

Bit	Field	Width	R/W	Reset	Description
31-28	Claim	4	RW	2	This resource must be claimed before use.
27-20	Reserved	8	R	0	
19	nEMU1O	1	R	1	Reading the nEMU10 bit ***shall*** indicate the current value of the trigger 1 output.
			W		When in BIT IO Mode, writing to this bit shall set the state of the trigger 1 output signal
18	nEMU1OE	1	R	1	Reading the nEMU1OE bit ***shall*** indicate the current value of the trigger 1 output-enable.
			W		When in BIT IO Mode, writing to this bit ***shall*** set the state of the trigger 1 output enable signal
17	nEMU1I	1	R	0	This bit ***shall*** indicate the current value of the trigger 1 input.
16	Reserved	1	R	0	
15-12	Trigger1Control[3:0]	4	RW	0000	Trigger1 control. See Table B.2.
11-8	Reserved	4	R	0	
7	nEMU0O	1	R	1	Reading the nEMU00 bit ***shall*** indicate the current value of the trigger 0 output.
			W		When in BIT IO Mode, writing to this bit shall set the state of the trigger 0 output signal
6	nEMU0OE	1	R	1	Reading the nEMU00E bit ***shall*** indicate the current value of the trigger 0 output-enable.
			W		When in BIT IO Mode, writing to this bit ***shall*** set the state of the trigger 0 output enable signal
5	nEMU0I	1	R	0	This bit ***shall*** indicate the current value of the trigger 0 input.
4	Reserved	1	R	0	
3-0	Trigger0Control[3:0]	4	RW	0000	Trigger0 control. See Table B.2.

Table B.1: Trigger Control Register

TriggerX Control [3:0]	Trigger Action	In/ Out	NEMUxO (output)	NEMUxOZ (output enable)	NEMUINTx	EMUDBGREQx
0000	Off	Off	High	High	High	Low
0001	Cross-trigger Halt	I	High	High	High	Not nEMUxI
0010	Cross-trigger Out	O	Low	Not DBGACK pulse	High	Low
0011	Cross-trigger	I/O	Low	Not DBGACK pulse	High	Not nEMUxI
0100	Interconnect	O	Low	Not (InterconnectError & /DBGACK)	High	Low
0101	Cross-trigger Interrupt	I	High	High	nEMUxI	Low
0110	Low Counter Overflow EMU0: Counter 0 EMU1: Counter 1 y = counter number (0,1)	O	Low	Not (CTRy_TRGREQ & /DBGACK)	High	Low
0111	Reserved for High Counter Overflow	O	Low	High	High	Low
1000	CPU WP Match EMU0: WP 0 EMU1: WP 1 w = watch point range (0,1)	O	Low	Not(RANGEOUTw & /DBGACK)	High	Low
1001	Reserved	O	Low	High	High	Low
1010	Reserved	O	Low	High	High	Low
1011	Reserved	O	Low	High	High	Low
1100	Reserved	O	Low	High	High	Low
1101	EXTERN	O	Low	Not (EXTERN & /DBGACK)	High	Low
1110	Reserved	O	Low	High	High	Low
1111	Bit IO	I/O	Register Bit	Register Bit	High	Low

Table B.2: Emulation Trigger Control

B.3.1 EMUx Trigger Action – OFF

The EMU0 and EMU1 trigger control functionality can be completely disabled by setting the TriggerXControl bits. The control value is ‘0000’ for disabling the emulation pin functionality. As see in Table B.2, the following conditions hold true when the trigger control functionality is disabled:

When the TriggerXControl field in the TCR is set ‘0000’ the EMUx output enables ***shall*** be set to off.

When the TriggerXControl field in the TCR is set ‘0000’ the EMUx inputs ***shall*** be ignored.

When the TriggerXControl field in the TCR is set ‘0000’ the corresponding EMUDBGREQx ***shall*** be driven LOW to ensure that the CPU does receive a debug request from the EMU triggers.

B.3.2 EMUx Trigger Action – Halt on EMUx input LOW

EMU0 and EMU1 can be used to halt the CPU by driving the corresponding input pin LOW. A high to low transition on either of the EMUx pins will cause the corresponding EMUDBGREQx to be asserted. This will effectively drive the external debug request (EDBGRQ) signal into the CPU. The EMU0 and EMU1 inputs are independent. This function is enabled by writing a ‘0001’ into the TriggerXControl bits. The following conditions hold true when this trigger control functionality is enabled

When TriggerXControl field in the TCR is set ‘0001’ then a high-to-low transition on the EMU input ***shall*** drive the EMUDBGREQ0 signal HIGH.

A low level on the EMU input pin ***shall*** not cause a debug request.

If both EMU0 and EMU1 are configured to cause a halt when a low is seen on the corresponding input, a LOW on EMU0 ***shall*** not generate EMUDBGREQ1.

If both EMU0 and EMU1 are configured to cause a halt when a low is seen on the corresponding input, a LOW on EMU1 ***shall*** not generate EMUDBGREQ0.

The CPU must be configured for ‘Halt Mode’ debug via the Debug Status Control Register (DSCR) within Debug TAP (DBGTAP) on the CPU.

B.3.3 EMUx Trigger Action – Drive EMUx LOW on CPU Halt

The EMU0 and EMU1 pins can be used to notify external clients which may also be connected to these lines, that the core has entered debug state and it has halted, since the last time it was issued a RUN command. The EMU0 and EMU1 outputs are independent. This function is enabled by writing a ‘0010’ into the TCR TriggerXControl bits. The following conditions hold true when this trigger control functionality is enabled:

When TriggerXControl field in the TCR is programmed to the value of ‘0010’, the EMU pin ***shall*** be driven LOW when DBGACK makes a low to high transition.

The EMU pin ***shall*** be driven low for 32 clock cycles and then return high.

This configuration ***shall*** not set EMUDBGREQ0 or EMUDBGREQ1

If, for example, the emulation pins are configured to drive the EMU0 output LOW and the EMU1 is configured to monitor for a high-to-low transition on its input, then a LOW seen on the EMU0 event will not be read into the EMU1 pin unless it is explicitly tied to the EMU0 pin.

DBGACK is the Debug Acknowledge signal that is driven by the CPU when it halts due to a debug event. DBGACK is driven LOW whenever the CPU exits debug state and starts running. Thus the output is driven LOW, and the tri-state is controlled by the NOT DBGACK signal.

B.3.4 EMUx Trigger Action – Bi-directional Cross-triggering

Bi-directional cross-triggering is selected on the emulation pins EMU0 and EMU1 by writing a '0011' to the TCR TriggerXControl bits. The following conditions hold true when this trigger control functionality is enabled:

The EMU0 input ***shall*** operate independent of the EMU0 output

The EMU1 input ***shall*** operate independent of the EMU1 output

If a High to Low transition is seen on the EMUx pins, the EMUDBGREQx signal ***shall*** be asserted high. The detection of a high to low transition on the EMUx pin is an asynchronous event.

A low level on the EMUx input pin ***shall*** not cause a debug request.

The assertion of EMUDBGREQx ***shall*** be inhibited for 1 clock cycle each time the DBGACK signal transitions from the high-to-low state. A high-to-low state transition on the DBGACK signal is an indication that the CPU is about to go out of debug state and start running. This must be done to prevent self-triggering.

A debug event such as a user halt or a breakpoint ***shall*** drive the EMUx pins LOW.

The EMU pin ***shall*** be driven low for 32 clock cycles and then return high.

Clearing the input synchronizers on a high to low transition of DBGACK will prevent retriggering from debug events that were registered on a previous halt.

B.3.5 EMUx Trigger Action – Notify Interconnect Error

The EMU0 and EMU1 trigger control can be used to notify the environment outside the chip that an application error has occurred on the L3/L4 interconnect. This function is selected by writing a '0100' to the TriggerXControl field. The following conditions hold true when this trigger control function is enabled:

The notification of Interconnect Error ***shall*** happen only at run-time when DBGACK is LOW.

The application error flag [Serror_App] reports both Application errors and not attributable errors

When the triggers are configured as "interconnect" and the SuppressInterconnIntr bit in the Debug Control and Status Register is set, then a non-attributable interconnect error, reported through an application error flag (Serror App) occurring while DBGACK is high, will not be reflected on the EMUx lines

The change in state on the EMUx pins may be used at the system level to either stop another CPU or start a trace.

B.3.6 EMUx Trigger Action – Generate EMUINT Interrupt

The EMU0 and EMU1 trigger control can be used to generate an interrupt on the target. This function is selected by writing a '0101' to the respective TriggerXControl field. The following conditions hold true when this trigger control function is enabled:

This function will effectively result in the generation of an interrupt to the CPU.

If EMU0 is driven LOW it will result in the generation of GNEMUINT0, which will in turn generate the IC_EMUINTR.

If EMU1 is driven LOW it will result in the generation of GNEMUINT1, which will in turn generate the IC_EMUINTR.
IC_EMUINTR is an input of the CPU interrupt controller.

B.3.7 EMUx Trigger Action – Notify on Low Counter Overflow

The EMU0 and EMU1 trigger control can be used to notify the environment outside the chip that the benchmark counters have overflowed. This function is selected by writing a ‘0110’ to the TriggerXControl field. The following conditions hold true when this trigger control function is enabled:
The change in state on the EMUx pins may be used at the system level to either stop or interrupt another CPU or signal an external piece of equipment.

B.3.8 EMUx Trigger Action – Notify on Watchpoint Match

The EMU0 and EMU1 trigger control can be used to notify the environment outside the chip that the watchpoints from the CPU matched. This function is selected by writing a ‘1000’ to the TriggerXControl field. The following conditions hold true when this trigger control function is enabled:
The change in state on the EMUx pins may be used at the system level to either stop or interrupt another CPU or signal an external piece of equipment.

B.3.9 EMUx Trigger Action - Notify on Extern Event

The EMU0 and EMU1 trigger control can be used to notify the environment outside the chip that an event has occurred on the EXTERN input signals. This function is selected by writing a ‘1101’ to the TriggerXControl field.

B.3.10 Trigger Control – Bit IO

The EMU0 and EMU1 trigger control can be used to directly drive and/or read the levels present on the EMU pins. This function is selected by writing ‘1111’ to the TriggerXControl field. The following conditions hold true when this trigger control function is enabled:
Each EMU pin can be tri-stated or driven to a high or low level
The current level of each EMU pin can be read

B.4 SOC Integration

The SOC manages two emulation triggers: trigger0 and trigger1. Based upon settings in a device or emulation pin manager, these triggers may be connected to the emulation device pins called EMU0 and EMU1. These triggers are also driven by signals from each core and are used to facilitate co-emulation. Cross triggering between cores within the device can occur even while the triggers are not connected to the external device pins.

Functionality

A trigger is used for both input and output. Each trigger consists of 3 signals.

Trigger Signal Name	Description
nEMUxO	Trigger-x output. Active pulse low.
nEMUxOz	Trigger-x output-enable. Active low.
nEMUxI	Trigger-x input. Active falling edge.

Table B.3: Emulation Trigger Description

Polarity

All trigger inputs and outputs are active low.

B.4.1 Device Pin Triggers

The SOC supports 2 external triggers. Each external emulation trigger corresponds to a device pin. The device pin EMU0 can be configured to contribute to trigger0. The device pin EMU1 can be configured to contribute to trigger1.

Internal to the chip, each EMU pin is made up of 3 signals. This means that there are 6 signals (2x3). All the external modules will have to have default states "1" on those signals (inactive states).

Signal Name	I/O	Description
POEMU0ON	Output from C-TRIG to EMU0 device pin.	Trigger-0 output. Active low.
POEMU0OEN	Output enable from C-TRIG to EMU0 device pin.	Trigger-0 output-enable. Active low.
PIEMU0I	Input to C-TRIG from EMU0 device pin.	Trigger-0 input. Active low.
POEMU1On	Output from C-TRIG to EMU1 device pin.	Trigger-1 output. Active low.
POEMU1OEN	Output enable from C-TRIG to EMU1 device pin.	Trigger-1 output-enable. Active low.
PIEMU1I	Input to C-TRIG from device EMU1 pin.	Trigger-1 input. Active low.

Table B.4: Emulation Signals for each EMU Pin

Input and Output

The EMU device pins are bi-directional pins. Even when the pin is driving an output, the input still senses the level on the device pin.

Input

By default, EMU0 and EMU1 are configured to be inputs that contribute to the trigger generation.

Output

For an EMU pin to act as an output, both the PODEMUpON signal must be driven with the desired value, and the corresponding DnDEMUpOEN signal must be driven low, where p is 0 for EMU0 or 1 for EMU1.

B.4.2 Module Triggers

C-TRIG supports many sets of emulation trigger groups. Each core or module in the chip can have up to 2 triggers. Each trigger consists of 3 signals. For example, if 7 secondary TAPs exist, C-TRIG will connect to 42 signals (7x2x3). In this case TAP doesn't mean JTAG programming. The programming model could be memory mapped and still support the described C-TRIG interface. This is one more advantage of mapping the hardware to the OCP fabric.

Signal Name	I/O	Description
PIEMU0ON[m]	Output from core to C-TRIG	Trigger-0 output. Active low.
PIEMU0OEN[m]	Output enable from core to C-TRIG	Trigger-0 output-enable. Active low.
POEMU0I[m]	Input to core from C-TRIG	Trigger-0 input. Active low.
PIEMU1ON[m]	Output from core to C-TRIG	Trigger-1 output. Active low.
PIEMU1OEN[m]	Output enable from core to C-TRIG	Trigger-1 output-enable. Active low.
POEMU1I[m]	Input to core from C-TRIG	Trigger-1 input. Active low.

Table B.5: Emulation Triggers for each TAP Core (m = TAP number)

B.4.2.1 Wiring for Modules without Triggers

If a module does not support a trigger, then the trigger inputs (PIEMUxON[m] and PIEMUxOEN[m]) to the C-TRIG for that TAP must be tied high external to the C-TRIG. This requirement does disappear if C-TRIG is distributed in OCP fabric and OCP Debug Socket is implemented only where desired- Then it's just configuration. EMUxI[m] is an output from C-TRIG. It can be left with no connection. For example, if TAP 5 does not have any triggers associated with it, then PIEMU0ON[5], PIEMU0OEN[5], PIEMU1ON[5], and PIEMU1OEN[5] will be tied high. This signals POEMU0I[5] and POEMU1I[5] will have no connection.

Signal Name	Tie-off External to C-TRIG
PIEMU0ON[m]	Tie high
PIEMU0OEN[m]	Tie high
POEMU0I[m]	No connect
PIEMU1ON[m]	Tie high
PIEMU1OEN[m]	Tie high
POEMU1I[m]	No connect

Table B.6: Tie-off for Modules without Emulation Triggers (m = TAP number without triggers)

B.4.2.2 Un-powered Modules

Holding Triggers High When Un-powered

When PISDOMAINPOWEREDx is 0, C-TRIG must hold POEMU0Ix and POEMU1Ix high. (Check if PISDOMAINPOWEREDx control is available in your design - It is in that case associated to an OCP master. We may see this redundant if power isolation is properly implemented. We want to keep it for robustness.) If an OCP debug interface is instantiated on the bus but not used in a design then all inputs have to be tied correctly high or low. Best is not to instantiate superfluous sockets.

Power Isolation

If isolation is required on any of these signals, the trigger signal should be held in the high state.

B.4.3 Trigger Generation

C-TRIG uses PIEMU0ON[m] and PIEMU0OEN[m] to create Trigger0. Similarly, C-TRIG uses PIEMU1ON[m] and PIEMU1OEN[m] to create Trigger1.

B.4.3.1 Trigger Output Enable

There are two trigger output enables; one for each trigger. TriggerxOZ is active low.

Generation

The output enable for trigger 0 is called Trigger0_N. It is 1 unless one or more of the following is true:
PIEMU0OEN[m] = 0 and PIEMU0ON[m] = 0, for any m = 0…M, where M is the number of secondary TAPs – 1
The output enable for trigger 1 is called Trigger1_N. It is 1 unless one or more of the following is true:
PIEMU1OEN[m] = 0 and PIEMU1ON[m] = 0, for any m = 0…M, where M is the number of secondary TAPs – 1
In other words, if any one or more modules call for the emulation trigger to be used as an output and the module is driving the trigger signal to zero, then the Trigger0_N will become 0.

Connections

The value of each trigger's enable is driven out to the device pins via the PODnEMU*x*OZ signals, where *x* is the trigger number. On-chip cross triggering shall function even if the EMU0,EMU1 pins are not available or any other debug / application function is mapped to these pins.

Trigger0_N drives:	POEMU0OEN and POEMU0ON
Trigger1_N drives:	POEMU1OEN and POEMU1ON

When configured externally to C-TRIG, trigger0 can drive EMU0, while trigger1 can drive EMU1.

B.4.3.2 Trigger Output

There are two triggers. Triggerx_N is active low.

Generation

The default state for a trigger is 1. It is held at this value unless one of the following conditions forces the trigger to be driven to 0. In the following equations, *X* or *x* represents the trigger number. This will be 0 or 1, depending on the trigger that is in question. If any one or more of these conditions is true, then the trigger should be 0.

Each TAP trigger signals are qualified by PISTAPENABLED and PISDOMAINPOWERED for the respective TAP. These are all combined as shown in Figure B.1. If any of the qualified trigger signals from the TAPs is set and nTRST is not being asserted, the Trigger signal is driven low. This signal drives both the active low output enable, POEMU0OEN and the data output signal, POEMU0ON. The tristate buffer shown shaded is implemented outside of the C-TRIG Module.

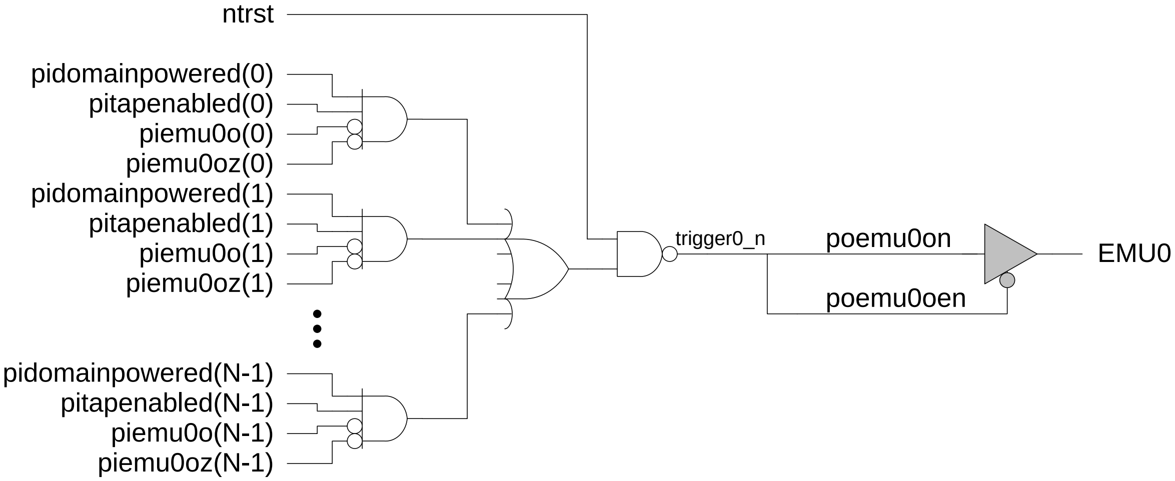


Figure B.1: Trigger Generation

If less TAPs are instantiated in the debug bank in the implementation of C-TRIG than inputs are defined, not all of the contributing trigger equations list above will exist. Trigger inputs only exists for TAPs that exist. This doesn't apply in case of a OCP distributed implementation where just the desired TAPs will be implemented.
The PISDOMAINPOWERED[m] and PISTAPENABLED[m] signals are used in the trigger builder equations.

If triggers are originated from different power domains, appropriate level shifters must be implemented external to the C-TRIG trigger generation block.

B.4.3.3 Trigger Input

The value of each trigger input is driven out to each core via the POEMU*x*I*m* signals, where *x* is the trigger number and *m* is the module TAP number. Hence, a signal is driven for each instantiated secondary TAP, regardless of whether the TAP is selected. In a C-TRIG implementation with 16 TAPs, 16 signals are always driven with the current value of the trigger.

Triggers to the cores can be created from the external trigger input signal, PIEMUxI or from the trigger output signal, Triggerx_N. If either of these two signals is low a trigger may be generated to each core, provided it is qualified. The trigger input is qualified by nTRST and PISTAPENABLED and PISDOMAINPOWERED for the respective TAPs as shown in Figure B.2. The POEMU1I Trigger is generated in a similar manner.

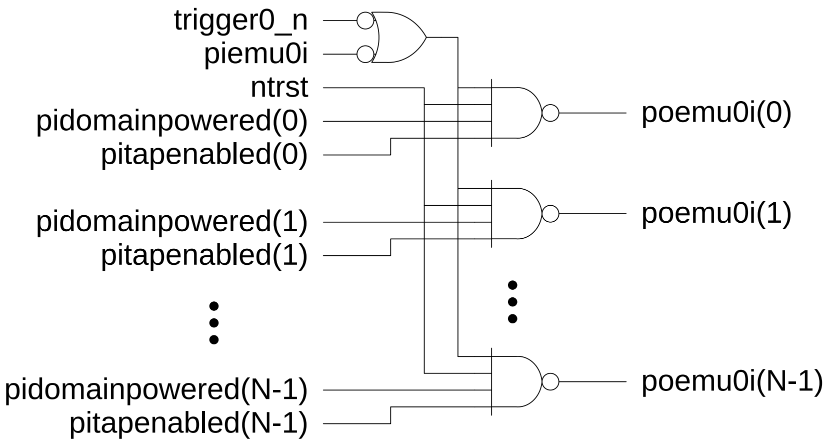


Figure B.2: Trigger Input Logic

Trigger outputs exist only for debug TAPs that exist.
When configured externally to C-TRIG, trigger0 can drive EMU0, while trigger1 can drive EMU1.

C. Nexus Example for OCP Debug Interface Implementation and Protocol

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. The Nexus infrastructure supports multicore development and multi featured trace and configuration/control. Nexus at its simplest (Class 1) level is compatible with JTAG, but recognizes that JTAG bandwidth limitations are not realistic for the debug requirements for complex or multicore environments. This discussion is supported by the Nexus 5001 specification which is freely available for download from the Nexus website. http://www.nexus5001.org/ The current version of the specification was released in 2003. Versions of Nexus architectures have been used extensively in US Automotive applications.

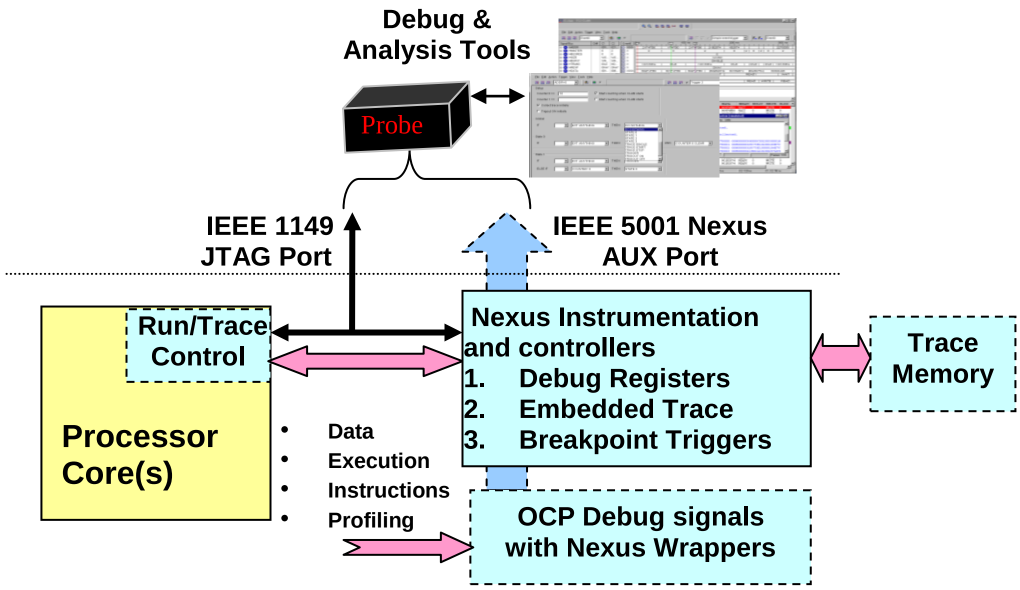


Figure C.1 : Nexus Interfaces

The Nexus architecture defines high performance data interface, protocol, and register that can be used to implement a variety of trace and control instrumentation.

C.1 Nexus Debug Signal Interface

The Nexus specification defines a vender-neutral IO signal interface and communications protocol that supports parallel debug and instrumentation support. 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 to allow higher read/write throughput between the target and debug and analysis tools compared to JTAG.

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.

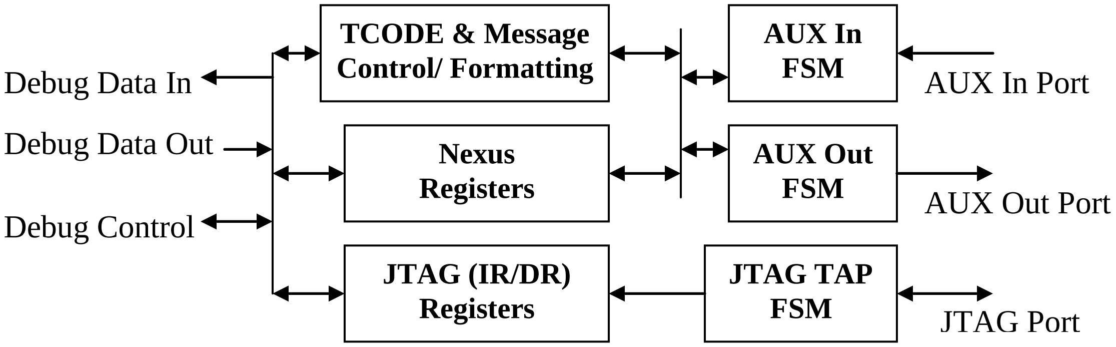


Figure C.2: Nexus Internal Architecture

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.*

C.2 Nexus Message Format

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 set of TCODEs for common identification and trace operations – including :

- 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.

C.3 Nexus Debug Registers

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. Standard definition and location of register set allows simpler integration and control of the instrumentations with embedded debuggers and related tools. Nexus defined recommended registers for debug purposes include:

- Device identification register (DID)
- 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)

C.4 Nexus Multi-core Debug Example

Nexus 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 package debug information into Nexus messages. Nexus messages can 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.

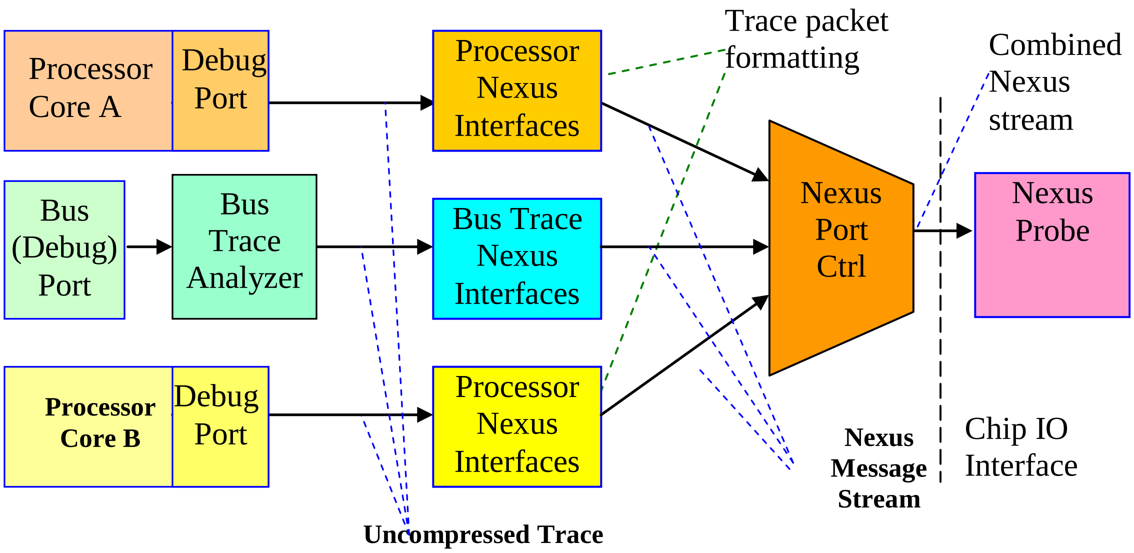


Figure C.3 – Basic Nexus Multi-core Debug flow

Figure C.3 shows this debug data flow, supporting a multi-core architecture consisting of 2 processor (or other) cores with debug interfaces and a bus level debug interface. Native debug blocks can be made Nexus compliant via a translation layer that supports debug information made into Nexus compliant messages, including any additional compression; A Nexus Port interface or multiplexer allows selecting message streams through 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 over many different core or subsystems remains largely independent. Coordinated control and synchronization of different debug resources can significantly improve debug efficiency. A Multi-core Embedded Debugger architecture (Fig. C.4), in addition to the Nexus interfaces for each of the on chip debug resources, may include cross triggering and system timestamping resources to help synchronize and cross-reference concurrent debug operations occurring at different parts of the architecture, allowing different off chip debugger environments to better comprehend the context of operations occurring in other parts of a design.

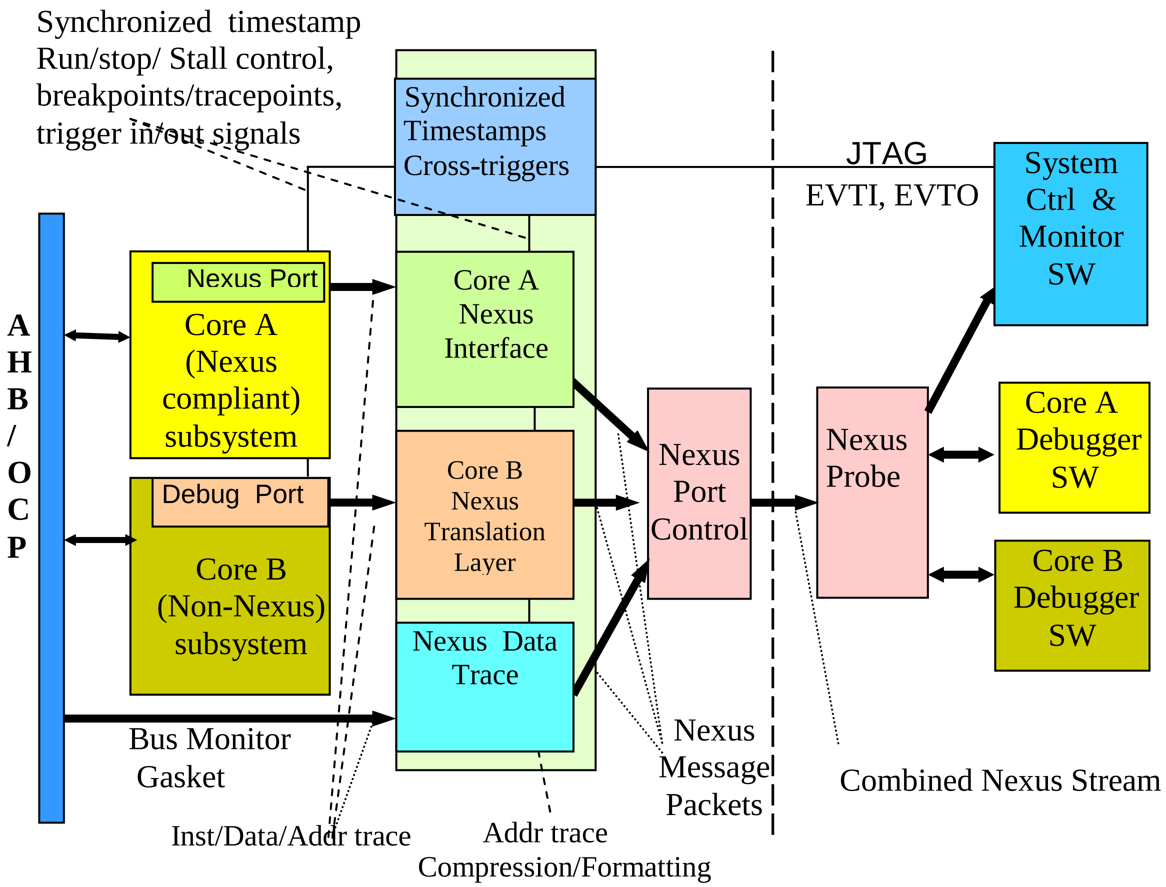


Figure C.4 – A Nexus compliant OCP Multicore Embedded Debug environment

Nexus 5001 Forum has ongoing collaboration with industry debug related efforts, including OCP-IP http://www.ocpip.org/pressroom/releases/2007_press_releases/OCP_Nexus.pdf 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.

Bibliography

OCP Debug Interface Standard 1.0
OCP Standard 3.0
MCDS Debug Block IP (Property of Infineon)
MCDS Debug Software (Property of PLS)
ARM Debug System CoreSight (Property of ARM)
MIPS multicore debug (Video Multiprocessor Debug System property of Mobileye)
NEXUS (Open Standard for Debug Tools in Automotive and Related)
LIT1: Extending Open Core Protocol to Support System-Level Cache Coherence; Konstantinos Aisopos, Chien-Chun Chou, Li-Shiuan Peh; CODES + ISSS, 2008

OCP International Partnership
3855 SW 153rd Drive
Beaverton, OR 97006
Ph: 503-619-0560
Fax: 503-297-1090
admin@ocpip.org
www.ocpip.org

Part Number: 1003