A White Paper
Presented by IPextreme

IPextreme

Multi-Core Debug Solution IP

SoC Software Debugging and Performance Optimization >> May 2007

Dr. Albrecht Mayer, Infineon Technologies AG
Harry Siebert, Infineon Technologies AG
Christian Lipsky, IPextreme, Inc.

Today's multi-core, multi-bus SoCs lack the necessary externally visible observation points needed for real-time system debug and performance optimization. Developers need access to a vast amount of data from deep within the chip while the chip is operating in its system environment. And, for effective debugging, that trace data needs to be filtered down to a manageable quantity of relevant data.

In this paper, we show how Infineon's Multi-Core Debug Solution makes real-time, in-system debug and performance analysis practical by tracing the relevant information at the relevant time(s).

HIGHLIGHTS

- Limitations of traditional debug methodologies
- Approaches to on-chip trace
- MCDS—a proven solution
- Debug tools adapt to advanced on-chip trace technology

ABSTRACT

The speed and density of today's multi-core SoCs have outgrown traditional debugging methodologies. To debug a system in its target environment, where problems often only occur, a debugger needs access to an enormous amount of trace data from various processors, buses, and signals within the SoC. Getting this data off-chip to the debugger in real time requires on the order of 100 Gbits/sec of bandwidth at the chip I/O, which is not practical using either dedicated debug pins or shared debug/functional pins. The problem is further compounded by the need to analyze all of that data.

Infineon has successfully developed and deployed a technology known as the Multi-Core Debug Solution (MCDS) to address that problem. Using advanced on-chip trace techniques that include on-chip trigger generation, trace data compression, and trace storage, MCDS provides only the relevant trace data to the debug tool. Without adding pins to the chip, MCDS enables real-time, in-system debug and performance optimization.

TABLE OF CONTENTS

WHY COMPLEX SOC DEBUGGING NEEDS A SOLUTION

Introduction

The migration from system-on-printed circuit board (PCB) to system-on-chip (SoC) has moved more and more parts of the system onto the SoC. An unintended side effect of this higher level of integration is the loss of observation possibilities (Figure 1).

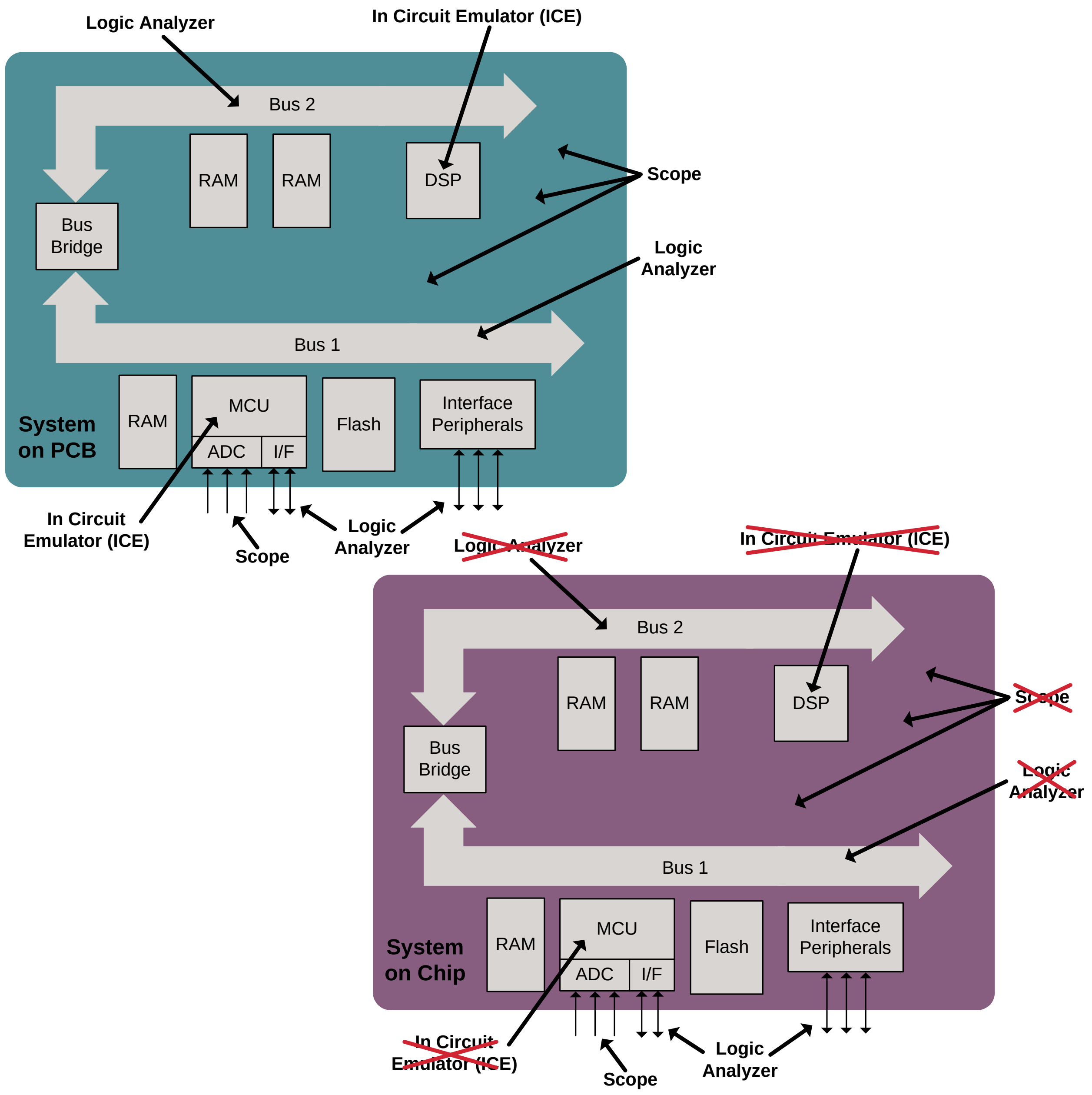

Figure 1: Tool Probing Points for System on PCB and SoC

When the program memory or a cache is on-chip, it is no longer possible to observe the program flow by just tracing the fetched code at the external bus interface. On the data access side, the trend is similar.

The latest generation of SoCs goes one step further with the introduction of multi-core architectures. Also, due to active peripherals like DMA or serial interfaces with bus master capabilities, such systems are increasingly more difficult to design and to debug. This is particularly true for real-time systems such as for automotive, where traditional debugging with breakpoints and single-stepping is definitely not adequate.

For example, even if the SoC in Figure 1 contains trigger logic on bus 2 that can stop the whole system when there is a wrong data access, this is not always sufficient to find the root cause of the problem. The bug could be in an interrupt service routine running on the processor core, which has already returned when the pipelined write access is propagated over the bus bridge from bus 1 to bus 2. In addition, the bug could be sporadic and may only occur when the SoC is used in its hard real-time environment. The obvious solution for such situations is to have trace support on the SoC.

Zero Defect Culture

Not only is the hardware part of a system is getting more and more complex, it is the same for the software (Figure 2). Traditional debugging still searches and finds a significant number of bugs when all parts of the system work together. However, ensuring a completely bug-free design becomes nearly impossible as the size and complexity of the system exceeds a certain limit. The time needed to analyze a bug, where the effect is visible in a totally different place and the link to the root cause is across several component boundaries and hierarchy levels, can obsolete any time-to-market planning. On the other hand, a bug that slips through testing can lead to a very expensive recall of the product.

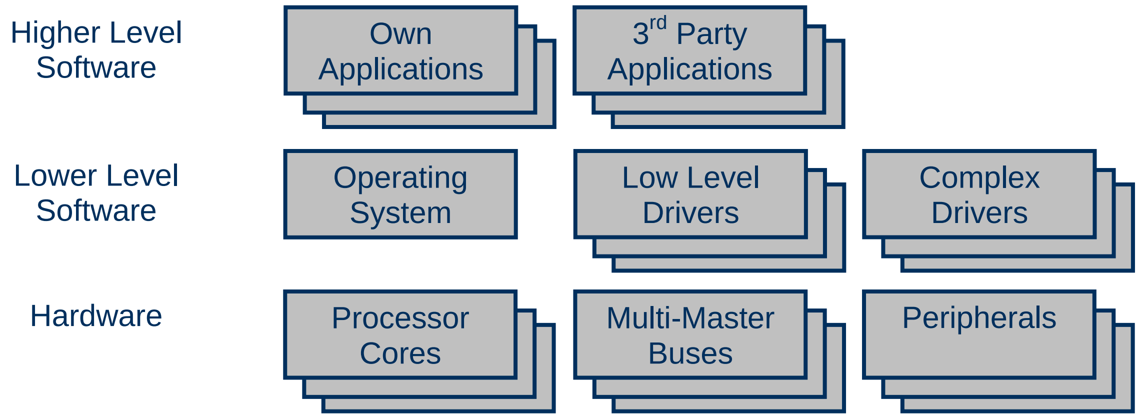

Figure 2: System Components on Hardware and Software Levels

The obvious and proven way out of this complexity trap is to build a system the employs well-defined, hierarchically structured components, with straightforward interfaces between them. For each single component, the verification complexity is manageable, so a bug-free component is not a completely unrealistic target. The debugging scope is more local, starting hierarchically from bottom to top. A lower-level component is merged into the next higher level of hierarchy only when it is fully tested and bug-free (Figure 3).

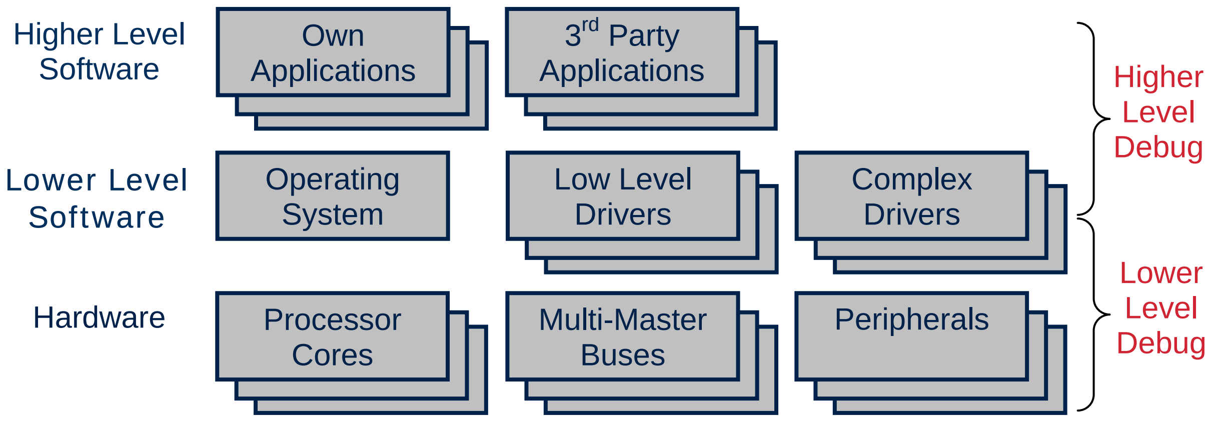

Figure 3: Debugging Levels

SoC Debug Challenge

Debugging of complex systems requires parallel tracing of selected system states. Ideally, the program flow of all cores, their data accesses, data transfers on multi-master buses, special signals, and certain states of peripherals would be available for analysis with cycle-accurate alignment. The key challenge is the getting the required bandwidth for the trace data. For a multi-scalar processor core, the Instruction Pointer (IP) and up to two data accesses (address and data) need to be recorded in the same cycle. Depending on the processor architecture, this amounts to 100–200 bits per clock cycle; so, for a multi-core SoC with fast processor cores, the order of magnitude of the required trace bandwidth is 100 Gbit/s. Even with trace data compression—which works fine for IP trace (compression factor of 10 to 20) but is quite limited for data trace (maximum compression factor of 2)—it is impossible to get this bandwidth off-chip in a cost-effective way.

The increased cost of getting the trace data off-chip can be limited to only a test version of chip, but the next hurdle is the external trace capture and storage unit. Gigabytes of trace memory are sufficient only for seconds of trace; and the trace data still needs to be uploaded to a computer. This will take about 20 seconds for one gigabyte over a gigabit Ethernet link, after which the debug tool software still needs to analyze and filter the data, which can take up to several minutes.

From Pin-Based Trace to On-Chip Trace

Figure 4 shows the general trend for required trace bandwidth for leading edge SoCs using the latest technology. The required trace bandwidth scales with the number of cores and their interconnect busses, multiplied by the operating frequency. This trend is balanced with the falling cost of gates and memory cells, which are the building block elements of on-chip trace solutions. However, it is not balanced with the cost of bandwidth per pin. At the physical transport layer level, there is no benefit from newer technologies. The silicon size is determined by mechanical constraints (bonding) and the required driver strength for a high-frequency signal with a given external load capacitance. Also, the packaging cost per pin is falling much more slowly than the cost per transistor.

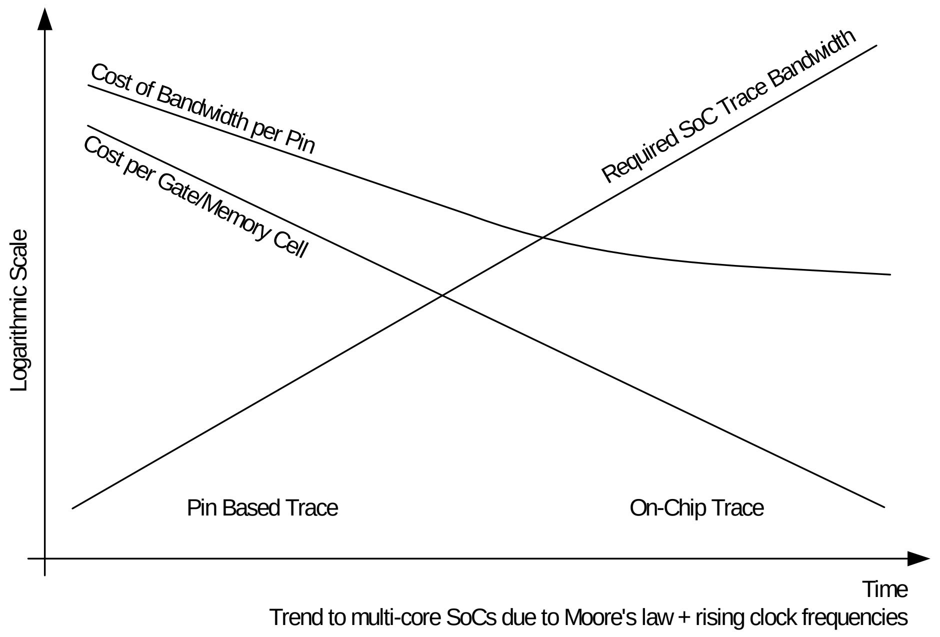

Figure 4: Trace Performance Trend

One option to gain trace pins is to overlay tracing on to existing pins. However, this has two drawbacks:

- The requirement for such pins is that they are not needed by the system during the debugging phase. It is usually difficult, if not impossible, to identify such pins.
- High trace data frequencies require strong output drivers. These drivers will cause high leakage currents and bad electromagnetic compatibility (EMC) behavior even when they are not being used for tracing, since they are also used for application functions during normal operation.

ON-CHIP TRACE

The following sections describe the reasons for on-chip trace, the limitations of on-chip trace, and how those limitations can be overcome.

Debugging

When used for debugging, the trace data will eventually be analyzed by a human being. Since nobody can step through millions of cycles of data at a computer screen, the debug tool must offer search functions for the trace. These search functions provide conditions, of various levels of complexity, to search for within the recorded trace.

The idea of on-chip trace is to apply these conditions to restrict the trace recording to this very small fraction of relevant trace data. The search condition becomes the trigger condition to stop the trace recording. There are even tools on the market that promote the feature that the trigger and trace qualification graphical user interface (GUI) is the same GUI used for trace search.

With this approach, the size of the trace memory can be kept so small that it is acceptable to have it on-chip. It can be also shared with user RAM, which is only used for higher-level applications, if a bottom-up system verification and debugging approach is used as previously described. With on-chip trace memory, the amount of trace data that needs to be uploaded is quite small, so there is no delay and the user can work interactively.

Profiling and Performance Optimization

In addition to debugging, trace is also used for profiling and performance optimization. Profiling is the analysis part of performance optimization—finding the sections of code where the application spends most of the time. These sections of code (for example, functions) can then be optimized on a higher level, which is sufficient in most cases. However, for certain critical algorithms, reducing the execution time of the core loops by just a few cycles can have a significant effect. For this optimization, a cycle-accurate trace is needed to enable analyzing the effects when this core loop is optimized on the assembler level (for example, by reordering the instructions).

Developers of automotive applications are using MCDS to tune powertrain control software—from the passenger compartment of the car while the engine is running.

Profiling is a statistical analysis of the system behavior. It is mainly applied for the program flow, but other aspects such as data accesses can also be profiled. In essence, profiling should answer two questions:

1. Where is the largest lever for performance optimization?
2. How can the performance in this part be optimized?

With statistical sampling, the required data rate can be reduced to a level that allows avoiding a high-speed trace interface. The sampling rate just needs to be kept low enough that the bandwidth of the tool interface is sufficient for continuously reading out the trace data. For individually analyzing the performance-relevant functions (for example, with cycle accurate trace), the on-chip trace memory is sufficient.

The reasons for sub-optimal performance can be such things as cache misses, pre-fetch buffer misses, and/or bus contention, which are not directly visible even from a cycle-accurate IP trace. In some cases, they can be guessed; but this is a cumbersome approach. The better option is direct recording of such events, aligned with data and IP trace.

Devices with On-Chip Trace

Based on these considerations and after intensive discussions with key customers, Infineon decided for an on-chip trace solution several years ago. Currently, there are two Emulation Devices (EDs) available in the market: TC1766ED and TC1796ED, which cover a whole family of high-end automotive powertrain microcontrollers.

Figure 5 shows the TC1766ED on the unchanged evaluation board of the TC1766. The automotive-specific requirement of a large overlay RAM for calibration drove the decision to build dedicated EDs. A typical calibration use case is to optimize the engine-control parameter tables, residing in flash memory, from the passenger compartment. The TC1766ED features 256 KB of emulation RAM whose 64-KB tiles can be assigned to flash overlay or trace buffering. The TC1766ED is created from the unchanged hard macro of the TC1766 SoC, extended on the side with the overlay/trace RAM and the trigger, trace qualification, and trace compression module known as MCDS (Multi-Core Debug Solution).

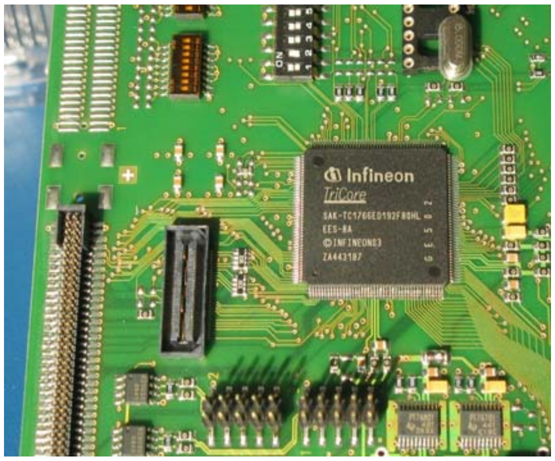

Figure 5: TC1766ED Emulation Device with MCDS

On-Chip Trace Architecture

In the MCDS-based on-chip trace architecture, the complete emulator from trace message generation down to trace storage is located within the device package (Figure 6). At the heart of this emulator logic is Infineon's on-chip MCDS, which features trace, triggers, and performance monitoring. The architecture is independent of the physical interface between the chip and the debug host; no additional hardware is required except for the host PC. Instead of an additional emulator box, a software layer running on the host computer can be used to configure MCDS and to read the stored trace data.

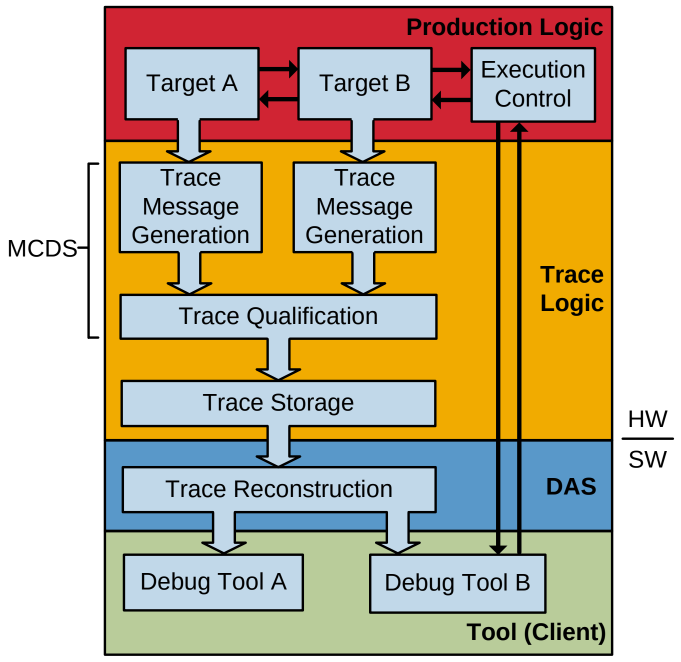

Figure 6: Data Flow through an MCDS-Based Debug System

Infineon developed and uses such a layer, called DAS (Device Access Server, see www.infineon.com/DAS). With this concept, any existing chip interface can be shared to get the trace data off the chip; so no additional pins are required. A popular solution is to use the standard JTAG pins as debug interface. Especially for debugging or calibration measurement in the field (for example, engine control in a car while driving), a narrow tool interface allowing a long thin cable is much more convenient than the bulky cables, headers, or adaptors that are required for pin-based trace architectures.

MULTI-CORE DEBUG SOLUTION (MCDS)

Architecture

Figure 7 shows the functional blocks of MCDS in an example system for two debug targets (Cores A and B). Depending on the target type (processor, bus, or set of signals), the relevant traced information can be different. For instance, the instruction pointer can obviously only be traced for a processor target. For other target types, information such as data and status signals can be of interest.

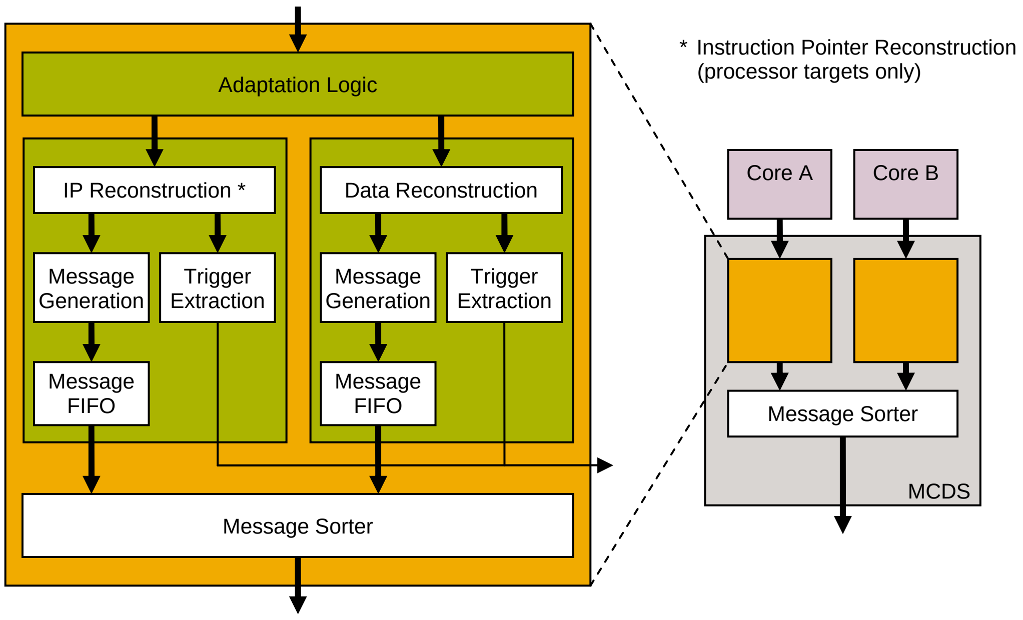

Figure 7: MCDS Functional Blocks in Two-Target Example

Figure 8 shows an example MCDS subsystem with the JTAG option for access to MCDS and the emulation memory (EMEM). For a different hardware interface (such as USB or CAN), the JTAG specific blocks in Figure 8 (JTAG controller, JTAG client, and bus master) have to be replaced by the corresponding interface logic.

The JTAG controller in the system as shown in Figure 8 could be shared with other JTAG clients. Appropriate JTAG instructions are required to ensure the correct mode of operation.

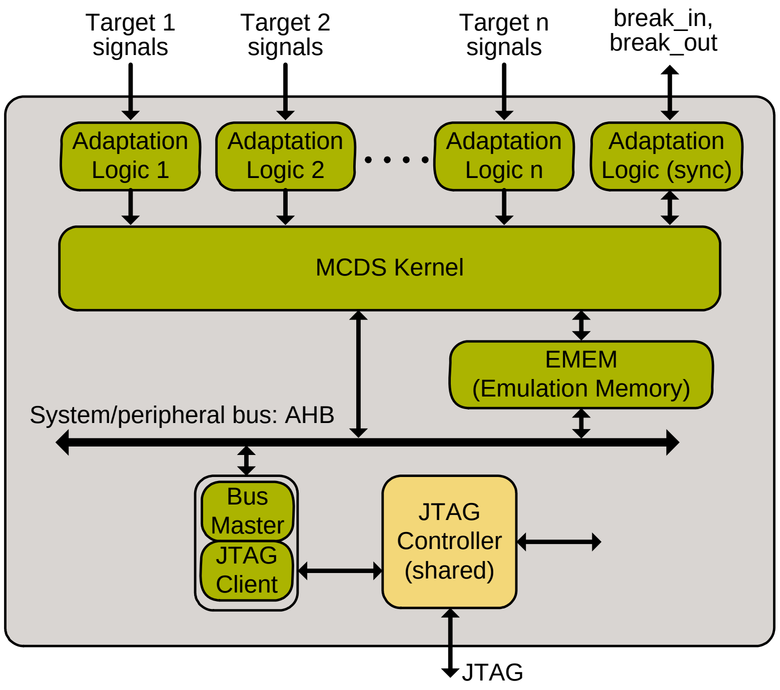

Figure 8: MCDS Sub-System with JTAG Access

MCDS supports debug targets of different types:

- Processor
- Bus
- Set of signals

For the different debug target types, different information is available for tracing. For example, a bus type of target usually does not provide any instruction pointer information. Table 1 lists the different target types and the corresponding relevant trace information.

Table 1: Trace Information from Different Target Types

Traced Information	Processor	Bus	Set of Signals
Process ID	X		
Instruction pointer (IP)	X		
Data	X	X	
Status	X	X	X
Performance data	X	X	
Watchpoint	X	X	

Each debug target is connected to MCDS through an Adaptation Logic block. The design of such a block is specific to the debug target. The Adaptation Logic block connects the target's custom interface (for example, instruction pointer, data, address, etc.) to a generic standardized interface that is used by MCDS. In addition, the Adaptation Logic block synchronizes the signals from the clock domain at the target side to the MCDS clock domain. Of course, this feature only applies if the emulation clock (the MCDS clock) and the target clock belong to different clock domains.

The trace messages that are generated in the MCDS kernel are stored in the EMEM. The EMEM contains memory cells (RAM) and the control logic that is required to ensure that MCDS (write) and the debug tool (read) can both access the trace memory without conflicts.

The architecture of MCDS depends on the number and type of debug targets. Figure 9 shows an overview of the blocks in the MCDS kernel. Each target is connected to a dedicated Observation Block (OB), in which trace qualification and trace message generation take place. Several Trace Units of different types are contained in an Observation Block. The number and type of these Trace Units depend on the debug target to which the Observation Block is connected.

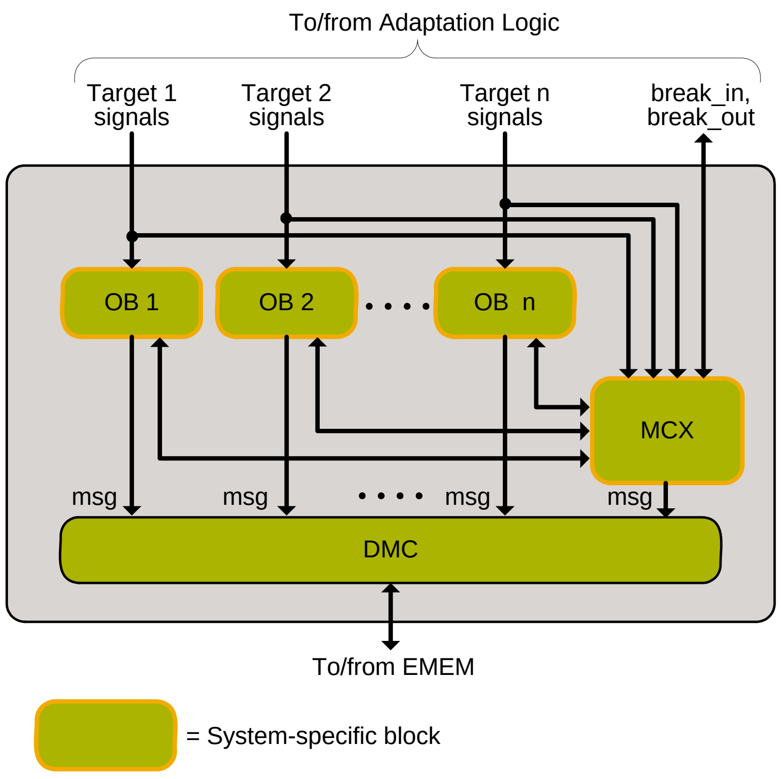

Figure 9: MCDS Kernel Overview

Table 2 gives an overview of which type of Trace Unit is usually used for a specific kind of trace information. The combination of Table 1 and Table 2 results in an appropriate set of Trace Units for each type of debug target.

Table 2: Trace Information by Trace Unit Type

Traced Information	Trace Unit Type
Process ID	Ownership Trace Unit (OTU)
Instruction pointer (IP)	Program Trace Unit (PTU)
Data	Data Trace Unit (DTU)
Status	Debug Status and Control Trace Unit (DCUU)
Performance data	Ownership Trace Unit (OTU)
Watchpoint	Watchpoint Trace Unit (WTU)

The Multi-Core Cross Connect (MCX) block provides the functionality for the generation of cross-target triggers. This kind of trigger is generated from several conditions in different Observation Blocks. The MCX also contains a component for timestamp generation and a Watchpoint Trace Unit (WTU). The architecture of the Observation Blocks and the MCX is specific to the target system.

The Debug Memory Controller (DMC) is a generic block that is responsible for writing the trace messages to the EMEM in the correct order with respect to the point in time of their creation.

Features

Some of the key features of MCDS are:

- Non-intrusive debugging
- Cycle-accurate correlation and view of traced debug events with internal debug target cores (Time Correlation)
- Complex trigger system including cross-target triggers and state machines
- Core runs at full speed
- Access to internal buses
- Code profiling through performance counters
- Implementation split is optimized for easy adaptation to different cores
- Covers gap between raw trace hooks and abstract messages
- Compression of trace messages to save memory
- Can be placed outside of production die
- Independent of physical interface between chip and debug host

Some of these features are described in more detail in the following sections.

Time Correlation

The MCDS is able to trace multiple cores in parallel and in real-time while they are running at full operating speed. Timestamps are used to sort the trace messages in the trace memory (EMEM), so trace reconstruction with a scalable granularity down to the emulation clock cycle level is possible. Trace messages from different debug targets have an exact time correlation.

Complex Triggers

Highly configurable, scalable, complex trigger generation is used for trace qualification and for run control of the processor debug targets. Triggers can be generated internally in the MCDS block. As Figure 10 shows, triggers from the debugged cores can be fed into MCDS where they can be combined with internal or external triggers from other cores (cross-target). A configurable break and suspend switch (outside of MCDS) allows control of multiple processors as a group by user-defined combinations of triggers.

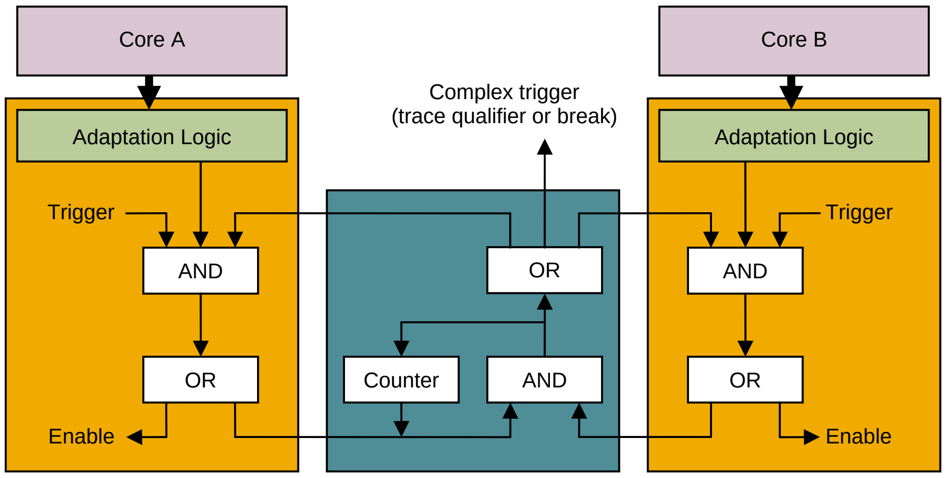

Figure 10: MCDS Trigger Control Cross-Connect for Two-Target Example

Figure 11 shows the basic principle of the trace qualification and what can be programmed by the debug tool. The AND/OR matrix scheme applies to all trace qualification blocks contained in the OBs and in the MCX. The trigger input values can be evaluated directly or negated. Also, edge or level sensitivity can be selected. Triggers are generated in multiple different sources:

- External core triggers directly driven by the debug targets
- Triggers generated in Trace Units
    - Programmable (bound, range, equality)
    - Example: Instruction pointer is in range [0x0000..0x00f0]
- Cross triggers
    - Trigger input is through the MCX (from another OB)
- Count triggers
    - Programmable (limit)
    - Only available in MCX
    - Example: (trigger_k && trigger_l) occurred n times

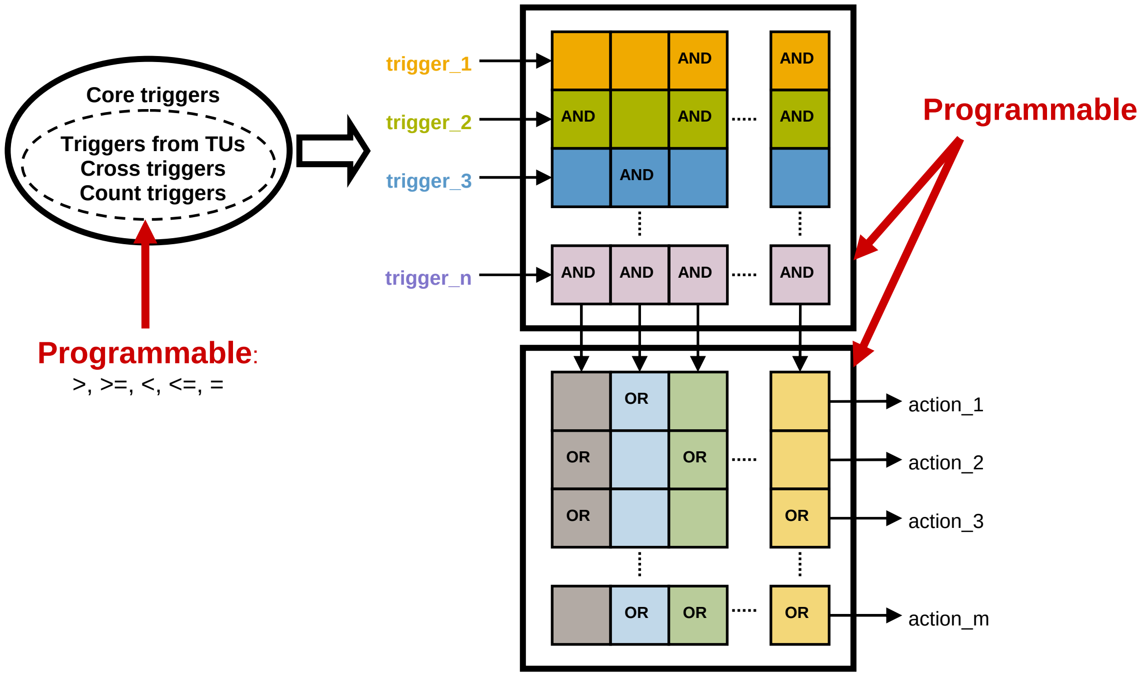

Figure 11: Trace Qualification Programming

The trace qualification logic allows for feeding actions back to the trigger inputs. So, with appropriate programming (for example, cascading counters), it is possible to generate very complex state machine based actions.

Due to the structure of the AND/OR matrix implementation, the required amount of silicon area would be high if all possible trigger combinations for action generation were programmable. To reduce the required gate count, a well thought out set of possible combinations is elaborated for a specific target system before MCDS is implemented.

In addition to the standard counters, so-called performance counters can be configured to count special performance signals that are driven by a dedicated debug target. Depending on the debugged core, such performance information can be used to measure characteristics such as:

- Cache hits/misses
- Number of executed instructions
- Number of stall cycles
- Number of interrupts acknowledged
- Number of bus transactions done

When a certain counter limit is reached, a message can be generated that contains the corresponding performance value. This information relates to a programmable temporal resolution. This methodology allows accurate code profiling and optimization.

Demands on the Debug Tool

The complex trace qualification features of MCDS make high demands on the debug tool. For example, approximately 500 32-bit MCDS registers are programmable in Infineon’s TriCore based TC1766/96ED (4 debug targets: 2x CPU, 2x bus; 2 of these targets are 64-bit). The effort for a software developer to program the configuration registers directly would be much too high. Therefore, new concepts for utilizing the full power of the MCDS debug logic are indispensable. Appropriate concepts have already been introduced and are being used by software tools [1].

IP-BASED MCDS

Overview

Infineon partners with IPextreme to make MCDS technology available to the industry as IP in order to create an attractive market for associated tooling, which benefits tool vendors (market size) and users (tool cost and quality). The biggest challenge in offering MCDS is the fact that it must fit all different kinds of SoCs, comprising multiple cores from different vendors. So the question is: what is the best approach to offer an appropriate MCDS architecture that fulfills this requirement?

To answer that question, two possible approaches for adding a debug solution like MCDS to a given target system shall be investigated.

The first approach is to develop debug logic that is completely customized for the given target system. All components are designed for specific target core requirements and features. Of course, this solution offers exactly the debug capabilities that have been defined as requirements. No logic overhead is implemented, which leads to optimal silicon area and clock speed values. On the other hand, the design and verification effort is high. Even small changes for derivative SoCs of the same product family can require costly updating of the debug logic. But what impacts development cost and effort even more is the required debug tool support for this proprietary solution. As the market for such a debug solution is relatively small, the tool cost is accordingly high.

The other approach is to add a generic debug architecture like a standard IP block to the SoC. The design and verification effort is comparatively low, which is a typical feature of re-used IP blocks. As the debug solution comprises a standard architecture, it can be expected that a certain market size is covered. The prospect for business opportunities raises the interest of tool development companies, which leads to competition. As a consequence the required debug tools feature high quality and low price. Opposed to these advantages, the generic debug architecture has always a clearly defined and limited set of debug features. Usually this set is either too narrow or too powerful compared to the debug features required for a certain target system. In the first case, the architecture is out of question because it does not fulfill the basic debug requirements. In the second case, it has to be accepted that the superfluous debug features mean logic overhead, which may lead to higher gate count and lower clock frequency.

Comparing these approaches and trying to create a solution that incorporates the advantages of both while at the same time narrowing down the drawbacks to a reasonable level led to development of the MCDS IP-based architecture. The term “IP-based” indicates that the solution is not a standard IP core that can be configured simply by adjusting some top-level hardware parameters. In fact, the IP-based solution is customized to the target system by (manually) creating the HDL descriptions for the upper levels. At subblock level, re-used, verified IP blocks are instantiated. The hardware parameters of these IP components are configured according to the target system’s specific requirements.

Architecture

Figure 12 shows the architecture of the MCDS kernel. MCDS is programmed through an AMBA AHB interface [2]. This bus standard was chosen because it is widely used in the industry and it is an open standard. AMBA APB cannot be used because it does not support the required wait-state mechanism for bus transfers. The Bus i/f AHB block translates the AHB accesses for the Internal Bus Controller (IBC), which controls the internal SFR (Special Function Register) bus. This custom bus is required because the large number of SFRs makes specific demands on the bus architecture in order to achieve the required timing. Each OB and the MCX contain programmable register banks. The IBC decodes addresses to select the appropriate register bank.

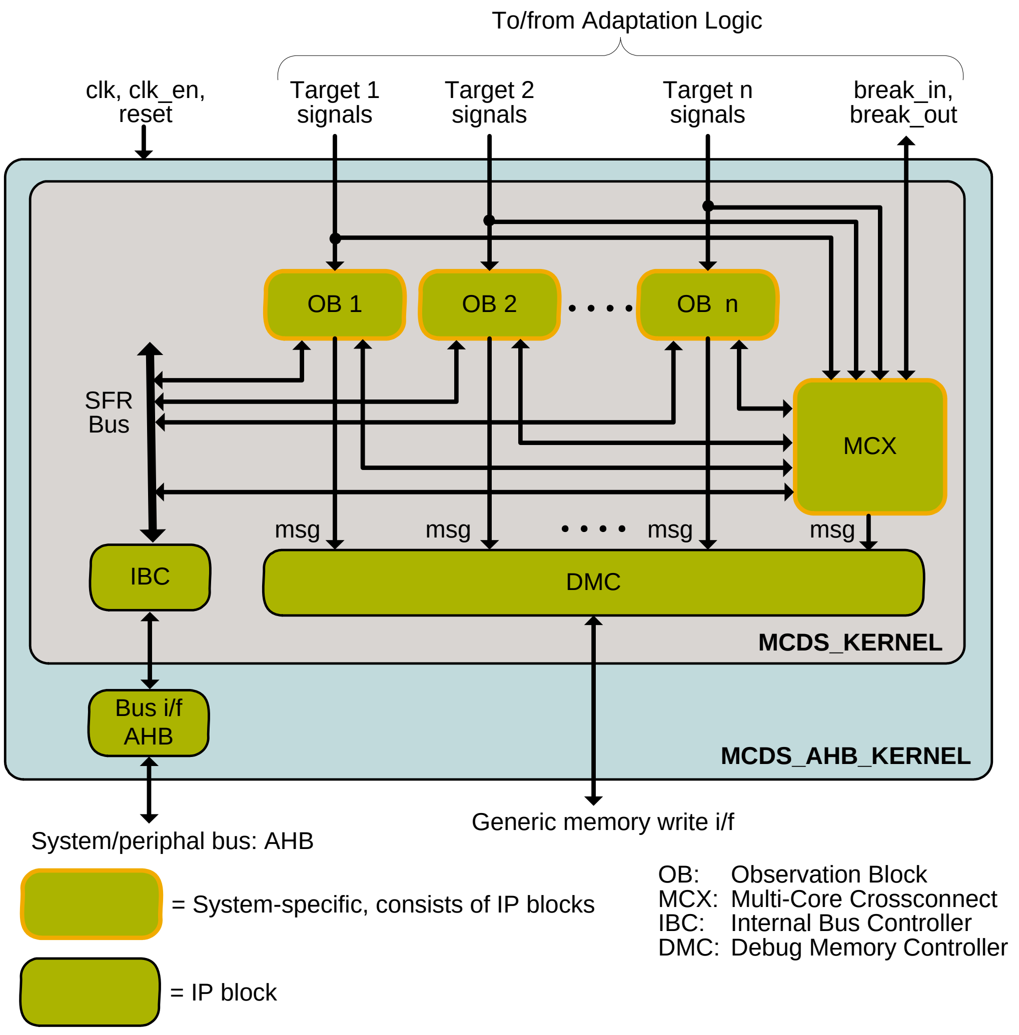

Figure 12: MCDS Kernel

The IBC, the DMC, and the Bus i/f AHB are IP blocks that are configured through their hardware parameters. The OBs and the MCX are containers for target system specific sets of sub-level IP blocks that have to be configured and connected. Figure 13 shows the scheme of composing an Observation Block. Depending on the trace requirements (see Table 1 and Table 2), a certain set of Trace Unit IP blocks is instantiated. Each Trace Unit converts the information that is forwarded by the Adaptation Logic into a specified trace message format. The Trace Qualification Unit (TQU) is programmed by the debug tool and embodies a filter mechanism that controls which messages are passed on to the Message Sequencer Unit (MSU). Here the messages from the different Trace Units are sorted regarding their time tags, then passed on to the DMC.

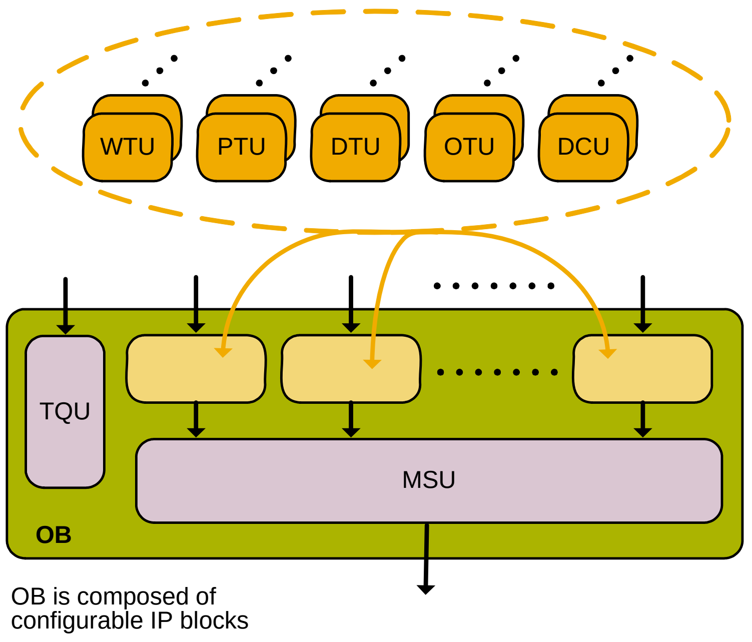

Figure 13: Observation Block Composition

The architecture of the MCX is similar to the OB architecture. The MCX usually contains only one Trace Unit of type Watchpoint Trace Unit. A Time Stamp Unit (TSU) delivers time information to all other parts of MCDS for internal use and for generated messages. The TQU in the MCX uses event counters to generate count triggers. In addition, it contains the performance counters used to generate performance information for a debug target. If the SoC contains a break switch, break signals (break_in, break_out) from and to this break switch are connected to the TQU in the MCX.

Adaptation to the Target System

As mentioned before, some “manual” implementation work is required to adapt the MCDS IP-based solution to a specific target system. First of all, the MCDS kernel has to be created. Sub-levels IP blocks need to be configured according to the target system requirements. In particular, the configuration of the TQUs in the OBs and the MCX requires detailed knowledge about the SoC and the desired debug features. The setup of the AND/OR matrices as depicted in Figure 11 is realized in the TQUs. To avoid logic overhead, only well thought out possible combinations of triggers should be realized. If the SoC to be debugged contains dedicated run control logic for the system (such as a break switch), appropriate signals (break_in, break_out in Figure 12) can be connected to/from MCDS. The MCDS output signal used to control such external on-chip debug logic is generated in the TQU of the MCX, just like all the other actions (see Figure 11). The incoming break signal is handled by the TQU in MCX like the other incoming triggers from the OBs.

On OB and MCX level, the configured IP blocks are then instantiated and connected. On the kernel level, again only hardware parameter configuration of IP blocks (IBC, DMC – see Figure 12) and interconnection of the instantiated blocks is required.

For the complete MCDS subsystem, additional components have to be created. The specific interface of each debugged core must be adapted to the interface standard supported by MCDS. The corresponding logic is contained in the Adaptation Logic blocks (see Figure 8). As their architecture is very target specific, these blocks are custom designs. If the clock domains of the debugged core and MCDS are not synchronous, the Adaptation Logic is responsible for synchronizing the observed information from the target clock domain into the MCDS clock domain. Depending on the clock ratios, information buffering may be required. The emulation memory block (EMEM) is also a custom design. In some SoCs, available system memory could be shared and used as trace memory. In other systems, dedicated RAM cells are used. In any case, appropriate control logic has to be implemented to avoid memory access conflicts between MCDS and the debug tool. Furthermore, custom MCDS and EMEM access logic (for example, JTAG client and AHB bus master) is required. For the popular JTAG option, reusable IP is available to implement the required hardware.

Impact on the Debug Tool

As previously mentioned, new concepts in debug tool development are required to leverage the full power of MCDS. The IP-based solution with its customized architecture places even higher demands on the debug software. The programming of the trace qualification is SoC (device) specific; therefore, tool providers need to apply intelligent methods to ensure reusability of their software across different SoCs. Easy adaptation of the existing software for a new target system is a key requirement for reasonable development costs and success on the market. A modular and scalable software architecture is necessary to support the hardware configuration space of the MCDS architecture.

In particular, the trace qualification capabilities of each specific MCDS implementation must be realized in the software. Typical SoC-specific trace qualification features include:

- Number of triggers
- Meaning of trigger
- Setup of the AND/OR matrices
  - Number of columns/rows
  - Position of triggers
  - Possible combinations

In the ideal case, the debug tool software has a generic architecture that can be adapted to a specific MCDS implementation by using a configuration file that contains the required specific hardware information.

CONCLUSION

Traditional run control debugging (start, stop, step, breakpoints) is not adequate for complex parallel real-time software running on a multi-core SoC. Only with non-intrusive trace support is it possible to observe and analyze such a system within its target environment. The challenge is the required trace bandwidth for cores and on-chip buses, which already has an order of magnitude of about 100 Gbit/s and it is still growing at the same pace as silicon integration technology (from a few cores to many cores).

On-chip trace buffering is the only long term sustainable solution, which was the reason that Infineon decided on this approach several years ago for the high-end automotive powertrain microcontroller line. The MCDS IP features trace, triggers, and performance monitoring. Devices with MCDS and associated tooling are available in the market and are being used with very positive feedback from automotive customers.

Infineon decided to partner with IPextreme to make the MCDS technology available to the industry to create an attractive market for associated tooling, which benefits tool vendors (market size) and users (tool cost and quality). The MCDS IP-based architecture allows tailoring an optimum debug solution for a specific multi-core SoC with minimum effort and minimum risk.

REFERENCES

1. Irrgang K., Weisse S., Groeger T., “A New Concept for Efficient Use of Complex On-Chip Debug Solutions in SoC Based Systems”

2. AMBA Specification, Rev 2.0, ARM, www.arm.com

IPextreme

www.ip-extreme.com

IPextreme, Inc.
307 Orchard City Drive
M/S 202
Campbell, CA 95008
800-289-6412 (toll-free)
408-608-0421 (fax)