

Generating the Trace Qualification Configuration for MCDS from a High Level Language

Jens Braunes
pls Development Tools
Technologiepark
02991 Lauta, Germany
Jens.Braunes@pls-mc.com

Rainer G. Spallek
Technische Universität Dresden
01062 Dresden, Germany
rainer.spallek@tu-dresden.de

Abstract—This paper introduces a high level trace qualification language and compiler which enables the user defining analysis tasks efficiently and fully utilize the powerful features of Infineon’s Multi-Core Debug Solution (MCDS) without the need of getting into the internals. The language and the compiler are already in industrial use where software development is based on MCDS enabled SoCs to support the developers to achieve better product quality and shorter product development cycles.

I. INTRODUCTION

In recent years the system complexity of modern embedded platforms was exponentially increasing. Single core solutions were replaced by Systems-on-Chips (SoC) integrating multiple heterogeneous processor cores and several different busses. Driven by the higher achieved performance, the size and complexity of embedded software projects have grown too. Errors which are naturally unavoidable are harder detectable or even undetectable using traditional debug methods. Due to the high degree of integration the visibility of internal chip operations, essential for debugging purposes, had been lost. The way out was to integrate dedicated debug support directly on-chip. The high degree of core interaction on multi-core systems and additionally the demand to meet hard real-time constraints require extensive trace capabilities. Only this way non-intrusive system observability is feasible.

Driven by requirements of automotive applications Infineon developed the Multi-Core Debug Solution (MCDS), a generic on chip trace and debug hardware IP block (intellectual property) for SoCs with multiple processor cores and busses. Non-intrusive observation of system states and signals combined with a sophisticated programmable trace qualification enables it to detect even errors which are caused by intensive process interactions and non-deterministic system behavior.

For effective industrial use of MCDS we developed a configuration tool which is based on a high level configuration language and a compiler to program the complex trace qualification mechanisms of MCDS. Instead of managing the trace and debug capabilities manually the user is now able to define even complex analysis tasks in an efficient way without any knowledge about the MCDS architecture itself.

This paper is organized as follows. Before in section III the developed trace qualification language will be introduced and important aspects of the compilation process will be discussed,

a brief overview of the MCDS architecture should be given in the next section for better understanding. Based on an example and the gained results, in section IV the advantages of the language concept for MCDS configuration should be substantiated. Finally, section V concludes the paper.

II. MULTI-CORE DEBUG SOLUTION (MCDS)

A. Overview

MCDS is a configurable and scalable trigger, trace qualification, and trace compression IP block developed by Infineon Technologies AG [1,2]. It allows simultaneous recording of multiple trace sources into one single, time aligned trace stream. That makes MCDS suitable as debug environment in multi-core and multi-bus systems. The generic concept of MCDS allows reusing and integration it into SoCs with heterogeneous processor cores and busses.

Figure 1 shows the MCDS core architecture with sub-components and the integration into an existing SoC with multiple trace and debug targets (e.g. CPUs, busses). The MCDS kernel consists of a number of observation blocks (OB), which are connected via an adaptation logic (AL) to the debug targets, the multi-core cross connect (MCX) with cross trigger connections to all OBs, and a data management controller (DMC). The latter collects all trace messages from

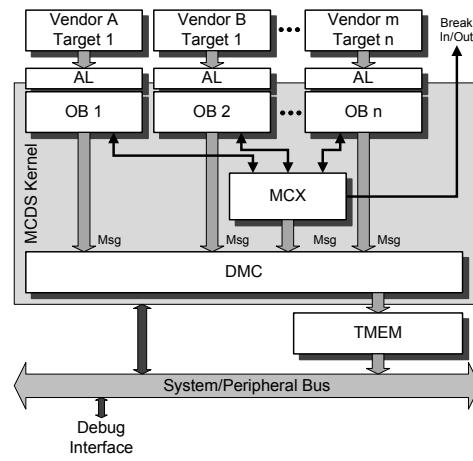


Fig. 1. MCDS architecture

the trace data sources (OBs and MCX) and writes them into one single trace while ensuring the correct temporal order of all trace messages. In contrast to other trace solutions, like NEXUS [3], where the trace data is getting off-chip during the capture using a costly high bandwidth trace port, MCDS implements an on-chip trace memory (TMEM) for trace data storage to overcome the trace port bottleneck [4]. To cope with the low capacity of on-chip memory – typically it is in the range of Kbytes – MCDS provides a sophisticated trigger and trace qualification logic, which will be introduced later, to control the trace recording.

B. Observation Blocks (OB)

Each debug target is connected via an adaptation logic to a standardized interface of the assigned OB. Depending on the target, various types of trace information can be derived (e.g. instruction pointer, data, process ID, counters, etc.). For each type a dedicated trace unit has to be implemented within the OB. The trace units reconstruct the target information and signals and generate appropriate trace messages send to the DMC if enabled by the trace qualification logic of the OB.

C. Multi-Core Cross Connect (MCX)

The MCX is the centralized cross trigger block, which is responsible for distributing cross triggers from one OB to all others (fig. 2). The routing of cross triggers is programmable using the trace qualification and trigger logic described in the next subsection. In addition to that, MCX provides central time stamps. All trace messages written to the trace memory are prefixed by them to make sure that all trace data are time aligned with the same time base and before-after relations of events, coming from different sources, can be reconstructed. In addition time stamps written to the trace stream can be used for time measurements.

A number of counters inside MCX allow to count events and rise an own event when a programmed counter value is exceeded. Based on that, state machines can be built to enhance the trigger generation and trace control.

D. Trace Qualification and Trigger Logic

To control the trace capture in order to record only relevant data and hence save trace memory, each OB and the MCX contains a complex trigger logic for trace qualification [5]. The trigger logic is build up from an event logic, an action logic and optionally a counter control logic as shown in figure 3.

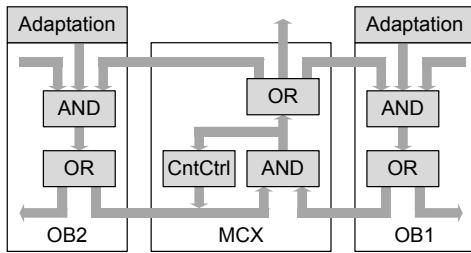


Fig. 2. Cross triggering

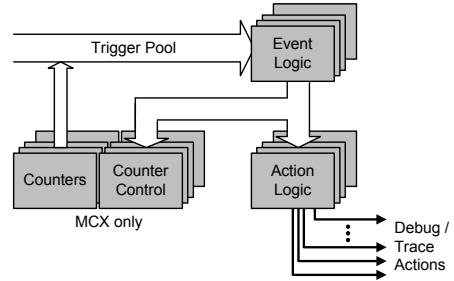


Fig. 3. Trigger logic for trace qualification (simplified)

The latter and the counters themselves are implemented only in MCX.

The event logic is realized as a matrix of l AND blocks with m inputs each. Each AND can combine a fixed subset of n ($m < n$) trigger signals from a trigger pool. Every input trigger can be disabled, when not needed, or its negated value can be applied. The trigger pool contains all trigger signals which are available for the particular OB or the MCX:

Core triggers: Signals directly driven be the debug targets.

Trace unit triggers: Programmable triggers on conditions. E.g. comparators and range triggers for addresses and data.

Cross triggers: Distribute events between MCX and all OBs and vice versa.

Counter triggers: Counters can be programmed to emit a trigger signal when a certain counter value is exceeded. Counter triggers are internal triggers of MCX. Cross triggering has to be used to distribute them to OBs.

The outputs of the event logic are connected to the action logic and – in MCX – to the counter control logic. The action logic combines the outputs of the event logic using programmable OR blocks. Each OR block is assigned to a trace or debug action which is enabled or disabled accordingly.

Each OR can be programmed to combine 0 to k of l ($k < l$) outputs from the event logic. This way a Boolean function of the form $y = (x_0 \wedge x_1 \dots) \vee (x_j \wedge x_k \dots) \vee \dots$ can be realized, where $x_1, x_2, x_k, x_l, \dots$ are triggers from the trigger pool and y represents a condition under which a trace action (e.g. start/stop trace, write value to trace memory, etc.) will be initiated. The counter controls allow to increment and reset the counters depending on each of one output of the event logic. Additionally edge detection can be enabled for each selected input of OR blocks as well as for the counter controls.

The OR, AND, and counter control blocks are programmable by a set of memory mapped registers which are accessible via the debug interface and the system bus.

III. COMPILE CONFIGURATION DATA FOR MCDS TRACE QUALIFICATION

A. High-Level Configuration Language for Trace Qualification

The powerful trace qualification and trigger logic of MCDS can be hardly programmed manually in particular when cross triggering is involved. To make programming more feasible

for every day's engineering work, a high level language for trace qualification and a compiler for it were developed. The compiler translates a described analysis task directly into the register settings controlling the MCDS in the end.

The used language is quite similar to high level programming languages and contains statements to initiate debug and trace actions (e.g. enabling trace recording, generating core events, break the targets) depending on conditions (IF-THEN-ELSE) as well as statements in order to realize state machines often required when an analysis task becomes more complex. Core triggers, triggers from trace units and counters are represented as structures which can be assigned to signals and initialized with data values or expressions. For examples see figure 5(a). These signals are used later in expressions to form conditions for trace actions and state transitions.

B. Transforming State Machines into Combinatorial Logic

MCDS does not support state machines directly. For this reason counters in MCX are used to represent every state. In order to map state machines, described using the trace qualification language, for all actions the corresponding combinatorial terms have to be created based on state counters and conditions. All conditions for action statements which will be finally mapped to the same MCDS action are combined to one single combinatorial term. Depending on the origin, either from IF part or from ELSE part, single conditions are used unchanged or negated. The result is a combinatorial term which has to be transformed into a canonical form and afterwards simplified in order to save MCDS' event resources.

Additional transformations might be needed if edge detection should be applied to one or more inputs of the combinatorial. The trace qualification language allows edge detection (`rise(event)` / `fall(event)`) for each input of both, disjunctions and conjunctions. However in MCDS edge detection is only implemented for action logic inputs (OR) and counter controls. Edge detection applied to inputs of the event logic needs to be transformed. Conjunctive subterms of the form

$$x_1 \wedge \text{rise}(x_2) \wedge x_3 \dots x_n$$

where edge detection should be applied to one or more inputs can be transformed to

$$\text{rise}(x_1 \wedge x_2 \wedge x_3 \dots x_n)$$

The function $y = \text{rise}(x_1 \wedge x_2 \wedge \dots \wedge x_n)$ is defined as follows:

y becomes true (high) if at least one input x_i has a transition from low to high meanwhile all other inputs are already high.

For $y = \text{fall}(x_1 \wedge x_2 \wedge \dots \wedge x_n)$ the definition is analogue. A mix off rise and fall within one conjunction is not allowed.

C. Cross Trigger Routing

After transformation and simplification the canonical form may contain a mix of events from different OBs and MCX which cannot be mapped directly to the MCDS, so cross triggers, routed through MCX, have to be used.

For that purpose the conditions are split into several sub terms whose events are all generated by the same OB or by

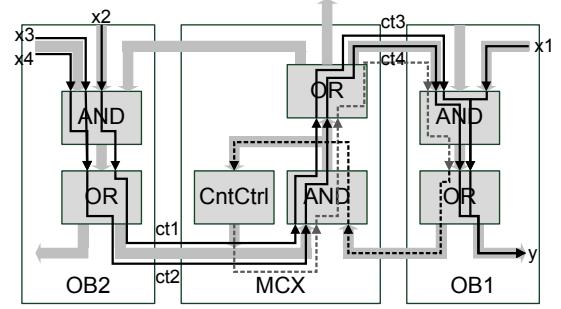


Fig. 4. Cross trigger routing

MCX. Originating from the target OB – implements the actual trace action – a route is searched through MCX to the event source OB. The sub terms are substituted by cross triggers ct_i as shown in the following example:

The condition

$$y_{OB1} = (x_{1_{OB1}} \wedge x_{2_{OB2}}) \vee (x_{3_{OB2}} \wedge x_{4_{OB2}})$$

under which an action y_{OB1} will be initiated is splitted into

$$ct_{1_{OB2} \rightarrow MCX} = x_{2_{OB2}}$$

$$ct_{2_{OB2} \rightarrow MCX} = (x_{3_{OB2}} \wedge x_{4_{OB2}})$$

which can be mapped to OB2 and use two different cross trigger lines $ct_{i_{OB2} \rightarrow MCX}$ from OB2 to MCX, and

$$ct_{3_{MCX} \rightarrow OB1} = ct_{1_{OB2} \rightarrow MCX}$$

$$ct_{4_{MCX} \rightarrow OB1} = ct_{2_{OB2} \rightarrow MCX}$$

which can be mapped to MCX and use cross trigger lines $ct_{j_{MCX} \rightarrow OB1}$ from MCX to OB1. The condition for y_{OB1} has to be finally substituted by

$$y_{OB1} = (x_{1_{OB1}} \wedge ct_{3_{MCX} \rightarrow OB1}) \vee ct_{4_{MCX} \rightarrow OB1}$$

The resulting cross trigger routing is illustrated in figure 4 (solid lines).

As you can see from figure 3 the inputs for counter controls do not follow the canonical form. They are directly connected to the event logic (AND) and do not provide OR-functionality for the inputs. For that reason the trace qualification logic of another OB has to be used to realize a full featured canonical form (fig 4, dotted lines).

D. Resource Management

After all cross trigger connections are routed the combinatorial resources of each OB and MCX are allocated.

The fixed assignment of input events to the AND blocks of the event logic limits the degrees of freedom when combining events. In order to save combinatorial resources the allocation has to be done carefully. For this reason the resource manager of the compiler examines at first which events has to be combined by AND and searches for that AND block which has the best utilization and allocates it finally. OR blocks of the action logic are implicit allocated due to their fixed assignment to trace and debug actions.

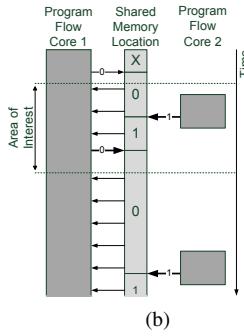
To save resources cross trigger routes are reused as much as possible. Explorations of typical analysis tasks have shown that

```

// Declarations
TPC.PCcore1 pc1,
TPC.PCcore2 pc2;
TPC.BusAddr addr;
TData.BusData data;
TData.BusAccess access;
// Definitions
TSignal Enter = pc1 == 0xD400049A;
TSignal Leave = pc2 == 0xD40005BC;
SharedMem = addr == 0xF0050040;
// Write to bus from any busmaster
MemWrite = 0x200 <= (access & 0x20e) <= 0x20F;
// State machine
state0:
  if (Enter) then
    goto state0;
state1:
  store pc1;
  store pc2;
  emit mcx.tick_enable;
  if (MemWrite and
      SharedMem) then
    trigger;
    store addr;
    store data;
  if (Leave) then
    goto state0;

```

(a)



(b)

Fig. 5. Analysis task to perform the multi-core memory write observation. (a) Task described using trace qualification language. (b) Illustration of the program behavior.

several debug actions depend on a small number of complex conditions. Thus allocated event/action resources of one OB can be reused frequently for different actions as well as for cross triggering.

IV. RESULTS AND CASE STUDY

The developed trace qualification language and compiler is already in industrial use. It enables the developers to fully utilize the capabilities of MCDS without extensive knowledge about its internals. The developers need neither care about the assignment of debug properties (e.g. instruction pointers, bus accesses, etc.) to OBs, nor need they deal with the complexity of cross trigger routing anymore.

Describing an analysis task and the compilation results should be illustrated by one representative example:

Two tasks, each executed by a separate processor core, are communicating via a shared memory locations. This communication should be observed.

Figure 5 shows the task described using the trace qualification language and the interesting interaction schematically. A state machine with two states is used to trace only the area of interest bounded by the two address comparators `Enter` and `Leave`. `state1` is representing the area and is also responsible for recognition the write accesses to the shared memory location using a bus observer.

Table I shows characteristic numbers of the described analysis task as well as of the compilation results. As seen from it the number of occupied debug blocks (4) and the intense interaction between them requires extensive use of cross trigger resources. Manually routing of cross triggers would be a time-consuming and error prone task and is for industrial use unacceptable.

TABLE I
ANALYSIS OF COMPILER SOURCE AND COMPILE RESULTS

Source file characteristics	
# of debug actions	6
# of states	2
# of state transitions	2
# of input events ¹	4
# of conditions	3
Characteristics of compilation results	
# of occupied debug blocks	4 ²
# of additional input events	2 ³
# of cross triggers ⁴	6
Cross trigger reuse rate ⁵	2,2
# of different action conditions	7
# of MCDS registers to set	46

¹ From trigger pools of all used OBs.² Two processor observation blocks, one bus observation block and MCX.³ Two counters representing the states.⁴ Adds one additional condition each.⁵ Cross triggers used by 13 action conditions.

V. CONCLUSION

Infineon's MCDS provides a high degree of system observability of multi-core SoCs while the run-time behavior is not influenced. With its sophisticated trace qualification logic, cross triggers, and state machines built from internal counters, MCDS is breaking the borders of visibility between the cores and busses of a SoC and the overall system behavior with all interactions getting visible.

To utilize the MCDS hardware software support was developed. Instead of programming MCDS using its configuration registers a high level trace qualification language was introduced which allows describing analysis tasks in an efficient and user friendly way, tailored to the users' need. A compiler was realized which translates the described tasks into the MCDS configuration data and takes care of resource allocation and the effective use of cross triggers.

The MCDS hardware and the high level trace qualification language and compiler have been proven already their abilities in industrial use where efficient debug support is essential for better product quality and shorter product development cycles.

REFERENCES

- [1] A. Mayer, H. Siebert, A. Kolof, and S. el Baradie, "Debug support for complex system-on-chips," in *Proc. Embedded Systems Conference (ESC)*, April 2003, www.techonline.com/learning/techpaper/193103950.
- [2] A. Mayer, H. Siebert, and K. D. McDonald-Maier, "Debug support, calibration and emulation for multiple processor and powertrain control socs," in *Proc. Conference on Design, Automation and Test in Europe (DATE)*, vol. 3, March 2005, pp. 148–152.
- [3] The Nexus 5001 Forum™, "IEEE-ISTO 5001™-2003, The Nexus 5001™Forum standard for a global embedded processor debug interface," www.nexus5001.org.
- [4] J. Braunes and S. Weiße, "Methods for real-time data trace in a system-on-chip debug environment," in *Proc. Embedded World Conference*, February 2007.
- [5] J. Braunes, S. Weisse, C. Lipsky, and J. Schicker, "A new debug methodology for efficient use of resources offered by complex on-chip debug solutions," in *Proc. 4th IEEE International Workshop on Silicon Debug and Diagnosis (SDD)*, May 2007.