Acquiring an Exhaustive, Continuous and Real-Time Trace from SoCs

Christian Hochberger\textsuperscript{1} and Alexander Weiss\textsuperscript{2}

\textsuperscript{1}Department of Computer Science, Dresden University of Technology, Dresden, Germany
\textsuperscript{2}Accemic GmbH & Co. KG, Flintsbach, Germany

christian.hochberger@inf.tu-dresden.de, aweiss@accemic.com

Abstract—The amount of time and resources that have to be spent on debugging of embedded cores continuously increases. Approaches valid 10 years ago can no longer be used due to the variety and complexity of peripheral components of SoC solutions that even might consist of multiple heterogeneous cores. Although there are some initiatives to standardize and leverage the embedded debugging capabilities, current debugging solutions only cover a fraction of the problems present in that area. In this contribution we show a new approach for debugging and tracing SoCs. The new approach, called hidICE (hidden ICE), delivers an exhaustive, continuous and real-time trace with much lower system interference compared to state-of-the-art solutions.

I. INTRODUCTION

Current chip technology allows the integration of many components on a single chip. Thus, embedded processing becomes more and more powerful and also more complex. Unfortunately, the integration of many peripherals, additional cores or bus systems also introduces a number of new problems during design and validation of such circuits.

Debugging the embedded software consumes a growing part of the design effort. Thus, debugging becomes a major part of the overall design cost. Additionally, debugging can play an important role in the validation of a design, so it is also part of the validation effort.

Various standards have to be taken into consideration for safety critical systems (IEC 61508 [1] in general, IEC62278 [2] and IEC 62279 [3] for railways, ISO 26262 [4] for the automotive area or RTCA/DO 178B [5] for the aircraft industry). In many cases real time traces of embedded software are necessary for system certification according to these standards. The level of detail of these traces depends on the criticality of the system. Providing this trace data requires a full trace of the application through all the relevant parts of the code.

Traditionally, the only way to gather this trace data was to use a full featured bond-out chip based in-circuit emulator (ICE). This approach uses specialized chips for the implementation of the processor/microcontroller to grant access to internal resources. Thus, an ICE allows the recording of the full instruction flow. Unfortunately, building an ICE is extremely expensive and time consuming. Also, at high CPU frequencies it is almost impossible to eliminate the physical restrictions of the probe cable.

As a consequence, industry and academia thought about solutions for the debugging problem. Major approaches of the last years have been embedded trace solutions, for instance the Nexus framework [6]. It is the combined effort of many companies in the embedded area which is intended to solve many of the debugging/trace problems.

II. RELATED WORK

Various debugging/tracing solutions for embedded processor cores exist. Typically, the processor designer has to choose a trade-off between the conflicting goals hardware-effort (chip area or pin count), level of detail and completeness/timeliness of acquired data.

Many approaches use serial interface to in internal debugging unit (like background debug mode in Freescale processors or JTAG embedded ICE in ARM processors). Such debug units allow the controlling host to insert breakpoints into the program flow and read/modify register values and memory locations. Breakpoints are either break instructions (e.g. in the BDM) or they are limited to simple address or data patterns combined with access flags (code or data fetch). Values of processor registers cannot be used to qualify breakpoints. An overview of these debugging techniques can be found in [7, 8].

Recently, more applications need to have a deeper understanding of the trace of execution in the embedded system. Thus, several approaches have been investigated to deliver such traces with minimal system interference. ARM embedded trace macrocell (ETM) [9] or Infineon’s Multi-Core Debug System (MCDS) [10] implement interfaces that allow the controlling host to follow the addresses of the executed instructions. Both variants are not able to deliver a continuous and complete real-time trace of executed instructions, data read, data written, DMA operations and CPU register values.

As the debugging of embedded processor cores becomes more and more important, there are also efforts to standardize the debugging interfaces. The most prominent approach here is the Nexus initiative, which tries to define different categories of debugging/trace support and associates standardized interfaces with these categories. Nexus also requires enormous amounts of resources, especially pins in the package to deliver reasonable results.

Recent academic research is mainly focused on reducing the bandwidth of trace data [11]. Although this might lower...
the external bandwidth requirements and eventually even might reduce the pin count, it still only records the addresses of executed instructions.

III. THE NEW TRACE METHODOLOGY

The fundamental idea behind the new methodology for capturing trace data is to equip the microcontroller with a facility that allows the synchronization with an external emulator [12]. Thus, only data is communicated from the SoC to the emulator that is required to reconstruct all internal data and the program flow. All other system responses are defined by the program code. Fig. 1 shows the emulation principle.

Conventional methods transfer all read data and write data as well as all jumps to the emulator. Whereas our proposed methodology only transfers all read data that originates from the peripheral components of the controller and the interrupts, because only this data/these events can change the program flow unpredictably.

Therefore, the emulator must meet these requirements:

- The emulator must replicate the microcontrollers bus master cores (one or more CPUs and DMA controller), but none of the peripherals such as ADC, UART or CAN.
- Its RAM memory must be the same size or larger and have the same or faster access times as the SoC.
- The emulator must have ROM of the same size or larger, same or faster access times and the same content as that of the SoC.

Therefore, the emulator must meet these requirements:

- It must receive the clock signal of the SoC.
- It must also receive the results of the read operations in the peripheral area.
- Finally, the emulator must receive interrupt and DMA requests from the SoC via the synchronization interface.

Given these properties, an emulation will precisely match the behavior and the instructions carried out by the microcontroller being emulated. Both SoC and emulator start with the same internal state. Also, both have the same content of ROM. Thus, the information read from peripheral components is sufficient to exactly reconstruct the RAM.
content in the emulation, since writes to the local RAM are reflected in the emulation. Note, a read operation to non-initialized RAM addresses is not allowed.

All branch decisions will be the same in the SoC and the emulator. The only remaining changes of program flow which can not be predicted in the emulator are interrupts or DMA requests, which are communicated to the emulation to replicate the exact behavior of the SoC.

The parallel emulation introduces an additional source of potential cost savings as well:

The developer can assign the pins required to output the synchronization information to simple output functions such as LED driver, pulse generators or LCD drivers. Once the emulator is attached to the microcontroller, it then takes over emulation of the pins which are temporarily used to output trace data. A ribbon cable allows the emulated signals to be transferred back to the target hardware (see Fig. 2).

Depending on complexity and speed, the emulation can be run in an FPGA for slower CPUs or in an ASIC for faster CPUs. The ASIC / FPGA emulation must include the CPU core(s), the DMA controller and the internal memory. In difference to the traditional bond-out chips, only one implementation of the emulation core is required for each CPU series. A new bond-out chip for each new device with new or different periphery is no longer required, since the implementation of peripheral units is not necessary for the emulation. The cost for new bond-out chips as well as associated delays is no longer encountered. The proposed principle is particularly suited for microcontroller families which have identical cores and varying periphery, as for each new derivative full trace support can be provided immediately at no additional cost.

For slower CPUs (up to ~50 MHz), emulating the CPU in an FPGA lowers tool cost dramatically because only the existing FPGA needs to be reconfigured and one emulator can support different CPU families. Also the emulator logic which analyzes or preprocesses the available trace data can be implemented in the same FPGA. This will provide a very compact and cost efficient emulation system.

In case of an ASIC implementation, the trace data can be made available on a configurable interface. Due the very high width of the available trace data, it seems reasonable to provide a configurable interface, which provides a subset of the available trace data depending on the current demand. For instance, for a branch / decision code coverage analysis of a CPU the program counter and the data read by the CPU can be made available on the output. For another problem e.g. stack analysis, the stack pointer and program counter may be selected for output.

Currently, we are discussing a convenient interface with some major emulator vendors.

Alternatively, the principle can also be used with software emulation. A fast buffer captures the synchronization data and a software emulation of the CPU core computes the executed instructions. Yet, this approach does not work in real time for most applications and the available trace interval is limited by the size of the buffer.

Fig. 3 illustrates the differences between the two traditional techniques (bond-out chip and embedded trace based emulators) and the new approach.

On bond-out chip based emulators the application is executed in a special evaluation chip inside the emulator. Basically, the bond-out chip can provide all trace information. The disadvantages are the physical limitations of this concept, which allows a limited speed only, and the high system costs. Due to these reasons bond-out chip based emulators become less important.

Over the last years, embedded trace based emulators have become the state of the art approach. Here the trace capturing capabilities are implemented on chip. Due the cost pressure only limited chip space is available for the embedded trace support. The result is that the developer has to accept an incomplete trace or the system will be slowed down when a complete trace has to be captured. Through increasing CPU clock frequencies, this limitation gets more and more inconvenient.

The new approach combines the advantages of both traditional technologies: the trace can be accessed at very high CPU clocks in the same quality as from the bond-out chip. The proposed solution is well applicable for multicore SoCs, but in case of independent clock domains and high I/O bandwidth the synchronization information may need a
very high bandwidth. However, in this case other comparable technologies are also not able to deliver a full, continuous and real-time trace.

For the implementation of the synchronization support the internal architecture of the SoC has to be analyzed. First we have to extract all signals which are required for the synchronization. In a second step we have to analyze the signal interdependence for further compression of the synchronization information. For instance, in case of starting an interrupt sequence, no I/O operations may be possible on the bus and in this case the pins used to transfer the interrupt number can be shared with the pins which are used to transfer the data read from the CPU. As a third step the system integrity IP has to be implemented. The complexity of this IP is scalable.

A correct synchronization of both systems may be achieved with a simple hash function, whereas a more complex hash may provide information on the problems in case of a system integrity violation.

A generic synchronization protocol is not reasonable, since the signal interdependence often differs from SoC to SoC. In some architectures it makes sense to capture the sync data very local to the CPU, in other architectures it may be more advantageous to capture the sync data on the bridge from local bus to periphery bus.

In general, the required bandwidth can be estimated with the following scheme:

Most actual microcontrollers provide a fast CPU and fast memory interface in combination with a significantly slower interface to I/O components. If the sync data interface is operated with double data rate, based on the CPU clock, the amount of pins required to output the synchronization data can estimated as follows:

\[ n = \left[ \frac{w + s}{2 + c} \right] + 1 \]  \hspace{1cm} (1)

Where \( n \) is the number of pins required, \( s \) is the number of status bits (SoC specific: interrupt state, bus wait flag, DMA state, DMA channel number), \( w \) is the bus width of I/O accesses and \( c \) is the minimum number of clock cycles per CPU read instruction and clock cycles per DMA read operation. The additional pin is the clock signal for synchronization of the data.

To illustrate this formula a bit more, we give three typical examples here. The first example is an 8 bit SoC for which we will also present the amount of additional hardware required to implement the emulation interface:

\[ n = \left[ \frac{8 + 1}{2 + 4} \right] + 1 = 3 \]  \hspace{1cm} (2 data, 1 clock)

The second example is that of a 16 bit SoC with additional DSP functionality:

\[ n = \left[ \frac{16 + 6}{2 + 4} \right] + 1 = 4 \]  \hspace{1cm} (3 data, 1 clock)

Finally, we present the example of a 32 bit SoC which uses only 16 bit for peripheral accesses:

\[ n = \left[ \frac{16 + 6}{2 + 3} \right] + 1 = 5 \]  \hspace{1cm} (4 data, 1 clock)

At higher CPU I/O read bandwidth the data stream to be transferred by the emulator interface may be buffered by a FIFO. In this case the statistical frequency of I/O load operations (\(-5\%\)) determines the actually required bandwidth.

The required bandwidth is dependant on the instruction set and the size of the FIFO (among other not so dominant factors) and can be estimated by the following formula:

\[ n = I_{IO} \times \left[ \frac{w + s}{2 + c} \right] + 1 \]  \hspace{1cm} (5)

The additional factor \( I_{IO} \) is the percentage of I/O load instructions. Of course, in a real implementation any case of FIFO overflow needs to be prevented by the following methods:

- Automatic insertion of NOPs by the compiler in case of frequent I/O read accesses.
- Pause the CPU in case of impending FIFO overflow.

Another method is the increasing of the clock line frequency. In many CISC systems the CPU clock is a multiple of the bus clock. Outputting this higher frequency also reduces the required pin count. E. g. because of the doubled CPU clock (ICGOUT) of the Freescale S08 microcontroller the pin count of formula (2) can be reduced to two pins. Alternatively, a high frequent external clock can be used to output the sync data.

In case of very high bandwidth I/O access (e.g. USB or Ethernet on CPU local bus) the presented approach gets more and more unsuitable. But also other approaches are limited here to non-continuous traces only.

Due to the availability of full trace data inside the emulator, it is possible to capture a very detailed trace. Not only the program counter and the data read and written, also the clock cycle synchronous bus control signals are available for further analysis. The quality of this trace data is similar to a state analysis by a logic analyzer, which has access to all internal bus signals of the microcontroller. This enables the deep analysis of bus performance, cache and DMA operation and makes a new generation of in-circuit emulators possible.

Until new in-circuit emulators will be available, a connection to existing interfaces like Nexus can easily be implemented and thus, it extends the applicability of those emulators to new chips that incorporate the synchronization interface.

The synchronization interface can be easily implemented as an add-on to existing on-chip debug support modules (OCD), which are available in a wide range of implementations (e.g. Renesas NDS or Freescale BDM). The OCD module is required for initial flash memory programming, setting breakpoints and in some cases for capturing some frames of trace data. An additional implementation of a synchronization facility extends this
very limited trace to a continuous and complete real-time trace of executed instructions, data read, data written, DMA operations and CPU register values.

IV. IMPLICATIONS/APPLICATIONS

The presented methodology has several implications and applications. We will discuss them in detail in this section.

A. New Debug Features

The emulation core carries out exactly the same operations with exactly the same data. Thus, it is possible to build an emulation core that uses this detailed data to enhance the debug functionality extraordinarily. E.g. it is possible to implement an almost arbitrary number of complex breakpoints which can be qualified by register contents of any processor register. The only drawback of these breakpoints is their delay. As the emulation runs with a well-defined delay of a few clocks, the detection of the breakpoints will be delayed by the same amount of clocks. But in most cases this can be tolerated. Also, it is possible to generate a full trace of the internal bus, processor registers or memory locations.

B. Access to trace data

In comparison to the embedded trace approach, the presented approach provides a huge amount of data. Potentially, all user addressable registers, internal registers (like status or pc) and bus information can be made available by the emulation core. This easily sums up to several hundred binary signals. Depending on the technology that is used to implement the emulator core, there are different options for the solution of this problem.

In case the emulator is implemented with FPGAs, the FPGA can be reconfigured for every new debugging process. The relevant data can be selected by the design and can be made available at some of the unused pins of the FPGA.

In case the emulator is implemented in a non-reconfigurable technology, there are two choices. Either the emulator chip provides as many signals as possible on its pins and these signals are wired to a large number of connectors, or the emulator chip is designed with an internally configurable selection mechanism, that lets the user decide at runtime, which signals will be routed to the external connectors.

In all cases, external equipment like logic analyzers or 3rd party trace probes can be used to record the trace data.

C. Debug and trace support in mass produced chips

As the implementation cost of the Sync TX and the Hash

![Fig. 4. ARM Cortex M1 based demonstration system. The cores (SoC and emulator) of the system were generated with the Actel CoreConsole IP Deployment Platform. On both systems the same hash functions are executed. When the results are identical, both systems are executing the same code and the trace data are valid.](image-url)
IP is quite small, chip manufacturers can afford to put the synchronization facility on all produced controller chips. Thus, debugging and validation can be done with the same chips that are used for the final product. Current in-circuit emulators are typically based on special versions of the controller (so-called evaluation chips) that grant access to internal signals and thus are different in their timing behavior and in some cases even expose a different logical behavior. The usage of mass produced chips for debugging and validation gives additional certainty that the obtained results are still valid for the final product. Of course this only works if the emulation chip perfectly duplicates the behavior of the original core.

D. System Integrity Control

Many discussions have shown that certain doubts remain about the emulator running perfectly synchronized with the original core. Some people doubt the identical behavior of the emulated core, while others doubt the synchronization process itself. Thus, we have added an optional module that computes a hash over the current core state and transmits this hash value whenever no other information needs to be sent from the SoC to the emulator. Optionally, a fixed part of the bandwidth can be dedicated to the transmission of this hash value. This hash value can be computed in the emulation in an identical way. Thus, the two values can be compared and will differ as soon as the state of the emulated core and the original core are not the same. Such a comparison can help to assure the same behavior of the core and the original core.

E. Failure detection in returned chips

Sometimes regularly produced chips fail in the field. For the manufacturer it is extremely interesting why those chips failed and which part of the chip failed. The system integrity check allows to reconstruct the exact situation in which the chip failed (namely the time when the hashes differ) and in some cases even enable the identification of the failing component.

V. IMPLEMENTATION EXAMPLE

To evaluate the presented methodology and to prove its concept, we have implemented different demonstration systems based on FPGAs and CPU soft cores.

One of the demonstration systems consists of two Actel M1-enabled ProASIC3 boards. The implemented CPU is an ARM Cortex M1 [13, 14]. Using the Actel CoreConsole IP Deployment Platform a SoC was designed, consisting of a Cortex M1 core, which is connected via an AHB lite bus [15] to a memory interface (CoreMemCtrl) and an AHB-to-APB bus bridge (CoreAHB2APB). On the APB bus [16] a GPIO unit (CoreGPIO), an UART (CoreUARTapb) and a Timer (CoreTimer) is connected (figure 4). For capturing trace data with the new technology, the system was extended by two modules.

One module (Hash IP) calculates recursive hash values of the HADDR and HRDATA busses. The calculation of these hash values considers the different bus states, for this purpose the HPROT[0], HWRITE, HREADY and HTRANS signals have to be interpreted.

A second module (Sync TX IP) encodes the CPU clock (SYSCLK), the CPU reset signal (NSYSRESET), the timer interrupt signal (IRQ), the data read from the APB bus (PRDATA) and the hash values. Due to time multiplexing of these signals, only three pins of the FPGA are required for transferring.

Table 1. Raw trace data of the ARM Cortex M1 AHB and APB

<table>
<thead>
<tr>
<th>Opcode</th>
<th>AHB trace</th>
<th>APB trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000c8c: ldr r3, [pc, #76]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000c8c: ldr r3, [r3, #0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000c8c: ldr r3, [r3, #0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000c8e: ldr r3, [pc, #76]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000c8e: ldr r3, [r3, #0]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The AHB bus signals HBURST, HIMASTLOCK, HPROT[3:1], HSIZE, HRESP, HWRITE and PRDATA and PWDATA are also traced, but not displayed in the table.
The logic was synthesized and optimized for an Actel ProASIC3 FPGA. The adequate gate may be lower by synthesizing and optimizing for non-FPGA targets.

<table>
<thead>
<tr>
<th>SoC</th>
<th>Required Pins (*)</th>
<th>Required Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freescale S08</td>
<td>2</td>
<td>~1k</td>
</tr>
<tr>
<td>ARM Cortex M1</td>
<td>3</td>
<td>~1.2k</td>
</tr>
<tr>
<td>Freescale S12X +</td>
<td>7</td>
<td>~2k</td>
</tr>
<tr>
<td>XGATE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Freescale MPC55xx</td>
<td>17</td>
<td>~4k</td>
</tr>
</tbody>
</table>

(*) In case of using port replacement, all pins are available for the application, even during recording trace data (see Fig 2)

The emulator core consists of the same units as the SoC except for the peripheral units. The transmitted synchronization information are decoded by the Sync RX IP, which provides the SYSCLK, NSYSRESET, IRQ, PRDATA and the hash values.

The emulated system is clocked with the same clock SYSCLK as the SoC. After power-on, the emulator waits for the SoC’s reset signal NSYSRESET. After receiving the reset signal, the emulator executes the same instructions as the SoC. Due to the time required for transferring the synchronization information, there is a well-defined delay of 3 SYSCLK clock cycles between SoC and emulation. In case of a read operation from the APB bus, the AHB-to-APB bus bridge of the emulator reads the same data as just read by the SoC’s AHB-to-APB bus bridge. Something similar happens in case of an interrupt: At the corresponding SYSCLK cycle in the emulator, the IRQ signal will get active and the emulator will also call the interrupt service routine.

Similar to the SoC, a hash value of the HADDR and HRDATA will be calculated. When both hash values are identical, we can be sure that the behavior of both systems is identical.

Now we can collect all trace information inside the emulator. Due to the easy access to all internal bus signals, it is possible to capture a low level bus signal trace of the AHB and the APB bus. This trace enables an extremely deep insight in the bus operation, similar to a logic analyzer (see table 1). In the demonstration system the internal block RAM resources of the FPGA are used for single-shot capturing the trace data.

To the best of our knowledge, the required resources for the Sync TX and the Hash IP in the SoC are always smaller than traditional solutions, which are necessary for capturing trace data (Table 2 and Table 3).

VI. CONCLUSION

The presented technology enables a variety of debug/test/validation related activities in mass produced SoCs. The amount of resources that is required on the SoC is so small that it can be implemented on almost any chip. Debug related activities include deep detailed traces of almost any processor internal information and arbitrary number of complex breakpoints. Test activities include system integrity control which allows inspection of returned devices. Validation activities include full traces of program execution to comply with regulations for safety critical systems.

The presented technology is patented [12] and several large chip manufacturers are considering to implement the synchronization interface on their new designs.

REFERENCES