Real-Time Debugging of Digital Integrated Circuits

Jürgen Haufe\(^1\), Christoph Fritsch\(^2\), Matthias Gulbins\(^1\), Volker Lück\(^2\), Peter Schwarz\(^1\)

\(^1\) Fraunhofer Institut für Integrierte Schaltungen, Germany
\(^2\) Bosch Telecom, Germany

Abstract

The detection of the reason for an incorrect behaviour of an integrated circuit in its real-time environment is a very time consuming task. Using logic analysers in combination with an extensive logic simulation is the today’s only way to isolate possible errors. In this paper, we introduce a hardware debugger to perform the debugging process in a more efficient way. Goal of the hardware debugger is detecting a failure of an integrated circuit in real-time processing and the supply of data for the determination of the cause of the failure.

1 Introduction

The detection of the reason for an incorrect behaviour of an integrated circuit is a very time consuming task. In many cases, errors arise after a real time of seconds or minutes. Furthermore, the errors can often not be reproduced. For the error isolation under real-time conditions, the designer can connect logic analysers or oscilloscopes to external pins of the IC hoping that the error will occur once more. Usually, integrated circuits are not prepared for accessing internal signals by such external devices when running at speed. Therefore, time consuming simulations partly at gate level are required to analyse chip internal causes of an error.

In this paper, we introduce a hardware debugger to perform the real-time debugging process in a more efficient way. The goal of the hardware debugger is detecting a failure of an integrated circuit (Device Under Test, DUT) in real-time processing and the acquisition of information to analyse the failure.

By real-time processing it is understood that debugger functions (trace and breakpoint detection) are executed while the integrated circuit and its system environment are running at speed. The timing of the DUT may not or less be influenced by the integrated debugger functions.

Our main focus for developing a hardware debugger was FPGA design and prototyping but our approach is not limited to. For broader application, the hardware debugger is independent of any circuit technology and of any circuit vendor. Our hardware debugger solves the above problems with integrated circuits which usually provide no means to observe and to modify internal states under real-time conditions. A stop of the system environment is normally not possible during the DUT in-system test. In order to meet these real-time debugging requirements, the hardware debugger has to be implemented as hardware consisting of DUT on-chip and DUT off-chip components.

We developed the following requirements for our efficient real-time debugger for in-system tests of digital FPGA and ASIC designs:

- **Trace mode**: Non-intrusive sampling and storing of all hardware-internal states and continuously tracing and storing of all input-signals.
- **Breakpoint mode**: Non-intrusive detection of user defined states and initiation of corresponding actions.
- **Update mode**: Presetting of user defined hardware-internal states.

In section 2, we give a brief overview of existing methods for ASIC and FPGA debugging. Our debugger approach is described in section 3 in more detail. Finally in section 4, we present experiences which we have made applying our hardware debugging environment.

2 Related methods

In this section, we present existing methods for hardware debugging.

In "classical" real-time hardware debugging approaches, the hardware behaviour is observed using logic analysers or oscilloscopes connected to externally accessible I/O pins or probe pins of the DUT. Hardware internal nodes are usually routed to external pins which are not used by the DUT. These probe pins allow access to internal nodes specified by the designer. Because of the limited number of unused external pins, only a small number of signals may be traced by that technique. Multiplexing pins [1] or using virtual probes [3], [7] increases the number of possible probes but this usually prevents real-time debugging. Some approaches use the hardware simulation environment instead of logic analyser or oscilloscopes to stimulate and to observe the DUT. The simulation environment running on a host computer is online [3], [4], [7] or off-line [5] coupled to the DUT. Due to the extremely slow host interface, real-time debugging is not possible.

Various scan path techniques [2] enable tracing and updating of those internal and I/O registers which are lined up in one or more user defined scan chains. The hardware internal scan chain is off-chip addressed by a JTAG port. The serial JTAG protocol...
does not support at speed debugging. The same disadvantage can be said for the *Xilinx Readback Capability* [9].

A completely new approach to hardware debugging is provided by Altera with the *SignalTap* as a built-in feature inside Altera’s APEX devices. FPGA-internal nodes are captured into an embedded RAM array (not used by the DUT) synchronously to a user selected internal global clock while the chip is running at speed. The data stored inside the RAM gets transferred off-chip via a JTAG port. This approach is limited to Altera APEX devices. The number of signals that can be stored in the embedded RAM and their sample buffer depth depends strongly on the amount of free embedded RAM space.

An approach to the automated insertion of additional **breakpoint** hardware into the DUT is presented in [10]. Here, the user specifies breakpoints in the behavioural description of the DUT. The breakpoint hardware is generated automatically for these user-defined breakpoints. This approach supports the breakpoint detection in real time.

A **hardware monitor** for real-time tracing of signals is introduced in [11]. This monitor is part of a system for communication analysis in distributed real-time systems.

In summary it can be said: no method for hardware debugging is actually known that offers the possibility to trace, to detect and to update chip-internal circuit states under real-time conditions as a technology-independent solution.

3 A hardware debugger approach

3.1 Methodology

Our methodology [12] of hardware debugging is strongly driven by the demand to shorten the time consuming error analysis process drastically. Analysis of the DUT behaviour starts when an error is detected during the in-system run or prototyping of the DUT.

Our idea behind hardware debugging is to sample and to store all hardware-internal states of a DUT during the real-time in-system run. This real-time in-system run starts at time point 0 and is executed until an error occurs. The data sampled during this real-time execution describe the real-time execution scenario which is then used for an analysis of the circuit behaviour. It can be divided into the following two different sets: the hardware-internal states describing the internal behaviour of the DUT, and the inputs to the DUT, representing the corresponding environment. The combination of the two data sets describes one real-time execution scenario of the DUT within its system environment.

The error analysis process is done using gate level simulations. The gate level simulation model of the

---

**Fig. 1:** Hardware debugging methodology

DUT is initialized with the data sampled for the hardware-internal states which belong to the point in time of the real-time execution scenario where the error analysis will start. Thus, the simulation does not need to start at the very beginning of the real-time execution scenario. As stimuli for the DUT inside the gate level simulation, those inputs are used which are sampled during the interval between the start of the error analysis and the occurrence of the error. Now, only the time interval between the initialization point and the error point has to be simulated for the detailed analysis (Figure 1). The effect is an overall reduction of the required simulation time.

To analyse the DUT and possible errors, it is necessary to restart the debugging process several times. The need to rerun the debugging process substantially defines the strategy for the debugging environment. In order to be able to reproduce a certain error situation, the DUT has to be executed to this specific state by applying the same set of stimuli each time. This requires that the set of stimuli is available or can be generated each time and the DUT behaves deterministically on this set of stimuli. If one of both conditions is not fulfilled, the DUT debugging process is not reproducible. In practice, this is often true with in-system real-time tests. Here, the stimuli may come from a real system environment and therefore they are often neither predictable nor repeatable.

In the case of a reproducible debugging process, debugging can be arbitrarily often repeated. The user can set breakpoints and read out circuit states incrementally in order to encircle the error cause. The same procedure is also possible with the simulation model.

In the case of a non-reproducible debugging process, each repetition of the debugging will result in a different DUT behaviour because the system environment may generate differing stimuli for each repetition. For a detailed analysis of the DUT behaviour (e.g. by simulation), the debugging has to be made

---

[236]
reproducible. One solution is to sample all relevant data and to store them in a memory as continuously as possible during the first execution. Especially, this has to be done for the input data that will be used as input stimuli for the simulation interval (Figure 1). As a drawback of this method, technical limitations arise caused by the limited sampling rate (technology dependent) and the limited RAM space for large number of signals (large designs, long real-time intervals).

Additionally, the DUT can also be set to a user defined state by using the update mode. The actual state of the DUT can be defined without having to take the device to this state by a sequence of external stimuli. This mode is used to test corner cases effectively. Corner cases describe DUT states which are hard to achieve by external stimuli, e.g. testing the ability of state machines to recover from illegal states that will never be reached during normal operation of the system. Before a real-time in-system test starts, the DUT is set to the user defined state for a certain corner case. Beginning with this DUT-state, the system gets executed. The DUT can then be observed and analyzed using our debugger.

3.2 Architecture

3.2.1 Overview

An overview of different versions of our hardware debugger architecture is shown in Figure 2.1 to Figure 2.3.

All presented versions of the debugger consist of components which resides inside the DUT and components that are implemented outside of the DUT. The components inside the DUT realize the communication between HW debugger and DUT internal registers implementing the trace/update functionality and implementing a breakpoint mechanism. The components implemented outside of the DUT are the HW debug controller, the trace/update RAM (organized as a ring buffer) and an interface to a host computer. In addition to the debugger components implemented as hardware, a debugger software is running on the host computer controlling the debugger. The three proposed versions of our debugger architecture differ in the amount of debugger components which are integrated into the DUT and the amount of components which are realized as a separate hardware device.

The advantage of version 1 (Figure 2.1) where the debugger is realized mostly outside the DUT is based on its independence from the DUT technology. Only few debugger components and the debugger interface have to be integrated into the DUT, fairly similar to scan path designs. Large off-chip RAMs are practicable. The external components can be combined to a standard debugging device. This debugger device can be attached to any device with the embedded parts of the debugger, e.g. FPGAs or first ASIC prototypes.

Architecture version 1

![Fig. 2.1: HW debugger separated from DUT](image1)

With a completely integrated version as shown in Figure 2.3, higher sample rates can be achieved because of the shorter internal signal paths. Version 3 provides the best performance for debugging, but it requires a large amount of internal RAM occupied by the debugger.

Architecture version 2

![Fig. 2.2: HW debugger kernel inside DUT](image2)

An intermediate solution is shown in Figure 2.2. Some parts of the debugger functionality are embedded into the DUT. A debugger interface with relaxed timing constraints connects the DUT internal parts to the external debugger components. The timing critical debugging task is carried out by the debug controller by tracing and storing the DUT states and the external stimuli applied to the DUT. Therefore, the
interface to the debugger inside the DUT in version 3 is simpler than the interface in version 1. This is due to the lower data rates for less data which have to be sent to the host interface and the debugger software. The intermediate solution of version 2 combines the advantages of version 1 and version 3. We achieve the high sample rates of version 3 and we can provide sufficient RAM space for the trace and update data.

**Architecture version 3**

![Fig. 2.3: HW debugger integrated into DUT](image)

As mentioned earlier, the input data has to be sampled continuously if the inputs are not reproducible. Currently, for simplification, we use a logic analyser to meet this task. A better solution may be a special hardware monitor as introduced in [11].

**3.2.2 Debugger execution modes**

The hardware debugging process is controlled by the user via the host interface. Three debugging modes are available:

*Trace mode*: During the trace mode, the register outputs of all DUT internal registers are sampled synchronously to a user-defined internal global clock. The sample clock may differ from the register clock.

*Update mode*: During the update mode, user-selected DUT-registers can be set to user-defined values. The update mode is not timing-critical and usually executed when the DUT is in a wait state (reset or breakpoint).

*Breakpoint mode*: Breakpoints are user-defined DUT states, which initiate a pre-defined action depending on the type of the occurred breakpoint. Generally, the DUT is stopped and the DUT state is extracted by the debugger. Depending on the cause of the breakpoint, the user can proceed with continuing, resetting or aborting the debugging process. There exist three classes of breakpoints:

- internal breakpoints: a definite internal circuit state is achieved, e.g. internal arithmetic overflow
- external breakpoints: in the DUT environment a certain state is achieved, e.g. incorrect stimuli or incorrect response
- user intervention: the user interrupts manually the DUT, e.g. starting or halting the debugging process

**3.2.3 Debugger components**

The debugger components integrated into the DUT are of special interest to the designer. These debugger components have direct impact on the DUT design and the corresponding design flow. They connect the DUT registers to the debugger interface. Depending

![Fig. 3: Input trace](image)

![Fig. 4: Architecture of a shadow cell](image)
cells for the DUT register, breakpoint cells, hardware debugger kernel, and hardware debugger interface.

**Shadow cells**

All or a user defined subset of the DUT registers are substituted by shadow cells. The architecture of the shadow cells is shown in Figure 4. The approach can be regarded as an extension of the clocked scan methodology described in [2]. The main feature of our shadow cells is characterized by the ability to capture the output of the DUT register into the shadow cell register without any impact on the DUT register. The output is captured while the DUT register is clocked at real-time speed.

Using simple scan techniques like clocked scan to sample a circuit state, the content of the DUT registers will be destroyed during capturing the data. Our inserted shadow registers are lined up to one or more shadow cell chains.

**Breakpoint cells**

Breakpoint cells are hardware components introduced into the DUT to control the detection of DUT internal states. We distinguish between two kinds of internal breakpoints (Figure 5): RT level breakpoints and gate level breakpoints.

![RT Level Breakpoint Cell](image)

VHDL Example:

```vhdl
if <breakcondition> then
  ce <= 8#1;
end if;
```

**Gate Level Breakpoint Cell**

![Gate Level Breakpoint Cell](image)

**Fig. 5: Breakpoint cells**

RT level breakpoints can be applied to define complex break conditions using complex signal data types (e.g. records) with complex operators. Currently, the RT level breakpoint macros have to be inserted manually into the DUT VHDL or Verilog code. The breakpoint management is up to the user. For example, the DUT can be disabled addressing the signal ce.

Gate level breakpoints are implemented by pre-designed cells which are automatically inserted into the DUT gate level description at user-defined locations. Thus, the RT level code can be kept untouched.

Due to the abstraction level, only single nodes or vectors can be observed. The reference value for the comparator can be downloaded using a separate breakpoint scan path. A reference value is valid only, if a single enable bit is set. Thus, the user can activate the comparator functionality for each breakpoint cell.

**On-chip HW debugger kernel**

The HW debugger kernel is a state machine managing the communication between the debugger interface and the debugger components within the DUT. It is implemented independently from the DUT. In the case of debugger architecture version 2 (Figure 2.2), the HW debugger kernel has to control the reading/writing of one or more shadow cell chains, breakpoint detection, user intervention, and the debugger interface.

![Debugger interface](image)

Reading the shadow cell chains is time-critical. The debugger kernel supplies the shift clock and has to read from one or more scan out ports. The time spent for that action determines the sample period essentially. Therefore, reading from shadow cell chains and writing to off-chip RAM is organized as a pipeline mechanism and it is done with a maximum of parallelism. Downloading data to the shadow cells during the debugger update mode is less time-critical. In this mode, the DUT is usually in a wait state.

As mentioned above, the breakpoint detection hardware for internal breakpoints is located within the DUT to react as fast as possible. In addition to the internal breakpoints, the HW debugger kernel has to process external breakpoints and user interventions too (see Section 3.2.2) and has to perform the corresponding actions (usually disabling the DUT by setting signal ce to low).
**Debugger interface**

Via the debugger interface, data between debugger kernel and both off-chip RAM and host interface (Figure 6) are transferred. We need 36 ports to access the external RAM. Six host interface signals are necessary to control the debugging process. The advantage of the selected interface is the high data transfer rate that can be achieved. As a drawback, we need 42 unused pins of the DUT to interface the debugger and the RAM. If there are not enough unused pins available, the debugger interface can be serialized in some manner. We try to avoid the serial interface whenever possible with respect to the increased hardware overhead for serialization and the poor performance.

### 3.3 Designflow

Figure 7 shows the integration of our hardware debugging methodology into a standard design flow for integrated circuits. The design flow can be divided into two phases: the synthesis phase and the verification phase.

- **Synthesis phase**
  - DUT-Design.vhd
  - RTLBreakInsert
  - RTLSynthesis
  - GTLBreakInsert
  - ShadowInsert
  - Linking
  - Place & Route

- **Verification phase**
  - System Environment
  - External Breakpoint
  - HW2sim
  - Sim2HW

**Fig. 7: Design Flow**

The synthesis phase starts with the VHDL or Verilog description of the DUT. Breakpoints at RT level can be optionally inserted by the user. The final RTL description is synthesized with known tools (e.g. Synopsys Design Compiler). As described above, additional breakpoints can be introduced into the gate level description after synthesis. The next step is the insertion of our shadow registers into the gate level description and the construction of the scan chains. Here, we apply two alternative methods, first a Synopsys debian script and second the Synopsys Test Compiler. After the shadow register and the scan path insertion was finished, the on-chip hardware debugger kernel (see Figure ) is linked to the DUT using the synthesis tool. Finally, the place & route step finishes the synthesis phase.

The verification phase starts with executing the DUT within its system or prototyping environment. If an error appears or an external breakpoint occurs, the sampled data can be used to initialize a simulator (HW2sim). Sim2HW can be used to initialize the DUT to a specific state before another verification cycle starts.

### 4 Application

A Viterbi decoder is used to apply our debugger to an actual project. The Viterbi design is an industrial standard code compliant Viterbi decoder IP developed at Bosch Telecom to be integrated into Bosch’s modem ASICs. The decoder instantiated 918 register cells and it requires about 12k byte of embedded RAM.

The Viterbi verification strategy consists of three steps (Figure 8 to Figure 10):

#### Classical RT level simulation

A testbench is used to read stimuli from a file. The testbench applies the stimuli to the Viterbi, samples the responses from the DUT and compares the Viterbi responses with a set of known good responses generated beforehand. If the DUT produces wrong responses the testbench stops the simulation and gives the designer several information to analyze the error.

**Fig. 8: RT level simulation**

**Stand-alone prototype test**

After the design of the Viterbi is verified at RT level, a FPGA prototype gets synthesized. According to the design flow shown in Figure 7, the hardware debugger kernel is inserted. We apply the same stimuli for both the RT level simulation and the prototype test. Therefore, the RT level stimuli are transformed automatically into a Flash ROM programming file. For this prototype test, the Flash
ROM works as a kind of pattern generator. The PROmetheus prototyping environment [6] based on Altera Flex10k250 devices is used for this prototype test.

**In-system prototype test**

The FPGA is integrated into a system prototype where several other modules of the modem design are tested too. For in-system test, the input stimuli to the Viterbi are now unpredictable and checking for correct Viterbi responses is not easily possible. We used the debugger to apply our stimuli and to observe the internal behaviour. This finished the basic verification of the FPGA prototype with the stand-alone prototype test. After this verification phase we could give the FPGA to our system group for in-system testing. During verification of the overall system prototype, it still remains possible to observe the state of the Viterbi at any time without interrupting the system. At the time when further FPGA of the modem prototype are realized with the embedded debugger, it is possible to look deeply into the system prototype even during long verification cycles. This gives us the possibility to understand better the systems behaviour and to determine exactly where errors arise.

**Acknowledgements**

The authors would like to thank MEDEA for sponsoring this work within the research program "European Intellectual Property In Designing Electronic Systems" (EURIPIDES).

**References**