Automatic Generation of JTAG Interface and Debug Mechanism for ASIPs

Oliver Schliebusch
ISS, RWTH-Aachen
52056 Aachen, Germany
+49-241-8027884
schliebu@iss.rwth-aachen.de

David Kammler
ISS, RWTH-Aachen
52056 Aachen, Germany
+49-241-8028256
kammler@iss.rwth-aachen.de

Anupam Chattopadhyay
ISS, RWTH-Aachen
52056 Aachen, Germany
+49-241-8027871
anupam@iss.rwth-aachen.de

Rainer Leupers, Gerd Ascheid, Heinrich Meyr
ISS, RWTH-Aachen, 52056 Aachen, Germany

Abstract

Application Specific Instruction Set Processors (ASIPs) combine the high performance of dedicated hardware with the flexibility of programmable solutions. Architecture Description Languages (ADLs) describe ASIPs on an instruction-accurate or cycle-accurate abstraction level. Accurate information about the physical parameters cannot be obtained on this level of abstraction. For this, a hardware model on Register Transfer Level (RTL) and a subsequent gate-level synthesis is required. Several projects deal with the topic of generating a hardware description model from an ADL. Currently, those approaches focus only on the desired processor core and omit architectural features, such as a debug mechanism or JTAG interface. The acceptance of ASIPs compared to other heterogenous solutions, dedicated to combine performance and flexibility, can only be achieved if well known processor features are supported. In this paper, we propose an automatic generation of JTAG interface and debug mechanism from an ADL. This generation is embedded into our RTL processor synthesis tool, which is based on the Language for Instruction Set Architectures (LISA).

1 Introduction

ADLs are used to map a given application to an optimized programmable architecture. These ASIPs combine high performance of dedicated hardware with the flexibility of a programmable solution. The required design space exploration is based on ADLs, which enable a fast incorporation of architectural changes. The architecture description is used to automatically derive software tools, for example C-compiler, assembler, linker and simulator within a few minutes.

The design space exploration, based on ADLs, is performed on a high level of abstraction and lacks accurate information about physical parameters, leading to a sub-optimal solution. These parameters (like chip area, clock speed and power consumption) can be accurately derived from a hardware model on RTL and a subsequent gate-level synthesis. To obtain accurate performance figures, while preserving the speed of design space exploration using an ADL, an automated approach to generate the RTL description from the ADL is mandatory. Several projects deal with the topic of generating a hardware description model from an ADL [1] [2] [3]. Currently, those approaches focus on the processor core and omit architectural features, such as a debug mechanism or JTAG interface. The acceptance of ASIPs can only be achieved if popular processor features are supported.

One of these features is a hardware debug mechanism. It enables the designer to debug software in its final hardware environment by giving access to the state of the processor core via an additional interface. The JTAG interface is commonly used for this.

During ASIP design, there are two possibilities to integrate a processor feature (for instance, a debug mechanism):

- Changes can be implemented manually either to the ADL description or to the already generated description on RTL. These changes are not possible without accepting increased development time and the risk of fatal errors.
- The solution, we propose, is to integrate the generation of processor features into the synthesis step from ADL to RTL. Here, the influence of a generated processor feature on the physical parameters can be taken into account during design space exploration without affecting the speed of the exploration.

The contribution of this paper is to present the first automatic generation of a JTAG interface and debug mechanism from an ADL. This generation is embedded into our RTL processor synthesis [4] [5] from the LISA [6] processor description language. This approach has been used successfully to generate different architectures completely on RTL [7] [8]. Using our approach, the designer is able to include necessary debugging capabilities into the target architecture and to evaluate the impact on the performance early in the designing process.
The paper is organized as follows: section 2 lists related work and introduces the Nexus and JTAG standard. Afterwards, section 3 discusses the changes required in the generated processor core in order to support debug functionality. Section

4 describes the functionality of the generated JTAG interface and section 5 presents the functionality concerning the generated debug mechanism. The results are discussed in section 6. This paper ends with conclusion and future work.

2 Related Work

In this section, the related work concerning RTL synthesis from ADLs, architecture design systems, the debugging standard Nexus and the JTAG standard for a test access port are discussed.

2.1 RTL Synthesis from ADLs

Several ADLs support HDL-code generation from higher level of abstractions than RTL. So far, publications about an automatic generation of JTAG interface and debug mechanism from ADLs are not known.

Some of the ADLs strongly oriented towards the instruction-set are ISDL [9] and nML [10]. For example, the synthesis tool HGEN [1] generates synthesizable Verilog code from an ISDL description. The HDL generator GO from Target Compilers Technologies [11], which is an industrial product, is based on the architecture description language nML. The project Sim-HS [3] is also based on the nML description language and generates synthesizable Verilog models from Sim-nML models.

There are also approaches based on a combination of instruction set/architecture description. Some of them are the ADL EXPRESSION [12] [2], FlexWare [13] which is more related to RTL than the level of ADL, the PEAS-III [14] and the derived ASIP-Meister [15] that work with a set of predefined components. Information on JTAG Interface and debug mechanism generation from any of these approaches is currently not available.

2.2 Architecture Design Systems

In addition to the work based on ADLs, architecture design systems have to be discussed here. The XTensa [16] environment from Tensilica [17] allows the user to select and configure predefined hardware elements. Hence, the design space exploration can be performed very efficiently and synthesis results are convincing. This approach is known to generate a JTAG interface and debug mechanism automatically. Detailed information about the performance is not publicly available. The S5000 family from Stretch [18] is based on the XTensa architecture and is enhanced with a flexible FPGA part, which is used to extend the instruction set. This design does contain a JTAG interface which makes the in-circuit debugging possible. The PICO (program in, chip out) [19] system developed by the HP-labs is based on a configurable architecture, including nonprogrammable accelerators and cache subsystems. Information about the generation of a debug mechanism is not known for the PICO system.

2.3 The Nexus Standard

In 1999, the Nexus 5001 Forum released a standard for a global embedded processor debug interface called Nexus standard [20]. Nexus compliant debug interfaces are divided into four classes. The standard specifies features supported for each class. Class 1 compliant devices implement least and class 4 compliant devices implement most features. The standard also declares specifications about the download and upload rate of the debug port and whether it is full- or half-duplex. Additionally, the Nexus standard defines an Application Programming Interface (API) for the debug mechanism. A complete list of the features required for a certain class can be found in [20].

A processor with Nexus debug interface is capable of switching to a special operation mode for debugging, when the processor core is halted. The current state of the processor core can be analyzed or modified via the debug mechanism. This operation mode is called *debug mode*. The usual operation mode is called *user mode*. The basic features required for making use of the debug mechanism are listed below.

Entering debug mode and returning to user mode: The debug mode can be requested via an external pin. Also, debug mode is entered whenever a breakpoint interrupt occurs.

Single step instruction in debug mode and re-enter debug mode: Sometimes it is very important to observe every step of execution of a critical part of an application. In this case, it is indispensable to run the instructions step by step.

Register access in debug mode: The current state of the registers of the processor core (the so called user registers) have to be read in order to determine the current state of the processor. Writing to the user registers is useful to intervene during execution of a program and to force a special processor state.

Memory access in debug mode: Similar to the state of the user registers, the content of the memory may be of interest during debugging. Therefore, it is useful to have read access to the memory in debug mode. Write access again gives the opportunity to manipulate the execution of a program.

Hardware breakpoints: Similar to breakpoints in debuggers used for software development, Nexus debug interfaces also support hardware breakpoints. If a breakpoint is hit, the processor switches to debug mode.

For low rate half-duplex debug ports the Nexus standard makes use of the IEEE 1149.1 port, commonly known as JTAG port.

2.4 The IEEE Standard Test Access Port (JTAG)

The IEEE standard 1149.1 ”Standard Test Access Port and Boundary-Scan Architecture” was originally introduced in February 1990 by the Joint Test Action Group (JTAG). It has been modified several times up to its current version 1149.1-2001 [21]. The standard defines a test access port and a boundary-scan architecture for digital integrated circuits and for the digital portions of mixed analog/digital integrated circuits. The acronym JTAG became synonymous with this standard over the years.

2.4.1 The Boundary-Scan

The boundary-scan test architecture provides means to test interconnects between integrated circuits on a board without using physical test probes. It adds a boundary-scan cell that includes multiplexers and latches to each pin on the device (figure 1). A detailed description of a boundary scan cell can be found in [21]. For a JTAG implementation, according to the IEEE standard, the integration of a boundary scan using pad cells is mandatory.

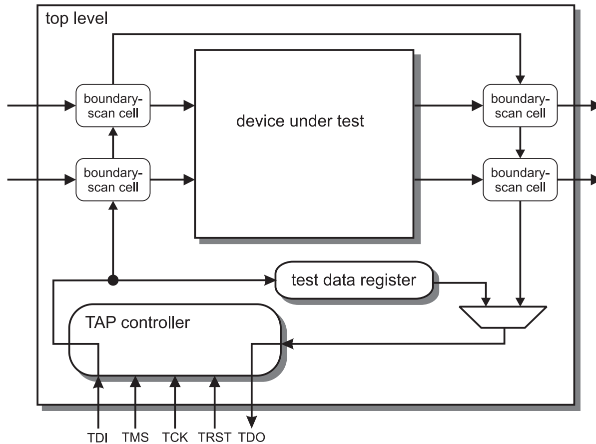

Figure 1. JTAG Interface and Boundary-Scan

2.4.2 The Test Access Port

The Test Access Port (TAP) is a five pin (TCK, TMS, TRST, TDI, TDO) general purpose port, which can be used to access additional functionality of the chip (e.g. the boundary scan chain). Data is propagated serially via TDI towards TDO into and out of the architecture. A detailed description of the ports and their purpose can be found in [21]. The TAP controller is the major component of a JTAG interface. Its functionality is well defined and not subject of our investigations.

In order to provide access to vendor-defined functionality via the JTAG interface, it is possible to connect special registers, the so called *test data registers*, serially between TDI and TDO. The TAP controller selects the current connected register. Not only a *test data register* may be selected, but also the boundary scan chain as shown in figure 1.

3 Extensions to the Processor Core

This section describes required changes to the synthesized processor core in order to support debugging functionality.

The LISA model description does not have to be modified to include the debug mechanism. Configurations for this feature are applied via a Graphical User Interface (GUI), which guides the complete synthesis process.

The general structure of a processor core generated from a LISA processor description is briefly introduced, before the changes are explained in detail.

3.1 Processor Core Structure

The following three terms will be referred to:

Entities model the hierarchical structure of the architecture. Every entity is implemented as an *entity* (VHDL) or a *module* (Verilog) on RTL.

A *unit* implements a cohesive functional block. It is defined by its purpose, its input and output. A unit is implemented as one or several *processes* (VHDL) or *always blocks* (Verilog).

A *path* establishes links between units. Paths contain one or more signals. For instance, a write path contains a signal for enabling the access, a signal propagating the data and a signal transmitting the address (in case of an array access). By using paths, signals can be established, that transmit information over the boundary of an entity. The signals contained in paths are represented on RTL as *signals* and *ports* (VHDL) or *regs/wires* and *ports* (Verilog).

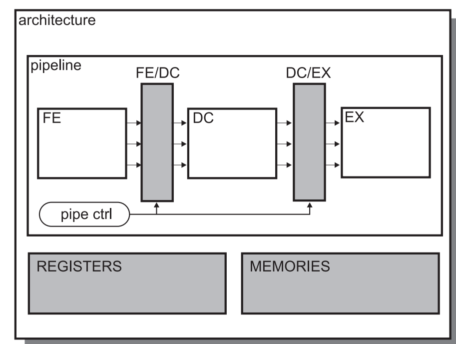

Figure 2. Example Architecture Structure

An example entity structure is shown in figure 2. Entities are represented as rectangles, units as round-cornered boxes and paths as arrows. For simplicity, only the pipeline controller unit is shown here as a representative unit.

Regarding the different components synthesized from LISA, the user registers, memories and pipeline registers are affected when generating the debug mechanism (indicated by the gray color in figure 2). These are the elements, which store the current processor state and thus must be accessible by the debug mechanism in two ways. On the one hand, the processor has to be halted in debug mode. Therefore, any write access to storage elements of the core has to be blocked. On the other hand, read and write access to these elements are necessary to determine and manipulate the state of the core.

Required changes to the storage elements of the core, namely user registers, pipeline registers and memories are discussed in the following subsections.

3.2 Registers with Debug Support

The structure of the register implementation is depicted in figure 3. This schematic describes the fundamental principle of the general register access and the debug functionality. In this example, we are assuming *n* read and write paths to a register file with *m* elements. The white boxes illustrate the regular processes required for the register implementation. The

write paths coming from functional units are routed to the particular element of the register file via a cross-connect. The synchronous register implementation leads to the read cross-connect, which routes the register values to the read paths.

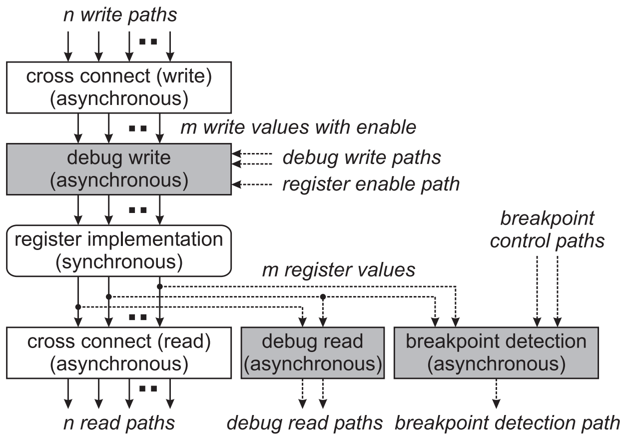

Figure 3. Register Implementation

The additional gray boxes are only necessary for the implementation of debug functionality. Those represent processes for setting a register value, reading a register value and detecting breakpoints. Additional paths are shown as dotted arrows.

The debug write process takes care of two functionalities required for debugging. First, it blocks the *enable* signals whenever the *register enable* is inactive. As a result, the register does not change its value anymore. The *register enable* path is a general path indicating whether the processor is halted and therefore has to maintain its current state. Similar functionality is implemented for all other storage elements of the architecture. Thus, it is possible to halt the complete core in its current state by setting the *register enable* inactive. Second, the register value can be overwritten by the debug write process. The value to be written and the according control signals are provided via debug write paths.

Through the debug read process the current register values are propagated via debug read paths. Note, that the debug read and write access is only generated for those register elements, which were configured with the GUI to provide this access. In figure 3, this is done for only two elements.

The detection of breakpoints is also only applied for registers selected with the GUI. Hardware breakpoints are divided into program and data breakpoints. Whenever a breakpoint is hit, the processor core is switched to debug mode.
Program breakpoints (sometimes also denoted as instruction breakpoints) are activated before the instruction associated with a pre-selected address, changes the state of the architecture. Thus, they are related to the program counter register.
Data breakpoints are related to any other user register. A data breakpoint is activated when the selected user register is written to. Moreover, data breakpoints may be value sensitive. This means the breakpoint is activated only if a certain predefined value is written.
In the breakpoint detection process, the register value is compared with the according breakpoint value provided via breakpoint control paths. In case of a match, the breakpoint hit is signaled via the breakpoint detection path. In order to detect a breakpoint correctly, the *write enable* also has to be taken into account, such that the breakpoint is only hit if the value has just been written. If the breakpoint is not value sensitive, the *write enable* is the only signal evaluated for the detection.

3.3 Pipeline Registers with Debug Support

The modifications to the pipeline register implementation in order to support debug functionality are very similar to those for user registers. The implementation only differs, because stalls and flushes have to be treated properly. A detailed discussion is omitted here, since this difference is not important for the implementation changes resulting from debug functionality.

3.4 Memories with Debug Support

The general idea of accessing memories in debug mode is equivalent to the mechanism for registers. The basic difference is, that in the case of register implementation the debug access can be included directly into the register implementation. Memories, being inseparable basic blocks with a limited number of ports, do not allow an embedded generation of debug mechanism like registers. Thus, the default read and write ports have to be used for debug access.

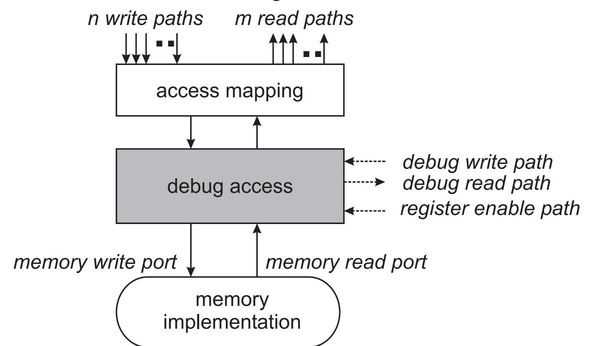

Figure 4. Memory Implementation

Figure 4 shows an example of a memory with one dedicated read and one dedicated write port. There are *n* write and *m* read paths connected to the memory. They are mapped to the existing memory ports in the access mapping process. The gray illustrated debug access process is used in two ways. By evaluating the *register enable* signal write accesses can be blocked in debug mode. This is necessary in order to maintain the current processor state. Additionally, the process bypasses a debug read and write access during debug mode.

3.5 Effort Estimation

As shown in the previous sections, the effort for changing the core implementation on RTL manually in order to support debug features cannot be neglected. Especially, an exploration of different configurations of debug functionality would be very time consuming, if changes to the register implementation and the implementation of access mapping for memories would have to be applied manually.

4 JTAG Interface Generation

We developed a debug mechanism, suited to architectures of various domains. Its features are accessed by sending debugging instructions through the JTAG interface. The standard compliant way to establish test features, which are accessible via the JTAG interface, makes use of vendor defined *test data registers*.

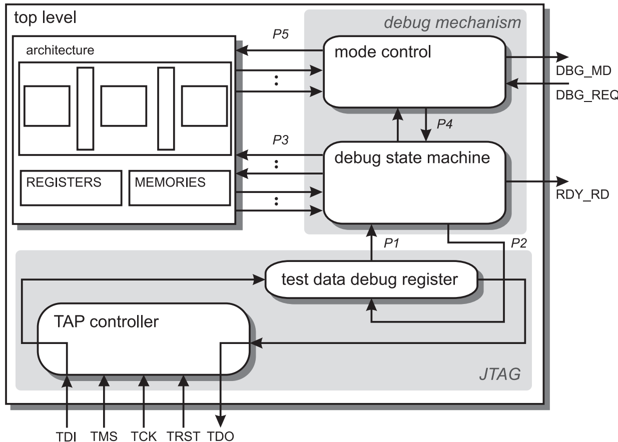

Figure 5. Extended Model Structure with Debug Mechanism

As shown in figure 5, the architecture entity is embedded in the top level entity, where the *TAP controller* and the *test data register* is also placed. In order to control the automatically generated debug mechanism, one *test data register* is instantiated. The generated JTAG interface and debug mechanism communicates (only) through this single register. Thus, the debug mechanism can also be generated without JTAG interface. In this case, the control of the debug interface is up to the designer. Due to this concept, our approach is highly flexible.

The *test data register* is the connection between the JTAG interface and the debug mechanism. A direct connection of the paths resulting from the core modifications to a *test data register* is hardly possible. Thus, special units have to be introduced in order to support debug functionality (*debug state machine* and *mode control*). They are connected between the *test data register* and the modified processor core. A detailed description is included in the next section.

The *TAP controller* writes the *test data debug register* serially in order to control the debug mechanism. Data, which is written, is stored left most in the register and data to be read is stored right most. By this, unnecessary shifting of the register is avoided. The *TAP controller* unit is mapped to a DesignWare component [22] during the synthesis process.

Our ASIP generation targets complex System-on-Chip (SoC) designs and the generated ASIP is only one part of the whole system. Thus, we are not dealing directly with pad cells. It is up to the designer to complete the implementation according to these requirements.

5 Debug Mechanism Generation

In figure 5, the additional units required to implement the debug mechanism are also shown.

The *debug state machine* accesses the *test data debug register*. It receives data words (like instructions, *P1*) and writes out the requested data words (such as a register value which has been read, *P2*) parallel. The *debug state machine* is the central element which implements the core functionality of the whole debug mechanism. It decodes and executes the debug instructions. Whenever requested data is stored in the *test data debug register*, it is indicated via the additional pin RDY_RD. This is required to establish a reasonable fast communication through the JTAG interface, since the interface itself is passive and not able to indicate events itself. The *debug state machine* is connected to the architecture with several paths as shown in figure 5 (*P3*). Via these paths, registers can be accessed (debug read and write paths) and breakpoint values are provided (breakpoint control paths). The mapping of the *debug state machine* unit to processes during the synthesis is strongly influenced by the configuration of the debug mechanism given via the GUI.

The *mode control* is a state machine to change the processor core from user mode to debug mode. The paths between the *mode control* and the *debug state machine* (*P4*) are used to indicate the current mode and to transmit mode changing instructions (e.g. single cycle execution) to the *mode control* unit. Two additional pins are directly connected to the *mode control*: DBG_REQ is dedicated to switch the processor to debug mode via interrupt; DBG_MD indicates the current mode of the processor to the outside. The paths connecting the *mode control* and the architecture (*P5*) are used to notify breakpoint hits (breakpoint detection paths) and to indicate a required halt of the processor (register enable path).

6 Results

The impact of a debug mechanism to the gate level synthesis results, such as clock speed and area, are discussed in this section.
The sample architecture is derived from the Motorola 6811 architecture [23]. This pipelined architecture with three pipeline stages and 16/32 bit instruction word is assembly compatible to the original architecture. All syntheses were done using a 0.18 μm technology. The area without debug mechanism (table 1, run 1) is 31.6 kGates, whereas an implementation with full debug support (table 1, run 2) takes 45.6 kGates. This is an increase of 44%. Full debug support means, that each of the 26 user registers can be written/ read while the processor is in debug mode and one breakpoint can be set to every register. Additionally, four program breakpoints can be set.
By analyzing the results, the major portion of area overhead is caused by the state machine and the extended register implementation according to section 3 and 5. The main area increase results from the fact, that breakpoints are implemented independently for each register and thereby introduce additional flipflops and comparators for each register.
Only five of the 26 registers are accessible directly using assembler instructions. The others are used for internal purposes

(e.g. for the state machine for division instructions). Thus, data breakpoints related to these registers are not required.
Consequently, in a third synthesis (table 1, run 3), we only added breakpoints and debug access to the five main registers. With this reasonably reduced amount of debug capabilities the required chip area for the complete architecture takes 36.4 kGates. This is only an increase of 15 % compared to the area used for the architecture without debug interface. The slight decrease of area consumption for the pipeline results from dynamic optimization criteria of the synthesis tool and range within its accuracy.

	area (kGates)					
	run 1		run 2		run 3	
architecture part	no debug capabilities		full debug capabilities		reduced debug capabilities	
total	31.6	100%	45.6	144%	36.4	115%
pipeline	20.4	100%	20.4	100%	19.7	97%
register file	11.1	100%	15.9	143%	12.1	109%
debug SM	-		7.7		3.1	
mode control	-		0.2		0.2	
test-data-reg	-		0.6		0.6	
TAP controller	-		0.6		0.6	

Table 1. Area Consumption for Three Configurations of the M68HC11

The implementation has been chosen in such a way that, potential critical paths are least affected. However, as can be seen in section 5 the debug-write features could not be fully removed from this critical path. Thus, there is a theoretical influence on the timing. Nevertheless, for this case study the deviations concerning timing range within the accuracy of the gate level synthesis tool. The data arrival time of the critical path amounted to about 4.5 ns. Thus, the values of the resulting maximum frequencies all are located in the range from 215 MHz to 225 MHz.

7 Conclusion and Future Work

In this paper we presented the first automatic generation of a JTAG interface and debug mechanism from an ADL. This generation is embedded into our RTL processor synthesis flow, which uses the language LISA to generate an RTL representation of the architecture. The JTAG interface is used as interface to the architecture and thus to the debug mechanism. The debug mechanism enables features like program and data breakpoints or register read and write access. The presented synthesis results clarified that debugging capabilities should be chosen judiciously, as they have a major effect on the chip area of the target architecture. Using our RTL processor synthesis tool, the designer is able to perform this selection during design space exploration.

In our future work we will investigate the suitability of our debug mechanism for architectures of various domains. Moreover, the results showed, that the current breakpoint implementation has a major impact on the required area consumption. Different improvements to the current implementation of breakpoints are being investigated. Their implementation will be addressed in our future work.

References

[1] G. Hadjiyiannis and S. Devadas. Techniques for Accurate Performance Evaluation in Architecture Exploration. *IEEE Transactions on VLSI Systems*, 2002.

[2] P. Mishra, A. Kejariwal, and N. Dutt. Rapid Exploration of Pipelined Processors through Automatic Generation of Synthesizable RTL Models Rapid System Prototyping. *Workshop on Rapid System Prototyping (RSP)*, 2003.

[3] V. Rajesh and R. Moona. Processor Modeling for Hardware Software Codesign. In *Int. Conf. on VLSI Design*, Jan. 1999.

[4] O. Schliebusch, A. Hoffmann, A. Nohl, G. Braun, and H. Meyr. Architecture Implementation Using the Machine Description Language LISA. In *Proc. of the ASPDAC/VLSI Design - Bangalore, India*, Jan. 2002.

[5] Schliebusch, O. and Chattopadhyay, A. and Steinert, M. and Braun, G. and Nohl, A. and Leupers, R. and Ascheid, G. and Meyr, H. RTL Processor Synthesis for Architecture Exploration and Implementation. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE) - Designers Forum*, Paris, France, Feb 2004.

[6] A. Hoffmann, H. Meyr, and R. Leupers. *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, 2002.

[7] M. Steinert, O. Schliebusch, and O. Zerres. Design Flow for Processor Development using SystemC. In *SNUG Europe Proceedings - Munich, Germany*, Mar. 2003.

[8] H. Scharwaechter, D. Kammler, A. Wieferink, M. Hohenauer, J. Zeng, K. Karuri, R. Leupers, G. Ascheid, and H. Meyr. ASIP Architecture Exploration for Efficient IPSec Encryption: A Case Study. In *Proc. of the Workshop on Software and Compilers for Embedded Systems (SCOPES)*, Sep. 2004.

[9] G. Hadjiyiannis and S. Hanono and S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. In *Proc. of the Design Automation Conference (DAC)*, Jun. 1997.

[10] A. Fauth and J. Van Praet and M. Freericks. Describing Instruction Set Processors Using nML. In *Proc. of the European Design and Test Conference (ED&TC)*, Mar. 1995.

[11] Target Compiler Technologies. *http://www.retarget.com*.

[12] A. Halambi and P. Grun and V. Ganesh and A. Khare and N. Dutt and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 1999.

[13] P. Paulin and C. Liem and T.C. May and S. Sutarwala. FlexWare: A Flexible Firmware Development Environment for Embedded Systems. In P. Marwedel and G. Goosens, editors, *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

[14] A. Kitajima and M. Itoh and J. Sato and A. Shiomi and Y. Takeuchi and M. Imai. Effectiveness of the ASIP Design System PEAS-III in Design of Pipelined Processors. In *Proc. of the Asia South Pacific Design Automation Conference (ASPDAC)*, Jan. 2001.

[15] ASIP Meister. http://www.eda-meister.org.

[16] R. Gonzales. Xtensa: A configurable and extensible processor. *IEEE Micro*, Mar. 2000.

[17] Tensilica. *http://www.tensilica.com*.

[18] Stretch. *http://www.stretchinc.com*.

[19] V. Kathail and S. Aditya and R. Schreiber and B.R. Rau and D. Cronquist and M. Sivaraman. Automatically Designing Custom Computers. *IEEE Computer*, 35(9):39–47, Sept. 2002.

[20] IEEE-ISTO. *The Nexus 5001 Forum, Standard for a Global Embedded Processor Debug Interface*, December 1999.

[21] IEEE, Inc., 3 Park Avenue, New York, NY 10016-5997, USA. *IEEE Std 1149.1-2001, Standard Test Access Port and Boundary-Scan Architecture*, June 2001.

[22] Synopsys. *DesignWare Components http://www.synopsys.com/products/designware/designware.html*.

[23] Motorola. *68HC11: Microcontroller http://www.motorola.com*.