Modeling Instruction Semantics in ADL Processor Descriptions for C Compiler Retargeting

Jianjiang Ceng, Weihua Sheng, Manuel Hohenauer,
Rainer Leupers, Gerd Ascheid, Heinrich Meyr
Aachen University of Technology (RWTH)
Integrated Signal Processing Systems
Aachen, Germany
and Gunnar Braun
CoWare, Inc.
Aachen Germany

Abstract. Today's Application Specific Instruction-set Processor (ASIP) design methodology often employs centralized Architecture Description Language (ADL) processor models, from which software tools, such as C compiler, assembler, linker, and instruction-set simulator, can be automatically generated. Among these tools, the C compiler is becoming more and more important. However, the generation of C compilers requires high-level architecture information rather than low-level details needed by simulator generation. This makes it particularly difficult to include different aspects of the target architecture into one single model, and meanwhile keeping consistency.
This paper presents a modeling style, which is able to capture high- and low-level architectural information at the same time and drives both the C compiler and the simulator generation without sacrificing the modeling flexibility. The proposed approach has been successfully applied to model a number of contemporary, real-world processor architectures.

1 INTRODUCTION

Today, application specific instruction-set processors (ASIPs) are used in a number of System-on-Chip (SoC) designs, because of their balance between computational efficiency and flexibility. Due to the diversity of the application domains that ASIPs are specialized in, one of the most important steps in designing ASIPs is architecture exploration, i.e. iteratively refining the architecture for the target application through exploiting different design space parameters such as instruction-set, pipeline structure, etc. This iterative approach demands that each time when the architecture is tuned, the software tools such as compiler, assembler, linker and simulator should be updated and available as soon as possible so that the tuning result can be examined to find out the potential improvement for the next iteration. For this reason, architecture description languages (ADLs) are developed to aid the design of ASIPs. From an ADL model, software tools

such as C compiler, assembler, linker, and instruction-set simulator, can be automatically generated, which significantly improves design efficiency.

However, the generation of C compilers has different requirements on instruction modeling than other tools, e.g. simulators, which makes it difficult to include different aspects of the architecture into a single model. Instruction-set simulators need the knowledge in detail about how the architecture executes an instruction, e.g. internal data manipulations, side effects calculation, cycle-accurate pipeline activities, etc. In contrast, C compilers view these instructions from a much higher, semantics-oriented abstraction level. They need to know the purpose of instructions rather than their execution in the architecture.

In this paper, we describe an extension of LISA 2.0 [13], a widespread industrial ADL, towards the modeling of instruction semantics for C compiler retargeting. The design of this extension aims at enabling the description of high-level instruction behavior with a minimum design effort. With this extension, embedded processor designers can generate a C compiler conveniently from a LISA processor model. Moreover, our approach helps not only the C compiler generation. In [6], we described the technique of simulator generation based on the work described in this paper and the related model consistency issues. Combined with the C compiler generation capability, our approach fulfills the demands for consistent tool generation from a single ADL model, and does not sacrifice flexibility. As the tool generation exceeds the scope of this paper, here we will focus on the new language constructs.

The rest of this paper is organized as follows: section 2 shortly discusses the approaches of related works. An overview of the LISA ASIP design methodology is given in section 3. Section 4 reviews several important principles in the LISA language. Section 5 describes in detail the design criteria and the extension of the language, which is the core of this paper. Section 6 presents the modeling results of several real-world processors. Finally, section 7 concludes the whole paper.

2 RELATED WORK

Within the last decade, a variety of ADLs has been developed to support ASIP design. However, not all of them support the generation of compilers. One important architecture-specific component in compiler is the code selector. It performs the task to translate the intermediate representation (IR) of the applications into assembly instructions. To generate a code-selector for a processor architecture, the knowledge of instruction *semantics*, i.e. what instructions do, is needed. Because most of the ADLs known today were originally designed to automate the generation of a specific embedded processor design tool, e.g. simulator, and later extended to other tools, different modeling styles were developed to support the generation of code-selectors. Based on the nature of the information provided, these ADLs can be classified into three categories: structural, behavioral and mixed.

The MIMOLA [5] language belongs to the structural ADL category. Its modeling style is similar to that of the VHDL hardware description language. The instruction set information is extracted from the register-transfer level (RTL) module netlist for use in code selector generation [16]. The software programming model of MIMOLA is an extension of PASCAL.

ISDL [9] is classified as behavioral ADL. It provides the means for hierarchical specification of instruction sets. During the code selector generation, the correlation between the target processor operations and the compiler basic operations comes from the behavior description of each instruction [11]. Because ISDL cannot model the structural details for pipelining, cycle accurate simulator generation is not possible.

nML [8] and EXPRESSION [10] are two mixed ADLs. nML was designed from its beginning to provide a formalism for instruction set modeling. The instruction set of the processor is described in a hierarchical style. The roots of the hierarchy are instructions, and the intermediate elements are partial instructions. Both instructions and partial instructions have *action* sections, which describe the behavior of instructions. Although nML is claimed as a behavioral/structural ADL, it lacks the capability of describing detailed micro-architecture structure, which limits the capability of simulator generation. The code-selection from EXPRESSION processor models relies on a so-called "Generic Machine" [1], which has a RISC instruction set architecture similar to that of the MIPS. *Operation-mapping* sections in EXPRESSION processor models are exclusively used to define the mapping between target processor instructions and one or more generic machine instructions on assembly level. The EXPRESS compiler first translates the input application to generic instructions, which are then replaced by target instructions.

The LISA 2.0 language [12] belongs to mixed behavioral/structural ADLs. The language allows the description of the micro-architecture behavior with arbitrary C/C++ code, which achieves high flexibility of modeling and very fast simulation speed for a broad range of contemporary RISC, VLIW, NPU, DSP, and ASIP architectures. In [14], we have described our overall LISA 2.0 based C compiler generation framework. Using ACE's CoSy system [2] as a backbone, it enables semi-automatic retargeting of C compilers from LISA processor models. A restriction of this earlier version is the need for manual interaction to specify the code-selector description in the compiler backend. The extension described in the following sections makes this manual interaction largely superfluous, and thus, permits to generate the code selector from a LISA model.

3 SYSTEM OVERVIEW

The work described in this paper is based on the *LISATek Processor Designer*, an embedded processor design platform available from CoWare, Inc [7]. The core of the platform is the LISA 2.0 ADL. It supports the automatic generation of efficient ASIP development tools such as instruction-set simulator, debugger/profiler, assembler, and linker. Furthermore, the platform also provides the

capability of generating VHDL, SystemC and Verilog hardware description language models for hardware synthesis. A retargetable C compiler, which is driven by the same ADL model used for the generation of other tools, has been recently seamlessly integrated into the platform, too. Because the problem of model inconsistency and the need to use various special-purpose description languages are avoided by using one single ADL model, this ADL-driven ASIP design approach can achieve very high design efficiency.

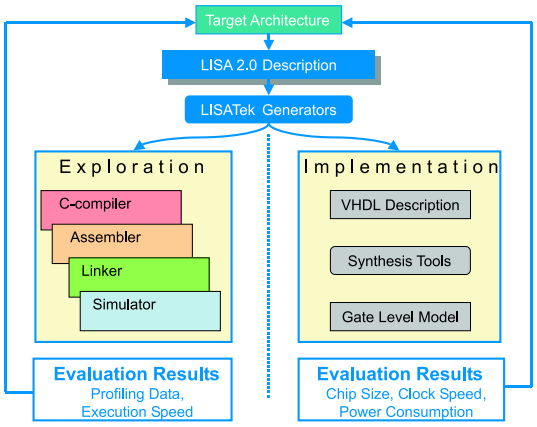

Fig. 1. LISA 2.0 Based ASIP Design Flow

4 LISA LANGUAGE

A LISA model basically consists of two parts: resource declarations and operations. Resource declarations define the processor resources like registers, buses, memories, pipelines, etc. The major part of a model consists of operations. `OPERATION` is the basic element of the instruction set description. The binary coding, assembly syntax, behavior and timing information are distributed over a number of operations, which are organized hierarchically. One of the advantages of the operation hierarchy is that the commonality of instructions can be easily exploited. Figure 2 shows three example operations `arithm`, `ADD` and `SUB`. Because `ADD` and `SUB` use the same type of operands, their operand fetching behavior is modelled in the operation `arithm` which belongs to the higher hierarchy. Their relationship is realized through the definition of GROUPs, whose members correspond to a list of alternative operations. In the example, the information about coding, timing and syntax is omitted for simplicity.

As mentioned in section 2, the LISA language allows arbitrary C/C++ descriptions of instruction behaviors, which is shown in the example. If a pipeline is modelled, this C/C++ instruction behavior description will be distributed over

```
OPERATION arithm IN pipe.ID{
    DECLARE{
        GROUP opcode = { ADD || SUB || ... };
        GROUP Rs1, Rs2, Rd = { reg_32 };
    }
    ...
    BEHAVIOR{
        PIPELINE_REGISTER(pipe, ID/EX).src1 = GPR[Rs1];
        PIPELINE_REGISTER(pipe, ID/EX).src2 = GPR[Rs2];
    }
}
OPERATION ADD IN pipe.EX{
    ...
    BEHAVIOR{
        int op1 = PIPELINE_REGISTER(pipe, ID/EX).src1;
        int op2 = PIPELINE_REGISTER(pipe, ID/EX).src2;
        PIPELINE_REGISTER(pipe, EX/WB).dst = op1+op2;
    }
}
OPERATION SUB IN pipe.EX{
    ...
    BEHAVIOR{
        int op1 = PIPELINE_REGISTER(pipe, ID/EX).src1;
        int op2 = PIPELINE_REGISTER(pipe, ID/EX).src2;
        PIPELINE_REGISTER(pipe, EX/WB).dst = op1-op2;
    }
}
```

Fig. 2. Operation Hierarchy

different pipeline stages. In the example, the BEHAVIOR sections of only two stages are shown, and the code inside does not model any feature like register bypassing, side-effects, etc. If these are modelled, the C/C++ behavior description will be much more complex than what is shown in the example. Taking this into account, it is nearly impossible to extract instruction semantics from the BEHAVIOR sections. For this reason we introduce the SEMANTICS section to the LISA language.

5 SEMANTICS SECTION

The SEMANTICS section is designed under certain criteria. Firstly, because SEMANTICS sections and BEHAVIOR sections both describe the behavior of instructions but from different perspectives, the new section should be concise so that the redundancy is reduced to minimum and the legacy LISA models can be easily extended to aid the compiler generation with few additional work. Moreover, the extension should be flexible so that a broad range of instruction set architectures can be described. For the purpose of compiler generation, ambiguity should be strictly avoided.

5.1 Micro-operations

After the examination of the instruction set architectures of a number of modern processors, we see that the high-level behavior of most of the instructions used

in these processors are normally either arithmetic calculations based on several operands or branches. The calculations carried out by the instructions can be further decomposed into one or several primitive operations, and the set of primitive operations is quite limited. Therefore, we model these primitive operations with so-called *micro-operation*s in the SEMANTICS section.

```
OPERATION ADD IN pipe.EX{
    DECLARE{
        GROUP Rs1, Rs2, Rd = { reg_32 };
    }
    ...
    SEMANTICS{
        _ADD|_C|(Rs1, Rs2)<0,32>->Rd;
    }
}
```

Fig. 3. An Example of SEMANTICS Section

Figure 3 shows the `ADD` operation of figure 2 with SEMANTICS section added. One statement is used to describe the semantics of an `ADD` instruction. The `_ADD` symbol is a micro-operator representing an integer addition. The following `_C` specifies that the carry flag is affected by the operation. `Rs1` and `Rs2` in the parentheses are the operands of the addition. They are GROUPs with one member, the `reg_32` operation. That means, the semantics of the operands is defined in the SEMANTICS section of `reg_32`. The `<0,32>` after the brackets explicitly specifies that the result of the addition is 32-bit wide and bit 0 is the first bit. Assignments in SEMANTICS sections are specified with `->` with the source on its left hand side and the destination on the right. Compared with the BEHAVIOR sections shown in figure 2, the description in SEMANTICS section is much simpler.

Generally, a micro-operation contains four parts, micro-operator, side-effects declaration, operands and bit-field specification. The operands of the micro-operators can be terminal elements, e.g., integer constants, `OPERATION`s, or other micro-operations. The constraint of the `OPERATION`s used as operands is that they must contain a SEMANTICS section. Side-effects in SEMANTICS sections are all predefined to avoid ambiguity. The behavior of four commonly used flags are provided, carry, zero, negative and overflow flags. If an instruction has any of the predefined side-effects, they can be declared by putting the corresponding shortcut after the micro-operator like what is shown in figure 3. The bit-field specification provides the bit-width and offset information of the micro-operations. It is compulsory for those micro-operations whose output bit-width cannot be deduced from their operands, such as sign/zero extension. Besides, allowing the use of micro-operations as the operands of other micro-operations is very useful in modeling complex operations. The next subsection will discuss the modeling of complex operations in detail.

5.2 Modeling Complex Operations

Many DSP processor architectures have instructions doing combined computation, e.g. multiply and accumulate. Such behavior is captured in SEMANTICS sections with *chaining*, i.e. using a micro-operation as the operand of another micro-operation. A simple example of a `MAC` operation can be found in figure 4. `_MULUU` is the micro-operator that denotes the unsigned multiplication. Its result is used as one of the operands of the `_ADD`, which forms a micro-operation chain. The chaining mechanism helps to describe some complex operations without introducing temporary variables, and keeps the statements in a tree-like structure. Such a structure is suitable for code selectors, because most of the code selection algorithms use the tree-pattern matching technique [4] [3].

```
OPERATION MAC{
    DECLARE{
        GROUP Rs1, Rs2, Rd = { reg_32 };
    }
    ...
    SEMANTICS{
        _ADD(_MULUU(Rs1, Rs2)<0,32>, Rd) -> Rd;
    }
}
```

Fig. 4. Micro-Operation Chaining

With *chaining*, most of the RISC instructions can be modelled with one statement, but obviously this is not enough for those instructions transferring data to multiple destinations. They are modelled with multiple statements in SEMANTICS sections. If a SEMANTICS section contains multiple statements, they are assumed to execute in parallel. That means, a preceding statement's result cannot be used as the input of the following statement. To illustrate this, an example is given in figure 5. The `SWP` operation swaps the content of a register by exchanging the higher and lower 16 bits. Because the execution is in parallel, the data in the register are exchanged safely without considering sequential overriding.

```
OPERATION SWP{
    DECLARE{
        GROUP Rs = { reg_32 };
    }
    ...
    SEMANTICS{
        Rs<0,16>->Rs<16,16>;
        Rs<16,16>->Rs<0,16>;
    }
}
```

Fig. 5. Multiple Statements

Another kind of important behaviors used in modern processors is conditional execution, i.e., an instruction is executed according to certain conditions. In order to model such instructions, `IF-ELSE` statements can be used. a total of 10 comparison operators can be used in SEMANTICS sections to model all kinds of conditions. Comparisons can be chained, too. In figure 6, the `_EQ` operator checks whether its two operands are equal or not. According to the result, the `IF` statement will execute the code specified in the braces.

```
OPERATION CADD{
    DECLARE{
        GROUP Rs1, Rs2, Rd = { reg_32 };
    }
    ...
    SEMANTICS{
        IF(_EQ(_CF,1)){ _ADD(Rs1, Rs2)->Rd; }
    }
}
```

Fig. 6. IF-ELSE statement

ASIPs often accelerate specific applications by using *intrinsics*, i.e. instructions performing specific operations frequently used in application source codes. With intrinsics, time consuming parts of the code can be executed fast by hardware support. The behavior of intrinsics is very heterogeneous, their C behavior code can vary from a few lines to several hundred. Such complex behavior is normally impossible to describe with micro-operations. Even if they would be modelled with primitive micro-operations, the C compiler could hardly use them. For this reason, intrinsics are viewed as special user defined micro-operations in SEMANTICS sections, as in figure 7. For simulator generation, the behavior of intrinsics must be specified explicitly in the BEHAVIOR section.

```
OPERATION DCT2d{
    DECLARE{
        GROUP Rs = { reg_32 };
    }
    ...
    SEMANTICS{
        "_DCT2d"(Rs);
    }
}
```

Fig. 7. Intrinsic Micro-operation

With the capability of defining intrinsics, almost all instructions can be described in SEMANTICS sections. Moreover, hierarchical description of instruction semantics is supported to achieve modeling flexibility and simplicity.

5.3 Semantics Hierarchy

In section 4 we have already illustrated the the LISA operation hierarchy. Taking advantage of hierarchical descriptions, SEMANTICS sections can be very concise. Figure 8 shows the hierarchical SEMANTICS sections of the three operations in figure 2. In the `arithm` operation, the GROUP `opcode` is used as micro-operator instead of predefined ones, which means that the concrete micro-operators can be found in the SEMANTICS sections of the GROUP members. Accordingly, the SEMANTICS sections of the `ADD` and `SUB` operation contain simply a micro-operator. The similarity of the `ADD` and `SUB` operations' semantics is well exploited here to simplify the description.

```
OPERATION arithm IN pipe.ID{
    DECLARE{
        GROUP Rs1, Rs2, Rd = { reg_32 };
        GROUP opcode = { ADD || SUB || ... };
    }
    ...
    SEMANTICS{ opcode|_C|(Rs1, Rs2)->Rd; }
}
OPERATION ADD IN pipe.EX{
    ...
    SEMANTICS{ _ADD; }
}
OPERATION SUB IN pipe.EX{
    ...
    SEMANTICS{ _SUB; }
}
```

Fig. 8. Hierarchical Operators

A SEMANTICS section can return not only a micro-operator but also a complete micro-operation expression. In figure 9, the SEMANTICS sections of the `SHL` and `SHR` operations do not contain a complete statement with assignment but micro-operators with operands (`_LSL` and `_LSR` are logical left and right shift micro-operators). As a result, the semantics of these two operations is not self-contained, because the data sink is missing. The use of these two operations is actually doing operand pre-processing for the `ADD` operation, which can be seen in its SEMANTICS section. The `opd` GROUP, which contains the previous two operations, is used as one of the operands of the `_ADD` micro-operation. Thereby, depending on the binary encoding of the instruction, one of the operand registers will be left or right shifted before addition.

In short, the formalism in SEMANTICS sections is very flexible and well integrated into LISA 2.0. If the commonalities of instructions are fully exploited, their instruction semantics can be described with only a few lines of code.

5.4 Difference Between SEMANTICS and BEHAVIOR

Though SEMANTICS and BEHAVIOR sections are similar in terms of describing the behavioral model of the processor, they are different in several ways:

```
OPERATION ADD IN pipe.EX{
    DECLARE{
        GROUP Rs1, Rd = { reg32 };
        GROUP opd = { SHL || SHR };
    }
    ...
    SEMANTICS{ _ADD(Rs1, opd)->Rd; }
}
OPERATION SHL IN pipe.EX{
    DECLARE{
        GROUP Rs2 = { reg_32 };
        GROUP imm = { imm8 };
    }
    ...
    SEMANTICS{ _LSL(Rs2, imm); }
}
OPERATION SHR IN pipe.EX{
    DECLARE{
        GROUP Rs2 = { reg_32 };
        GROUP imm = { imm8 };
    }
    ...
    SEMANTICS{ _LSR(Rs2, imm); }
}
```

Fig. 9. Hierarchical Operands

- **Contents:** in BEHAVIOR sections, C/C++ code can be used without limitation. However, in SEMANTICS sections, only a limited set of micro-operators (31 in total) are allowed, and their usages are fully formalized.
- **Operands:** in BEHAVIOR sections, nearly all processor resources and arbitrary variables can be used. However, in SEMANTICS sections, only compiler related resources can be accessed directly, e.g. registers, memories, etc.

6 CASE STUDIES

In order to prove the concept described in this paper, five architectures have been examined totally, namely ARM's ARM7 core, CoWare's LTRISC core, STMicroelectronics' ST220 VLIW multimedia processor [15], Infineon's PP32 network processing unit, and Texas Instruments' C54x digital signal processor. The LTRISC processor is a very small RISC core, which is provided with CoWare's LISATek Processor Designer. The PP32 is an evolution of [17] and comprises instructions which are able to operate on bit-fields. The existing LISA 2.0 models of these five processors have been extended with SEMANTICS sections. Although the SEMANTICS section is not intended for the extension of already existing models, our test approach proved that the new section does not impose any particular modeling style. This is very important to keep the flexibility of LISA 2.0.

The modeling result is summarized in table 1. Note that the design effort is calculated in man-days. It can be seen that the work for adding SEMANTICS sections scales with the number of instructions in the architecture. The complexity of the instructions (RISC vs. CISC) also has influence on the effort.

	ARM7	LTRISC	ST220	PP32	C54x
ISA	RISC	RISC	RISC	RISC	CISC
No. instructions	62	17	82	41	110
Design effort	4d	2d	10d	8d	15d

Table 1. Modeling Results of Five Processors

Generally, the predefined micro-operations and the bit-field specification make the behavioral description code size in SEMANTICS sections significantly less than that of the C code in BEHAVIOR sections. Furthermore, the ambiguity of the C description is avoided, which is important for C compiler generation.

7 CONCLUSION

In this paper, we presented an approach for modeling instruction semantics based on an existing ADL with the main purpose of C compiler generation. Our approach incorporates a new SEMANTICS section into the structure of LISA 2.0 without influencing the existing flexibility, and achieves a concise formalism for instruction-set description which is important for code selector generation. Besides providing instruction semantics for C compiler generation, it is also possible to generate instruction-set simulator and documentation with the information provided by SEMANTICS sections.

A further interesting result of our approach is that both instruction- and cycle-accurate descriptions of the processor architecture are able to coexist in a single model. This allows for a very high design efficiency on different abstraction levels, while maintaining consistency through using a single model during the entire design process. Our future research activities will be in the area of processor specific code optimization based on instruction semantics.

References

1. *EXPRESSION User Manual (version 1.0)*
   http://www.ics.uci.edu/~express/documentation.htm.
2. ACE – Associated Computer Experts bv. *The COSY Compiler Development System*
   http://www.ace.nl.
3. A. Aho, M. Ganapathi, and S. Tjiang. Code generation using tree matching and dynamic programming. *IEEE Transactions on Programming Languages and Systems*, 11(4):491–516, Oct. 1989.
4. A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools.* Addison-Wesley, Jan. 1986. ISBN 0-2011-0088-6.
5. S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer. The MIMOLA Language, Version 4.1. Reference Manual, Department of Computer Science 12, Embedded System Design and Didactics of Computer Science, 1994.

6. G. Braun, R. Leupers, G. Asheid, and H. Meyr. A Novel Approach for Flexible and Consistent ADL-driven ASIP Design. *Proc. of the Design Automation Conference (DAC)*, Mar. 2004.
7. CoWare Inc., http://www.coware.com. *LISATek product family.*
8. A. Fauth, 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.
9. G. Hadjiyiannis, 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. Halambi, P. Grun, V. Ganesh, A. Khare, 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.
11. Silvina Hanono and Srinivas Devadas. Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator. In *Design Automation Conference*, pages 510–515, 1998.
12. A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr. A Novel Methodology for the Design of Application Specific Instruction Set Processors (ASIP) Using a Machine Description Language. *IEEE Transactions on Computer-Aided Design*, 20(11):1338–1354, Nov. 2001.
13. A. Hoffmann, R. Leupers, and H. Meyr. *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, Boston, Jan. 2003. ISBN 1-4020-7338-0.
14. M. Hohenauer, H. Scharwaechter, K. Karuri, O. Wahlen, T. Kogel, R. Leupers, G. Ascheid, and H. Meyr. A Methodology and Tool Suite for C Compiler Generation from ADL Processor Models. *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 2004.
15. F. Homewood and P. Faraboschi. ST200: A VLIW Architecture for Media-Oriented Applications. In *Microprocessor Forum*, Oct. 2000.
16. R. Leupers and P. Marwedel. A BDD-based frontend for retargetable compilers. In *Proc. of the European Design and Test Conference (ED & TC)*, pages 239–243, 1995.
17. X. Nie, L. Gazsi, F. Engel, and G. Fettweis. A new network processor architecture for high-speed communications. In *Proc. of the IEEE Workshop on Signal Processing Systems (SIPS)*, pages 548–557, Oct. 1999.