Retargetable Tools for Embedded Software

Gabriele Luculli
Advanced System Technology
STMicroelectronics
Grenoble, France
gabriele.luculli@st.com

Abstract—**Today, the advent of the multi-millions transistor chip offers the opportunity to provide more integrated functionalities into a single silicon die. This is both a clear advantage to improve existing products and an hard challenge to cope with. Unfortunately, the today available design methodologies and tools are not sufficiently advanced to exploit the full potential of such large silicon flexibility. On one hand the design of complex system-on-chip requires new tools for the analysis and optimization of embedded software, which should extend the standard tool chain consisting of compiler, debugger and simulator. On the other hand the development of such tool chain for a new microprocessor requires a substantial time effort which is no more compatible with actual market dynamics. In STMicroelectronics we developed a technology which tries to tackle such double-face problem by adding several retagetability features to a standard tool chain and by providing new tools for code analysis.**

I. Introduction

In the past years, several approaches on tool chain retargetability have been explored. They all share the same basic idea of exploiting an abstract processor specification to generate automatically a tool chain. However, they differ a lot on the characteristics of the used specification language and, hence, on the automation process used for tool generation. Moreover, most of them just focus on retargetability issues of the instruction-set simulator (ISS) rather than considering a complete tool chain.

The approach proposed in [7] is probably one of the most interesting. They use the machine description language LISA to generate an almost complete tool chain consisting of assembler, linker, debugger and simulator. In particular, they describe their technology for compiled simulation providing wide evidence of the state-of-art performance of the simulators they can generate. Due to several open issues in the generation of production-quality code using customized compilers, they did not implement a retargetable compiler back-end yet, but this could eventually happen in the future as research on retargetable compilers will advance. A quite similar approach is described by Jeremiassen [3], even if he just focuses on simulation technology.

Many specification languages for instruction-set architectures are currently available in literature. Some of them just provide simple constructs to label the different parts of a specification. So they don't support any kind of semantic check for its validation. Unfortunately, LISA belongs to such class of languages. Others formalisms, like ISDL [2], provide weak forms of correctness checks. Finally, the most complex ones make use of classic language-theoretic approaches, such as attribute grammar, to validate most of the formal properties which can be attributed to this kind of specification: completeness, consistency, unambiguity and, when it is feasible, correctness. The nML formalism is probably the best known of such class [1]. Its main characteristic is that the semantic actions of the instructions are composed of fragments that are distributed over the whole grammar tree. The SLED language belongs to such class of formal languages as well, but it does not address any issue related to the description of instructions semantic, only the instructions encoding may be described [8].

In this paper we propose an overview of the retargetable technology we developed in STMicroelectronics. The main retargetability feature we consider refers to the instruction-set architeture (ISA) of a microprocessor. We take advantage of a SLED specification [8] of the microprocessor encoding to automatically generate a partial tool chain which consists of disassembler, code analyzer and extended simulator. These are the basic tools usually required for the analysis and optimization of embedded software. The focus of this paper is intentionally on the role of a single formal specification as key enabling factor for the automatic retargetability of the overall toolset. The reported results are innovative at least for:

- The successful development of a fully retargetable framework which meets the requirements of each specific tool for potentially any instruction set architecture.
- The leverage of a formal specification to generate automatically a software tool chain, rather than using a code-cut-and-paste approach [7], [3].

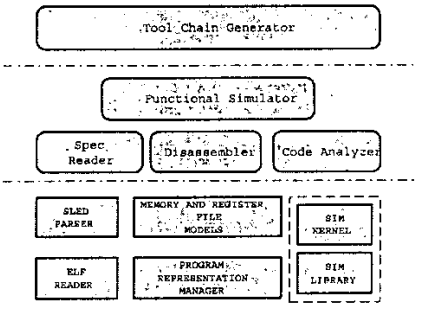

Fig. 1. Global software architecture of our ISA-retargetable environment

The rest of the paper is organized as follows. Section 2 starts with an introduction to our implementation. Then, a description of the main requirements for ISA retargetability is reported. Finally, major details about our retargetable decoder technology are provided, together with its generalization as ISA *internal representation* for any tool of our tool chain. Section 3 concludes the paper.

II. Retargetable Tool Chain

The overall software architecture of our retargetable framework is depicted in Figure 1. It consists of three layers of software which incrementally add up to provide different views of the same basic technology to different users. In the lowest layer we implemented the *building blocks* which are common to one or more tools of the tool chain. In the second one, by composing several building blocks of the previous layer, we provide the real *ISA-retargetable tool chain* which consists of a reader of processor specs, a disassembler, a code analyzer and a functional simulator. In the last one we implemented a *software generator* which is able to generate automatically a *custom tool chain* for the given ISA spec. Obviously, the main difference between the last two layers is that, whether the processor description is used by any retargetable tool to adapt its functionality at run-time to the specified processor, the software generator uses the same description to generate a version of the tool chain (i.e. a *retargeted tool chain*) which is customized for the specific processor and, eventually, customizable for additional require-

0-7803-7763-X/03/$17.00 ©2003 IEEE

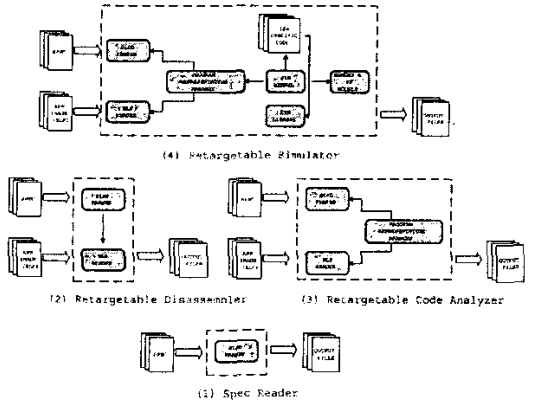

Fig. 2. Structural implementation of the retargetable tools set

ments.

It is interesting to note that the adoption of a hierarchical software architecture was not only necessary to master the complexity of the development but, much more important in our case, it is also useful to provide different functionalities to different users in an efficient way. In fact, today semiconductor companies heavily rely on the availability of a tool chain for most of the activities which are related to microprocessors. First of all, the design of the microprocessor itself requires the definition of both an effective instruction set and the necessary architectural features to sustain any expected workload. The adoption of a formal specification for the instructions encoding permits to explore several alternative designs of the ISA in a quite short time. Different encodings result in very different structures of the decoder which, as a matter of fact, directly limits the processor performance. So the availability of a formal specification language for ISA encoding and its related parser, as our SLED reader in Figure 1, are surely relevant for such activity. Then, any processor architect has to decide which instructions to add to the instruction set. The availability of a technology for the fast adaptation of an ISS to an updated ISA, like it is practically our retargetable ISS, is essential for this. In addition there are many other activities, still related to the processor architecture definition, implementation and validation, which require several sort of simulation capabilities. Most of the tools of our second layer can be easily enhanced to fulfill them. As well as, they can be improved to satisfy the requirements of any team involved in the integration of microprocessors into a sub-system or system-on-chip. The second layer of our architecture is also interesting for compiler developers. One of their main goal is to define the best strategy for code generation, they usually need retargetable disassemblers and simulators for the fast validation of such strategies. As we previously remarked, the teams which are involved in embedded software optimization require new advanced tool chains to gain a complete system-level view of the hardware effects. The retargeted tools of the third layer have them as important users. Last but not least, semiconductor companies usually provide the tool chain as an essential add-on to their microprocessors customers. The automatically generated tools of our third layer may permit the deployment of solutions which fulfill both quality-of-product and eventual customer-specific requirements without severely impacting the overall cost of development.

A. Tool Chain Requirements

The tools which compose our ISA-retargetable tool chain have obviously different requirements. They are here shortly described.

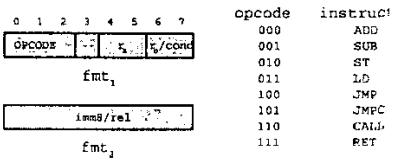

Fig. 3. Encoding formats of the SIMPLE processor

The basic functionality of the *disassembler*, which is the simplest tool, is to recognize the instructions of the input binary file and to print them in output using the correct assembler syntax. The main difference of a retargetable disassembler vs. a classic disassembler is that the encoding format of the instruction set and the assembler syntax are fixed for the latter whereas they are variable for us. In order to grasp the meaning of such variability, a formal specification is required.

The *code analyzer* has, as main functionality, the recovery of the program structure in term of global control-flow graph, local control-flow graph to each procedure and call graph. Classic tool chains don't usually include such kind of code analyzer because it is not useful to design general-purpose software: code implementation and functional verification are traditionally done by a try-and-fix approach using a debugger. The situation is quite different for embedded software which requires a substantial performance tuning for each specific hardware platform. Following the simulation, software engineers make use of the code analyzer to identify the potential performance bottlenecks inside the software structure so that they can hopefully re-write the source code to enhance its performance. It is important to note that general-purpose code profilers don't provide such functionality because they just annotate the program structure with information about the execution frequency and not the hardware metrics. Neither it is done by typical simulators because, even if they produce an output report with performance or power metrics, they just furnish their global figures and not their local decomposition inside the program structure.

Finally, the code analyzer is also useful to understand how the compiler changed the structure of the source code. In contrast to the general-purpose case where there are few compiler options to drive the code optimization, embedded compilers provide many options and pragma directives which can drastically change how the code is generated (e.g. loop unrolling or call-return convention for procedures) and where static/dynamic data are allocated in main memory. Our code analyzer is clearly a very useful tool to visualize such changes.

From an implementation point of view, the code analyzer includes all the functionalities of the disassembler and it extends them with several algorithms (quite standard) that try to recover the code structure from its binary files. Since such algorithms partially depend on the instruction set, it was necessary their generalization so that they could work for any generic instruction set.

The most complex tool of the overall chain is the *functional simulator*. It has the responsability of simulating the execution of the application on the target hardware architecture, taking into account the target ISA as well as the additional hardware models which are present. Usually, the main output is a text report which contains global figures about several performance metrics such as the code execution time or the cache miss count. In the past, when code optimization was mainly a compiler task, such global information was sufficient to identify the useful compiler options to adopt. Today, as we already said, the strong optimization of code is became a manual activity of many software engineers. The evaluation of performance metrics is still required but, in addition, they need to identify how each part of the program contributes to such global metrics. A simulator which is able to provide the local decomposition of the global metrics into the program structure seems to be the ideal support required by them. Accepting such vision, our

```
token FIRSTBYTE (8),
  fields
    opc 0:2 imm 3:3 ra 4:5 rb 6:7 cond 6:7

  fieldinfo cond is [names [EQ NE LT GT]]
  fieldinfo [ra rb] is [names [R0 R1 R2 R3]]

patterns
  [ADD SUB ST LD JMP JMPC CALL RET] is opc=[0 to 7]

  arith is ADD | SUB
  ldst  is LD  | ST

constructors
  arith Ra, Rb is arith & Ra=ra & Rb=rb
  ldst Ra, [Rb] is ldst & Ra=ra & Rb=rb

  JMP [Ra] is JMP & Ra=ra
  JMP^COND [Ra] is JMPC & Ra=ra & COND=cond

  CALL Ra is CALL & Ra=ra
  RET is RET
```

Fig. 4. SLED specification of the SIMPLE processor (fmt_1)

```
token SECBYTE (8),
  fields
    imm8 0:7 rel 0:7

constructors
  arith Ra, Rb is arith & imm=0 & Ra=ra & Rb=rb
  arith Ra, #IMM8 is arith & imm=1 & Ra=ra; IMM8=imm8

  JMP [Ra] is JMP & imm=0 & Ra=ra
  JMP TARGET { TARGET=PC+2+REL } is JMP & imm=1; REL=rel1

  JMP^COND [Ra] is JMPC & Ra=ra & COND=cond
  JMP^COND TARGET { TARGET=PC+2+REL } is JMPC & imm=1 & COND=cond; REL=rel1
```

Fig. 5. Update of the SLED specification of SIMPLE processor (fmt_2)

```
token FIRSTTOK (7),
  fields
    opc1 0:1 ra 3:4 rb 5:6 cond 5:6

token SECTOK(1),
  fields
    opc0 0:0

token FIRSTBYTE(8),
  fields
    opc 0:2 ra0 4:5

fieldinfo cond is [names [EQ NE LT GT]]
fieldinfo [ra rb] is [names [R0 R1 R2 R3]]

patterns
  [ADD SUB]  is opc0=[0 to 1]
  [ST LD]    is opc0=[0 to 1]
  [JMP JMPC] is opc0=[0 to 1]
  [CALL RET] is opc =[6 to 7]

  [arith ldst jmp] is opc1=[0 to 2]

constructors
  ADD Ra, Rb   is arith & Ra=ra & Rb=rb; ADD
  SUB Ra, Rb   is arith & Ra=ra & Rb=rb; SUB
  LD  Ra, [Rb] is ldst  & Ra=ra & Rb=rb; LD
  ST  Ra, [Rb] is ldst  & Ra=ra & Rb=rb; ST

  JMP [Ra] is jmp & Ra=ra; JMP
  JMP^COND [Ra] is jmp & Ra=ra & COND=cond; JMPC

  CALL Ra is CALL & Ra=ra0
  RET is RET

classes for CPG
  jmp  is JMP
  jmpc is JMP^COND
  call is CALL
  ret  is RET
  others is ADD | SUB | LD | ST
```

Fig. 6. SLED specification with three-tokens decomposition for the SIMPLE processor

extended simulator should include all the functionalities of a classical simulator, the functionalities of the code analyzer and some kind of back-annotation mechanism to update the program structure, recovered by the analyzer, with local values of the chosen metrics.

A completely different problem is how to extend such simulator for ISA retargetability. There is just a single functionality which needs to be generalized, basically how to implement the semantic of a generic instruction set. So it could seem an easy affordable task. Unfortunately, the development of any ISA simulator always raises the issue of having state-of-art simulation speed. The development of a retargetable simulator makes the picture even worse because it opens the question of how to generalize for retargetability without loosing simulation performance or, at least, loosing the minimum necessary amount. We partially answered to such question in a previous research [6], where we showed that the necessary element for an efficient retargetable simulator is the adoption of a simulation library whose execution times are optimally balanced for any possible workload. We will not develop further such arguments in this paper.

The last tool is the *spec reader*. Its functionalities are simply to parse and perform all the relevant correctness checks on a given spec. Clearly, such reader is mainly used to validate a new spec during its development.

B. Role of the SLED Specification Language

A possible manner of adding ISA retargetability to a whatever tool is to generalize its software structure in such a way that it can automatically adapt at run-time to a new configuration of the instruction set. This can be done basically by abstracting all the common features of any instruction set into a *specification language* and using it to build up, at run-time, the *internal representation* of the instruction set which is currently described. The description of a specific instruction set in the specification language is mainly the process of gathering all the useful information in a formal and ordered manner. A retargeted tool should use the internal representation as the original tool used the information about the specific instruction set.

Following a review of the available languages for ISA specification, we decided to use the SLED language as driver of our retargetable environment. There are several unique advantages we envisioned in its use:

- *The language is based on a set of syntactic objects (i.e. tokens, fields, patterns and constructors) which well adapt to represent the content of ISA architectural manuals. Shortly, the token is the basic unit of processing of the language and it is characterized by its length in bits. Different tokens relate to different instruction lengths. A field is a contiguous range of bits within a token, so each token is partitioned into several fields. Fields contain opcodes, operands, modes or other information about ISA encoding. Patterns are used to represent the values which can be contained by fields in a single token or in a sequence of tokens. So, in some ways, they constrain the values of fields. Finally, constructors represent the encoding of a class of instructions by connecting a set of fields with their values and patterns, actually they specify a mapping between fields and the values space. Each constructor is also associated with a string which is the assembly-language syntax of the recognized instruction. A very simple example of SLED specification is reported in Figure 4 for the first instruction format of Figure 3.*
- *The language is well suitable for different kinds of processor architectures. Examples of specification of RISC and CISC machines were already available in literature few years ago [8]. Recently, we developed the first SLED specification of a VLIW architecture, the ST200 embedded processor of STMicroelectronics [4].*
- *Since SLED formally addresses the encoding structure, it can be efficiently used as common reference model across different teams inside a company for what concern the ISA specification. This can seem trivial but it is not indeed. In general, it is quite difficult to find a unique model which well adapts to the needs of different teams because they are used to expose different views or partial aspects of the same instruction set. E.g. people working on the definition of a new microprocessor start by defining just few basic instructions for the expected data path and then, if required, they split some of them into several ones or simply add new classes. So, it is essential for them the capability of defining a specification in an incremental way but without the hurdle of using an exhaustive description. People working on compilers development are in a opposite situation in some ways. It may happen they need to evaluate if it is useful to enrich the instruction set with few*

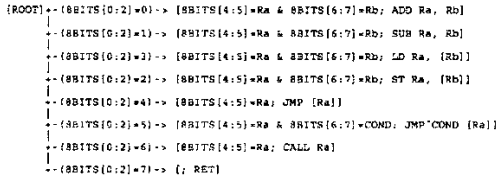

Fig. 7. Decoding tree derived from the specification of Figure 4 (fmt_1)

application-specific instructions in order to improve the quality of the generated code. This time the simple capability of adding a flat specification of the additional instructions without the need of rearranging the overall specification is what they mainly require. SLED easily supports this kind of needs. An example of incremental extension of the SIMPLE encoding for the second format is available in Figure 5.

- The language constructs support the specification of the same ISA at different levels of granularity, starting from a very concise description up to the exhaustive one. So the complete structure of a class of instructions is exposed just if it is really useful for some purposes. Moreover, the same ISA can be described by several specs which provide different encoding decompositions of the same input bits stream but, which are functionally equivalent from the decoding perspective. For example, Figure 6 report a three-tokens specification of the SIMPLE processor which is functionally equivalent to the one of Figure 4.

- Finally, any new microprocessor requires several sorts of technical documentation. Its writing-up is a time consuming activity that not directly impact the quality of the product itself but it can seriously impair its usability. Even if we did not implemented this feature, it is clearly feasible to use any SLED specification to automatically derive a skeleton of ISA architectural manual.

C. Technology of the Retargetable Decoder

As we suggested in a previous section, and as it is clearly exposed in Figure 2, there is an important functionality which is common to any tool. That is the retargetable decoder. We implemented an algorithm that starts from a correct specification in SLED, builds (as *internal representation* for decoding purpose) an optimized decoding tree like the one reported in Figure 7 for the specification of Figure 4. In short, each node and each edge of the tree is associated to a set of bindings between a field (or a sequences of fields) and its actual value in the binary code. The edge bindings are equivalences that have to hold true in order to proceed the decoding in the following edges. While on the contrary the node bindings are value assignments that must be performed by the decoder in order to recover the actual values of the fields which compose the recognized instruction. In addition, leaf nodes report the complete assembler-like syntax of the recognized instruction.

There are two main opportunities to optimize this kind of tree structure:

- Two or more contiguous fields, which should be normally checked for value equivalence in two different steps, can be concatenated into a single field in the decoding tree and then evaluated in a single step. An obvious example is the register-immediate ADD instruction in Figure 8, where the concatenation of the fields `imm` and `opc` is checked for their joint value (i.e. `8BITS[0:3] = 8`).

- Hierarchical encoding formats usually result on decoding

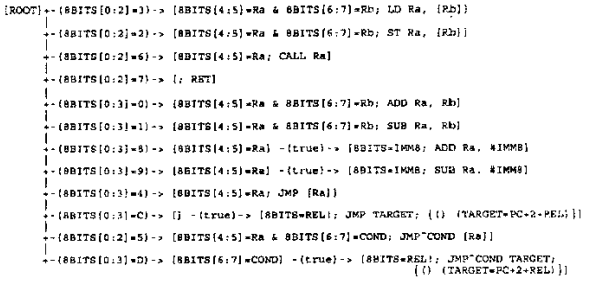

Fig. 8. Decoding tree derived from the specs of Figure 4-5 (fmt_1, fmt_2)

trees with several equivalent edges. Two edges are equivalent if they have associated the same bindings field-value, they have the same source node and their sink nodes have the same set of value assignments. In this case, the two edges can be reduced to just one as in the example of Figure 9 for the ADD/SUB and LD/ST instructions. Please, note as the fulfillment of only the first two conditions is not sufficient to establish the equivalence; all the three conditions are necessary. As example, the decoding tree of Figure 9 has two separated edges for the instructions JMP and conditional JMP, otherwise the decoding process should not result correct.

It should be quite obvious that we adopted as main criteria of optimization the minimization of the number of decoding steps, assuming each step has the same cost. This is reasonable in our context because the decoding time severely impacts the performance of interpretive simulators, where each instruction is decoded each time that it is simulated. Instead, it is completely negligible for both the disassembler and the code analyzer. Different criterias can be easily implemented to fulfill other kinds of optimizations. For example, to improve further the simulation time we could generate a decoding tree where the edges are ordered with respect to the statistics of instructions usage. Or, the optimized generation of a VHDL decoder implementation could result from adopting non-uniform costs for the tree edges. We didn't explore such alternatives yet, but they are clearly feasible.

D. ISA Internal Representations

Since ever decoding trees are a suitable representation of the ISA encodings, when the purpose is the decoding of an input bits stream of instructions. So it was quite obvious to adopt a sort of generalized decoding tree, like the ones we reported, as *internal representation* of the ISA for our retargetable decoder. What was not obvious is the discovery that it is possible to enhance such generalized decoding tree with additional information in order to obtain a data structure which is suitable as internal representation of the overall set of retargetable tools we developed.

For implementing the ISA retargetability feature in the code analyzer, it was required to generalize any of the already available algorithms of code analysis [5] in order to cope with a generic instruction set. This has been quite easy to achieve because all such algorithms are based on the manipulation of some sort of control-flow graph (CFG). So, following the recovery of the global control-flow graph of the application, they can be applied for free. On the other hand, to generalize the identification of the basic blocks we found that it is sufficient to associate an attribute to each instruction class. Such attribute distinguishes the instructions which can potentially changes the flow of control from the others (i.e. jump, conditional jump, procedure call, procedure return). According to this, we extended the SLED language with a new construct `classes for CFG`, as shown in Figure 6, which follows the SLED syntax and it applies to any constructor. Thus, each leaf node

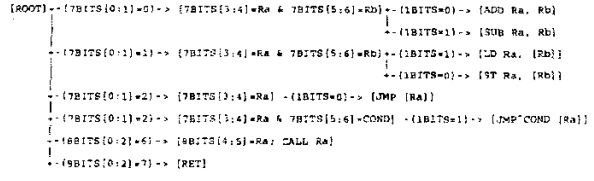

Fig. 9. Decoding tree for the three-tokens specification of Figure 6

of our decoding trees has associated such additional attribute and it is exactly this attribute which drives our retargetable code analyzer at runtime. We know that such extension is not general enough to apply to any ISA, in fact it doesn't cover the case of e.g. a call instruction which is implemented as a sequence of instructions terminating by a jump (this kind of code is quite common on some embedded processors). Anyway, such limitation is just aesthetic. It can be easily removed by generalizing the added construct to sequences of instructions and extending the related algorithm for basic blocks recognition.

Additional information were required also for the simulator retargetability. There is a large spectrum of ISA simulation technology which spans between the two main cornerstones of compiled and interpretive simulation. Depending on the chosen approach the implementation has quite different characteristics, different principles are exploited to gain simulation performance and different information have to be recovered in the binary files. Due to space limitations, we will not describe such details for our retargetable simulator in this paper. Anyway, we would like to summarize here its main properties:

- It is a mix of compiled and interpretive simulation technology. Instructions are decoded just one time, as in compiled simulation, but then they are executed like in interpretive simulation (i.e. there is a single behavioral code for each instruction class).
- The complete internal representation of the instructions for simulation purpose is quite complex. We did several experiments to define the best tradeoff between the complexity of the internal representation and the simulation performance, here we summarize the main results. No extension of the SLED language is required to support our retargetable simulation technology. However, it has been essential to extend the decoding tree in order to classify the instruction operands among several classes: statically-known in the spec, statically-known in the binary file, dynamically-determined at decoding-time and dynamically-determined at run-time. When the decoding tree is built, such attribute is associated to any instruction operand which is present in any node. At run-time, the operands classes are utilized to generate an optimized skeleton for each instruction class.

III. Conclusions

We have presented an overview of our ISA retargetable technology. In particular, several facts related to the SLED specification language have been described: first of all, the reasons of its choice as main driver of ISA retargetability; then, how a single internal representation, that matches the requirements of the many retargetable tools, can be derived by a single SLED spec. A single SLED specification of the instruction-set architecture is sufficient for the automatic customization of a retargetable tool chain. Moreover, the quality of the tools is comparable to full custom implementations.

We feel that the use of automatic retargetable frameworks is already a major step to tackle the time-to-market pressure for microprocessor-oriented products, and we believe that future research on the automatic generation of behavioral code for functional simulation will permit further important improvements.

References

[1] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. *IEEE Proc. of the European Design and Test Conference*, pages 503–507, 1995.

[2] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. ISDL: An instruction set description language for retargetability. *IEEE Proc. of the Design Automation Conference*, pages 299–302, 1997.

[3] Tor E. Jeremiassen. Sleipnir – an instruction-level simulator generator. In *Proceedings of the 2000 IEEE International Conference on Computer Design: VLSI in Computers & Processors*. IEEE, 2000.

[4] Gabriele Luculli. An ISA retargetable simulator for embedded software analysis. *IEEE Proc. of the 10th Annual International Conference and Workshop on the Engineering of Computer-Based Systems*, April 2003.

[5] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, 1997.

[6] Francesco Papariello and Gabriele Luculli. Optimization of a retargetable functional simulator for embedded processors. *IEEE Proc. of the 9th Annual International Conference and Workshop on the Engineering of Computer-Based Systems*, April 2002.

[7] Stefan Pees, Andreas Hoffmann, and Heinrich Meyr. Retargetable compiled simulation of embedded processors using a machine description language. *ACM Transactions on Design Automation of Electronic Systems*, 5(4):815–834, January 2000.

[8] Norman Ramsey and Mary F. Fernández. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524, May 1997.