Modeling and Simulation of Embedded Processors Using Abstract State Machines

Dirk Fischer, Jürgen Teich, Ralph Weper

Electrical Engineering Department
University of Paderborn
D-33098 Paderborn, Germany
email: {fischer,teich,weper}@date.uni-paderborn.de

ABSTRACT

We demonstrate how application specific instruction set processors (ASIPs) may be systematically described using the formalism of Abstract State Machines (ASM) as introduced by Gurevich [10]. We developed a modeling environment, called *ArchitectureComposer*, which automatically generates an ASM model from the graphical input of an architecture's description on register transfer level. *ArchitectureComposer* emits XASM [1] code which in turn is processed by the tool Gem-Mex [3] which automatically generates a cycle-accurate simulator for the specified architecture.

This framework is part of the project BUILDABONG at the University of Paderborn dealing with joined architecture/compiler co-generation for application specific instruction set processors (ASIPs).

Here, we focus on the underlying mathematical model of ASMs and introduce the applied tools, e.g. *ArchitectureComposer*, in particular the features of the graphical architecture editor and the automatic XASM-code generator, and briefly describe how these tools are embedded in the BUILDABONG project.

1. INTRODUCTION

Special application domains such as digital signal processing require the design of special instruction set processors (ASIPs) in order to fulfill efficiency, cost, and other design criteria.

The final architecture of an ASIP is not a priori fixed but is the result of an iterative process of prototyping. Thereby, the simulation of a preliminary architecture design is of utmost importance.

In this paper, we introduce a methodology for the automatic generation of efficient yet cycle-accurate and bit-true simulators from graphical entry of processor building blocks.

In the first step, a processor's control- and data path is entered graphically using a library of customizable building blocks such as register files, memories, arithmetic and logical units, buses, etc. From this description, a mixed behavioral and structural model of the customized processor is generated using the formalism of Gurevich's Abstract State Machines (ASMs) [10].

We describe ASMs using the specification language XASM [1]. XASM is supported by the Gem-Mex [3] tool that is able to automatically generate a debugging and simulation environment for the given XASM specification.

This methodology is part of the project BUILDABONG (*Building special computer architectures based on architecture and compiler co-generation*) [5].

In this paper, first, we give an overview of related work. Then, we introduce the basics of ASMs as the underlying mathematical model of our methodology and explain the major advantages of ASMs for describing the behavior of a processor. In Section 5, we present the features of the graphical editor tool *ArchitectureComposer* and demonstrate how this tool automatically generates XASM-code from a graphical architecture description.

2. RELATED WORK

In the following, we list some significant approaches aiming at architecture/compiler co-design.

The RECORD system (University of Dortmund, Germany) [15] aims at automatic code generation for fixed-point DSPs with a fixed instruction word length. The target processor has to be specified by the user in the structural hardware description language MIMOLA [4].

LANCE (University of Dortmund, Germany) [12] is a development environment for C compilers. The system serves as a basis for developing new code generation and optimization techniques for embedded processors, with emphasis on DSPs.

The hardware description language nML [7] maintains concise, hierarchical processor descriptions in a behavioral style. An example of the use of nML is the CBC/SIGH/SIM framework [6] consisting of the retargetable code generator CBC which uses a standard code-generator generator for instruction selection, and the instruction set simulator SIGH/SIM. Another approach using nML is CHESS [8, 14], developed at IMEC in Leuven, Belgium, a retargetable code generation environment for fixed-point DSP processors. CHESS uses a mixed behavioral/structural processor model and is supported by the simulator tool CHECKERS [14].

The machine description language LISA [18] is the basis for a

retargetable compiled simulator approach [17] developed at RWTH Aachen, Germany, which reaches simulation speeds in the order of 100K instr./sec.

The FLEXWARE (*SGS Thompson, Bell Northern Research*) [16] framework consists of the retargetable code generator CODESYN and the instruction set simulator INSULIN. CODESYN takes one or more algorithms expressed in a high-level language and maps them onto a user defined instruction set to produce optimized machine code for a target ASIP or a commercial processor core. INSULIN is based on a reconfigurable VHDL model of a generic instruction set processor.

At the ACES laboratory of the University of California, Irvine, the architecture description language EXPRESSION [11] has been developed. From an EXPRESSION description of an architecture, a retargetable compiler and a cycle-accurate simulator can be automatically generated. The tool is supported by the graphical design environment V-SAT.

CASTLE [13] is an automata-theoretic approach describing the behavior of the data path as *extended finite state machines* which are extracted from a register transfer level description of the target processor based on a VHDL-template.

3. BUILDABONG

BUILDABONG ("Building special computer architectures based on architecture and compiler co-generation") is the name of our project at the University of Paderborn [5] which aims at architecture and compiler co-generation for special purpose processors, i.e., DSP type and VLIW processors. The project started in 1999 and consists of the following four phases of development, see also Figure 1:

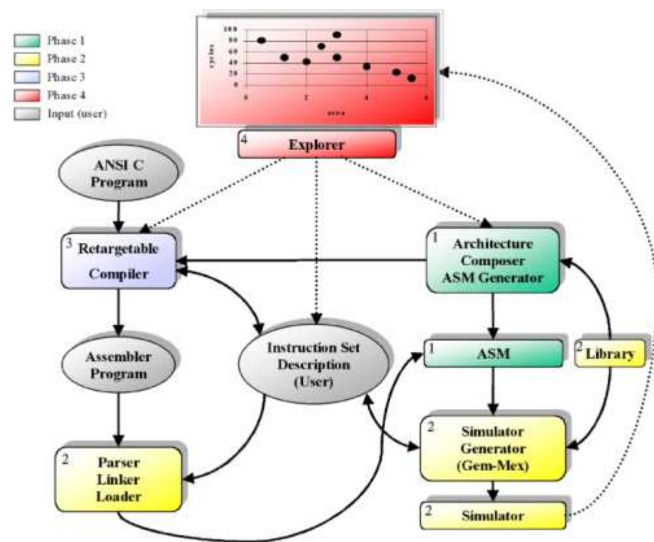

Figure 1: Overall software architecture and design flow in the BUILDABONG project

phase1 architecture entry and composition:
An object-oriented tool for hierarchical graphical entry and composition of a processor architecture (*ArchitectureComposer*) has been developed from which an ASM description of the resulting architecture is generated automatically. For this purpose, a library of common high-level components to compose an architecture (e.g., address generation units, buses, ALUs, memory units, register files, etc.) has been defined being parameterizable in bit width, number of inputs, etc.

phase2 architecture simulation:
ArchitectureComposer generates the ASM description of the processor's behavior from which subsequently, a cycle-accurate and bit-true instruction set simulator is generated using the Gem-Mex environment for ASM prototyping [3]. Currently, we have reached this level of implementation.

phase3 compiler generation (retargeting):
In order to allow the compiler to exploit architectural changes, e.g., number of functional units, distributed register files, etc., the necessary information for code generation and instruction scheduling is extracted from the graphical description (editor, phase 1).

phase4 architecture/compiler optimization:
The final goal of the project is to provide an exploration framework for joint architecture/compiler co-generation. For this purpose, an exploration tool has to be developed which adds profiling functions to the ASM description that are evaluated during simulation, and is responsible to explore the design space of architecture and compiler changes.

4. ABSTRACT STATE MACHINES

Here, we introduce the basic concepts of Abstract State Machines (ASM) as the fundamental mathematical theory of our model and demonstrate their suitability for modeling computer architectures on a high level of accuracy.

4.1 Basics of ASMs

In mathematics, domains, also called *universes* [10], functions, and relations constitute a *structure*. Structures without relations are traditionally called *algebras*. A sequential *Evolving Algebra*, or now called *Abstract State Machine* [10] $\mathcal{M} = (\mathcal{V}, f_1, ..., f_r)$ is a first order algebra with:

- $\mathcal{V}$ being a finite vocabulary (called *SuperUniverse*) and
- f_i being a finite set of n-ary functions over $\mathcal{V}$.

States of $\mathcal{M}$ are structures (resp. algebras) over $\mathcal{V}$. Now, an ASM $\mathcal{M}$ is defined by a an initial state $\mathcal{S}_0$ and a program $\mathcal{P}$ (`Update Rules`) consisting of a finite set of transition rules each of the form:

- `if <Cond> then <Rule> endif` (Conditional)
  with `<Cond>`, `<Rule>` being terms over $\mathcal{V}$,
- `Rule_1 Rule_2 Rule_3 ... Rule_n` (Block)
  with `<Rule_i>` being terms over $\mathcal{V}$,

- $f(t_1, ..., t_n) := t$ (Update) with $f(t_1, ..., t_n), t$ being terms over $\mathcal{V}$. Let f denote an arbitrary $n-$ary function and $t_1, \cdots, t_n$ denote a sequence of parameters. Then, a (function) update $f(t_1, \cdots, t_n) := t$ sets the value of f at $(t_1, \cdots, t_n)$ to t. Note, that `Updates` are computed simultaneously.

In general, `<Rule>` is a finite set of update rules and `<Cond>` may be an arbitrary boolean valued expression (first-order logic formula). Since algebras do not include relations, the equality-relation is expressed by its characteristic function. So, the evaluation of a condition is the comparison of a term (an n-ary function) and a nullary function `true` resp. `false` or `undef`.

The operational semantics of an ASM may be described as follows [10]: The effect of a transition rule $\mathcal{R}$ when applied to a state $\mathcal{S}_i$ (which itself is an algebra) is to produce another algebra (or state) $\mathcal{S}_{i+1}$ which differs from $\mathcal{S}_i$ by the new values for those functions at those arguments where the values are updated by the rule $\mathcal{R}$. If `Cond` is true, the rule can be executed by simultaneously executing each update in the set of `Updates`. An ASM terminates when it runs into a fixed point $\mathcal{S}_n$, i.e. $\mathcal{S}_n$ evokes no further `Updates`.

4.2 Modeling Processors with ASMs

We aim at a cycle-accurate model of computer architectures. This means that we have to describe an architecture's behavior on at least *register transfer level* (RTL). In general, the execution of a register transfer may depend on a machine's internal state (e.g. state of mode registers, drivers, and/or the value of certain instruction bit variables). Leupers [15] demonstrates that the behavior of a processor architecture may be described on register transfer level by a set of so-called *guarded register transfer patterns* (GRTP), where a *register transfer pattern* denotes a set of operations to be performed if the associated *guard*, a boolean function representing a machine's internal state, is evaluated to true. With this concept, we can consider a processor as a machine which in every control step executes the same set of parallel *guarded operations*, each of which has the form: **if** `<register_transfer_condition>` **then** `<register_transfer_pattern>` The correspondence of GRTPs and ASM transition rules is obvious.

4.3 Advantages of ASM Models

The main advantages of using ASMs for processor modeling may be summarized as follows:

- Shortness of description (e.g., 200 lines XASM code for the ARM7 processor [19])
- readability of the specification (no syntactic overhead),
- cycle accuracy and acceptable simulation speed,
- bit-true simulation of irregular arithmetic operations on arbitrary large word-lengths using a C-based library of configurable standard functions based on arbitrary precision numbers (GNU-MP) since the inclusion of C-libraries is strongly supported by the XASM environment.

5. ARCHITECTURE COMPOSER

ArchitectureComposer is a tool for the automatical generation of an ASM model of a processor given by a graphical description of the architecture's control- and data path.

5.1 The Architecture Editor

The tool supplies a library of standard components necessary for designing the basic elements of a processor, e.g. registers, memory, logical and arithmetical units. Figure 2 shows the graphical composition of a simple architecture, the lower left window displays the set of basic architecture modules to choose from.

These basic elements are parameterized. The parameters of each element can be manipulated at any time via a dialog. See, e.g., the upper right dialog of Figure 2 for changing the parameters of the multiplexer *AluOpMux* with three inputs of six bit each and a control port width of two bits.

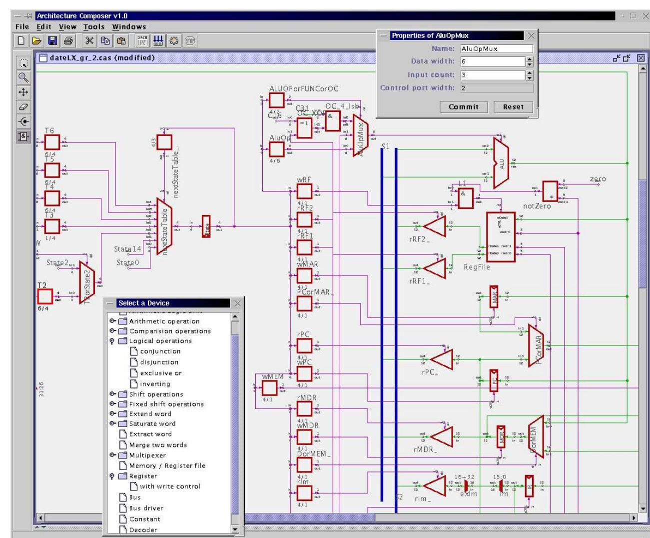

Figure 2: Designing a MIPS like processor architecture using *ArchitectureComposer*

Interconnections of basic elements are automatically validated. Finally, the editor allows to cluster nets and define new components this way (component generation) [2]. Thus, an overall architecture is represented hierarchically.

After architecture entry, the XASM-code generator (see Section 5.2) creates a corresponding behavioral and structural description in XASM notation [1].

5.2 Automatic XASM-Code Generation

In our approach, code generation is performed according to the following scheme:

1. *Generation of output signal declarations*: For each output signal of each hardware component, a function declaration is generated. Thus, a function represents the output signal of a hardware component.
2. The functions corresponding to sequential elements (e.g. memory) are *initialized* in a separate block encapsulated by the keywords `init` and `endinit`.

3. Memory and register assignments are encoded as *guarded update rules.*

We distinguish two kinds of hardware components:

a) combinational elements (e.g. comparators, multipliers, adders, etc.) and
b) sequential elements (registers, memories, etc.)

Non-hierarchical combinational elements are modeled by an external library of C-functions providing the necessary arithmetic and logical functions. Using the XASM construct *derived function*, an alias for the respective function call is declared. For each sequential element, an XASM *function* is generated which in turn is used in the update rules, see Section 4.1.

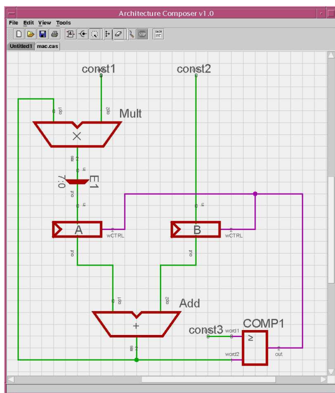

Figure 3: Example of a simple data path

Example: Here, we consider the XASM code generation for the simple architecture displayed in Figure 3. The generated XASM code looks as follows:

```
asm MAIN is
   use cpucore
   derived function Mult_res ==
         c_mult (Add_res, const1_out, 8, COMPLEMENT_2)
   derived function Add_res ==
         c_add (A_out, B_out, 8, COMPLEMENT_2)
   derived function E1_out ==
         c_extract (Mult_res, 16, 0, 8)
   derived function COMP1_out ==
         c_gteq (const3_out, Add_res, 8, COMPLEMENT_2)
   function A_out
   function B_out
   function const1_out
   function const2_out
   function const3_out
   init
      A_out := "00000000"
      B_out := "00000000"
      const1_out := "00000010"
      const2_out := "00000001"
      const3_out := "00001000"
   endinit
   if COMP1_out = "1" then A_out := E1_out
   endif
   if COMP1_out = "1" then B_out := const2_out
   endif
endasm
```

In the above code, the construct **use cpucore** is needed for linking the library of C functions (**c_mult, c_add, c_extract**) mentioned above. These *external functions*, marked by a prefix **c_** call the generic simulation engine **gen_lib** which is independent of an architecture's internal number representation or word length. This simulation engine is supported by the GNU Multiple Precision Arithmetic Library [9] operating on arbitrary word-length integers. So, operands, represented by bitstrings, are converted to integer values, then the respective operation is performed on the integer operands and the result is converted back to a bitstring of the machine's representation format.

5.3 Automatical Simulator Generation

After generation of the XASM description by *Architecture-Composer*, the XASM-compiler translates this description into C-code and uses the gcc-compiler to compile these C-files. Subsequently, the resulting object files are linked with the simulation engine mentioned in Section 5.2 and the XASM runtime library. The result is an interactive simulation and debugging environment controled by Tcl/Tk scripts provided by the Gem-Mex tool [1, 3].

Figure 4 displays the design workflow including a snapshot of a typical simulation and debugging session (case study Texas Instruments TMS3200C6201) using the Gem-Mex environment. In this case study [20], the ASM model has still been handwritten. The application has been programmed in C code, the corresponding TI-C6 assembly code was produced by the Texas Instruments *Code Composer Studio* compiler.

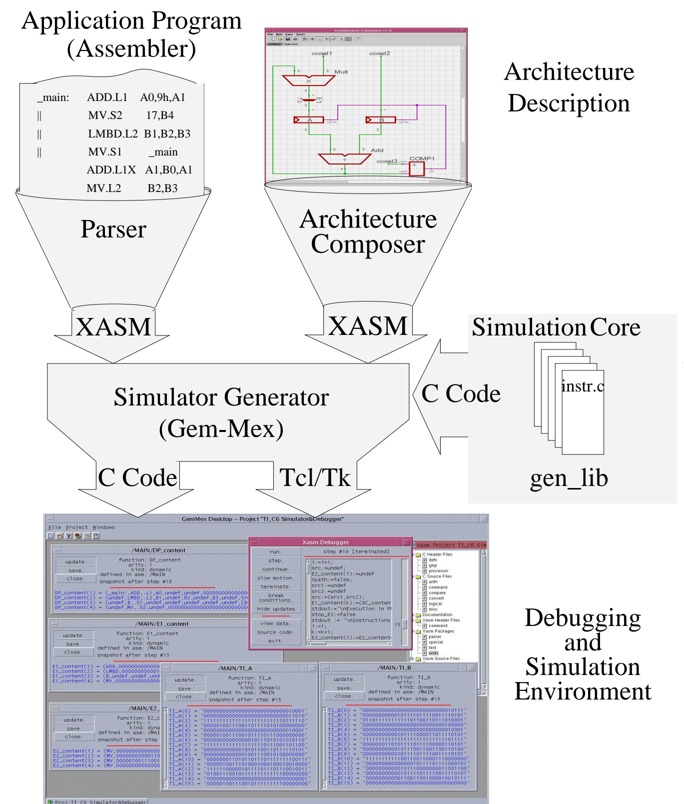

Figure 4: Workflow of the Simulation

6. CONCLUSION AND FUTURE WORK

In this paper, we describe how cycle-accurate processor behavior may be efficiently modeled using the formalism of Abstract State Machines [10]. We introduce the tool *ArchitectureComposer* for hierarchical graphical entry and composition of processor architectures. Given the graphical representation of a processor's control- and data path, *ArchitectureComposer* automatically generates an ASM model in the ASM specification language XASM. From this XASM-code, a simulator and debugging environment can be generated automatically by the Gem-Mex tool.

The presented methodology is part of the first two phases of the project BUILDABONG which aims at joined architecture/compiler co-generation. In the future, given a specification of a problem and constraints on the hardware design, our task is to explore architecture tradeoffs. For this purpose, we need the editor described above and a compiler tool which is retargetable to the architecture emitted by the editor to guarantee optimal scheduling and resource utilization. The development of such a compiler tool is the current field of research in the BUILDABONG project (phase 3).

7. REFERENCES

[1] M. Anlauff. XASM - an extensible, component-based abstract state machines language. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *International Workshop on Abstract State Machines*, volume 1912 of *Lecture Notes on Computer Science (LNCS)*, pages 69–90. Springer, 2000.

[2] M. Anlauff, D. Fischer, P. Kutter, J. Teich, and R. Weper. Hierarchical microprocessor design using XASM. In *Proceedings of the 7th International Conference on Computer Aided Systems Theory: formal Methods and Tools for Computer Science (EUROCAST 2001)*, Las Palmas de Gran Canaria, Spain, Feb. 19-23, 2001. (to appear).

[3] M. Anlauff, P. Kutter, and A. Pierantonio. Formal aspects of and development environments for Montages. In *Second Int. Workshop on the Theory and Practice of Algebraic Specifications*, Workshops in Computing. Springer-Verlag, 1997.

[4] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer. The MIMOLA language - version 4.1. Technical report, University of Dortmund, 1994.

[5] `http://www-date.upb.de/RESEARCH/BUILDABONG/buildabong.html`.

[6] A. Fauth. Beyond tool-specific machine descriptions. In P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*, pages 138–152. Kluwer Academic Publishers, 1995.

[7] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. In *Proceedings on the European Design and Test Conference, Paris, France*, pages 503–507, March 1995.

[8] G. Goossens, J. Van Praet, D. Lanneer, W. Geurts, and F. Thoen. Programmable chips in consumer electronics and telecommunications. In G. de Micheli and M. Sami, editors, *Hardware/Software Co-Design*, volume 310 of *NATO ASI Series E: Applied Sciences*, pages 135–164. Kluwer Academic Publishers, 1996.

[9] T. Granlund. The GNU multiple precision library, edition 2.0.2. Technical report, TMG Datakonsult, Sodermannagatan 5, 11623 Stockholm, Sweden, 1996.

[10] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36, Oxford University Press, 1995.

[11] Ashok Halambi, Peter Grun, Asheesh Khare, Vijay Ganesh, Nikil Dutt, and Alex Nicolau. Expression: A language for architecture exploration through compiler/simulator retargetability. In *Proc. Design Automation and Test in Europe (DATE'1999)*, 1999.

[12] `http://ls12-www.informatik.uni-dortmund.de/leupers/lanceV2/lanceV2.html`.

[13] M. Langevin, E. Cerny, J. Wilberg, and H.-T. Vierhaus. Local microcode generation in system design. In P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*, pages 171–187. Kluwer Academic Publishers, 1995.

[14] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. Chess: Retargetable code generation for embedded dsp processors. In P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*, pages 85–102. Kluwer Academic Publishers, 1995.

[15] R. Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1997.

[16] P. Paulin, C. Liem, T. May, and S. Sutarwala. Flexware: A flexible firmware development environment for embedded systems. In P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*, pages 67–84. Kluwer Academic Publishers, 1995.

[17] S. Pees, A. Hoffmann, and H. Meyr. Retargeting of compiled simulators for digital signal processors using a machine description language. In *Proceedings Design Automation and Test in Europe (DATE'2000)*, Paris, March 2000.

[18] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. Machine description language for cycle-accurate models of programmable DSP-architectures. In *Proceedings of the 36th Design Automation Conference (DAC'99)*, New Orleans, June 1999.

[19] J. Teich, P. Kutter, and R. Weper. Description and simulation of microprocessor instruction sets using ASMs. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Int. Workshop on Abstract State Machines*, volume 1912 of *Lecture Notes on Computer Science (LNCS)*, pages 266–286. Springer, 2000.

[20] J. Teich, R. Weper, D. Fischer, and S. Trinkert. A joined architecture/compiler design environment for ASIPs. In *ACM SIG Proceedings International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES 2000)*, pages 26–33, San Jose, CA, U.S.A., November 2000.