IP-SOC 2010

TLM 2.0 STANDARD INTO ACTION: DESIGNING EFFICIENT PROCESSOR SIMULATOR

Luca Fossati, European Space Agency - Luca.Fossati@esa.int

Keplerlaan 1 - Noordwijk, The Netherlands

Abstract:

The increasingly growing complexity of integrated Systems-on-Chip forces the early use of simulation in order to explore the huge design space characterizing them. Simulators also enable software development before the final hardware prototype is ready. Unfortunately, manually building simulatable models is a lengthy and error prone task; this, in particular, concerns complex elements such as microprocessors: Architectural Description Languages (ADLs) come to a help, enabling the automatic generation of such simulators starting from a high-level abstract description. This article presents TRAP (*T*ransactional *A*utomatic *P*rocessor generator), an innovative ADL targeted to the creation of fast and flexible simulators based on the SystemC language and on the new OSCI Transaction Level Modeling methodology. The modeling of the LEON2 and LEON3 processors is presented to show and assess the features, performance, and capabilities of TRAP.

1 Introduction

The continuous increase of transistor density for each die is leading towards the production of more and more complex *systems on a single chip*, with an increasing number of components, as clearly shown in Figure 1. The design of such complex embedded

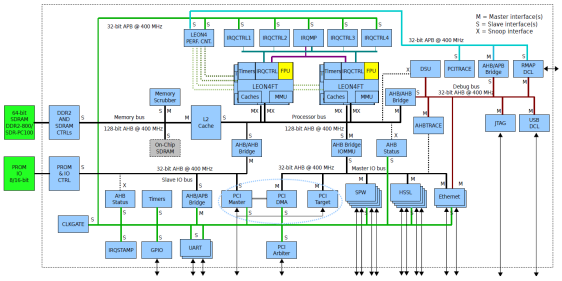


Figure 1: Next Generation Multi-Purpose Microprocessor, Aeroflex Gaisler Research [1]

systems is a difficult problem, requiring designers with skills and experience to identify the best solution. In this context, the ability to accurately, quickly, and easily predict properties of computer systems is useful for computer architects, designers, and software developers. Because simulators are not subject to the same constraints as their physical counterpart, they are easier to create, modify, and observe. Among such hardware simulators, the so called *Virtual Platforms* (VPs) are gaining more and more attention. They combine high-speed micro-processor simulators and functional C/C++ models of the remaining system's building blocks, to provide a high-level model of the hardware to the software developer and, in the preliminary development stages, to the system architect.

Describing Fast Simulatable Hardware Models

In order to enable effective *Virtual Platform* design there is the need for hardware description languages which are both expressive, fast, and which enable targeting detailed low level hardware structures as well as system-level models. In particular SystemC [4] has grown more and more popular; this language fuses a well-known syntax with powerful constructs, enabling the modeling and simulation of complex systems. SystemC consists of a set of C++ libraries devoted to building system-level executable models of mixed hardware-software systems at several levels of abstraction; many constructs are provided, among which *modules*, *channels*, and *interfaces*. The true advantage of SystemC as a specification language lies in the fact that it encompasses all the important hardware modeling features, as well as providing powerful modeling constructs for system level design. This ensures that the transition to a SystemC-based methodology entails no compromise in terms of expressive power at the lower levels of abstraction, and yet it forms a useful framework for modeling at the higher levels.

Using a language for System-Level modeling, such as SystemC, is not enough for building an effective simulatable model of systems as complex as the one in Figure 1: it is also necessary to define modeling styles and interoperability rules among the various models. Transaction Level Modeling (TLM), first introduced in [6], is now widely accepted as an efficient technique for *abstract modeling of communication* and, according to recent research [7], computation. The underlying concept of TLM is to model only the level of detail that is needed, hence both boosting simulation speed and improving de-

sign flexibility, as changes to the design are also relatively easy and cost effective, not having yet delved in low-level details.

Recently, the OSCI committee has proposed version 2.0 of their TLM library [12] composed of a set of SystemC primitives that allow designers to implement several Transaction Level *communication protocols* using different *coding styles* concerning the modeling of time and providing different tradeoff between simulation accuracy and speed. The *loosely*

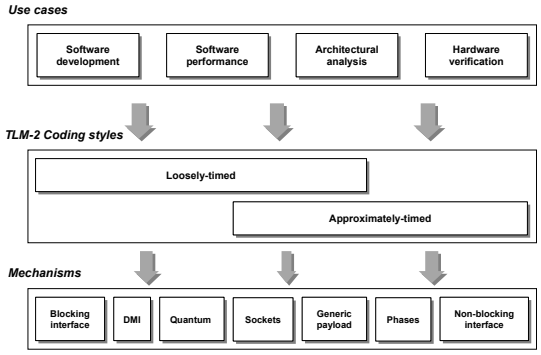


Figure 2: Codying Styles and Use Cases for the OSCI TLM 2.0 Standard

timed coding style (LT-TLM) is appropriate for the use case of software development; it supports the modeling of timers and interrupts, sufficient to boot an operating system and run arbitrary code on the target machine. Opposite, the *approximately-timed* (AT-TLM) coding style is used mainly for architectural exploration and detailed performance analysis. With respect to AT-TLM, LT-TLM enables much higher simulation speed both by abstracting every communication event (e.g. read/write to/from a memory location) with a single transaction and by enabling the so called *temporal decoupling*: part of the models to run ahead of the global system time in a local "time warp" until they reach the point when they need to synchronize with the rest of the system.

Processor Description Languages

As clearly exemplified by the System-on-Chip of Figure 1 and presented by the international Semiconductor Industry Association (ITRS) [18], most of nowadays hardware architectures are centered on single or multiple processing units; hence, the respective system-level models (called *Instruction Set Simulators*, ISS) must include detailed, accurate, and efficient models of such processing elements.

Those models could be hand-crafted, but this is a tedious, lengthy, and error-prone task [5]. For those reasons, *Architecture Description Languages* (ADLs) [14] have been devised to enable automatic generation of tools to support the microprocessor design process starting from high-level abstract descriptions. Such descriptions enable the generation of other tools, in addition to ISSs, aiding processor's development flow: (a) Register Transfer Level models (enabling hardware synthesis of the described processor), (b) Compilers, and (c) Model Checkers, Formal Verifiers, etc. Partly automating

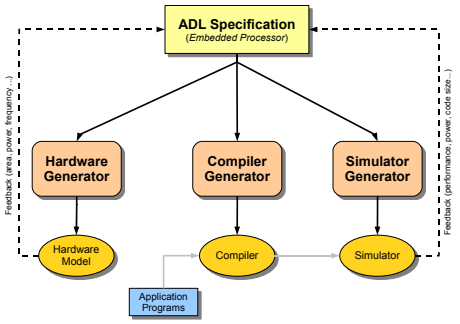


Figure 3: Design Flow using Architectural Description Languages

the processor design process and centralizing its specification into a single high level description brings numerous advantages: i) speeds-up the exploration and the evaluation of the different design alternatives, ii) aids the validation of early design decisions, iii) helps keeping the model and its implementation consistent, and iv) it improves communication among team members.

After presenting a brief overview of the ADL landscape, this article focuses on presenting TRAP an ADL designed to generate fast simulators based on the newest SystemC 2.2 and TLM 2.0 standards. Last part of the article shows a use case where TRAP was employed for the description of LEON2 and LEON3 processors [1] and the generation of the respective Instruction Set Simulators.

2 *TR*ansactional *A*utomatic *P*rocessor generator

In the past decade many ADLs were introduced, each one with different characteristics and capabilities; as described in [14], they can be classified into three broad categories: structural, behavioral, and mixed; this classification is based on the nature of the information that the developer provides through the particular ADL. While *structural languages* (e.g. MIMOLA [11]) focus on the description of the architectural elements and their connectivity, *behavioral languages*, like nML [8], explicitly specify the instructions' semantic, almost completely ignoring detailed hardware structures and timing information. Finally *mixed languages* (of which the most famous examples are LISA [13, 19], EXPRESSION [10, 15], MADL [16], and ArchC [17]) are a mixture of the previous two types, as they include both behavioral and structural information.

TRAP (*TR*ansactional *A*utomatic *P*rocessor generator) [3] is a behavioral ADL including the tools for the automatic generation of processor simulators.

The tool is written in Python and it produces SystemC based *Instruction Set Simulators*.

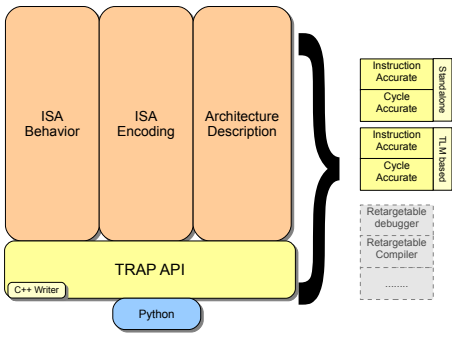


Figure 4: Architecture of the TRAP language and tools

TRAP, as shown in Figure 4 consists of a Python library: the processor specification is given through appropriate calls to its APIs. With respect to standard ADLs, which use custom languages, directly specifying the input in Python eliminates the need for an ad-hoc front-end; such feature simplifies the development of the ADL and its use by the designer: (a) there is no need to learn a new language, (b) during model creation the full power of the Python language can be exploited, and (c) no ad-hoc parser is needed. The Instruction Set Simulators generated by TRAP are based on the SystemC library and on the new TLM 2.0 standard for modeling the processor's communication interfaces. Depending on the desired accuracy/simulation speed tradeoff, different flavors of simulators can be automatically created; the developer can choose between Loosely-Timed TLM interfaces or Accurate-Timed ones and, orthogonally, between taking into account pipeline details or not.

2.1 Language Structure

Three elements (Listings 1, 2, 3) compose a processor model in TRAP:

1. *architecture description*, where the structural elements (registers, pipeline stages, etc.) are described.
2. *ISA coding*, specifying the encoding of each instruction.
3. *ISA behavior*, which contains the behavior of each instruction of the Instruction-Set

A fourth element, *ISA testing*, can be part of a TRAP description; it defines the tests to be applied to the Instruction-Set description to make sure that it has been correctly implemented. The developer only needs to specify the processor state (in terms of the registers relevant to the instruction under test) before the execution of the instruction and the desired state after the instruction execution; testing then automatically takes place.Per-instruction testing increases the confidence in the correctness of the simulator and, in case of implementation bugs, it consistently reduces the effort needed to locate and correct the problem.

Listing 2: Application binary interface

```
abi = trap.ABI('REGS[24]', 'REGS[24-29]', 'PC', '
    LR', 'SP', 'FP')
abi.addVarRegsCorrespondence({'REGS[0-31]': (0,
    31), 'Y': 64, 'PSR': 65, 'WIM': 66, 'TBR':
    67, 'PC': 68, 'NPC': 69})
```

2.2 Modeling Methodology

The modeling of a processor with TRAP starts with the architectural description (Listing 1); as for most of the behavioral languages, few details need to be modeled: (i) the number and size of the registers, (ii) the internal memory and/or the memory ports, (iii) the interrupt ports with the behavior associated with the interrupts, and (iv) the pipeline stages. The behavior of the instructions (Listing 3), is specified directly using C++ code, separating it into the different pipeline stages for which the particular instruction is active; the same mechanism is used to describe the reaction of the processor to an incoming interrupt signal.

Following what introduced in most ADLs [8, 19] both instructions' behavior and encoding are given in a hierarchical way: the specification is given only for the basic building blocks, which are, then, composed into the final instructions. Such organization consistently reduces the development efforts and it simplifies the description.

Listing 1: Structural Description of the LEON3 processor

```
processor = trap.Processor('LEON3')
processor.setBigEndian()
processor.setWordsize(4, 8)
globalRegs = trap.RegisterBank('GLOBAL', 8, 32)
processor.addRegBank(globalRegs)
tbrBitMask = {'TBA' : (12, 31), 'TT' : (4, 11)}
tbrReg = trap.Register('TBR', 32, tbrBitMask)
tbrReg.setDefaultValue(0)
processor.addRegister(tbrReg)
regs = trap.AliasRegBank('REGS', 32, ('GLOBAL[0-7]
    ', 'WINREGS[0-23]'))
processor.addAliasRegBank(regs)
fetchStage = trap.PipeStage('fetch')
processor.addPipeStage(fetchStage)
```

The **Application Binary Interface (ABI)**, Listing 2, concludes a processor description with TRAP, formalizing the rules with which the tools (compiler, debugger, etc.) access and use the processor and with which the compiler compiles the software (for example the routine call conventions).

Through the ABI description TRAP enables:

- generation of a debugger interface to enable the use of the GDB debugger for checking the correctness of the software running on the ISS.
- system call emulation: allowing the redirection of the system calls contained in the simulated software to the host Operating System.
- generation of a software profiler, for gathering statistics on the software running on top of the Instruction Set Simulator.

Once completed, the Python files containing those elements are *executed* (remember that they are composed of calls to TRAP APIs): the whole TRAP description can indeed be thought as a Python program having the aim of producing text files containing the C++ code of the processor simulator. Communication ports for the interfaces towards memory and for the interrupt signals are also created; as previously anticipated, such interfaces follow the TLM 2.0 standard and use the corresponding OSCI library.

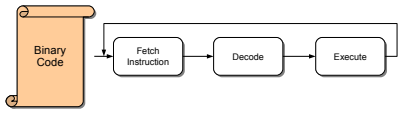
With respect to traditional Architectural Description Languages, TRAP has the following advantages: (a) it is Open Source, (b) the descriptions are based on the Python language, enabling the use of the full capabilities of this language (any valid Python statement can be employed in the processor description) and eliminating the need to learn a new language, (c) it has a simple structure, as it is restricted to generating only Instruction Set Simulators, and (d) it automates most of the steps necessary for simulator generation, including pipeline hazards management and verification of the correctness of the model description; finally (e) it is deeply integrated with SytemC and TLM libraries, generating processors based on the latest hardware modeling technologies.

Listing 3: ISA Description of the LEON3 processor

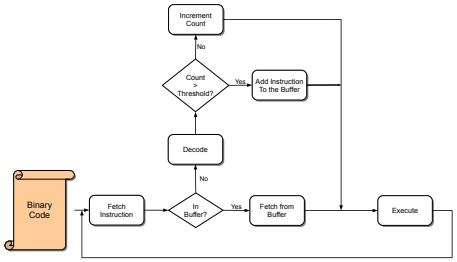
```
dpi_format2 = trap.MachineCode([('op', 2), ('rd',
    5), ('op3', 6), ('rs1', 5), ('one', 1), ('
    simm13', 13)])
dpi_format2.setBitfield('op', [1, 0])
dpi_format2.setVarField('rd', ('REGS', 0), 'out')
dpi_format2.setVarField('rs1', ('REGS', 0), 'in')

opCodeRegsImm = cxx_writer.writer_code.Code("""
rs1_op = rs1;
rs2_op = SignExtend(simm13, 13);""")
opCodeExec = cxx_writer.writer_code.Code("""
result = rs1_op + rs2_op;""")
add_imm_Instr = trap.Instruction('ADD_imm', True,
    frequency = 11)
add_imm_Instr.setMachineCode(dpi_format2, {'op3':
    [0, 0, 0, 0, 0, 0]}, ('add␣r', '%rs1', '␣', '
    %simm13', '␣r', '%rd'))
add_imm_Instr.setCode(opCodeRegsImm, 'regs')
add_imm_Instr.setCode(opCodeExec, 'execute')
add_imm_Instr.addBehavior(WB_plain, 'wb')
add_imm_Instr.addBehavior(IncrementPC, 'fetch',
    pre = False)
add_imm_Instr.addVariable(('result', 'BIT<32>'))
add_imm_Instr.addVariable(('rs1_op', 'BIT<32>'))
add_imm_Instr.addVariable(('rs2_op', 'BIT<32>'))
isa.addInstruction(add_imm_Instr)
```

Decoding Buffer

In order to improve the execution speed of the generated simulators, they incorporate a *decoding buffer* for caching individual decoded instructions, thus avoiding the need of re-decoding them when re-encountered. With this mechanism, shown in Figure 5, the slow instruction decoding process is amortized by the high hit rate of the buffer. With respect to existing alternative solutions, TRAP's decoding buffer enables the presence of self modifying code in the program being simulated.



(a) Standard Fetch/Decode/Execute loop



(b) Fetch/Decode/Execute loop using the decoding buffer

Figure 5: Standard instruction decoding mechanism compared with the one based on the instruction buffer

2.3 Modeling Effort

Processor modeling using TRAP requires a moderate effort, since as many steps as possible are automated (like, for example, pipeline hazards detection, or instruction testing). In addition checks are performed, during processor generation, to make sure that the specification is consistent, signaling to the user, in case problems exist, where they are located in the specification. Examples of such checks are: conflicts in the instructions' machine code, syntax errors (such as use of non-declared registers in the specification of the instruction behavior), and checking that the user specified all the necessary details (e.g. identified both the register read and write back pipeline stages), etc.

Table 1 shows the overall modeling efforts necessary for the processors currently present in TRAP; to show the usefulness of such ADL, a comparison between the lines of code (LOC) of TRAP's description and of the generated processor is given.

Table 1: Processor modeling effort using TRAP

Processor	**TRAP** (LOC)	**ISS** (LOC)	**Effort** (m/m)	**Status**
Microblze	2234	14788	1	Instruction Accurate
ARM7	2457	11336	1.2	Cycle & Instruction Accurate
LEON3	3612	32292	1.5	Cycle & Instruction Accurate
MIPS	2351	17634	1	Instruction Accurate

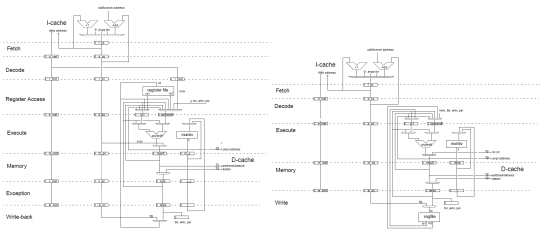
3 Case Study: Modeling the LEON Processor Core

The modeling of the LEON2 and LEON3 processors was used as a case study to test, validate, and exper-

iment with TRAP and to show its effectiveness and capabilities for modeling modern RISC processors.

3.1 LEON Processor

LEON is a 32-bit processor core conforming to the IEEE-1754 (SPARC V8) architecture, including hardware multiply and divide instructions. The number of register windows is configurable within the limit of the SPARC standard (2 - 32), with a default setting of 8. The pipeline consists of 7 stages for the LEON3 (Figure 6(a)) and 5 for LEON2 (Figure 6(b)), with a separate instruction and data cache interface (Harvard architecture). It is designed for embedded applications, combining high performance with low complexity and low power consumption. Refer to



(a) LEON3 (b) LEON2

Figure 6: LEON3 and LEON2 Pipeline Structure

Aeroflex Gaisler Research [1] for more details on the LEON processor family.

Being targeted to low power, relatively simple embedded systems, this processor family features an in-order single issue pipeline, built on purpose to minimize the stalls due to data hazards. This means that if instruction `B` needs in input the result of a preceding operation `A`, such result is forwarded to `B` as soon as it is ready without the need to stop the flow of instructions in the pipeline. With the exception of `B` being a `branch`, such mechanism holds for all instructions, whose latency can, thus, be statically computed. This consistently simplifies the design and the timing verification of the processor simulator.

3.2 Processor Modeling

Processor modeling was performed according to the methodology outlined in Section 2; taking the LEON3 processor as an example, the specification consists of 5 `Python` files containing, in total, 3612 lines of code. From such files the different simulator flavors were created, producing around 32000 lines of code for the Instruction Accurate simulator and above 90000 for the one featuring accurate description of the pipeline. This already gives an idea of how the use of an Architectural Description Language (TRAP in this case) helps in consistently reducing the modeling effort. In addition to the Instruction Set simulator itself, individual tests for each assembly instruction were automatically created, with an average of 10 tests for each instruction; the generated `C++` code for such tests amounts to over 110000 lines of code.

Completed the simulator generation, accurate tests were performed to ensure its correct behavior and to measure, and eventually correct, the timing accuracy.

3.2.1 Testing

Behavioral Testing has been performed at three different levels: 1) Single instruction level: each instruction has been separately tested using TRAP according to the mechanisms previously described. 2) Syhnthetic benchmarks: simple C programs checking single issues, such as the correct behavior of memory loads, or memory store, shifts, etc. 40 different C benchmarks were written, each one compiled with different optimization flags, yielding a total of 160 different application programs. 3) Full-fledged benchmarks: real-world C programs (video compression/decompression, signal processing, etc.) taken from the MiBench [9] suite and manually coded, for a total of over 100 programs.

Single instruction tests mainly help eliminating the coding-errors, synthetic benchmarks help detecting the errors due to wrong synchronization and communication among the instructions (such benchmarks are short, so when there is failure it is easy to detect and correct the error). Finally correct execution of real-world benchmarks ensures that everything is indeed working as expected.

3.3 Quantitative Results

3.3.1 Execution Speed

Simulator performance has been measured on a set of 26 applications taken from the the full-fledged benchmark set introduced above. Figures 7 and 8 show, for each model flavor, the average simulation speed over all the benchmarks and the maximum value obtained. `IA-LT` refers to Instruction-accurate accuracy with Loosely-timed interfaces, `IA-AT` with Accurate-timed interfaces, and `PA-LT` and `PA-AT` are the same but for models containing an accurate pipeline description.

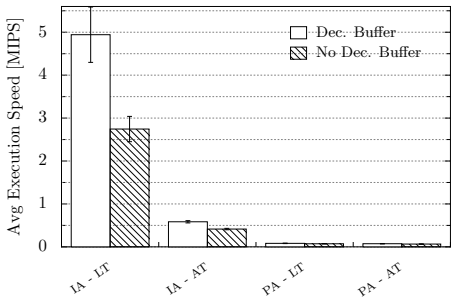


Figure 7: Average Execution Speed of the LEON3 processor simulator

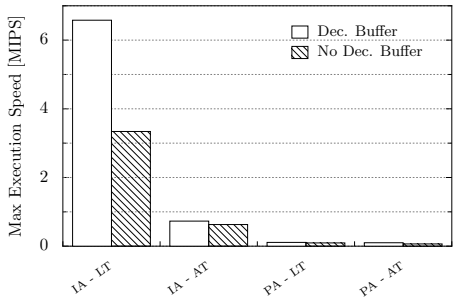


Figure 8: Maximum Execution Speed of the LEON3 processor simulator

It is clear that the use of the decoding buffer greatly improves the average execution speed; unfortunately we note a much higher variability over different benchmarks for the decoding buffer version with respect to the plain version (blue bar of Figures 7 and 8). Such variability is explained with the relative frequency of the instructions in the applications being simulated: while the applications executing faster execute many times a small set of instructions (thus they greatly benefit from the decoding buffer mechanism), the slower applications' execution path goes through a large set of different instructions. Such motivations drove us to believe that the overall simulation speed can be improved by addressing this high variability when the decoding buffer is used.

3.3.2 Timing Validation

The methodology used to assess the timing accuracy (i.e. how accurate is the simulation time reported by the simulator model) is more complicated, requiring the use of a golden model as a mean of comparison. The GRSIM [1] simulator was used; as it models the whole System-on-Chip (including caches, AMBA bus, etc.) there has been the need to isolate the processor core from the other architectural elements.

The same full-fledged benchmarks used for the behavioral validation have also been used for the timing validation, yielding a *99.3%* accuracy for the Instruction Accurate simulator flavor and a *100%* accuracy for the cycle accurate one. The 0.7% skew in the Instruction Accurate model with respect to the actual processor is due to the fact, as explained in Section 3.1, that in such model flavor data hazards due to `branch` instructions are not taken into account.

4 Summary and Conclusions

Architectural Description Languages (ADLs) prove an invaluable tool for the modeling and the design of micro-processors; they are often used only with the purpose of generating Instruction Set Simulators helping early software development, assessing the architecture performance, and even performing preliminary partition of the system's functionalities into hardware and software. The use of the SystemC language and of the Transaction Level Modeling methodology within such simulators, provide high execution speed, reduced modeling effort and it enables interoperability with external IP-models, easing the modeling of complex Systems-on-Chip. The LEON processor models presented in this article are indeed being used by the SoCRocket project [2] for the creation of a Virtual Platform representing the Systems-on-Chip commonly used in the space industry, with the aim of optimizing such systems and helping software development.

References

[1] Aeroflex Gaisler Research. http://www.gaisler.com/.

[2] SoCRocket Project. http://www.vlsilab.org/.

[3] Transactional Automatic Processor generator. http://trapgen.googlecode.com/.

[4] G. Arnout. SystemC standard. In *ASP-DAC 2000*, pages 573–577, 2000.

[5] G. Braun, A. Nohl, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr. A universal technique for fast and flexible instruction-set architecture simulation. *IEEE Transactions on CAD*, 23(12):1625–1639, 2004.

[6] L. Cai and D. Gajski. Transaction level modeling: an overview. In *CODES+ISSS '03*, page 1924, 2003.

[7] R. Domer. Transaction level modeling of computation. Technical report, Center for Embedded Computer Systems University of California, Irvine, 2006.

[8] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. In *European conference on Design and Test*, page 503, 1995.

[9] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: a free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE Workload Characterization Workshop, 2001*, pages 3–14. IEEE Computer Society, 2001.

[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 *DATE*, pages 485–490, 1999.

[11] P. Marwedel. The mimola design system: Tools for the design of digital processors. In *DAC*, pages 587–593. IEEE Press, 1984.

[12] Open SystemC Initiative (OSCI). OSCI TLM2 USER MANUAL, November 2007.

[13] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. LISA-machine description language for cycle-accurate models of programmable DSP architectures. In *DAC*, pages 933–938, 1999.

[14] M. Prabhat and N. Dutt. *Processor Description Languages*. Elsevier Science Ltd, June 2008.

[15] M. Prabhat, N. Dutt, and A. Nicolau. Functional abstraction driven design space exploration of heterogeneous programmable architectures. In *ISSS '01*, pages 256–261. ACM, 2001.

[16] W. Qin, S. Rajagopalan, and S. Malik. A formal concurrency model based architecture description language for synthesis of software development tools. In *ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 47–56, 2004.

[17] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. ArchC: a systemC-based architecture description language. In *SBAC-PAD 2004*, pages 66–73, 2004.

[18] SEMATECH. International technology roadmap for semiconductors (ITRS), 2007. http://www.itrs.net/, 2007.

[19] V. Zivojnovic, S. Pees, and H. Meyr. LISA-machine description language and generic machine model for HW/SW co-design. In *VLSI Signal Processing, IX*, pages 127–136, 1996.