Teaching Computer Architecture Using an Architecture Description Language

Sandro Rigo, Marcio Juliato, Rodolfo Azevedo, Guido Araújo, Paulo Centoducatte

Computer Systems Laboratory - Institute of Computing, University of Campinas
Cidade Universitária Zeferino Vaz, Po. Box 6176, Campinas-SP, Brazil
{srigo, marcio.juliato, rodolfo, guido, ducatte}@ic.unicamp.br

Abstract

This paper presents the use of the ArchC Architecture Description Language (ADL) as a support tool for computer architecture courses. ArchC enables students to perform several experiments using its automatically generated SystemC simulators, covering topics from simple single-cycle (functional) models to pipeline and memory hierarchy simulation. We show how instructive may be the process of modeling a processor using an ADL and suggest several possible exercises, following the course development structure presented in the classical Hennessy and Patterson's computer architecture didactical book. Moreover, we report how the experience of assigning students to study and to model modern embedded architectures has provided good results on an undergraduate computer architecture course at IC-UNICAMP. The simplicity and flexibility of the ADL, along with its simulation features, proved to be an useful tool not only for research, but also for computer architecture education.

1 Introduction

Architecture description languages (ADL) have been introduced to help designers face the development challenges that have arisen in the past few years, due to the increasing complexity of modern architectures. These difficulties have forced hardware architects and software engineers to reconsider how designs are specified, partitioned and verified. As a consequence, designers are starting to move from hardware description languages (VHDL, Verilog) and also beyond the RTL level of abstraction toward the so called *system level design*, where a tool for evaluating a new designed instruction set architecture, which automatically generates a software toolkit composed of assemblers, simulators, etc is mandatory. Such tools are commonly based on an architecture description language.

Besides their application and well known suitability for designing and experimenting with new architectures in the industry, architecture description languages can be very useful for academic purposes, like teaching/researching computer architecture at undergraduate and graduate level. On one hand, at the undergraduate level, models of well known architectures are appropriate to learn how a pipelined architecture works, including interlocking, hazard detection and register forwarding. If allowed by the ADL, this model can be plugged to different memory hierarchies in order to illustrate how the performance of a given application can vary, depending on the choice made for cache size, policy, associativity, etc. On the other hand, at the graduate level, researchers can use ADLs to model modern architectures and experiment with their ISA and structure with all the flexibility demanded in research projects. This paper is focused on the application of an ADL in a computer architecture course.

A common structure of an introductory computer architecture course is presented in the classical computer architecture book by Hennessy and Patterson [5]. The course starts with the instruction set architecture (ISA), i.e., presenting different instruction formats and how the processor manage to decode each instruction during execution. Some knowledge of assembly language programming is exercised at this moment. After understanding how an ISA is built, the student is ready to learn how instructions are really executed, how the data and computation flow inside the processor. The truth is: it is difficult to students to realize how all these features are implemented, and how they really work together inside a micro-processor without a tool to experiment with. This is the point were a software toolkit based on an ADL becomes very useful. Students can grab the knowledge about ISAs and pipelines from books and classes, and then fix it through the implementation of a processor model using an ADL, and get a simulator to experiment with and really *see* the whole thing running.

ArchC [6, 9] is an open-source ADL that fits very well in this context. It is being used as a support tool for computer architecture courses in the Institute of Computing, at University of Campinas, Brazil. Since the language, documentation, its parser, and simulator generator tools are all in public domain on the Internet, it is easy to students to get and to start using ArchC. This paper shows how to use ArchC MIPS models to illustrate computer architecture courses following exactly the structure presented in [5]. Moreover, we present how modeling modern computer architectures may be a good exercise for students.

The remaining of this paper uses the Hennessy and Patterson's book as a guide for presenting architecture concepts were ArchC may be a useful tool for illustration and experimentation. The text is organized as follows: Section 2 mentions some related work, Section 3 contains a brief introduction to the ArchC language, Section 4 shows how the ADL may be a useful tool on a computer architecture course, covering instruction set introduction, single-cycle, multi-cycle, pipelined, and memory hierarchy examples, Section 5 shows that an ADL enables teachers to introduce modern architectures even in an introductory course. Finally, we present our conclusions in Section 6.

2 Related Work

Considering automatic generation of a software toolkit for architecture exploration, one can find several ADLs on the literature, like: nML [2], ISDL [3], EXPRESSION [4], and LISA [11]. But no work has been published reporting and/or exploring the didactical capabilities of these languages. In fact, they have a serious drawback considering their application on computer architecture education, since none of these languages has all its tools and models published on public domain. All ArchC tools mentioned in this work, along with several architecture models can be freely obtained from [6].

Architecture simulators like SPIM [7] or SimpleScalar [1] may be used for didactical purposes. SPIM is a MIPS assembly simulator, compatible with the R2000/R3000 processors. It reads and executes assembly language code, but is not capable of executing binary files. SimpleScalar offers a MIPS like ISA, called PISA, for didactical purposes, along with a GCC port to this target. MASE [10] is a graphical simulation environment built on top of SimpleScalar. RaVi [8] comprises a set of multimedia MIPS based-modules for dynamic visualization of hardware behavior. These approaches do not provide automatic retargetability of their simulators and do not offer the flexibility of describing architecture behaviors in several levels of abstraction, or the easiness to model new architectures as provided by C++ based ADLs.

3 The ArchC Architecture Description Language

ArchC is an architecture description language initially conceived for processor architecture description, aiming to facilitate and accelerate processor description, combined with enough expression power to model several classes of architectures (RISC, CISC, DSPs, etc). ArchC allows users to fast explore a new ISA by automatically generating software tools, like SystemC simulators. Nowadays, ArchC is capable of describing processors as well as a memory subsystem. Memory hierarchies can be declared, containing several levels of memories and caches. Caches can be configured to simulate different set associativities, write polices, replacement strategies, and line sizes.

A processor architecture description in ArchC is divided in two parts, making clear the necessity of both behavioral and structural information. The `Instruction Set Architecture (AC_ISA)` description is where the designer provides details about instruction formats, size and names combined with all information necessary to decoding and the behavior of each instruction. In the `Architecture resources (AC_ARCH)` description, he/she informs ArchC about storage devices, pipeline structure, memory hierarchies, etc. Based on these two descriptions, ArchC will generate a behavioral simulator written in SystemC for the architecture, that may be purely functional or cycle-accurate, depending on the abstraction level used for instruction behavior descriptions. One important characteristic is that instruction behaviors, which are the largest part of the code in a processor model in ArchC, are described in pure C++ code. There is no restrictions, so model designers are capable of declaring their own methods and variables. C/C++ are largely used and it becomes very easy to students to start using ArchC. Moreover, there are complete GCC ports for MIPS and SPARC, including libraries to generate binary elf files ready to be loaded on ArchC simulators. This enables users to experiment with their own programs using ArchC simulators, and to execute real-world applications, including system calls emulation, like JPEG and MPEG coders.

We are going to use several pieces of ArchC code to illustrate our examples in this text, explaining some characteristics of the language as necessary, but readers should refer to the *Archc Language Reference Manual* [9] for a complete description of the ArchC's syntax and tools.

4 ArchC as a Support Tool for Teaching Computer Architecture

This section describes how ArchC can be a useful tool for developing projects and exercises, on a computer architecture course based on the classical didactical book from Hennessy and Patterson: "Computer Organization & Design" [5]. As this reference will be frequently mentioned throughout this text, for the sake of simplicity, we are going to refer to this book as *COD* from this point on.

```
AC_ISA(mips){

  ac_format Type_R = "%op:6 %rs:5 %rt:5 %rd:5 0x00:5 %func:6";
  ac_format Type_I  = "%op:6 %rs:5 %rt:5 %imm:16:s";
  ac_format Type_J  = "%op:6 %addr:26";

  ac_instr<Type_R> add, addu, subu, multu, divu, sltu;
  ac_instr<Type_I> lw, sw, beq, bne;
  ac_instr<Type_I> addi, andi, ori, lui, slti;
  ac_instr<Type_J> j, jal;

  ISA_CTOR(mips){

    load.set_asm("lw %rt, %imm(%rs)");
    load.set_decoder(op=0x23);

    store.set_asm("sw %rt, %imm(%rs)");
    store.set_decoder(op=0x2B);

    add.set_asm("add %rd, %rs, %rt");
    add.set_decoder(op=0x00, func=0x20);

    addu.set_asm("addu %rd, %rs, %rt");
    addu.set_decoder(op=0x00, func=0x21);

                                ...
  };
};
```

Figure 1: MIPS ISA Description in ArchC

4.1 Instruction Types, Assembly Mnemonics and Decoding

The use of ArchC in a computer architecture course can start as early as in the third chapter of COD, where the authors introduce the instruction representation inside a computer: the Instruction Set Architecture (ISA).

First, the MIPS assembly language is introduced, followed by information on how to translate it to the MIPS machine language. In order to do this translation students must learn about MIPS instruction formats and binary encoding, and finally how machine code is decoded by the processor. This is exactly the information contained inside an `AC_ISA` description, as illustrated by Figure 1. Students can do an `AC_ISA` implementation using the knowledge they are gathering from the book on instructions, assembly syntax, formats, fields and decodification, and also do some experiments with the decoder generated by ArchC, issuing some instructions in binary format to see if they supplied enough decoding information for each instruction in the ISA.

4.2 The Single-cycle and Multi-cycle Datapaths

By doing the simple exercises related in the previous section, the students can have their first contact with instruction set definitions and with the ArchC tools. This experience is important to the following tasks.

We call a functional model in ArchC a model that does not have any timing information, i.e., a model that executes one instruction per cycle. That is exactly the first example of a datapath construction presented in the book. Of course, the high abstraction level of ArchC models does not comprise functional units and signals, but the exercise of modeling the behavior of each instruction in C++ and trying to figure out which functional units and signals would be necessary to build a single-cycle datapath capable of executing such a behavior may be very instructive. COD authors suggest exercises like: write a functional simulator for the single-cycle and the multi-cycle versions of the datapath presented in the book using a hardware description language, like Verilog or VHDL. Authors predicted that students would take a week to develop each one of these simulators. Both of them can be easily coded in ArchC, for such a short and simple instruction set. Figure 2 (A) shows the functional version of the MIPS `add` instruction behavior, and Figure 2 (B) shows its multi-cycle version, according to the description given in COD pages 385–388. We estimate that a functional model of a fifteen or twenty-instruction of a MIPS-like ISA could be developed in three or four hours of work, after going through the theory presented in the book. Remember that instruction behaviors are written in C++, which is a language that most of the students are very familiar with. Another three or four hours of work would be enough to refine this functional model to a multi-cycle model, which is exactly the process of re-writing instruction behaviors to make them look like the example in Figure 2 (B).

4.3 The Multi-cycle Datapath with Pipeline

After the single and multi-cycle datapath concepts are sedimented, it is natural to introduce the concept of

```
void ac_behavior( add ){

    ac_pc += 4;
    RB.write(rd, RB.read(rs) +
                 RB.read(rt));
};
```

(A)

```
void ac_behavior( add, cycle ){

     switch( cycle ){
          case 1:
               ac_pc += 4;
               break;
          case 2:
               A = RB.read(rs);
               B = RB.read(rt);
               break;
          case 3:
               ALUout = A + B;
               break;
          case 4:
               Rb.write(rd, ALUout);
               break;
          default:
               break;
     }
};
```

(B)

Figure 2: Single-cycle and Multi-cycle Behavior Description

pipelining, where multiple instructions are overlapped in execution. The ArchC language contemplates this approach by supporting pipelining, in which we evolve from a functional model to a cycle accurate model, differing from the first in the timing precision. While functional models execute all instructions in one clock cycle, a cycle accurate model has instruction behavior descriptions reflecting the real number of clock cycles taken by the instructions. The great benefit brought by the use of an ADL like ArchC is that students can take advantage of his previous developed functional model to, gradually refine it to a new pipelined implementation, and get it running as a software simulator for the target architecture.

ArchC provides the necessary constructions for pipeline simulation but, in order to get a complete model of the MIPS architecture with a pipeline, students will have to consider its mechanisms, like register forwarding or data hazard detection, inside their behavior description. The first step is to insert information regarding the pipeline registers and pipeline stages into the functional model, as shown in Figure 3. It is important to emphasize that the same instructions continue to exist and the Instruction Set Architecture (ISA) remains unchanged for the pipelined model, i.e., no modification on the `AC_ISA` description file, showed in Figure 1, is required.

The pipelined model divides the instructions in minor execution units to be executed in the several pipeline stages, which are declared using the `ac_pipe` keyword. However, the stages communicate to each other through the pipeline registers, which have their particular structures, i.e, their fields. The students, while modeling the pipeline, have to declare the pipeline register structures, and thus, they have inevitably to have a complete understanding about how the pipeline works, what fields are necessary in each pipeline register, and what are their functions. Pipeline registers are declared by the combination of the `ac_format` and the `ac_reg` keywords. Let us take the ID/EX pipeline register of a MIPS processor, like described in COD, as an example. Such processor has a 5-stage pipeline (Fetch, Identification, Execution, Memory Access and Write-Back), and four pipeline registers (IF/ID, ID/EX, EX/MEM, MEM/WB). The student should declare a format (field structure) for each register and give a name for it, as it is done for registers in Figure 3.

From this point on, all the necessary structural information is already inserted in the model. But before running this new model, it is necessary to take the second step, which is the refinement of the instruction behaviors. This is necessary because in a pipelined model the instructions are split into several parts, and each of these parts are executed in a different pipeline stage. It is important to notice that, in despite of the higher abstraction level of ArchC models if compared with the datapaths presented in the book, student must have consolidated the concepts of pipelining and its physical structure in order to be able to model it correctly.

ArchC automatically fetches the instruction pointed by its program counter (`ac_pc`), i.e., the student does not have to worry about the instruction fetch, but do need to take care of the PC increment. ArchC also generate a decoder for the architecture, based on the information provided in the `AC_ISA` description. But in a pipelined model there are other tasks that must be performed at the ID stage. Let us take the instruction `add` as an example. Still in the ID stage, the contents of the ID/EX register must be filled-up, from where the execution stage will access the correct values for the

```
AC_ARCH(mips){
  ...
  ac_format F_IF_ID  = "%npc:32";
  ac_format F_ID_EX  = "%npc:32 %data1:32 %data2:32 %imm:32:s rs:5 %rt:5 %rd:5
                        %regwrite:1 %memread:1 %memwrite:1";
  ac_format F_EX_MEM = "%alures:32 %wdata:32 %rdest:5 %regwrite:1 %memread:1 %memwrite:1";
  ac_format F_MEM_WB = "%wbdata:32 %rdest:5 %regwrite:1";

  ac_reg<F_IF_ID>  IF_ID;
  ac_reg<F_ID_EX>  ID_EX;
  ac_reg<F_EX_MEM> EX_MEM;
  ac_reg<F_MEM_WB> MEM_WB;

  ac_pipe   PIPE = {IF,ID,EX,MEM,WB};

  ARCH_CTOR(mips) {
    ...
```

Figure 3: Inserting Structural Information Regarding the Pipeline and its Registers into the MIPS I ArchC Description

operand registers and the program counter of the current instruction, along with some *signals* to control execution into further stages. Figure 4 shows an excerpt of a possible `add` behavior description. After that, the `add` instruction goes to the execution stage (EX), where the actual computation of the sum takes place, followed by setting the fields in the next pipeline register. In the case of the `add` instruction, the memory access stage (MEM) has just to copy the EX_MEM register contents to the MEM_WB register and, finally the instruction reaches the write back stage (WB), where the result of the sum is stored into the correct destination register.

```
  ...
case ID:
  ID_EX.regwrite = 1;
  ID_EX.memread = 0;
  ID_EX.memwrite = 0;
  ID_EX.npc = IF_ID.npc;
  ID_EX.data1 = RB.read(rs);
  ID_EX.data2 = RB.read(rt);
      ...
case EX:
  EX_MEM.alures= ID_EX.data1 + ID_EX.data2;
  EX_MEM.regwrite= ID_EX.regwrite;
  EX_MEM.rdest   = ID_EX.rd;
         ...
  break;
   ...
```

Figure 4. Modeling Instruction Behaviors Considering the Pipeline

The main point of this example is that, assuming that students are familiar with the basics of C/C++ programming languages, this operations are quite simple to be implemented, because they are nothing else than simple C++ statements. One very important thing when applying ArchC to computer architecture classes, is that the simplicity of the language brings the focus of the work to the architecture being described, and do not add an extra burden to the learning process due to syntax details of the language. Another advantage of using ArchC is its flexibility, since students are able to call their own functions inside behavior description methods, in order to debug the simulation. This facilitates the visualization of the pipeline internals, i.e., the student is free to watch whatever he wants by printing such data on screen, while running the simulator.

Notice that the code presented in Figure 4 does not consider the possibility of data hazards. With such a model in hands, a teacher could give some small examples of MIPS machine code where, for example, an instruction needs to use a register, but this register is being used by another instruction inside the pipeline, i.e, it is still not written in the register bank. Asking students to identify the problem and to add, for example, a register forwarding mechanism to their model can be a very useful exercise, which would help to solidify some important pipelining concepts. When modeling a processor with ArchC, a student can implement data forwarding in a way that is very similar to the described by didactical books. Lets take the *if statement* showed in COD, page 480, as a didactical example on how to insert data forwarding to our pipeline. In Figure 5 (A) we see how register forwarding is shown in the book, and in Figure 5 (B), how it is modeled in ArchC.

An ADL that generates C++ based simulators is also a well suitable tool for exercises like those suggested at the end of Chapter 6 in COD, mainly the last two exercises. The first asks students to collect statistics on data hazards for a C program and write a subroutine to model the five-stage pipeline presented in the book. Authors are asking for statistics like: number of instructions executed, number of data hazards, etc. This could be accomplished by implementing a

```
            BOOK:                                    ArchC MODEL:

if(EX/MEM.RegWrite and                          if((EX_MEM.regwrite == 1) &&
   (EX/MEM.RegisterRd != 0) and                    (EX_MEM.rdest != 0) &&
   (EX/MEM.RegisterRd = ID/EX.RegiterRs))          (EX_MEM.rdest != ID_EX.rs))
   ForwardA = 10;                                  operand1 = MEM_WB.wbdata;
               ...                                            ...

               (A)                                            (B)
```

Figure 5: Register Forwarding for the rs Register in Didactical Books and in an ArchC Implementation

pipelined model in ArchC. The last exercise asks for students to elaborate a model of the single-cycle datapath in a HDL like Verilog, and then refine it toward a pipelined model. A single-cycle or multi-cycle model implemented in ArchC, like those described in Section 4.2, can be refined toward the pipelined implementation required for this exercise. By using an ADL, this task certainly would be accomplished faster than by a HDL. A interesting approach is to divide the class in groups of students, and assign some of them to use an HDL like Verilog or VHDL, some to use SystemC, and some to write models using ArchC. At the end, students could share their experiences, pointing out the advantages and disadvantages of each approach. As suggested by the authors, this would be a project that would take students at least a month to be done. ArchC simulators are also capable of executing pipeline stalls and flushes so, there are several other possibilities of experiments that can be suggested to students, while teaching pipelining in a computer architecture course.

4.4 Memory Hierarchy

Continuing with our course based on COD, the next topic would be memory hierarchies. ArchC is capable of describing hierarchies composed of caches and memories distributed at different levels. Caches can be customized by the user by choosing parameters for associativity, number of lines, words per line, replacement strategy, and write policies. This is illustrated in the cache declarations contained in the example in Figure 6. The user creates the hierarchy by describing the connections among these devices, through the method `bindsTo`, as illustrated in the last two lines of the example.

So, by adding such a memory hierarchy description to our functional model, students can experiment different cache and hierarchy configurations. A possible exercise would be to choose a particular application, or a small set of applications, and let students experiment with cache parameters for a given hierarchy. They would be able to analyze the simulation results, to compare miss rates, and determine the best configuration for each application. ArchC simulators automatically keep track of all access to storage devices, so reporting miss rates and total number of accesses to each device declared in the AC_ARCH description.

```
AC_ARCH(mips){

  ac_cache   icache("dm", 128, "wt", "war");
  ac_cache   dcache("2w", 64, 4, "lru", "wt", "war");
  ac_mem   MEM:256K;

  ac_regbank RB:34;
  ac_wordsize 32;

  ARCH_CTOR(mips){
    ac_isa("mips_isa.ac");

    icache.bindsTo( MEM );  //Memory hierarchy
    dcache.bindsTo( MEM );  //construction
  };
};
```

Figure 6. Memory Hierarchy Declaration in ArchC.

After going through all the theory in COD's 7th chapter, students have a bunch of exercises to work on, in order to fix the concepts presented in the book. In addition to the theoretical exercises, students may use ArchC simulators to experiment with memory hierarchies. For example, one of the exercises proposed by the authors ask students to analyze a trace produced by GCC for different cache organizations. For MIPS and SPARC architectures, there is a GCC port available for generating code to be run in ArchC simulators [6]. So, students can perform such an experiment using real-world applications, like JPEG or MPEG coders or some cryptography algorithm, using more than one processor and a number of different cache organizations. Moreover, they are able to compile their own programs, or examples provided by teachers, in order to do this kind of analysis. The simulation statistics provided by ArchC simulators combined with some pre-defined miss and/or hit penalty may be used to compute the performance numbers for each configuration tested. The most important part, they would actually see that the memory hierarchy may have a strong impact on the performance, and get a felling of how hard may be to tune a processor + memory system to a given real-world multi-media application, a common task for embedded systems designers.

5 Modeling Modern Architectures

An alternative approach for teaching computer architecture would be to use the MIPS architecture, following the COD book, in classes and to adopt different projects to be developed by the students. In one of the computer architecture courses at IC-UNICAMP, students were divided into several groups, and each one of those groups was assigned to a different project. The project was basically to develop a new functional model of a real-world architecture, most of them largely used in the industry as part of SoCs and embedded systems. For example, in this first semester of 2004, there are groups developing models of Intel XScale, IBM/Motorola PowerPC, Altera Nios, Infineon TriCore 2, OpenCores OR1K, and Motorola 68k/ColdFire ISAs as part of the computer architecture course. These functional models are heavily based on the instruction set, not concerning specific pipeline or cycle-accurate details of these complex architectures. They all execute one instruction per cycle. But the experience of getting into contact with different ISAs, more complex and modern than the simple RISC MIPS-I that is usually used in this kind of course, is instructive and attractive for students. They have the opportunity of learning details about architectures that are state-of-the-art in the industry, which is an extra motivation for the course. Moreover, students are getting some experience on how to build cross-compilers for GCC, in order to be able to create binary files to be loaded in their simulators.

6 Conclusions

ArchC is an architecture description language recently developed by the Computer Systems Laboratory (LSC), at IC-UNICAMP. Its based on C++ and automatically generates SystemC simulators from processor descriptions. ArchC is also capable of describing memory hierarchies. Its simulators have several capabilities to help simulation debugging and statistics collection that become useful in computer architecture education.

The language has been used for the last two semesters in computer architecture courses at IC-UNICAMP, both at the undergraduate and graduate levels, and the feedback received from the students was very positive. Among other projects, students developed functional models for real-world architectures like PowerPC and XScale. They got very motivated while developing their projects, and some of them even contributed with improvements on the ArchC tools that ended up as new features adopted by the ArchC Team in the official distribution, resulting in a kind of integration between education and research. The experience of developing architecture models gave students a deeper understanding of the concepts recently learned from the book and classes.

7 Acknowledgments

We would like to thank FAPESP (Grants 00/14376-2, 03/11674-0, and 2000/15083-9) and CNPq (ChameLeon Project) for the financial support to this work. We are also very grateful to all the students and teachers that are using ArchC, for education and/or research, whose feedback has been extremely valuable to the continuous improvement in ArchC tools.

References

[1] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-1996-1308, University of Wisconsin. Computer Sciencies Department., 1996.

[2] Markus Freericks. The nML Machine Description Formalism. Technical report, Technische Universitt Berlin, Fachbereich Informatiky, July 1993. Updated and Revised Version 1.5(Draft).

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

[4] A. Halambi, P.Grun, V.Ganesh, A.Khare, N.Dutt, and A.Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *in Proc. European Conference on Design, Automation and Test(DATE)*, March 1999.

[5] J.L. Hennessy and D.A. Patterson. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, 1998.

[6] http://www.archc.org. The ArchC Resource Center.

[7] http://www.cs.wisc.edu/ larus/spim.html. SPIM MIPS R2000/R3000 Simulator Homepage.

[8] Peter Marwedel and Birgit Sirocic. Multimedia Components for the Visualization of Dynamic Behavior in Computer Architectures. In *Proceedings of the Workshop on Computer Architecture Education*, 2003.

[9] The ArchC Team. *The ArchC Architecture Description Language Reference Manual*. Computer Systems Laboratory (LSC) - Institute of Computing, University of Campinas, http://www.archc.org, 2004.

[10] C. T. Weaver, E. Larson, and T. Austin. Effective Support of Simulation on Computer Architecture Instruction. In *Proceedings of the Workshop on Computer Architecture Education*, 2002.

[11] Vojin Zivojnovic, Stefan Pees, and Heinrich Meyr. Lisa - machine description language and generic machine model for hw/sw co-design. In *Proceedings of the IEEE Workshop on VLSI Signal Processing, San Francisco*, 1996.