# ArchC: A SystemC-Based Architecture Description Language

Sandro Rigo, Guido Araújo, Marcus Bartholomeu, Rodolfo Azevedo
Computer Systems Laboratory - Institute of Computing, University of Campinas
Cidade Universitária Zeferino Vaz, Po. Box 6176, Campinas-SP, Brazil
{srigo, guido, bartho, rodolfo}@ic.unicamp.br

## Abstract

*This paper presents an architecture description language (ADL) called ArchC, which is an open-source SystemC-based language that is specialized for processor architecture description. Its main goal is to provide enough information, at the right level of abstraction, in order to allow users to explore and verify new architectures, by automatically generating software tools like simulators and co-verification interfaces. ArchC's key features are a storage-based co-verification mechanism that automatically checks the consistency of a refined ArchC model against a reference (functional) description, memory hierarchy modeling capability, the possibility of integration with other SystemC IPs and the automatic generation of high-level SystemC simulators. We have used ArchC to synthesize both functional and cycle-based simulators for the MIPS, Intel 8051 and SPARC V8 processors, as well as functional models of modern architectures like TMS320C62x, XScale and PowerPC.*

## 1. Introduction

With the advent of System-on-Chip (SoC) designs, it is becoming possible to design an entire embedded system onto a single chip. As we approach the availability of 100 Million gates, for the 90nm VLSI technology, a new scenario has been drawn in which SoCs can be composed of a number of specialized processors interconnected by an elaborate *Network-on-a-Chip* (NoC). Moreover, as new specialized processor architectures are been proposed, it becomes evident the need for flexible simulation models that enables the designer to tailor processor ISA to the application. Such tools are commonly based on processor models written in some architecture description language (ADL).

In this paper we introduce a new SystemC-based architecture description language (ADL) called ArchC. SystemC [12] is among a group of design languages and extensions being proposed to raise the abstraction level for hardware design and verification. SystemC is entirely based on C/C++ and the complete simulation library source code is freeware. SystemC is composed by a set of C++ class libraries, that extends the language to allow hardware and system-level modeling. Though SystemC supports a wide range of computation models and abstraction levels, it is not possible to extract information from a generic SystemC processor description in order to automatically generate tools to experiment and evaluate a new *Instruction Set Architecture* (ISA). ArchC is a simple language, specialized for processor architecture description, that follows C++/SystemC syntax style. Its main goal is to provide enough information, at the right level of abstraction, in order to allow users to explore and verify a new architecture by automatically generating software tools like assemblers, simulators, and co-verification interfaces. ArchC's key features are a storage-based co-verification mechanism that automatically checks the consistency of a refined ArchC model against an ArchC reference description, memory hierarchy modeling capability and the possibility of integration with other SystemC IPs. Functional and cycle-based simulators have been synthesized starting from ArchC descriptions for the MIPS (implementing the MIPS-I ISA), Intel 8051 and SPARC processors. Functional models for the Texas TMS320C62x, Intel XScale, Motorola ColdFire, PowerPC, Altera NIOS, OpenCores OR1k, and Hitachi SH-4 where also developed in ArchC, some of them are still in the verification phase.

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. ArchC fits very well in this context, since the ArchC parser and simulator generation tools, along with several architecture models, are published on public domain and can be freely downloaded from [8]. To the best of our knowledge, ArchC is the first ADL being applied in universities for educational purposes.

The remaining sections of this paper are organized as follows. Section 2 discusses some previous work on architecture description languages. Section 3 introduces the ArchC syntax and semantics. Section 4.1 presents the co-verification mechanism provided by ArchC. Section 4 describes ArchC tools and simulators, Section 5 presents some experimental results, and finally, Section 6 summarizes our conclusions and describes the future work.

## 2. Related Work

The nML [1] ADL is based on the information typically available in the programmer's manual of a processor, which consists of a list of instructions and corresponding register transfers, binary encoding and assembly mnemonics. The abstraction level of nML is the instruction set, which is described through an attributed grammar. Hartoog et al [9] presented a set of tools generated from a nML description of a processor and concluded that nML does not support multi-cycle instructions and straightforward extensions of nML can only support very simple pipelines, i.e., in nML it is not possible to produce cycle-accurate simulators for processors with complex execution schemes, like many DSPs. The nML implicit program counter is inconvenient in some circumstances, as when multi-word and multi-cycle instructions are used. These limitations are not present in ArchC.

The LISA [17] ADL was primarily designed for automatic retargeting of fast compiled simulators that are cycle and bit-accurate. When introduced, the main contribution of LISA was its operation-level description of the pipeline and sequencing model. Hoffmann et al [7] used LISA to create a framework for hardware/software co-simulation, by connecting LISA models to SystemC hardware models. Nohl et al [11] presented a new technique to mix compiled and interpretive simulation. Efficient support of complex instruction formats seems to require extensive coding in LISA. Companies like Axys Design and LISATek have been able to deliver proprietary LISA based processor models for a number of architectures, but no public release compiler and/or toolkit is available for this language.

EXPRESSION [5] is an ADL focused on architecture design space exploration for SoCs and automatic generation of a compiler/simulator toolkit. EXPRESSION supports specification of memory subsystems, which seems to be the key feature of the language. Reshadi et al [13] presented a technique called instruction-set compiled simulation (IS-CS), designed to improve the flexibility of compiled simulators without slowing down the performance to the interpretive simulation level.

The Instruction Set Description Language for Retargetability (ISDL) [4] was designed to be an ADL for compiler retargetability, specially focused on VLIW machines. Like nML, ISDL is a purely behavioral language also based on an attributed grammar. It is not clear if ISDL tools are capable of extracting the correct behavior for pipelined architectures with complex execution schemes. Pipeline flushes does not seem to be possible. Another strong constraint: ISDL is not able to model multi-cycle instructions of variable length [3].

ArchC has its syntax totally based on C++ and SystemC. It was designed focused on SystemC users, i.e., aiming to make these users more comfortable with the language, since its syntax is very close to those they are already used with. The user has just to learn a small set of keywords. ArchC relies on both structural and behavioral information of the architecture. ArchC has many features that distinguish it from other ADLs, like: non-proprietary SystemC compatibility, behavior modeling at several abstraction levels, behavior hierarchy, etc. Its key features are a storage-based co-verification mechanism that automatically checks the consistency of refined ArchC model against ac ArchC functional description, memory hierarchy modeling capability, the possibility of integration with other SystemC IPs. To the best of our knowledge it is currently the only ADL totally committed to the SystemC effort, and thus it is published on public domain so that all source code and tools presented in this paper can be freely downloaded.

## 3. ArchC Syntax and Semantics

An architecture description in ArchC is divided in two parts: the *Architecture Resources* (`AC_ARCH`) description and the *Instruction Set Architecture* (`AC_ISA`) description. In the `AC_ARCH` description the designer provides ArchC with information about storage devices, pipeline structure, memory hierarchy and all the processor resource information available in the ISA manual. In the `AC_ISA` description, the designer provides ArchC with details about each instruction, like: (a) format, size and assembly language syntax; (b) opcode information required to decode it; and (c) instruction behavior. Based on these two descriptions, ArchC automatically generates a simulator for the architecture.

```
AC_ARCH(mips){
  ac_cache  icache("dm", 16, 4, "wt", "war");
  ac_cache  dcache("2w", 32, 4,  "lru", "wb", "wal");

  ac_regbank RF:32;
  ac_pipe    pipe = {IF, ID, EX, MEM, WB};
  ac_wordsize 32;

  ac_format RegFmt1 = "%npc:32";
  ac_format RegFmt2 = "%rd:5 %rs:32 %rt:32 %imm:32";

  ac_reg<RegFmt1> IF_ID;
  ac_reg<RegFmt2> ID_EX;
         ...
  ARCH_CTOR(mips){
    set_endian("big");
  }
};
```

**Figure 1. MIPS `AC_ARCH` resource declaration.**

## 3.1. Architecture Resources (`AC_ARCH`)

Architecture resources are described in the `AC_ARCH` description. ArchC provides a set of data types that can be used to declare registers (`ac_reg`), register banks (`ac_regbank`), caches (`ac_cache`), pipeline (`ac_pipe`), pipeline stages (`ac_stage`), and other (micro) architecture modules. We illustrate the basic resource data-types available in ArchC using a fragment of the MIPS `AC_ARCH` file, shown in Figure 1.

The MIPS processor has a five stage pipeline, 32 general purpose registers plus two special registers used for multiplication and division. In the MIPS example of Figure 1, the keyword `ac_regbank` is used to define the processor register file. Caches are declared using the `ac_cache` keyword. For example, in Figure 1 a direct-mapped, 16Kb, 4 words/line instruction-cache (`icache`) is declared. ArchC also provides other overloaded cache constructors that allow of several choices of memory update policies, set-associativity and other cache parameters. Notice the two-way set associative, 32Kb write-back, write-allocate data-cache declared in the example.

Microarchitecture designers frequently need to access instruction fields or control signals stored in registers. In order to enable that, ArchC allows the designer to declare register formats, exactly as he/she does for instructions (Section 3.2). In ArchC, a format is declared through the `ac_format` keyword. The designer must provide a name and a string, representing the format subdivision into fields. Take a look at `RegFmt1` format declaration in the example. The declaration of a field starts by a `%` character, followed by an identifier that becomes the field's name. The number after the colon indicates the size of the field. So, the string "%rs:5" declares a 5-bit field named `rs`. The designer assigns a format to a register by using the keyword `ac_reg` and a syntax similar to C++ *templates*. In the example, format `RegFmt2` is associated to register `ID_EX`. This allows the designer to assign to (and read from) each register field individually when describing instruction behaviors. The behavior description of the `add` instruction in Figure 3, uses `ID_EX.rt` to read the value stored in the `rt` field of the `ID_EX` pipeline register.

Pipelines are created through the keyword `ac_pipe` and are composed by a name and a list of pipeline stages. Pipeline stages listed in an `ac_pipe` declaration are assumed to execute instructions in the same order that they were declared. Stages also carry methods that allow pipeline flush and stall operations.

## 3.2. Instruction Set Architecture (`AC_ISA`)

The `AC_ISA` description provides ArchC with all information it needs to automatically construct a decoder and the skeleton of a SystemC (C++) source file where the designer inserts the behavior of each instruction in the ISA, as show in Figure 2 from our MIPS description.

The MIPS architecture uses the so called `R` format (or type) for arithmetic instructions. The example in Figure 2 starts with the declaration of this format (`Type_R`), by using the `ac_format` keyword, similarly as it was done before (Section 3.1) to assign a format to a register. Still following the example in Figure 2, notice that instructions are declared using the keyword `ac_instr`. The designer must give a name and assign a previously declared format to any new instruction. The example shows the declaration of the `add` and `load` instructions.

```
AC_ISA(mips){

 ac_format Type_R="%op1:6 %rs:5 %rt:5 %rd:5 0x00:5 %op2:6";

 ac_format Type_I="%op1:6 %rs:5 %rt:5 %imm:16";

 ac_instr<Type_R> add;
 ac_instr<Type_I> load;

 ISA_CTOR(mips){
   add.set_asm("add %rs, %rt, %rd");
   add.set_decoder(op1=0x00, op2=0x20);

   load.set_asm("lw %rt, %imm(%rs)");
   load.set_decoder(op1=0x23);

 };
};
```

**Figure 2. MIPS ISA Description**

The `ISA_CTOR` section of an `AC_ISA` declaration is used to construct instructions from their predefined formats. Instructions carry methods `set_asm` and `set_decoder` that are respectively used to define the instruction assembly syntax and the contents of its opcode

```
 void ac_behavior( add, stage ){
    switch(stage)
     {
     case IF: ...
     case ID: ...
       break;
     case EX:
       EX_MEM.alu_result = ID_EX.rs + ID_EX.rt;
              ...
       break;
     case MEM:
       MEM_WB.alu_result = EX_MEM.alu_result;
       MEM_WB.rd = EX_MEM.rd;
              ...
       break;
     case WB:
       RF.write(MEM_WB.rd, MEM_WB.alu_result);
     default:
       break;
     }
};
```

**Figure 3. Instruction `add` behavior description**

fields. The former information can be used to synthesize the processor assembler and the latter a decoder for the simulator.

**3.2.1. Modeling instruction behavior in ArchC** Figure 3 shows how the designer describes the behavior of each instruction in ArchC. This example is a fragment extracted from a description of the MIPS `add` instruction. The body of an `ac_behavior` method is a piece of code that executes operations by accessing declared storage devices and instruction fields. In general, if the designer declares a pipeline, he/she has to provide the behavior of each instruction for each pipeline stage. For example, during the execution of stage `EX` the designer uses fields `ID_EX.rs` and `ID_EX.rt` from pipeline register `ID_EX` to compute the result of instruction `add` into the `EX_MEM.alu_result` field of pipeline register `EX_MEM`.

If an instruction does not execute anything in a given pipeline stage, the `case` clause in the `switch` statement of the instruction behavior should be empty for that stage. Moreover, if instead of a pipeline, a processor has multi-cycle instructions, the `ac_behavior` method takes as argument the cycle number, also using a `switch` statement to select and execute the appropriate cycle, and `case` clauses to divide the instruction behavior into several cycles.

The main point about behavior description in ArchC is its flexibility. The `ac_behavior` method is written in pure SystemC (C++) code. Designers' own C++ functions can be called from an `ac_behavior` method, providing them with a lot of expression power for debugging and extension features. Moreover, there is no assumption that the designer is writing code at any determined abstraction level. For example, one could have written a functional model of the MIPS without a pipeline if cycle-accuracy was not an important feature at the early stages of the design. In that case, the instruction behavior could be as simple as the piece of code in Figure 4, i.e., just a sequence of C++ statements to execute the given MIPS instruction behavior, resulting in a simulator that runs at one instruction per cycle. This ability to express behaviors in several levels of abstraction is another strong feature in ArchC.

```
void ac_behavior( add ){
  RB.write(rd, RB.read(rs) + RB.read(rt));
}

void ac_behavior( load ){
  RB.write(rt, DM.read(RB.read(rs)+ imm));
}
```

**Figure 4. Functional Behavior Description in ArchC**

Often, there are many instructions in a particular architecture that execute exactly the same task as part of their behavior. In order to factor out this behavior, writing it once and using it for all instructions of the same type, ArchC provides the designer with the possibility of overloading the `ac_behavior` method, so that it can take an instruction format as argument. Consider the MIPS processor as an example, all instructions that were declared with the `Type_R` format associated to it execute a couple of tests, showed in Figure 5, *before* running its own behavior, so that they can do register forwarding for both instruction operands, `rs` and `rt`. The code in Figure 5 happens to be as simple and clear as it is presented in the Hennessy and Patterson's classical architecture book [6]. The same strategy can be used for coding data hazard detection. There is still another situation, where the designer wants that all instructions in the architecture execute a piece of code before running its own behavior method. This is also possible in ArchC by describing a generic instruction behavior, that is a behavior method that belongs to all instructions. The designer would follow the same style used above and pass the keyword `instruction` as the argument of the behavior method. This forms a behavior hierarchy, where generic instruction behavior is executed first, followed by the format behavior, and finally the specific instruction behavior is executed. The same sequence is performed by each pipeline stage in the case of a pipelined architecture. Notice that, for pipelined or multi-cycle instructions, both format and generic instruction behaviors can be written in a cycle-accurate fashion by means of a C++ switch statement, just like it is done for instruc-

```
void ac_behavior( Type_R ){
  switch( stage ) {
                  ...
  case _EX:
    /* Checking forwarding for the rs register */
    if ( (EX_MEM.regwrite == 1) &&
       (EX_MEM.rdest != 0) &&
       (EX_MEM.rdest == ID_EX.rs) )
           operand1 = EX_MEM.alures.read();
    else if ( (MEM_WB.regwrite == 1) &&
          (MEM_WB.rdest != 0) &&
          (MEM_WB.rdest == ID_EX.rs) &&
          (EX_MEM.rdest == ID_EX.rs) )
           operand1 = MEM_WB.wbdata.read();
    else
      operand1 = ID_EX.data1.read();

    /* Checking forwarding for the rt register */
    if ( (EX_MEM.regwrite == 1) &&
       (EX_MEM.rdest != 0) &&
       (EX_MEM.rdest == ID_EX.rt) )
           operand2 = EX_MEM.alures.read();
    else if ( (MEM_WB.regwrite == 1) &&
          (MEM_WB.rdest != 0) &&
          (MEM_WB.rdest == ID_EX.rt) &&
          (EX_MEM.rdest == ID_EX.rt) )
           operand2 = MEM_WB.wbdata.read();
    else
      operand2 = ID_EX.data2.read();
    break;
                ...
  }
}
```

**Figure 5.** `Type_R` **format behavior description**

tions. By using ArchC's behavior hierarchy, we were able to factor out many operations that would have to be repeated into several instruction behaviors, what ended up saving a great amount of redundant code in our models. A similar effect may be possible in an attributed grammar based ADL, but never with the same flexibility, cycle-accuracy and expression power as in pure C++ code.

## 4. ArchC Tools

### 4.1. Storage-based Co-verification

Due to the current growth on complexity and reduced time to market in embedded system projects, verification tools are gaining more and more importance in the whole process. The natural flow of a project is to start with a model in a higher level of abstraction and gradually refine it toward synthesis. As we see it, a design of an architecture model using ArchC should start by the elaboration of a description at the functional level. After validating the whole ISA using this high-level model, the designer can refine it, including pipeline structure, or making it a multi-cycle architecture model, in order to get a cycle-based description to experiment with. By describing an architecture in ArchC, the designer automatically gets a SystemC behavioral simulator of the processor in each of these phases.

Since it is performed manually, this refinement process is clearly error prone, making a tool for checking these refined

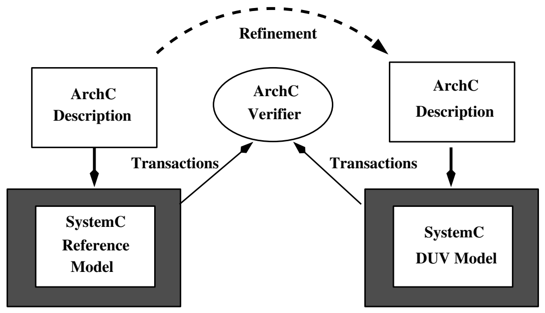

**Figure 6. ArchC Co-verification Methodology**

behaviors against a reference model very useful during the new model's verification. ArchC provides a co-simulation tool allowing the designer to couple these two different models of the architecture. Both models run the same application and the ArchC verifier checks if they are consistent. Figure 6 illustrates the ArchC co-verification methodology, where usually an ArchC functional model is used as the reference model and the refined model is the DUV(*Device Under Verification*), assuming that it contains all the storage elements the designer wants to verify. Here, the refined model may be generated from a refined ArchC description. The ArchC verification approach is a transaction-based verification methodology that we call *Storage-Based Co-Verification*. It is based on tracking down every update to the storage devices of both models, marking them with timestamps to show when they happened. By comparing the sequence of transactions generated throughout the execution, the ArchC verifier can tell if both models are consistent.

When co-simulating two ArchC models, the designer does not have to make any change in the descriptions, it is only necessary to run the ArchC Simulator Generator (Section 4.2 ), passing the command-line option to generate a simulator prepared to run in the co-verification mode. When the designer simulates the refined model, coupled with a correct ArchC reference model, ArchC's co-verification mechanism issues an error message and saves a transaction log every time the values matched are inconsistent. Using this information, the designer rapidly identifies which instruction was executing at that time and which values were stored in register banks, memories and pipeline registers. The whole process of identifying and correcting this kind of error is thus accelerated.

It is important to mention that both models run in parallel along with the *ac_verifier* tool, using three different Linux processes that communicate through IPC (Interprocess communication) mechanisms provided by the OS. We designed our co-verification algorithm to work in two situations. First, both the behavioral and the refined models are timing-accurate and the designer wants to assure that updates are executed at the same simulation (cycle) time on both models. Second, the ArchC reference model is not cycle-accurate and the designer just wants to verify the consistency of the update sequences generated by the models.

### 4.2. The ArchC Simulator Generator and Simulators

We called as `ArchC Simulator Generator (acsim)` the tool that takes an ArchC description, composed by an instruction set architecture description (`AC_ISA`) and an architecture resource description (`AC_ARCH`), and generates a behavioral model of the architecture. `Acsim` uses two other tools that are not visible to the user: the Archc Pre-Processor (`acpp`), which is composed by a lexical analyzer and a parser for the language, and a decoder generator. The parser extracts information from the description files and stores it into data structures that are used by `acsim` to create all C++ classes and/or SystemC modules necessary to build the architecture simulator. The decoder generated by ArchC is capable of handling ISAs from simple RISC machines till multi-word of variable length instructions, like in many DSPs.

ArchC can generate simulators using the interpreted technique. Interpreted simulators execute instruction decoding, schedule and behavior dynamically. Since the decoding process is too costly in terms of simulation performance, these interpreted simulators may use a cache for decoded instructions in order to speed-up simulation. Similar techniques are applied in some well known ISA simulators[2]. This technique can be disabled by command-line options passed to `acsim`, in order to enable the execution of self-modifying code.

ArchC SystemC simulators may be used for memory hierarchy exploration [15, 16] and have even been successfully connected to other SystemC IPs to compose more complex digital systems. Depending on command-line options that may be passed to `acsim`, the generated simulator can be instrumented with several features to help on architecture exploration, like simulation statistics collection, trace generation, co-verification interface, etc.

### 4.3. OS call emulation

Programming in a high level language like C/C++ is mandatory for the the current embedded software development flow. The majority of applications are designed using C/C++ together with some hand-tunned assembly code to improve performance of critical inner-loops. Any non-trivial C/C++ program uses the standard C library (`stdlib`), which provides the most essential routines for I/O, dynamic memory allocation and other utilities.

ArchC generate simulators capable of being instrumented with an OS call emulation mechanism, which enables ArchC models to simulate applications containing I/O operations. We have grouped the system call routines in a retargetable library written in the C language, so only a recompilation is needed to port it to any other target. The designer of a new architecture is able to tell from which storage elements (memory, register bank, etc) the arguments to the system call will come from. It is done by writing interface functions that provide the required information to the ADL compiler. Typical examples of information that is required by most of the operating system calls are: how to get the first three arguments provided by a function call and how to save the return value as an integer or pointer. Functions in this group are normally very small, with less then five lines. The information required to implement them is taken from the processor Application Binary Interface manual. This feature allows ArchC to simulate programs extracted from real-world benchmarks, like Mediabench and Mibench, running them with realistic data samples.

## 5. Experimental Results

Table 1 shows some performance measures of our SPARC-V8 and MIPS-I models running programs extracted from the Mediabench suite, including system calls emulation. Table 2 shows the performance measures for the Mibench suite, which has two execution modes were the difference is the work-load imposed by small and large input data samples. We have included the number of running instructions for each application to show the large coverage of these tests. For example, we can see that the LAME MP3 coder executes nearly 95 billion instructions in the MIPS architecture model. The output files provided by the simulators were checked against files generated by a real SPARC machine running the same benchmarks. The MIPS-I model reached 550.91 KIPS (thousands of instructions per second) in average, while the SPARC-V8 model reached 633.47 KIPS. We see that the SPARC model is about 15% faster, which is due to the fact that this model does not need the ArchC internal delayed assignment control, which is used to simulate delay slots on the MIPS-I model. This functionality brings an additional burden for the MIPS simulator. The use of a cache associated to the decoder boosted the performance of our interpreted simulators in approximately 80% for our MIPS-I model and more than 200% for the SPARC-V8 model. The greater impact on the SPARC-V8 model performance is due to the higher complexity of its ISA, when compared with MIPS-I, demanding more processing power to decode it. These simulators ran on a P4 2.8 Ghz 1GB machine. These results are from our functional model. I/O emulation for cycle-accurate models is in its final stage of tests and results will be available soon. Applications are compiled for the target architecture using `gcc` and loaded into the simulator. Our experiments also showed that, when running on co-verification mode, the simulators reach about 36% of their usual performance.

Up to this point, our group and collaborators have modeled the following architectures in ArchC: MIPS, SPARC-V8, Intel 8051, Intel XScale, Motorola ColdFire, PowerPC, Texas TMS320C62x, Altera NIOS, OpenCores OR1k, and Hitachi SH-4. Two of these architecture we have both cycle-accurate and functional models: MIPS and the Intel 8051 microcontroller. The development stage reached by each model can be checked at [8]. The first two are well-known general purpose architectures, the third is one of the most used processor in embedded control applications, and the following are modern architectures largely used in embedded systems. The SPARC-V8 functional model has been used to develop and explore memory hierarchy simulation in ArchC [15, 16], by performing experiments to evaluate several cache and hierarchy configurations using multimedia applications. This was our first step toward extending ArchC modeling capabilities beyond processor architecture boundaries, enabling designers to simulate complete systems, like SoCs, based on ArchC descriptions of processors and memory hierarchies. Our collaborators are currently working on connecting ArchC generated models to other SystemC IP through buses, like OCP/IP. There is a first prototype which uses the SPARC-V8 functional model, simulating other SystemC IPs, like memories or application specific modules and communicating through an OCP/IP interface. By adding this communication feature, we intend to provide ArchC users with a set of tools that are suitable for supporting platform-based design.

We are also applying ArchC as a tool to support computer architecture education [14]. It has been applied in computer architecture courses at IC-UNICAMP during the last year. In one of theses courses, 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. This experience was very useful and tested the language's expression power, and gave the opportunity to students to contribute with suggestions and improvements to ArchC tools.

## 6. Conclusions and Future Work

This paper presents ArchC, a new architecture description language. ArchC was created to provide designers with the possibility of using automatically generated SystemC executable models from the very early stages of the architecture design process, combined with the power of automatically generating tools for architecture exploration and verification. Designers can choose between interpreted and compiled simulators, they can also use ArchC models to simulate/evaluate more sophisticated memory hierarchies with several cache and memory levels. To the best of our knowledge, ArchC is the only ADL currently published in public domain.

We are currently working on improving and tunning our co-verification algorithm and interfaces. As stated above, models of real-world architectures, like PowerPC, TMS320C62x, Morotorola ColdFire, and Intel XScale, are under development. There is an ongoing work to generate compiled simulators from ArchC processor descriptions [10], which aims to speedup simulation time. We have collaborators working to enable SystemC models generated by ArchC to be connected, through automatically generated interfaces to buses like AMBA or OCP/IP, with other SystemC IPs. Finally, the automatic generation of an assembler and a debugger interface for an architecture described in ArchC is under development.

## 7. Acknowledgments

We would like to thank FAPESP (Grants 00/14376-2, 01/09424-1, 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 contributions and feedback have been extremely valuable to the continuous improvement in ArchC tools.

## References

[1] A. Fauth, J. Van Praet e M. Freericks. Describing Instruction Set Processors using nML. In *in Proc. European Design and Test Conf., Paris*, pages 503–507, March 1995.

[2] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):128–137, May 1994.

[3] G. Hadjiyiannis and S. Devadas. Techniques for Accurate Performance Evaluation in Architecture Exploration. *IEEE Transactions on VLSI Systems*, 2002.

[4] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. In *Design Automation Conference (DAC)*, pages 299–302, 1997.

[5] 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.

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

| Program | MIPS-I | | SPARC-V8 | |
|---|---|---|---|---|
| | #Instr. | Performance (KIPS) | #Instr. | Performance (KIPS) |
| ADPCM Coder | 7,491,407 | 542.91 | 7,256,633 | 636.22 |
| ADPCM Decoder | 5,902,038 | 553.76 | 6,261,169 | 652.16 |
| ADPCM Timing | 14,843,027 | 541.85 | 17,660,293 | 635.63 |
| JPEG Coder | 16,776,932 | 568.70 | 14,288,757 | 606.85 |
| JPEG Decoder | 5,205,441 | 559.48 | 4,187,863 | 625.48 |
| MPEG Coder | 11,699,577,710 | 540.59 | 10,834,240,031 | 637.89 |
| MPEG Decoder | 3,858,003,816 | 569.94 | 3,451,838,509 | 640.80 |
| PEGWIT Encode | 31,167,432 | 546.36 | 30,823,606 | 616.18 |
| PEGWIT Decode | 17,495,609 | 565.64 | 16,865,081 | 625.39 |
| PEGWIT Generate | 12,611,316 | 551.89 | 12,790,067 | 642.19 |
| GSM Encode | 220,916,822 | 575.00 | 157,404,486 | 639.92 |
| GSM Decode | 64,216,961 | 570.25 | 88,286,413 | 653.43 |

| Program | MIPS-I | | | | SPARC-V8 | | | |
|---|---|---|---|---|---|---|---|---|
| | #Instr. | | Perf. (KIPS) | | #Instr. | | Perf. (KIPS) | |
| | Small | Large | Small | Large | Small | Large | Small | Large |
| Basicmath | 1,386,360,829 | 22,469,864,764 | 540.56 | 553.76 | 1,305,099,574 | 21,365,365,698 | 636.87 | 619.22 |
| Bitcount | 45,593,412 | 684,250,120 | 549.73 | 550.90 | 49,862,750 | 748,368,499 | 642.22 | 636.14 |
| Qsort | 14,359,302 | 991,774,388 | 568.11 | 561.43 | 14,137,668 | 792,301,299 | 643.71 | 630.78 |
| Susan (smooth) | 35,318,140 | 423,335,581 | 542.73 | 545.31 | 30,084,781 | 349,096,326 | 628.76 | 658.56 |
| Susan (corners) | 3,458,894 | 44,558,848 | 541.04 | 565.41 | 3,264,029 | 42,298,235 | 615.10 | 659.97 |
| Susan (edges) | 6,887,655 | 177,394,682 | 540.92 | 545.58 | 6,705,342 | 174,339,811 | 626.96 | 620.78 |
| Dijkstra | 31,643,160 | 285,280,151 | 569.94 | 549.00 | 50,927,675 | 244,328,854 | 645.24 | 636.40 |
| Patricia | 288,823,865 | 1,828,671,354 | 556.44 | 557.35 | 278,070,182 | 1,760,759,794 | 651.40 | 643.82 |
| ADPCM Coder | 34,628,152 | 688,959,463 | 538.65 | 533.88 | 34,285,488 | 678,637,897 | 620.95 | 649.54 |
| ADPCM Decoder | 27,256,674 | 538,659,721 | 539.82 | 530.21 | 29,687,245 | 584,732,032 | 604.04 | 646.23 |
| ADPCM Timing | 14,842,939 | - | 541.54 | - | 16,535,279 | - | 628.35 | - |
| FFT | 760,568,611 | 15,244,218,349 | 534.72 | 560.88 | 763,966,571 | 12,936,715,454 | 658.10 | 642.34 |
| FFT INV | 1,822,953,504 | 14,750,402,897 | 534.00 | 561.03 | 1,801,388,682 | 12,559,257,170 | 629.47 | 626.07 |
| CRC 32 | 31,643,160 | 615,051,948 | 542.43 | 547.36 | 30,236,229 | 587,712,330 | 646.58 | 630.98 |
| JPEG Coder | 32,169,687 | 118,600,038 | 552.93 | 535.97 | 25,236,729 | 93,002,355 | 617.62 | 630.04 |
| JPEG Decoder | 9,473,307 | 32,333,052 | 550.74 | 562.32 | 7,279,954 | 24,773,993 | 628.82 | 648.63 |
| LAME | 8,033,704,002 | 94,314,747,771 | 550.96 | 564.87 | 7,641,414,891 | 89,899,090,464 | 634.34 | 629.76 |
| SHA | 13,036,287 | 135,696,013 | 560.71 | 561.65 | 13,254,952 | 137,975,709 | 648.08 | 626.74 |
| RIJNDAEL Encode | 33,637,647 | 350,251,921 | 542.31 | 542.67 | 32,560,404 | 339,042,970 | 612.76 | 613.42 |
| RIJNDAEL Decode | 34,684,744 | 361,155,494 | 540.57 | 541.36 | 32,482,970 | 338,231,604 | 612.69 | 613.54 |

[7] A. Hoffmann, T. Kogel, and H. Meyr. A framework for fast hardware-software co-simulation. In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, March 2001.

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

[9] M. R. Hartoog, J. A. Rowson, P.D. Reddy, S. Desai, D. D. Dunlop, E. A. Harcourt, N. Khullar. Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign. In *in Proc. Design Automation Conference*, pages 303–306, 1997.

[10] Marcus Bartholomeu, Rodolfo Azevedo, Sandro Rigo and Guido Araujo. Optimizations for Compiled Simulation Using Instruction Type Information. In *proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04), Foz do Iguaçu,* , October 2004.

[11] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. In *Proceedings of Design and Automation Conference (DAC)*, June 2002.

[12] OSCI. *SystemC Version 2.0 User's Guide*, 2001.

[13] M. Reshadi, P. Mishra, and N. Dutt. Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation. In *Proceedings of Design and Automation Conference (DAC)*, 2003.

[14] Sandro Rigo, Marcio Juliato, Rodolfo Azevedo, Guido Araujo and Paulo Centoducatte. Teaching Computer Architecture Using an Architecture Description Language. In *proceedings of the Workshop on Computer Architecture Education (WCAE), held in conjunction with the International Symposium on Computer Architecture (ISCA), Munich*, June 2004.

[15] P. Viana, E. Barros, S. Rigo, R. Azevedo, and G. Araújo. Exploring Memory Hierarchy with ArchC. In *Proc. of the 15th Symp. on Computer Architecture and High Performance Computing, São Paulo, (SBAC-PAD'03)*, November 2003.

[16] P. Viana, E. Barros, S. Rigo, R. Azevedo, and G. Araújo. Modeling and Simulating Memory Hierarchies in a Platform-Based Design Methodology. In *Proc. of the Design, Automation and Test in Europe (DATE'04), Paris*, February 2004.

[17] V. Zivojnovic, S. Pees, and H. 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.