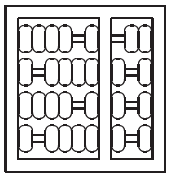

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

The ArchC Architecture Description Language

Sandro Rigo *Rodolfo J. Azevedo*
Guido Araujo

Technical Report - IC-03-015 - Relatório Técnico

June - 2003 - Junho

The ArchC Architecture Description Language

Sandro Rigo* Rodolfo J. Azevedo Guido Araujo

Abstract

In this report we introduce a new architecture description language (ADL) called ArchC. ArchC is an open-source SystemC-based ADL 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 simulators, assemblers and compiler back-ends. ArchC's key feature is a storage-based co-verification mechanism that automatically checks the consistency of refined SystemC RTL models against the behavioral reference model. We have used ArchC to synthesize cycle-based simulators for the MIPS, Intel 8051 and SPARC processors.

1 Introduction

Embedded systems designers are facing increasing development challenges at all stages of the design process due to increasing system complexity, rapidly changing architectures and a very difficult financial environment. Instruction set architectures used to be fairly stable, but no longer. There were fifty nine new architectures in one recent 12-month period [12]. In addition, the complexity of embedded software is increasing by leaps and bounds. A typical embedded system had 100,000 lines of code in 1995 and may have 1 million lines of code nowadays [12]. As a consequence, designers are reconsidering how designs are specified, partitioned and verified.

In the traditional design environment, hardware and software teams interact occasionally but mostly work separately. Both teams meet at the beginning of the project, define an architecture, and create a model. Once they agree on hardware/software partitioning they tend to follow separate paths, writing non-reusable application, testing and verification code that follows their own design viewpoint. In this methodology, major design problems arise from the use of different design languages when systems/software engineers are programming in C/C++ and hardware designers are working using hardware description languages (HDL) like VHDL or Verilog. Code reuse becomes rare and incompatible tools are almost inevitable. As a way around that, designers are starting to move from HDLs and also beyond the RTL level of abstraction toward the so called *system level design*.

An example of these changes in design style is system integration on a virtual prototype, for example: a simulation model of a SoC before fabrication. Software is run on an instruction set simulator(ISS) [6, 7] model of the processor, and the ISS communicates with a

*Research supported by FAPESP (grant 00/14376-2)

simulation of the rest of the hardware via a bus interface model or bus functional model. A functional model, also known as system-level simulator, allow hardware designers and software engineers to precisely reproduce an application behavior in a target system, even if this system is still under development [25]. Designers have started to create virtual platforms of their SoC designs for good reasons. First, they can provide an early simulation platform for the system integration that executes much faster than HDL-based co-verification. Second, they can verify the architecture at an early stage of the design process, testing different hardware/software partitions and evaluating the effects of architectural decisions before implementation. After all, by the time they are coding RTL, the hardware architecture is established and there's little room for flexibility. Third, they can start working on derivative or next-generation designs by reusing the platform even before the first generation design is complete.

Great part of the success in the system-level modeling area is due to the availability of tools and methodologies that are efficient enough to help designers meet the stringent time to market that has been imposing to them. In the new system-level design scenario, a tool for evaluation of a new designed instruction set architecture which automatically generates a software toolkit composed by assemblers, linkers, compilers and simulators became mandatory.

1.1 SystemC

SystemC [13, 30] 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 source code for the simulation kernel is freeware. SystemC is composed by a set of C++ class libraries, that extends the language to allow hardware and system-level modeling. Designers are allowed to model in low-levels of abstraction, like RTL, using SystemC. However, SystemC's main goal is not to replace HDLs (like VHDL and Verilog), but to allow system-level design.

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).

In this document, we introduce a SystemC-based architecture description language (ADL) called ArchC. ArchC is a simple language, specialized for processor architecture description, that follows 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, compiler back-ends and co-verification interfaces. ArchC has many features that distinguish it from other ADLs. The key feature though is a storage-based co-verification mechanism that automatically checks the consistency of refined SystemC RTL models against the behavioral ISA description. Cycle-based simulators have been synthesized starting from ArchC descriptions for the MIPS, Intel 8051 and SPARC processors.

The remaining sections of this report are organized as follows. Section 2 discusses some related works. Section 3 introduces the ArchC syntax and semantics. Section 4 presents the

co-verification mechanism provided by ArchC. Section 5 describes ArchC tools and some processor case studies, and finally, Section 6 summarizes our conclusions and describe the future work.

2 Related Work

In the following sections, we are going to discuss some related work on architecture description languages, software tool-set generation and instruction set simulators.

2.1 nML

The nML [3, 10, 11] formalism is based on the information typically available in the programmer's manual of a processor, which consists in a list of instructions and corresponding register transfers, binary encoding and assembly mnemonics. A programmer's model of the machine is usually provided in terms of a coarse schematic showing registers, functional units and the basic interconnection scheme. However, detailed information of the datapath is normally not available. nML is based on a paradigm that mixes structural with behavioral information, i.e., structural is described together with execution behavior. On one hand, the *skeleton* of the machine structure is constructed by declaring storage entities. On the other hand, register transfers between these entities describe the execution behavior of the machine. Addressing modes are also supported explicitly. However, originally, nML did not support self-modifying code, sub-instructions and multi-cycle instructions.

The abstraction level of the nML language is the instruction set. nML is based on a set of assumptions: a machine, when running, executes a program that is a series of instructions. The *program counter(PC)* points to the next instruction to be executed. A machine *state* is stored in *memory locations*. The sole purpose of a program is to change this state, in other words, all that instructions do is changing the values of locations. There are no inter-instruction control flow constructs. The program flow is changed by writing to the PC location.

In nML, the instruction set is described by an attributed grammar. This approach allows several instructions to share the same behavior, coding, or syntax since the semantic action of any instruction can be composed of fragments that are distributed over the whole grammar tree. These fragments are basically *AND* and *OR* rules, used to express composition of or alternative instruction parts, respectively. The semantic action attribute has to evaluate to a sequence of register-transfer operations.

Fault et al [3] presented an extension to nML in order to support some timing description. They introduced the concept of transitory registers. The transitory nature means that their values are valid only for a finite period of time (delay), and become undefined afterward. In this scheme, all *computing* operations have a duration of zero cycles and only the read from a storage can be delayed by a certain amount of cycles after a write has occurred. The authors claim that this feature enables the description of multi-cycle operations and pipelines.

Hartoog et al [26] presented an experimental set of tools generated from a nML description of a processor. The authors conclude that real retargetable assemblers and disassem-

blers can be generated with the inclusion of some additional information. However, nML does not support multi-cycle instructions and straightforward extensions of nML would support only the simplest pipelines, i.e., is not possible to produce cycle-accurate simulators for pipelined processor architectures, specially for processors with complex execution schemes, like many DSPs. The nML implicit program counter is inconvenient in some circumstances, like to properly handle multi-word instructions and zero overhead loops.

nML was developed at Technical University of Berlin (TUB). Using nML, TUB developed an instruction set simulator called SIGH/SIM [9] and a code generator called CBC [10]. Independently, IMEC (Interuniversity MicroEletronics Center) developed a code generator called CHESS [2] and an instruction set simulator called CHECKERS.

2.2 LISA

Vojin Zivojnovic et al [32, 33] first introduced LISA. LISA 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. This early version of LISA was not well suited for generation of code-generators and assemblers. In order to model the operation sequencer, LISA followed the basic ideas of reservation tables and *Gantt charts*, however it extends Gantt charts to enable modeling of data/control hazards and pipeline flushes [32].

Pees et al [29] presented a new version of LISA that was capable of automatically generating simulators and assemblers for several classes of architectures, like DSP, SIMD, VLIW and superscalar.

A LISA description is composed of *resource declarations* and *operation* declarations. An operation represents the programmer's view of the behavior, the structure, and the instruction set of the architecture. They are the basic elements in LISA. In order to collect different properties of the system, operation definitions have several sections:

- CODING: the binary image of the instruction;
- SYNTAX: the assembly syntax;
- SEMANTICS: specifies the instruction semantics;
- BEHAVIOR and EXPRESSION: describe components of the behavioral model.
- ACTIVATION: describes the timing of other operations relatively to the current one;
- DECLARE: local declarations of identifiers and lists of alternatives elements

The designer is allowed to add further sections in order to define other attributes.

The resources represent the storage devices of the hardware, which stores the state of the system. They are registers, memory, pipelines, etc. In the pipeline model of LISA, operations are assigned to pipeline stages and can be activated with or without delay. Operations will be executed synchronously to control steps, which can be defined as instructions cycles, clock cycles or phases.

Hoffmann et al [4] introduced the *LISA Processor Design Platform(LPDP)*. This is a platform for the design of application specific instruction set processors (ASIP), using a LISA description of the machine as base to generate software development tools. In this work, LISA descriptions consist of the following model components: memory, resource, instruction set, behavioral, timing and micro-architecture. If we compare LISA descriptions presented in [29] and [4], we can identify some modifications introduced in the language, mainly to support automatic generation of a C compiler back-end and of a synthesizable model in a HDL like VHDL. The memory model comprises registers and memories available in the system. The resource model reproduces properties of hardware structures which can be accessed by only one operation at a time. In other words, when a register is declared in the RESOURCE section, the designer have means of declaring how many simultaneous access the register supports. It is a necessary information for the instruction scheduler of the compiler back-end. Another new feature is the micro-architecture model. There is a new keyword ENTITY in the resource section of the LISA model, were operations can be assigned to functional units and structural components can be included in the model by inlining HDL code. Till this point, the platform seemed to be ready to generate good compiler simulators, assemblers and linkers but the work on retargetable compilers and HDL generator seemed to be preliminary.

Hoffmann et al [22] presented another LISA tool-suite. It is composed of an assembler, a linker, a simulation compiler, a simulator and a graphical debugger front-end for the simulator.

Hoffmann et al [21] used LISA to create a framework for hardware/software co-simulation. On the software side, LISA models are used to create a tool-suite composed of compiled simulator, assembler, linker, HLL compiler and a co-simulation interface. On the hardware side, SystemC is used to together with the simulation interface to enable the integration of external simulators. The goal is to couple models at various abstraction levels for hardware-software co-verification.

2.3 EXPRESSION

EXPRESSION [18] is a ADL focused on architectural design space exploration for SoCs and automatic generation of a compiler/simulator toolkit. EXPRESSION also follows the mixed-level approach of description, i.e., mixing behavioral and structural information and supports specification of novel memory subsystems. The later seems to be the key feature of the language. The authors invested in this feature claiming that SoC technologies permit the incorporation of novel on-chip memory organizations.

Grun et al [14] introduced an algorithm to extract the information needed to automatically build reservation tables from an architecture description in EXPRESSION. This is an important resource in order to allow the automatic generation of a compiler back-end [18].

Mishra et al [27] introduced a functional abstraction-based design methodology. Parameterized functions for functional units present in programmable architectures, such as fetch unit, branch-prediction unit, decode unit, etc were defined. These functional units were used in a description of the Texas TMS320C62x processor in EXPRESSION. The main purpose of the generic function based design space exploration is allowing designers to

compose new architectures by reusing the generic abstractions within an ADL and generate automatically a customized software toolkit.

2.4 ISDL

The Instruction Set Description Language for Retargetability (ISDL) [16] was developed at MIT. Hadjiyiannis et al [16] presented the language and an automatic assembler generator.

ISDL was projected to be a machine description language for compiler retargetability, allowing the description of many different architectures, in particular VLIW machines. ISDL is a behavioral language that lists the instruction set of the architecture based on an attributed grammar, as well as nML. ISDL is very similar to nML, except in the way it handles constraints. In nML, only valid instructions can be described, therefore, the invalid combinations must be worked around through additional rules in the grammar.

The ISDL tool set was extended with a code-generator generator (Aviv) [19], an instruction level simulator generator (GenSim) and a generator of hardware models written in Verilog(HGen) [15, 17]. ISDL is a purely behavioral language, so all structural information necessary to the simulator and hardware model synthesizers have to be inferred from operation descriptions, which are basically register-transfer operations, and from the instruction set, which has some attributes like latency and delay for each instruction. It is not clear if ISDL tools are capable of extracting the correct behavior for pipelined architectures with complex execution schemes. For example, pipeline flushes does not seem to be possible. The authors presented some results on models for VLIW, RISC and DSP (Motorola 56000) architectures. Another strong constraint: ISDL is not able to model multi-cycle instructions of variable length [15].

2.5 Other ADLs

RADL [28] has explicit support for multiple pipeline modeling, including delay slots, interrupts, hardware loops and hazards. RADL is presented with focus on cycle and phase-accurate simulators, but no results are provided on realized simulators based on this language.

MIMOLA [5] is an ADL based on the structure of the processor. MIMOLA descriptions are very low-level, to the point of the same description being used for both processor synthesis and code generation, which prevents this language from being used for architecture exploration and system-level design.

2.6 Instruction Set Simulators

Nowadays, instruction set simulators are an essential part of the toolkit used in processor design. The increasing number of new architectures appearing every day makes retargetability a mandatory feature for simulation tools. Simulators are used in the architecture exploration phase, to evaluate early design decisions as well as to validate the architecture and compilers.

Some simulators, like Shade [7], FastSim [8] and Embra [31], rely on a dynamic binary translation together with a result caching mechanism to improve simulation performance.

SimpleScalar [1] is another popular simulation environment, which instruction set is based on MIPS-IV. These are very fast simulators that support limited architectures and require specific host and the target machines so, they are not retargetable.

3 The ArchC Architecture Description Language

Basically, ArchC can be considered as an extension of SystemC, which is specialized for architecture description. ArchC's primary goal is to provide enough information, at the right level of abstraction, in order to allow users to explore a new architecture and automatically generate software tools like assemblers, simulators or even compiler back-ends. It is a simple language that follows SystemC syntax style, which makes clear the connexion between these two languages.

An architecture description in ArchC is divided in two parts: the `Instruction Set Architecture (AC_ISA)` description and the `Architecture elements (AC_ARCH)` description. Into the `AC_ISA` description, the designer provides to ArchC details about instruction formats, size and names combined with all information necessary to decoding and the behavior of each instruction. The `AC_ARCH` description informs ArchC about storage devices, pipeline structure etc. Based on these two descriptions, ArchC will generate a behavioral simulator written in SystemC for the architecture.

The designer must provide the `AC_ISA` and `AC_ARCH` descriptions in separated files. Both descriptions are parsed and pre-processed by ArchC. The result is several files written in SystemC/C++ which compose the behavioral simulator. The user can compile these files through a *makefile*, similar to those provided together with SystemC examples, and get an executable specification of the architecture to experiment with.

Up to this point, our group has used ArchC to model three architectures: MIPS (implementing the MIPS-I instruction set), Sparc-V8, and a microcontroller. The MIPS [20, 24] is a RISC machine with a five-stage pipeline, containing 32 general purpose registers plus two special registers used for multiplication and division. The Intel 8051 (i8051) microcontroller is one of the most used processors for embedded control. This is a CISC architecture with multi-cycle instructions of variable length. Throughout this text, we are going to use examples extracted from our ArchC descriptions of two of these architectures, MIPS and Intel 8051.

3.1 Describing Architecture Resources

ArchC uses some structural information about the resources available in the architecture in order to automatically generate a simulator. The designer must provide such an information in the `AC_ARCH` description, which is basically composed of storage elements and pipeline declarations.

The level of details used for this description will depend on the level of abstraction that the designer wants for his/her model. For example, one may want to simulate the instruction set of the MIPS architecture, but without concerning with pipelining. This makes the instruction behavior description very simple, as we are going to see in Section 3.2, but also demands few structural information, like showed in Figure 1.

An architecture resources description starts with the keyword `AC_ARCH`, like in the first line of our example. The designer is supposed to inform a name for the project, like is usually done for modules in SystemC. Then follows the storage device declarations. In this simple example, the designer uses the keyword `ac_cache` to declare instruction and data caches. The number after the colon represents the size, in bytes, of the device. Following with the example, a register file is declared through the keyword `ac_regbank`. We need 34 registers to model the MIPS, as mentioned above. The declaration of the `ARCH_CTOR` constructor finishes the `AC_ARCH` description and uses the `ac_isa` keyword to inform in which file the ArchC pre-processor will find the `AC_ISA` description.

```
AC_ARCH(mips){

  ac_mem   MEM:256k;
  ac_regbank RB:34;

  ARCH_CTOR(mips) {

    ac_isa("mips_isa.ac");

  };
};
```

Figure 1: MIPS `AC_ARCH` resource declaration.

In order to get a more detailed model, like a cycle-accurate model of the MIPS processor, the designer must provide more information about the architecture. Figure 2 shows our description of the architecture resources for the MIPS processor. This example illustrates some new features available in ArchC. The designer is allowed to declare the word-size for the architecture using the `ac_wordsize` keyword. Memories and caches are byte-addressed but their return type is always a word. If not declared, ArchC assumes that the word-size is 32 bits by default.

The example follows with a register bank declaration, similar to the previous example, and then we have a pipeline declaration. Pipelines are created through the keyword `ac_pipe` and are composed by a name and a list of pipeline stages attributed to it. Pipeline stages in ArchC are declared using keyword `ac_stage`, and are organized by means of keyword `ac_pipe`. Pipeline stages listed in the same `ac_pipe` are assumed to execute instructions in the order they appear in that declaration. In Figure 2, we show two equivalent pipeline declarations, which means that just one needs to be present in the description. Both will produce a five stage pipeline, where `IF` is the first and `WB` is the last stage. Both syntaxes are simple but are only applicable for single-pipelined architectures. In other complex multi-pipelined/superscalar architectures, the following stage could be dependent on which instruction is in the current stage. In such cases, the designer needs a mechanism to inform the simulator the next stage that its current instruction should be dispatched to. In order

```
AC_ARCH(mips){

  ac_wordsize 32;

  ac_mem  MEM:256K;
  ac_regbank RB:34;

  ac_pipe    pipe = {IF, ID, EX, MEM, WB};   //|\  equivalent,
  ac_stage   IF, ID, EX, MEM, WB;            //|/  use only one.

  ac_format Fmt_IF_ID = "%npc:32";
  ac_format Fmt_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 Fmt_EX_MEM =
            "%alures:32 %wdata:32 %rdest:5 %regwrite:1 %memread:1 %memwrite:1";
  ac_format Fmt_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;

  ARCH_CTOR(mips) {

    ac_isa("mips_isa.ac");

  };
};
```

Figure 2: MIPS `AC_ARCH` resource declaration.

to handle that, the designer declares only stages and ArchC provides a method named `ac_next`, that can be called from inside the instruction behavior description, to enable the designer to re-route the instruction to the correct next stage. Stages also carry methods that allow pipeline flush and stall operations.

Still following the example in Figure 2, we reach a set of declarations that represent another feature available in ArchC: formatted registers. Microarchitecture designers frequently need to access instruction fields or control signals available in registers. In order to enable that, ArchC allows the designer to declare a format, that is nothing more than a set of fields, and associate this format with a register, exactly as he/she does for instructions (see Section 3.2). In ArchC, a format is declared through the `ac_format` keyword. The designer must give a name to the format, like `Fmt_IF_ID` in the example. The string assigned to a format represents its subdivision into fields. 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. For example, the string "%rs:5" declares a 5-bit field named `rs`. It is also possible to tell ArchC that a field stores a signed value, by appending

the ":s" flag at the end of a field declaration, like is done to the `imm` field in the `Fmt_ID_EX` format in Figure 2. 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 `Fmt_ID_EX` is associated to register `ID_EX`. This allows the designer to assign to (and read from) each register field individually when describing behaviors. The four formatted registers declared in Figure 2 represent the four MIPS pipeline registers, as presented in [20].

3.2 Describing the Instruction Set Architecture

The `AC_ISA` description provides ArchC with all information it needs to automatically synthesize a decoder, along with the behavior of each instruction in the architecture. This description is divided in two files, one containing the instruction and format declarations and another containing the instruction behaviors.

Figure 3 shows an example of an `AC_ISA` description extracted from our MIPS model. The beginning is similar to the `AC_ARCH` description, starting with the keyword `AC_ISA`, along with the name of the project. The designer continued by declaring the instruction formats. This is done by using the `ac_format` keyword, following the same syntax introduced in the previous section. Next step is to declare instructions. Every instruction must have a previously declared format associated to it. The designer declares an instruction through the keyword `ac_instr` and he/she can assign it a format using a syntax similar to C++ *templates*. In the example, format `Type_R` is associated to instruction `add`. This allows the designer to access each instruction field individually when describing instruction behaviors.

The description follows with the `AC_ISA` constructor declaration (`ISA_CTOR`). This is where the designer will have to initialize some key values for each instruction. An assembly syntax is assigned to an instruction using the `set_asm` method. The decodification sequence is a key element to the automatic generation of an instruction decoder. It is provided to ArchC through the `set_decoder` method, and is composed by a sequence of pairs <*field_name* = *value*>. In Figure 3, examine the example for `add` instruction. The call to `add.set_decoder` method states that a bit stream coming from memory actually is an `add` instruction if, and only if, fields `op` and `func` contain the values 0x00 and 0x20, respectively. The decoder will know where to look for the field values in the bit stream, because this information is available in the format previously associated to the instruction. Exactly the same steps are taken to the declaration of the `load` instruction. Notice that `load` has a different format, and that is why just one field has to be checked to decode it.

The MIPS is a RISC architecture, so all instructions have the same size and takes the same number of cycles to be executed. However, in architectures like CISC, DSPs and VLIW machines, this may not be true. Figure 4 is an example extracted from our Intel 8051 description. The Intel 8051 (i8051) microcontroller is a CISC architecture with a more complicated ISA, when compared with a RISC machine. Notice that instructions now have different sizes, which are given by the size of the associated format in ArchC. The i8051 is also a multicycle processor, which means that instructions will take different number of cycles to be completed. So, this number of cycles taken by each instruction to execute must be another parameter informed by the designer. This is done through the `set_cycles` method used in the `AC_ISA` constructor in ArchC, like is illustrated by the declaration of

```
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, sub, instr_and, instr_or, mult, div;
  ac_instr<Type_R> mfhi, mflo, slt, jr;
  ac_instr<Type_R> addu, subu, multu, divu, sltu;
  ac_instr<Type_R> sll, srl;
  ac_instr<Type_I> load, store, beq, bne;
  ac_instr<Type_I> addi, andi, ori, lui, slti;
  ac_instr<Type_I> addiu, sltiu;
  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 3: MIPS ISA Description

the two-cycle `mov_r_iram` instruction in the example. Observe that the `add_r` instruction does not have a call to the `set_cycles` method, so ArchC assumes that it is an one-cycle instruction, by default.

```
AC_ISA(i8051){
                                  ...

  ac_format Type_3bytes = "%op:8 %byte2:8 %byte3:8";
  ac_format Type_2bytesReg = "%op3:5 %reg2:3 %addr:8";
  ac_format Type_OP_R = "%op1:5 %reg:3";

  ac_instr<Type_2bytesReg> mov_r_iram;
  ac_instr<Type_OP_R> add_ar;

  ISA_CTOR(i8051){

    mov_r_iram.set_asm("mov %reg2, %addr");
    mov_r_iram.set_decoder(op3=0x15);
    mov_r_iram.set_cycles(2);

    add_ar.set_asm("add A, %reg");
    add_ar.set_decoder(op1=0x05);

                                  ...
 };
};
```

Figure 4: Intel 8051 ISA Description

3.2.1 Providing Instruction Behavior

The behavior description file is where the designer provides a description of which operations are executed by each instruction in the architecture. By issuing both description files introduced so far, `AC_ISA` and `AC_ARCH`, to the ArchC pre-processor (`acpp`), the designer gets a template of the behavior description file. This template is a skeleton of a .cpp (SystemC source) file where the designer is going to fill out the behavior method of each instruction in the architecture. The behavior description file is named as *project_name*-isa.cpp and the template generated by ArchC is called *project_name*-isa.cpp.tmpl, by default.

This behavior information can be expressed in several levels of abstraction. For example, at the very early stages of the design, cycle-accuracy may not be important. Normally, the first model of a new architecture does not have timing information. So, for this preliminary model, the behavior of a given instruction is just a sequence of C++ statements representing the operations that this instruction would execute in the hardware. Figure 5 illustrates how we could model the behavior of `add` and `load` instructions in the MIPS architecture. At this abstraction level, the simulator will execute one instruction per cycle, so it is not necessary

to worry about data hazards and forwarding, because every time an instruction begins its execution, the previous will have been completed for sure. A so high-level modeling style will provide the designer with an executable specification of the architecture very rapidly, but this specification will be suitable only for experimenting with the instruction set, not for performance measurements due to its lack of timing information. We can also describe the i8051 instruction-set in such a high-level fashion. This is showed by Figure 6 for the `mov_r_iram` instruction. This instruction moves memory data, given by address `addr` into the memory position given by the address store in a register. In the Intel 8051 architecture, the register bank where one instruction is supposed to operate is given by a pair of bits of a special register called `PSW`, that have to be checked every time the instruction is executed. The much more complex i8051 CISC instruction-set does not get to be described in such a straightforward manner like the MIPS, but it still is just a sequence of C++ statements without timing information.

```
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 5: MIPS Instruction behavior description in a high-level model

But, as the design process moves forward, more detailed models are needed. The designer certainly reaches a point where a cycle-accurate model is necessary. At that time, the architecture description must be refined. For our MIPS model, we need to declare a pipeline along with its pipeline registers in the `AC_ARCH` description (see Figure 2), and then divide the behavior of each instruction, telling ArchC what is done at each pipeline stage. Figure 7 illustrates part of the `add` instruction behavior in a cycle-accurate fashion.

3.2.2 Providing Format and Generic Instruction Behavior

Often, there are many instructions in a particular architecture that execute exactly the same task as part of their behavior. As mentioned above, in the i8051 microcontroller, an instruction that operates on registers has to check two bits of the `PSW` register to discover which register bank it is going to use. As a consequence, a piece of code to check that would have to be inserted into the behavior method of every instruction in this class. In the MIPS processor, some tests have to be performed by several instructions in order to determine data hazards and to do register forwarding.

In ArchC, instead of repeating the code for every instruction, the designer is able to use formats in order to factor out this behavior, writing it once and using it for a whole class of instructions. This is possible because ArchC provides the designer with the possibility of

```
void ac_behavior ( mov_r_iram ){

  sc_uint<8> psw;
  int reg_indx;

  psw = IRAM.read(208);

  if(psw.range(4,3) == 0 ){
     reg_indx = reg2;
  } else if(psw.range(4,3) == 1 ){
     reg_indx = reg2+8;
  } else if(psw.range(4,3) == 2 ){
     reg_indx = reg2+16;
  } else{
     reg_indx = reg2+32;
  }

  IRAM.write( reg_indx, IRAM.read(addr));

};
```

Figure 6: i8051 Instruction behavior description in a high-level model

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 8, *before* running its own behavior, so that they can do register forwarding for both instruction operands, `rs` and `rt`.

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, i.e., a behavior method that belongs to all instructions. Following the same style used above, the designer has to pass the keyword `instruction` as the argument of the behavior method:

```
void ac_behavior( instruction ){

  switch( stage ) {
  case _IF:
          ...
    IF_ID.npc = ac_pc + 4;
           ...
  }
};
```

The example above shows a piece of code that computes the `npc` (next pc) value for our MIPS description. Notice that both format and generic instruction behaviors can be

```
void ac_behavior( add, stage ){
   switch(stage) {
   case IF:
      IF_ID.npc = ac_pc + 4;
      break;

   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:
      RB.write(MEM_WB.rd, MEM_WB.alu_result);
   default:
      break;
   }
};
```

Figure 7: Instruction `add` behavior description

written in a cycle-accurate fashion for pipelined or multi-cycle instructions, by means of a C++ switch statement, just like it is done for instructions.

The hierarchy of behavior methods in ArchC states that the simulator will, at a given simulation time, start the execution by the generic instruction behavior method, followed by the behavior of the format corresponding to the current instruction and, finally, the current instruction behavior itself. The same sequence is performed by each pipeline stage in the case of a pipelined architecture. Through this ArchC behavior hierarchy, we were able to factorize a lot of operations that would have to be repeated into several instruction behaviors, what ended up saving a great amount of redundant code in our models.

4 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. It is known that verification now consumes over 70% of the effort to design an ASIC [23]. The natural flow of a project is to start with a model in a higher level of abstraction and gradually refine it toward synthesis. So, by describing an architecture in ArchC, the designer automatically gets a SystemC behavioral simulator of the processor and, after

```
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 8: `Type_R` format behavior description

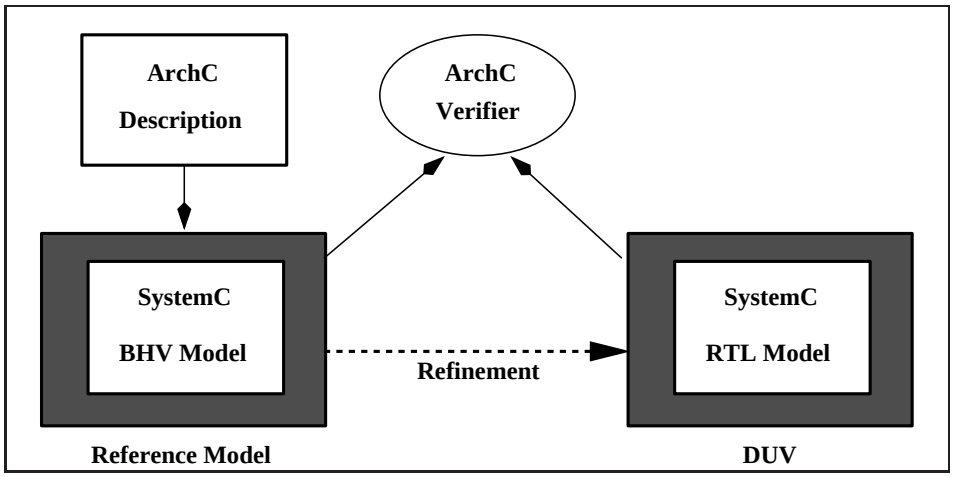

Figure 9: ArchC Co-verification Methodology

experimenting with the instruction set and some aspects of the structure, he/she concludes that it reflects the processor expected behavior. The designer can then refine this model, starting from the ArchC model or even writing a RTL model from scratch in SystemC.

ArchC provides an *interface* allowing the designer to couple these two different models of the architecture. Both models run the same application and the ArchC verifier checks if the models are consistent. Figure 9 illustrates the co-verification methodology of ArchC, where the behavioral model is used as the reference model and the refined model is the DUV(*Device Under Verification*). 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 down the timestamps when they happened. By comparing the sequence of transactions generated throughout the execution, the ArchC verifier can tell if the models are consistent.

Figures 10 and 11 illustrate how ArchC co-verification can help the designer to speed-up verification. Figure 10 illustrates part of the correct `EX` stage of the MIPS processor. In that figure, fields `rs`, `rt`, `alu_result` of pipeline registers `ID_EX` and `EX_MEM` are the same as those in the `EX` clause of the ArchC behavioral description of Figure 7. Notice that signals `read_data2` and `imm` are correctly bound to ports `0` and `1` of the multiplexer. But, notice also that the designer made a mistake and inverted this order when connecting these signals in the SystemC RTL model shown in Figure 11. This mistake resulted in an incorrect port assignment to the MUX inputs, thus producing an incorrect value being stored into field `EX_MEM.alu_result` of pipeline register `EX_MEM` (when compared to the execution of the ArchC `add` behavior). ArchC provides a verification method, called `ac_verify`, that allows the designer to couple these two different models of the architecture. By using `ac_verify` in Figure 11 the designer associates the `EX_MEM.alu_result` register field of the ArchC description to the `alu_out` signal in the RTL model. This is usually done at every point where the signal is being written in the RTL model. The `ac_verify` method can also be used to verify other storage elements like caches, register banks and memories.

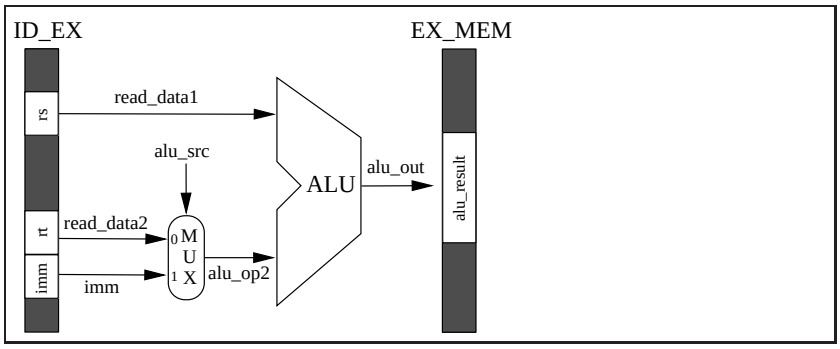

Figure 10: `EX` stage datapath with correct MUX binding

```
#include "dlx-ac.h"
              ...
  Mux_ALUsrc = new mux("MUX");
  Mux_ALUsrc->select( ID_EX_ALUsrc );
  Mux_ALUsrc->in1(ID_EX_read_data2);  // swapped
  Mux_ALUsrc->in0(ID_EX_imm);         // ports
  //Operand going to ALU
  Mux_ALUsrc->out(alu_op2);
              ...
  // connect ALU to signal alu_out
  ALU->out(alu_out);
  // connect to alu_result port
  EX_MEM->alu_result(alu_out);
              ...
  // verify against ArchC variable
  ac_verify(EX_MEM.alu_result, alu_out);
```

Figure 11: SystemC RTL incorrect MUX binding (wrt. ArchC behavioral description in Figure 7)

When the designer simulates the RTL model, coupled with a correct ArchC behavioral model through `ac_verify`, ArchC's co-verification mechanism issue an error message and saves a transaction log every time the values matched by `ac_verify` are inconsistent. Each transaction log contains: (a) the values assigned by the SystemC RTL and the ArchC models to the inconsistent storage elements; (b) the simulation time when the inconsistence occurred. 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.

We designed our co-verification algorithm to work in two situations. First, both the behavioral and the refined models are timing-accurate, if the designer wants to assure that updates are executed at the same simulation (cycle) time on both models. Second, the

behavioral ArchC (reference) model is not cycle-accurate, if the designer just wants to verify the consistency of the update sequences generated by the models.

5 The ArchC Pre-processor

Currently, the ArchC tools are organized around the `ArchC pre-processor (acpp)`. From the designer point of view, `acpp` takes an ArchC description, composed by an instruction set architecture description (`AC_ISA`) and an architecture resource description (`AC_ARCH`), and generates a SystemC behavioral model of the architecture.

`Acpp` is composed by a parser, a SystemC generator and a decoder generator. The parser extracts information from the `*.ac` files and stores into data structures. The SystemC generator comprises of a set of functions capable of generating all classes and SystemC modules necessary to build the the architecture simulator. It relies on some options that the designer must provide to `acpp`, so that it can decide which features will be enabled in the generated simulator. The decoder generator relies on the data structures extracted by the parser to compile all information from the `AC_ISA` description, so that it can automatically generate a decoder to be used by the SystemC simulator. To include a new instruction, the designer just adds a description of its format along with its behavior to the `AC_ISA` description and, after running `acpp` and the C++ compiler, a new simulator is produced. This process is illustrated by Figure 12.

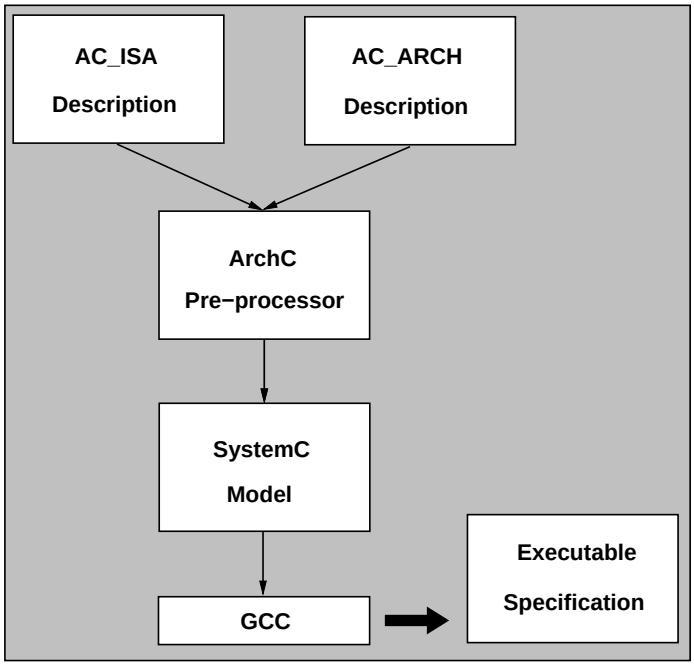

Figure 12: ArchC Simulator Generation Flow.

5.1 Command-line Options

The designer has some freedom to choose what features will be available in the simulator generated by ArchC. It is true that the simulator runs faster with no additional options turned on, but they may be very useful during the design exploration phase. Figure 13 shows the `acpp` help screen. The user can see this information by running `acpp --help`. It is composed by a description of the syntax used to run `acpp` with/without options, followed by a list, together with a brief description, of all command-line options available in ArchC. All options have a *full-name*, which is easier to memorize, and a equivalent *short-name*, like most options in `gcc`[1]. Options full-names and short-names are preceded by `--` and `-` respectively.

In order to generate a new simulator, `acpp` requires just one mandatory argument, that is the name of the `AC_ARCH` description file. The options that can be passed to `acpp` by command line are:

- **-debug, -g** :
  This option turns on the ArchC debugging features: trace and storage update logs. ArchC traces are dumps, into a text file, of the name and field values of all instructions executed by the simulator. Update logs are the list of changes suffered by each storage device, also dumped in a text file. Notice that both files can become very large for long simulation times.

- **-dumpdecoder, -dd** :
  When this option is turned on , `acpp` will dump, into a text file, the data structure built by the decoder. It is useful to debug the ArchC decoder generator, or to anyone interested in understand how it works.

- **-help, -h** :
  This option can be used by itself, i.e., it does not require the presence of a `AC_ARCH` file name, and just display the help screen showed in Figure 13.

- **-quiet, -q** :
  The simulator generated by ArchC will not issue messages and logs on the screen when this option is turned on.

- **-stats, -s** :
  ArchC is capable of collecting some statistics during simulation, like number of running instructions, how many times each instruction was executed, how many accesses each storage device has suffered, etc. This option turns this feature on and a statistics report will be saved in a file named *project_name*.stats.

- **-verify, -v** :
  When this option is chosen, `acpp` will generate a simulator prepared for execute the storage-based co-verification following the untimed model, i.e., when the ArchC behavioral description and the DUV are not time equivalent. See Section 4 for more details on the co-verification mechanism.

[1] The GNU Compiler Collection. http://gcc.gnu.org

```
========================================
 This is ArchC Pre-processor version 0.7
========================================

Usage: acpp input_file [options]
       Where input_file stands for your AC_ARCH description file.

Options:
    --debug, -g             Enable simulation debug features: traces, update logs.
    --dumpdecoder , -dd     Dump the decoder data structure.
    --help        , -h      Display this help message.
    --quiet       , -q      Do not display update logs for storage devices during simulation.
    --stats       , -s      Enable statistics collection during simulation.
    --verify      , -v      Enable co-verification mechanism. Untimed model.
    --verify-timed, -vt     Enable co-verification mechanism. Timed model.
    --version     , -vrs    Display ArchC version information.
```

Figure 13: Acpp command-line options

- -verifytimed, -vt :
When this option is chosen, `acpp` will generate a simulator prepared for execute the storage-based co-verification following the timed model, i.e., when the ArchC behavioral description and the DUV are time equivalent. See Section 4 for more details on the co-verification mechanism.

- -version, -vrs :
This option can be used by itself, i.e., it does not require the presence of a `AC_ARCH` file name, and just displays the ArchC version number to the user.

6 Conclusions and Future Work

This report 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. Table 1 shows a comparison among several ADLs, based on the table presented in [18]. We indicate with an **F** features that are planned in future ArchC releases or that are currently being implemented. Question marks in that table indicate that not enough evidence exist in the literature to support the ADL developers claim. ArchC has many features that distinguish it from other ADLs, in addition, it will have the language and the whole set of tools and models mentioned in this text published in public domain. This does not happen with any other ADL, making ArchC a powerful and suitable support tool for computer architecture research.

We are currently working on improving and tunning our co-verification algorithm and interfaces, paying attention to the new SystemC Verification Standard that is being developed and has already a beta version published. We have already used ArchC to model the MIPS, Sparc and Intel 8051 architectures, and we are starting a model of a real-world DSP

Feature	LISA	EXPRESSION	nML	ISDL	ArchC
Cycle-accuracy	•	•		?	•
Multi-cycle supp.	•	•			•
Pipeline supp.	•	•	?	?	•
Instr-set info	•	•	•	•	•
Compiler supp.	?	•	•	•	**F**
Assembler supp.	•	•	•	•	**F**
Memory Hierarchy supp.		•			**F**
Storage-based co-verif.					•
Format Behavior					•

Table 1: Comparison among ADLs

architecture, the TMS320C62x, in order to test and improve the ArchC design. We are considering techniques to speedup simulation time, though this is not yet the focus at this stage of the work. There is also some ongoing work in order to enable ArchC to simulate more sophisticated memory hierarchies with several cache levels. A system calls emulation library and a binary application loader are in the last steps of their implementation and tests. They will allow designers to run applications compiled for the target architecture using the simulator generated by ArchC, including I/O emulation through the host machine. Finally, though the task of automatically generating an assembler for an architecture described in ArchC is not very complicated, it still needs to be done.

References

[1] SimpleScalar homepage. In *http://www.simplescalar.com*.

[2] *Code Generation for Embedded Processors*, chapter Chess: Retargetable Code Generation for Embedded DSP Processors, pages 85–102. Kluwer Academic Publishers, Boston, 1995.

[3] A. Fault, 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.

[4] Andreas Hoffmann, Oliver Schliebusch, Achim Nohl, Gunner Braun, Oliver Wahlen, Heinrich Meyr. A Methodology for the Design of Application Specific Instruction Set Processors (ASIP) using the Machine Description Language LISA. In *in Proc. of International Conference on Computer Aided Design (ICCAD)*, pages 625–630, 2001.

[5] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer. The MIMOLA Language - Version 4.1. Technical report, Computer Science Dpt., University of Dortmund, September 1994.

[6] R. Bedichek. Some efficient architecture simulation techniques. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 53–64, Berkeley, CA, 1990. USENIX Association.

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

[8] Eric Schnarr and James Larus. Fast Out-Of-Order Processor Simulation Using Memoization. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, California, October 1998.

[9] M. Freericks F. Lohr, A. Fauth. SiGH/SIM - An Environment for Retargetable Instruction Set Simulation. Technical Report 43, Comp. Science Dept., T.U. Berlin, Berlin (Germany), 1993.

[10] A. Fauth and A. Knoll. Automated generation of DSP program development tools using a machine description formalism. In *Proc. IEEE Int. Conf. Acoustics, Speech, Signal, Minneapolis (Minn., U.S.A.)*, pages 457–460, 1993.

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

[12] Richard Goering. Keynoter Says Chip Value is in its Intellectual Property. In *http://www.eedesing.com/story/OEG20020614S0100*, June 2002.

[13] Thorsten Grotker, Stan Liao, Grant Martin, and Stuart Swan. *System Desing with SystemC*. Kluwer Academic Publishers, 2002.

[14] P. Grun, A. Halambi, N. Dutt, and A. Nicolau. RTGEN: An Algorithm for Automatic Generation of Reservation Tables from Architectural Descriptions. In *in Proceedings of International Symposium on System Synthesis (ISSS)*, 1999.

[15] George Hadjiyiannis and Srinivas Devadas. Techniques for Accurate Performance Evaluation in Architecture Exploration. *IEEE Transactions on VLSI Systems*, 2002.

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

[17] George Hadjiyiannis, Pietro Russo, and Srinivas Devadas. A methodology for accurate performance evaluation in architecture exploration. In *Design Automation Conference(DAC)*, pages 927–932, 1999.

[18] 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.

[19] Silvia Hanono and Srinivas Devadas. Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator. In *in Proc. of 35th Design Automation Conference(DAC)*, June 1998.

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

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

[22] Andreas Hoffmann, Achim Nohl, Stefan Pees, Gunnar Braun, and Heinrich Meyr. Generating production quality software development tools using a machine description language. In *Proceedings of Design, Automation and Test in Europe Conference(DATE)*, March 2001.

[23] J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, 2000.

[24] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, New Jersey, 1992.

[25] Peter S. Magnusson, Fredrik Larsson, Andreas Moestedt, Bengt Werner, Fredrik Dahlgren, Magnus Karlsson, Fredrik Lundholm, Jim Nilsson, Per Stenström, and Håkan Grahn. SimICS/sun4m: A Virtual Workstation. In *In Proceedings of the Usenix Annual Technical Conference*, pages 119–130, 1998.

[26] Mark R. Hartoog, James A. Rowson, Prakash D. Reddy, Soumya Desai, Douglas D. Dunlop, Edwin A. Harcourt, Neeti Khullar. Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign. In *in Proc. Design Automation Conference*, pages 303–306, 1997.

[27] Prabhat Mishra, Nikil D. Dutt, and Alexandru Nicolau. Functional abstraction driven design space exploration of heterogeneous programmable architectures. In *in Proceedings of International Symposium on System Synthesis (ISSS)*, pages 256–261, 2001.

[28] Chuck Siska. A processor description language supporting retargetable multi-pipeline dsp program development tools. In *in Proceedings of International Symposium on System Synthesis (ISSS)*, December 1998.

[29] Vojin Zivojnovic e Heinrich Meyr Stefan Pees, Andreas Hoffmann. LISA - Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures. In *in Proc. of ACM Design Automation Conference(DAC)., New Orleans*, 1999.

[30] System Collaborators. *SystemC Version 2.0 User's Guide*, 2001.

[31] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Measurement and Modeling of Computer Systems*, pages 68–79, 1996.

[32] 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.

[33] Vojin Zivojnovic, Stefan Pees, C. Schalger, Markus Willems, R. Schoenen, and Heinrich Meyr. DSP Processor/Compiler Co-Design: A Quantitative Approach. In *International Symposium on System Synthesis (ISSS)*, 1996.