The ArchC Architecture Description Language

v2.0

Reference Manual

The ArchC Team
http://www.archc.org

August 2007

Acknowledgments

This manual was developed by Sandro Rigo and Thiago Massariolli Sigrist. Please, if you have any question about this document email it to *archc@lsc.ic.unicamp.br*.

ArchC is an open-source architecture description language that has been designed at the Computer Systems Laboratory (LSC) of the Institute of Computing of the University of Campinas (IC-UNICAMP). The ArchC Team:

- Guido Araújo
- Rodolfo Jardim de Azevedo
- Paulo César Centoducatte
- Sandro Rigo
- Marcus Bartholomeu
- Bruno de Carvalho Albertini
- Márcio Rogério Juliato
- Alexandro Baldassin
- Thiago Massarioli Sigrist
- Marilia Felippe Chiozo
- Danilo Marcolin Caravana
- Luis Felipe Strano Moraes

ArchC also relies on the great work performed by our collaborators:

- Informatics Center of Federal University of Pernambuco (Cin-UFPE)
  - Edna Natividade Barros
  - Pablo Viana da Silva
  - Cristiano Coelho de Araújo
  - Tiago Sampaio Lins
  - Silvio Veloso
  - Diogo Az

- Systems Design Automation Lab of Federal University of Santa Catarina (LAPS-UFSC)
  - Luíz Claúdio Villar dos Santos
  - Olinto J. V. Furtado
  - Daniel Casaroto
  - José Otávio Carlomagno Filho
  - Leonardo Taglietti
  - Max Schultz
  - Alexandre Mendonça
  - Felipe Carvalho
  - Gabriel Renaldo Laureano
  - Luiz Penkal

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter presents some background information on Architecture Description Languages (ADLs) and related work, followed by a proper introduction of the ArchC language and more useful information on setting up ArchC and its toolkits on one of the supported platforms. Finally, an overview of ArchC 2.0 follows, describing it most important new features and changes.

1.1 Background and Related Work

Architecture description languages (ADL) have been introduced to help designers face the development challenges that have arisen in the past few years, due to the rapidly increasing complexity of modern architectures. These difficulties, that end up delaying the whole design process and preventing designers of meeting their stringent time to market, have forced hardware architects and software engineers to reconsider how designs are specified, partitioned and verified. As a consequence, designers are starting to move from hardware description languages (VHDL, Verilog) and also beyond the RTL level of abstraction toward the so called *system level design*, where automatic generation of a software toolkit (composed by assemblers, linkers, compilers and simulators) is mandatory. Such tools are commonly based on an architecture description language.

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

1.1.1 SystemC

SystemC [10, 2, 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 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 from a generic SystemC processor description all necessary information, in order to automatically generate tools to experiment and evaluate a new *Instruction Set Architecture* (ISA).

1.1.2 GNU Binutils

The GNU Binutils [14] are a collection of binary tools such as an assembler, a linker and object files inspectors. The package is used by GCC [15] (*The GNU Compiler Collection*) to assemble and link the files generated by the compilers. It allows one to build binary tools for a new architecture by rewriting the machine-dependent files, while still reusing the machine-independent ones.

The current version of the ArchC toolkit contains a tool, `acbingen`, capable of generating the machine-dependent files in order to retarget the GNU Binutils package to a machine described in the ArchC language.

1.2 The ArchC Architecture Description Language

In this document, we introduce a SystemC-based architecture description language called ArchC. ArchC is a simple language, capable of describing a processor architecture as well as a memory hierarchy, 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, linkers and debuggers.

An architecture description in ArchC is divided in two parts: the `Instruction Set Architecture (AC_ISA)` description and the `Architecture Resources (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 can generate interpreted simulators (using SystemC), compiled simulators and assemblers (using the Binutils framework). The following chapters cover both of these descriptions in details.

Throughout this text, we are going to use examples extracted from our ArchC descriptions of three architectures: MIPS (implementing the MIPS-I instruction set), SPARC-V8, and Intel 8051. MIPS [3,11] and SPARC [13] are both well-known RISC architectures with five-stage pipelines. 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. All these models are available at the ArchC website [5].

1.2.1 Download and Installation

The ArchC package can be found at the ArchC website [5], by following these links: Language→Download. The package comprises a tree of C/C++ source files. You will need a C/C++ compiler to build ArchC binaries. We suggest the use of the GNU Compiler Collection (GCC) [4], version 3.3 or higher, running on a Linux system. ArchC has also been reported to work in Windows, under the Cygwin environment [6].

ArchC may generate simulators using two different techniques: interpreted or compiled. The interpreted simulators use SystemC as the simulation engine. So, if you plan to use those kind of simulation it is mandatory to have a working SystemC installation in your machine. On the other hand, the compiled simulators do not use SystemC and thus it is not nescessary. All packages and information you need to install SystemC can be found at [10]. Please, make sure you are installing SystemC version 2.1v1 or higher. The assembler generator tool requires the GNU Binutils 2.15 [7] source distribution, or newer. Moreover, ArchC also requires the following softwares:

- Bison (version 1.5 or higher)
- Flex (version 2.5.4 or higher)
- GCC (version 3.3 or higher)

Notice that newer versions are OK, but we do not recommend older versions. You can install ArchC in any directory you want to. Root privileges are not necessary, just like it happens to SystemC.

ArchC relies on a configuration file to get all system information needed to generate the simulators. This file is located in the configuration directory under the installation prefix (usually `$PREFIX/etc`). It is named `archc.conf` and is automatically created by the `make install` command during installation. These are the variables contained into the configuration file:

SYSTEMC_PATH :
It is the path for your SystemC installation that must be used together with ArchC. This variable is also present in the makefiles generated for the examples distributed with SystemC during its installation.

CC :
The name and/or path to the C++ compiler that must be used by ArchC tools. Its normally set to `g++`.

OPT :
It is the optimization flags to be passed to gcc during model's compilation.

DEBUG :
It is the debugging flags to be passed to gcc during model's compilation. ArchC normally uses the `-g` flag.

OTHER :
It is the remaining flags that the user wants to be passed to gcc during model's compilation. The default is to use `-Wall` and `-Wno-deprecated`.

TARGET_ARCH :
It must be set to the same value used by this variable in your SystemC installation.

ArchC tools also allow user-specific configuration: any of the above variables may be overridden in the file `$HOME/.archc/archc.conf`. This feature is useful for system-wide installations of ArchC.

The ArchC source package is developed using the very popular **GNU Autotools** configuration management system. Therefore, installation can be performed simply by invoking the `configure` script, passing parameters like installation prefixes and `CFLAGS`, then running `make` to compile all the source code and `make install` to install it.

The `configure` script for the ArchC source package uses three specific options, which must be set in case the associated features are desired. These are the options, in detail:

`--with-systemc=PATH` :
Setting this option enables SystemC support, which is required for the interpreted simulator generator tool (`acsim`). Its value `PATH` must be set to the SystemC installation prefix.

`--with-tlm=PATH` :
Setting this option enables SystemC TLM 1.0 support for interpreted simulators (generated with the `acsim` tool). The `PATH` value must be set to the SystemC TLM 1.0 library installation prefix.

`--with-binutils=PATH` :
Setting this option enables support for binary utilities generation, via the `acbingen` tool. The `PATH` value must point to a directory containing the **GNU Binutils** source files. More information about `acbingen` can be found on its separate manual.

For more details on ArchC tools and simulators, please see Chapter 4.

1.3 Changes from ArchC 1.6

This version of ArchC, 2.0, introduces several significant changes since version 1.6. The main difference of ArchC 2.0 is in the greater usability of functional interpreted simulators, allowing these simulators to be used as CPU/microprocessor modules in system-level models.

Therefore, the functional interpreted simulators generated from an ArchC functional architecture description have the following main advantages:

Self-containment. ArchC 2.0 simulators rely only on instance variables or members, instead of depending on global variables or static members like in ArchC 1.6. Thus, multiple instantiations of simulator modules are possible, enabling multiprocessor system simulation.

TLM-based external communication. Simulator modules may have external communication ports, allowing them to read or write data to external devices or, likewise, to be interrupted by such devices. The protocol for such communication is based on the SystemC TLM 1.0 standard, therefore enabling integration of processor modules to a system-level model.

Moreover, other changes to functional interpreted simulators include:

Extra speed. Functional interpreted simulators generated with the `acsim` tool of ArchC 2.0 are, on average, 30 times faster than the same simulators generated with ArchC 1.6 tools.

Cleaner compilation. In ArchC 1.6, library classes like those for the instruction decoder (`ac_decoder`), storage elements (`ac_storage`) and so forth had their source (`.cpp`) files copied to the model directory for in-place compilation, effectively being recompiled at every simulator compilation. ArchC 2.0 solves the problem by compiling library source files at the ArchC package compilation time, grouping the resulting object files in a library which is effectively installed (instead of installing `.cpp` files). This effectively reduces simulator compilation time and restricts only model-specific files to the model directory, leading to a cleaner practice.

Other noteworthy changes include:

Separation of functional and timed simulators. ArchC 2 has two separate tools for generating interpreted simulators: `acsim`, which generates functional simulators, and `actsim`, responsible for timed (cycle-accurate) simulators. As of ArchC 2.0, the `actsim` tool is not part of the core ArchC distribution, but is available separately as a beta version.

Inclusion of `acstone`. The ArchC project's own benchmark and test suite for processor models, `acstone`, being a tool which use is highly recommended, is now part of the ArchC core distribution.

The development of ArchC 2.0 also has taken into consideration the issue of forwards compatibility, making sure models written for versions 1.6 and previous require as little modification as possible to work under 2.0. A guide for migrating models to ArchC 2.0 is provided on chapter.

However, ArchC 2.0 has the following known limitations:

`accsim` still at version 1.6. The ArchC compiled simulator generation tool, `accsim`, has not yet been updated for ArchC 2.0, being available only on the ArchC 1.6 package. The `accsim` tool is being revamped for ArchC 2 and will appear in the core distribution on upcoming versions.

Chapter 2

Describing Architecture Resources

ArchC needs some structural information about the resources available in the architecture in order to automatically generate software tools. The designer must provide such an information in the `AC_ARCH` description.

The detail level used for this description will depend on the level of abstraction desired for the model. For example, one may want to simulate the instruction set of the MIPS architecture, but without concerning with pipelining. In fact, this is exactly how we advice designers to start a new architecture model in ArchC, even if a cycle-accurate model is necessary. A model without timing, or cycle-accuracy, or pipelining information is called *functional*. Such a model simulates the behavior of an instruction set, executing all operations of a given instruction during a single SystemC delta cycle. By building a functional model, the designer gathers enough knowledge about the ISA to further describe a more detailed (refined) model, thus avoiding to propagate bugs that could have been fixed at early design stages. An architecture description at the functional level demands very simple instruction behavior descriptions, as we are going to see in Chapter 3, but also demands few structural information, like showed in Figure 2.1. This example illustrates the minimum amount of architecture resource information needed to build a MIPS functional model. In order to get a more detailed model, like a cycle-accurate model of a MIPS family processor, the designer must provide more structural information. Figure 2.2 shows our description of the architecture resources for the R3000 processor, which is an implementation of the MIPS-I architecture.

2.1 Structure of Architectural Resources Declaration

An ArchC architectural resources declaration (`AC_ARCH`) has a very simple structure, which is delimited by the `AC_ARCH` statement, described below:

`AC_ARCH` :

Synopsis :

`AC_ARCH` (*project_name*) {
// (...) *resource declarations*
};

Description :

An architecture resources description always starts with this keyword, like in the first line of Figure 2.1. The designer is supposed to inform a name for the project enclosed in parentheses (*project_name*), like it is usually done for modules in SystemC.

project_name : Name for the project. Can be any name, but is commonly a descriptor for the architecture (ISA) being described, possibly including version numbers if necessary. Examples: `mips1`, `sparcv8`. Timed (cycle-accurate) models are usually named after specific processor models instead of the ISA, since the timing information usually ties the ArchC description to a specific implementation. Example: `r3000`. In case it's appropriate for both a functional and a cycle-accurate to share the same name, it's common practice to include a '`_ca`' suffix in the cycle-accurate model project name.
A far more important convention refers to description file naming. The architecture resource description file for a project called `project` should be called `project.ac`. Likewise, the ISA declaration file (described on Chapter 3) for this project should be called `project_isa.ac`. Despite not being enforced by ArchC tools like `acsim`, this convention must be followed for two extremely important reasons:

- Every other file comprising an ArchC model will have the project name as a prefix, no matter whether generated by a tool or not. Therefore, the two files with `ac` extension will be the only ones not following this convention, which is confusing.
- Certain tools or frameworks (like ARP or Platform Designer?) that use ArchC as clients might require this naming convention to be followed, as it facilitates automation.

resource declarations : Declarations of all architecture resources of the processor being modelled. These include: internal storage (memory), individual registers, register banks, and so forth. A comprehensive list of every possible resource declaration in the ArchC language is presented in the next section.

2.2 Declarations of Individual Resources and Architectural Characteristics

In this section are presented all the ArchC keywords that may appear into an `AC_ARCH` descriptions, presented in the same style as the main `AC_ARCH` declaration in the previous section.

`ac_wordsize` :

SYNOPSIS :

`ac_wordsize` *wordsize*;

DESCRIPTION :

Defines the architecture word size.

In ArchC, this is is the default size for:

- words fetched from instruction memory;
- words read from (or written into) data memories or ports;
- registers.

There is also a way to override the default size for each of those. To fetch differently-sized words from instruction memory, use `ac_fetchsize`. Data memories and ports allow for reads and writes in bytes, half words, full words and double words. Finally, both individual registers and register banks can be declared with arbitrary register widths.

wordsize : Word size in bits. Can be either 8, 16, 32 or 64. ArchC currently doesn't allow for other word sizes, and it doesn't allow for operations in double words when word size is 64 bits.

`ac_fetchsize` :

SYNOPSIS :

`ac_fetchsize` *fetchsize*;

DESCRIPTION :

Defines the size of words fetched from instruction memory.

Please note this is not the same as the size of an instruction word: it is the size of a word fetched from instruction memory. Instruction words may be composed of one or more of those fetched words.

fetchsize : Word size in bits. Can be either 8, 16, 32 or 64. ArchC currently doesn't allow for other word sizes, and it doesn't allow for operations in double words when word size is 64 bits.

`ac_format` :

SYNOPSIS :

`ac_format` *format_name* `= "`(`%`*field_name*`:`*field_size*(`:s`)?)+`"`;

DESCRIPTION :

Defines a format and its fields.

Formats defined inside an `AC_ARCH` description are meant to be used by formatted and pipeline registers. Refer to the `ac_reg` keywords for more details on how to assign a previously defined format to a register.

format_name : Name of the format being defined.

FORMAT STRING :

A format is defined with a double-quoted format string. In the synopsis, the format string is concisely presented in a syntax similar to regular expressions.

A more detailed explanation goes as following: the string assigned to a format represents its subdivision into fields, therefore, it is a sequence of one or more fields (thus the + sign in the expression). Fields are separated from each other by whitespace, and are declared with a leading percent (`%`) character, followed by the field name, a colon (`:`) and the size of that field in bits. An optional suffix "`:s`" means the field contains a signed value.

`ac_mem` :

SYNOPSIS :

`ac_mem` *mem_name*`:`*mem_size*;

DESCRIPTION :

Declares a storage object of type `ac_mem`.

This type of storage object, `ac_mem`, is meant to be used as an internal memory. Internal memory objects are used in basically three situations:

- To model actual internal memories, such as those present in microcontroller chips like i8051, PIC, Atmel AVR etc;
- To model memories local to a processor model. This is useful when you don't want to communicate via that particular memory over a bus (or other TLM-modelled communication structure), for the sake of simplicity and performance;
- To model main memory in a processor model meant to be used only as a standalone ISA simulator. Even though this is a special case of the previous situation, it's important to mention it since all official ArchC processor models fall into this situation, since they were designed for use with ArchC 1.x, which allowed only standalone simulation.

mem_name : Name of the memory element being declared.

mem_size : Size of the memory element being declared, in bytes. Byte-multiple units may be used: kilobytes (2^{10} or 1024 bytes, suffix `k` or `K`), megabytes (2^{20} or 1048576 bytes, suffix `m` or `M`) or gigabytes (2^{30} or 1073741824 bytes, suffix `g` or `G`). Since the size is expressed as a number followed by an optional unit, a regular expression for *mem_size* is: $\big[$`0-9`$\big] + \big[$`kKmMgG`$\big]?$

`ac_tlm_port` :

SYNOPSIS :

`ac_tlm_port` *port_name*`:`*port_size*;

DESCRIPTION :

Declares an outbound TLM communication port.

Outbound TLM communication ports are often used to provide a means to connect the processor simulator to TLM models of devices such as buses, external memories and other TLM IPs.

port_name : Name of the TLM port being declared.

port_size : Size of the address space of the port being declared, in bytes. The size specification is exactly the same as in `ac_mem`.

`ac_tlm_intr_port` :

SYNOPSIS :

`ac_tlm_intr_port` *port_name*;

DESCRIPTION :

Declares a TLM interrupt port.

Interrupt ports are used in ArchC to model an ordinary interrupt port, into which any write transaction causes the simulated processor to be interrupted.

port_name : Name of the TLM port being declared.

`ac_regbank` :

SYNOPSIS :

`ac_regbank`$\big($`<`*reg_size*`>`$\big)?$ *regbank_name*`:`*num_regs*;

DESCRIPTION :

Declares a register bank.

regbank_name : Name of the register bank being declared.

num_regs : Number of registers that compose this register bank.

reg_size : Size of the registers that compose this register bank, specified in bits. Can be 8, 16, 32 or 64. If not specified, the default word size (defined with `ac_wordsize`) is used. The same caveat mentioned in `ac_wordsize` applies: 64-bit registers don't allow for double-word operations.

`ac_reg` :

SYNOPSIS :
`ac_reg`(`<`*reg_size_or_format*`>`)? *reg_name*;

DESCRIPTION :
Declares a register.

reg_name : Name of the register being declared.

reg_size_or_format : Either the identifier of a format for the register (which has to be previously defined with `ac_format`) or, for a non-formatted register, the size of the register, specified in bits (like in `ac_regbank`, can be 8, 16, 32 or 64). If not specified, a simple, non-formatted register is created, and the default word size (defined with `ac_wordsize`) is used for it.

`ac_pipe` :

SYNOPSIS :
`ac_pipe` *pipe_name* `=` { *stage1_name* (`,`*stageN_name*)* }

DESCRIPTION :
Declares a pipeline.

pipe_name : Name of the pipeline being declared.

stage1_name..stageN_name : A comma-separated sequence of pipeline stage names for the pipeline being declared. Being a sequence, it is, naturally, ordered, therefore the order provided by the user will be used in executing instructions.

EXAMPLE :
See Figure 2.2 for an example.

`ARCH_CTOR` :

SYNOPSIS :
`ARCH_CTOR` (*project_name*) {
// (...) *model initialization*
};

DESCRIPTION :
The `AC_ARCH` constructor declaration. It is the mandatory last declaration inside an `AC_ARCH` block.

project_name : Project/architecture name. Must match the one declared in the `AC_ARCH` statement.

model initialization : Statements initializing important parts of the model description. Currently only two are supported, both of which mandatory: `ac_isa` and `set_endian`.

`ac_isa` :

SYNOPSIS :

`ac_isa("`*isa_file*`");`

DESCRIPTION :

Informs the name of the ISA declaration file to the ArchC tools.

isa_file : Name of the file containing the `AC_ISA` declaration, which is the full ISA declaration of the model.

`set_endian` :

SYNOPSIS :

`set_endian("`*endianness*`");`

DESCRIPTION :

Sets endianness for the architecture.

endianness : Endianness for the architecture modelled. Can be either `little` or `big`.

2.3 Resources Declaration Examples

In this section, examples will be presented for all of the most important keywords and declarations/definitions that appear inside the `AC_ARCH` declaration body. To give users a better picture of this first, extremely important, step in writing a new ArchC model, full examples for both a functional and a cycle-accurate model (both based on MIPS) will be provided.

```
AC_ARCH(mips){

  ac_wordsize 32;

  ac_mem   MEM:256k;
  ac_regbank RB:32;
  ac_reg hi, lo;

  ARCH_CTOR(mips) {

    ac_isa("mips_isa.ac");
    set_endian("big");
  };
};
```

Figure 2.1: `AC_ARCH` Resource Declaration for a Functional MIPS Model.

```
AC_ARCH(mips){

  ac_wordsize 32;

  ac_mem  MEM:256K;
  ac_regbank RB:32;
  ac_reg hi, lo;

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

  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<Fmt_IF_ID>  IF_ID;
  ac_reg<Fmt_ID_EX>  ID_EX;
  ac_reg<Fmt_EX_MEM> EX_MEM;
  ac_reg<Fmt_MEM_WB> MEM_WB;

  ARCH_CTOR(mips) {

    ac_isa("mips_isa.ac");
    set_endian("big");

  };
};
```

Figure 2.2: `AC_ARCH` Resource Declaration for a Cycle-accurate MIPS Model.

Summarizing, the `AC_ARCH` description in Figure 2.1 gives all resource information we need to develop a functional model of the MIPS-I architecture: a memory for data and instructions, the demanded 32-entry register bank, registers `hi` and `lo` (used in multiplication and division operations) and the correct endianness.

However, more structural information is necessary in order to get a cycle-accurate model. Figure 2.2 illustrates how we declared the five-stage pipeline and the four pipeline registers, `IF_ID, ID_EX, EX_MEM` and `MEM_WB`, associating a format to each one of them. This allows the designer to assign to (and read from) each register field individually when describing behaviors, as we are going to see in Chapter 3.

Chapter 3

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 3.1 shows an example of an AC_ISA description extracted from our MIPS model. 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 3.2 is an excerpt of our Intel 8051 microcontroller description, which is one of the most used processors for embedded control. This is a CISC architecture with multi-cycle instructions of variable length. Based on these examples, let's analyze each ArchC keyword that may appear in an AC_ISA description:

`AC_ISA` :
: An ISA description always starts with this keyword. The designer is supposed to inform a name for the project enclosed in parentheses, as it is done for AC_ARCH descriptions (Chapter 2).

`ac_format` :
: Declares a format and its fields. The syntax is similar to the one used in Chapter 2, when we presented the AC_ARCH description. The difference is that here formats will be associated to instructions, not to registers. There is an aditional construct for instructions formats that allows fields to overlap. It can be used to facilitate the description of complex instruction-sets. A group of fields choices starts with a curly brace ("[") and additional groups are given after a vertical bar ("|"). Finally, after all fields groups have been declared, use a closing curly brace ("]"). As an example, we could define the following instruction format for the SPARC-V8 architecture:

```
ac_format Type_F3 = "%op:2 %rd:5 %op3:6 %rs1:5 %is:1 [ %asi:8 %rs2:5 | %simm13:13:s ]";
```

ArchC decodes all fields for instructions defined with this format and they can be acessed independently , but note that not all of them may be valid. The designer would have to chose which group is valid by testing the value of some other field (the "`is`" in this case). For the SPARC-V8, `is=0` means only "`asi`" and "`rs`" fields are valid, `is=1` means only "`simm`" is valid.

`ac_instr` <**fmt**> :
Declares an instruction. Every instruction must have a previously declared format associated to it. Formats are assigned to instructions using a syntax similar to C++ *templates*. In Figure 3.1, format `Type_R` is associated to instruction `add`.

`ISA_CTOR` :
Initializes the `AC_ISA` constructor declaration.

`ac_asm_map` : (assembler specific - check the `acasm` manual for details)
Specifies a mapping between assembly symbols and values. Example of Figure 3.1 defines the set of register names and values for the MIPS-I architecture.

`set_asm` : (assembler specific - check the `acasm` manual for details)
Associates an assembly syntax string and operand encoding to an instruction. The syntax of this construct is similar to the `printf` familiy used in the C language. Literal characters must be matched as it appears in the assembly source program, while conversion specifiers (`%`) force the assembler to recognize ranges of values or symbols for operands. For each operand, there must be an instruction field associated, specifying the operand encoding. Example of Figure 3.1, with the `add` instruction, uses three operands of type `reg`.

`set_decoder` :
Initializes the instruction decoding sequence, which is a key element to the automatic generation of an instruction decoder. The sequence is composed of pairs <*field_name = value*>. In Figure 3.1, 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. 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.

`set_cycles` :
Provide the latency of a multi-cycle instruction (used only for cycle-accurate models of multi-cycle architectures). The i8051 is a multi-cycle processor, which means that instructions will take different number of cycles to be completed. Take a look at the `AC_ISA` constructor declaration in Figure 3.2, where this value is initialized for the two-cycle `mov_r_iram` instruction. Observe that the `add_ar` instruction does not have a call to the `set_cycles` method, so ArchC assumes that it is an one-cycle instruction, by default.

`pseudo_instr` : (assembler specific - check the `acasm` manual for details)
Creates a pseudo instruction based on previously created instructions. Example of Figure 3.1 declares the pseudo instruction `li` based on the previously declared instructions `lui` and `ori` (they are not show in this example).

3.1 Behavior Description in ArchC

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 simulator generator (`acsim`), the designer gets a template of the behavior description file. This template is a skeleton of a .cpp (C++/SystemC source) file where the designer is going to fill out the behavior method of each instruction in the architecture. The template for the behavior description file is named as *project_name* _isa.cpp.tmpl, by default.

One strong feature in ArchC is the capability of modeling behaviors 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. As the design process moves forward, the model can be refined to express operations in a cycle-accurate fashion.

In order to model complex instruction sets, it is very important to be able to share operations among several instructions. Often, there are many instructions in a particular architecture that execute exactly the same operations as part of their behavior. For instance, 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 in 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.

The ArchC behavior hierarchy aims to solve this problem. Its goal is to factor out operations that are executed by instructions of the same format, or even by all instructions. So, the designer has three different kinds of behavior to describe in ArchC: the generic instruction behavior, the format behavior, and the specific instruction behavior. When executing an instruction, the simulator follows exactly this order, i.e., the behavior that is common to all instructions executes first, followed by the correspondent format behavior, and finally the behavior of the specific instruction is executed. We are going to analyze in details each one of them in the following sections.

3.1.1 Providing Format and Generic Instruction Behaviors

Consider, as an example, our MIPS family models. A typical operation that is performed by every single instruction is the program counter (PC) increment. In ArchC, the PC value is

```
AC_ISA(mips){

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

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

  ac_asm_map reg {
     "$"[0..31] = [0..31];
     "$zero" = 0;
     "$at" = 1;
     "$kt"[0..1] = [26..27];
     "$gp" = 28;
     "$sp" = 29;
     "$fp" = 30;
     "$ra" = 31;
  }

  ISA_CTOR(mips){

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

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

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

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

    pseudo_instr("li %reg, %imm") {
      "lui %0, \%hi(%1)";
      "ori %0, %0, %1";
                              ...
    }
  };
};
```

Figure 3.1: MIPS ISA Description

```
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 3.2: Intel 8051 ISA Description

controlled by the variable `ac_pc`. Both functional and cycle-accurate models can make use of the generic instruction behavior method to increment this value. Figure 3.3 shows how simple it is to perform such an operation in a functional model: just increment the desired variable. Figure 3.4 shows a more complex example, that is used for cycle-accurate models. The increment must be performed at the right pipeline stage, and its value must be copied to a field into a pipeline register, in order to be propagated through the pipeline. ArchC provides formatted registers, but the designer is responsible for storing values to fields and copying values from one register to another. Moreover, this cycle-accurate example also shows that the operations performed during the `write-back` stage may be codified in this generic behavior method, avoiding useless code repetitions in specific instruction behaviors.

Often, all instructions with the same format also perform many operations in common. ArchC provides the designer with the possibility of overloading the `ac_behavior` method so that it can take an instruction format as argument. Take a look again at the MIPS processor, all instructions that were declared with the format `Type_R` associated to it execute a couple of tests, showed in Figure 3.5, *before* running its own behavior, so that they can do register forwarding for both instruction operands, `rs` and `rt`. The code in Figure 3.5 happens to be as simple and clear as it is presented in the Hennessy and Patterson's classical architecture

```
void ac_behavior( instruction )
{
    ac_pc += 4;
};
```

Figure 3.3: Generic Instruction Behavior Description in a Functional Model

book [3]. The same strategy can be used for coding data hazard detection.

3.1.2 Providing Instruction Behavior

Specific instruction behaviors follow exactly the same syntax and rules as explained above. Now, the designer will provide operations that are unique to each instruction.

Let's first see some examples of instruction behavior description in functional models. Figure 3.6 shows the behavior of the `add`, `load word` and `store half word` instructions for the MIPS architecture. MIPS-I is a very simple instruction set, so many of its instructions can be as easily described as these examples. It is important to notice that the storage objects declared by the designer in the `AC_ARCH` description are accessed through `read` and `write` methods in behavior descriptions. Memories and caches are always byte addressed and these methods always return a word. But ArchC also provides methods for manipulating bytes and half-words from a memory address, which are: `read_byte/write_byte` and `read_word/write_word`. The behavior of the `sh` (store half word) instruction in Figure 3.6 illustrates the use of one of these methods.

Figure 3.7 shows an example extracted from our SPARC-V8 functional description. SPARC is also a RISC architecture, but has a more complex ISA when compared with MIPS-I. We have to mention one important feature illustrated by this example: the possibility of calling user defined functions from ArchC behavior methods. The behavior file is a regular C++ (.cpp) source file. Into it, the designer is able to insert any extra code he/she likes to. Sometimes, it is better to create auxiliary functions to perform some specific tasks. For example, the SPARC architecture makes use of register windows. In order to simulate this feature, the ArchC SPARC model designer decided to declare a register bank object containing 256 registers. SPARC machines do not have so many registers. The 256 registers are divided into 16 register windows. Take a close look at the example. Notice that the designer does not access the register bank directly through its read or write methods, like it is done in the MIPS model. Instead, new functions were created to accomplish this task, so its is possible to handle register window specific details before doing the actual access. The user defined functions `readReg` and `writeReg` are responsible for doing this job. Moreover, the SPARC architecture also demands a more complicated scheme for computing the PC increment, so this is also coded into a separate function, called `update_pc`. If the reader wants more details on the code of our SPARC-V8 model, it is available at the ArchC website [5].

Figure 3.8 shows a cycle-accurate example. For the MIPS model, we need to declare a

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

                 ...

  case WB:

    /* Execute write back when allowed */
    if (MEM_WB.regwrite == 1) {

      // Register 0 is never written
      if (MEM_WB.rdest != 0)
         RB.write(MEM_WB.rdest.read(), MEM_WB.wbdata.read());
    }

    break;

  default:
    break;
  }
};
```

Figure 3.4: Generic Instruction Behavior Description in a Cycle-accurate Model

```
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 3.5: MIPS Type_R Format Behavior Description

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

//Instruction load word behavior method.
void ac_behavior( lw )
{
  RB.write(rt, DM.read(RB.read(rs)+ imm));
};

//Instruction store half word behavior method.
void ac_behavior( sh )
{
  unsigned short int half;

  half = RB.read(rt) & 0xFFFF;
  MEM.write_half(RB.read(rs) + imm, half);
};
```

Figure 3.6: MIPS Instruction Behavior Description in a Functional Model

pipeline, along with its pipeline registers, in the `AC_ARCH` description (see Figure 2.2) and then divide the behavior of each instruction, telling ArchC what is done at each pipeline stage. Figure 3.8 illustrates part of the `add` instruction behavior in a cycle-accurate fashion. Notice that several operations that should be performed by an `add` instruction were transfered to the format (Type_R) and generic instruction behaviors, as they must also be performed by other instructions.

3.1.3 Utility Methods and Important Variables

This section describes some methods provided by ArchC in order to help in behavior descriptions. Its important to know that these methods may only be called from `ac_behavior` methods, since they will not be available *outside* the ArchC simulator classes.

`ac_stall` (***stage***) :
: Stalls a pipeline stage. Receives a stage name as argument. The instruction currently being executed will remain in the same stage on the next simulation cycle, and a *nop* will be inserted on the following stage.

`ac_flush` (***stage***) :
: Flushes a pipeline stage. Receives a stage name as argument. The instruction being executed on the stage is discarded.

`delay(` ***value***, ***cycles***`)` :

```
///!Instruction addcc_reg behavior method.
void ac_behavior( addcc_reg )
{
  dprintf("addcc_reg r%d,r%d,r%d\n", rs1, rs2, rd);
  int dest = readReg(rs1) + readReg(rs2);

  PSR_icc_n = dest >> 31;
  PSR_icc_z = dest == 0;
  PSR_icc_v = (( readReg(rs1) &  readReg(rs2) & ~dest & 0x80000000) |
       (~readReg(rs1) & ~readReg(rs2) &  dest & 0x80000000) );
  PSR_icc_c = ((readReg(rs1) & readReg(rs2) & 0x80000000) |
       (~dest & (readReg(rs1) | readReg(rs2)) & 0x80000000) );

  writeReg(rd, dest);
  dprintf("Result = 0x%x\n", dest);
  update_pc(0,0,0,0,0);
};
```

Figure 3.7: Sparc-V8 Instruction Behavior Description in a Functional Model

Delayed assignment method. The example on Figure 3.9 shows how an assignment to a storage device can be delayed for a given number of cycles. For example, it was used to simulate the branch delay slots for the MIPS architecture, in an older version of the `mips1` ArchC model. The use of `delay()` is deprecated in ArchC 2.0, since it severely slows down simulation.

`get_name ( )` :
Provides the name of the current instruction.

`get_size ( )` :
Provides the size, in bits, of the current instruction.

`get_cycles ( )` :
Provides the latency, in cycles, of the current instruction. Available only for multi-cycle instructions declared through the keyword `set_cycles`, in the `AC_ISA` description.

`ac_annul ( )` :
Annul the execution of the current instruction. May be called inside generic instruction and format behavior methods. It is available only for functional models. When called from generic instruction behaviors this method annuls the execution of the format and specific behavior for the current instruction. Clearly, if it is called inside a format behavior it is able to annul just the execution of the specific behavior.

ArchC has also a set of pre-defined variables that should be used to control some aspects of the simulation. These variables are meant to be used inside processor behavior methods,

```
void ac_behavior( add, stage )
{
   switch(stage) {
   case IF:
      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 3.8: MIPS `add` Instruction Behavior Description in a Cycle-accurate Model

especially `ac_behavior` methods.

`ac_pc` :
: The current program counter value. Must be explicitly updated by the user. See Section 3.1.1 for examples.

`ac_cycle` :
: The current cycle being executed for the running instruction. Must be explicitly updated by the user. Only available when multi-cycle instructions were declared.

`ac_instr_counter` :
: The number of instructions already executed during the current simulation.

```
//!Instruction beq behavior method.
void ac_behavior( beq )
{
  if( RB.read(rs) == RB.read(rt) ){
    ac_pc = delay(ac_pc + (imm<<2), 1);
  }
};
```

Figure 3.9: Delayed Assignment Example: MIPS `beq` Instruction Behavior Description

Chapter 4

ArchC Tools

4.1 The ArchC Preprocessor

The *ArchC Preprocessor* (`acpp`) is the tool compiled as a library that takes an ArchC description, composed by an instruction set architecture description (`AC_ISA`) and an architecture resource description (`AC_ARCH`), and extracts the information used in software tools generation. This preprocessor is not used directly by ArchC designers, but is internally called by ArchC tools like the simulator and assembler generators. All information extracted by `acpp` is stored in data structures in memory, that are available for the tool which invoked it.

`Acpp` is composed by lexical and syntactical analyser (parser), which were built using GNU Flex [9] and GNU Bison [8].

4.2 The ArchC Simulator Generator

The *ArchC Simulator Generator* (acsim) is composed by a simulator generator and a decoder generator. It uses the `acpp` to extract the information from the model description files, in order 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. The process of generating simulators from an ArchC description is illustrated by Figure 4.1.

4.2.1 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 4.2 shows the `acsim` help screen. The user can see this information by running `acsim --help`. It is composed by a description of the syntax used to run `acsim` with/without options, followed by a list, together with a brief description, of all command-line options available

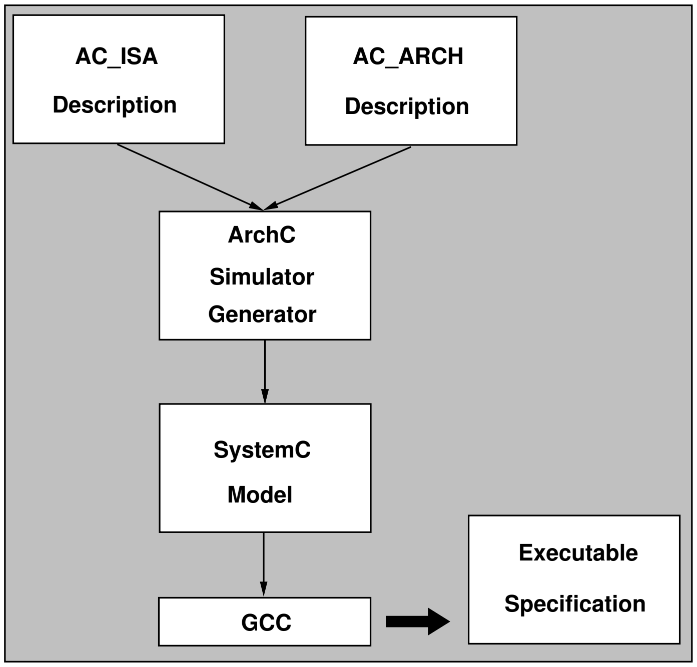

Figure 4.1: ArchC Simulator Generation Flow

in ArchC. All options have a *full-name*, which is easier to memorize, and an equivalent *short-name*, like most options in `GCC` [4]. Options full-names and short-names are preceded by `--` and `-` respectively.

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

`--abi-included, -abi`:
This option tells `acsim` that an *application binary interface* has been implemented to the architecture. In this case, the model generated will be able to perform OS call emulation.

`--debug, -g`:
This option turns on a debugging feature: execution traces. ArchC traces are dumps, into a text file, of all PC values visited by the simulator.

`--delay, -dy`:
Enables delayed assignment to storage elements. This is necessary to model architectures that make use of delay slots. The support for delayed assignment slows down the simulation, so it is optional to not penalty models where it is not necessary.

`--dumpdecoder, -dd`:
When this option is turned on , `acsim` will dump to the screen the data structure

built by the decoder. It is useful to debug the ArchC decoder generator, or to anyone interested in to 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 4.2.

`--no-dec-cache, -ndc`:
Disables the cache of decoded instructions. The simulator generated by ArchC usually makes use of a cache of decoded instructions to speed-up simulation. But this may not be suitable for architectures/applications which write to the program memory, like self-modifying code.

`--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 is output to stdout.

`--verbose, -vb`:
Displays update logs for storage devices during simulation. Used for simulation debugging.

`--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.

`--gdb-integration, -gdb`:
Enable suport for debugging programs using `gdb` remote target. Note that a GDB port has to be configured for the generated simulator in order to wait for remote conection. See Section 4.7 for details.

`--no-wait, -nw`:
Disables calls to SystemC `wait()` in the processor execution thread. Has the potential of speeding up execution but makes multithreaded simulation impossible (ie, multi-core/multiprocessor and system-level platform simulation) since the processor model will own the SystemC execution context all of the time.

4.2.2 ArchC Simulators

The `acsim` will automatically generate a behavioral simulator written in SystemC from your ArchC description files. ArchC generates interpreted simulators, which 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

```
===================================================
 This is the ArchC Simulator Generator version 2.0
===================================================

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

Options:
    --abi-included   , -abi       Indicate that an ABI for system call emulation was provided.
    --debug          , -g         Enable simulation debug features: traces, update logs.
    --delay          , -dy        Enable delayed assignments to storage elements.
    --dumpdecoder    , -dd        Dump the decoder data structure.
    --help           , -h         Display this help message.
    --no-dec-cache   , -ndc       Disable cache of decoded instructions.
    --stats          , -s         Enable statistics collection during simulation.
    --verbose        , -vb        Display update logs for storage devices during simulation.
    --version        , -vrs       Display ACSIM version.
    --gdb-integration, -gdb       Enable support for debbuging programs running on the simulator.
    --no-wait        , -nw        Disable wait() at execution thread.

For more information please visit www.archc.org
```

Figure 4.2: Acsim Command-line Options

instructions in order to speed-up simulation. Similar techniques are applied in some well known ISA simulators [1]. This technique can be disabled by command-line options passed to `acsim`, like showed in Section 4.2.

ArchC SystemC simulators may be used for design-space exploration. 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, etc.

4.2.3 Building ArchC Simulators

`Acsim` automatically generates a *Makefile* for building the SystemC models generated by ArchC. The Makefile is called `Makefile.archc`. The user may alter some variables in this file, in order to customize flags passed to the compiler, or the path for the preferred SystemC installation and compiler. Just as it is done for the SystemC models distributed as examples inside regular SystemC source packages.

The SystemC module that contains the processor model generated by ArchC must be instantiated in a `sc_main` function, another SystemC module etc. `acsim` creates a template of a `main.cpp` file for the designer. In this file, the model is instantiated and some code for setting debugging features is included. Notice that this code will only take effect if the model was generated with the correspondent option turned on using `acsim`. If a `main.cpp` file does not exist, the makefile will copy the `main.cpp.tmpl` file to `main.cpp` and will be ready to build the model. There are users who experiment with ArchC models by connecting them to other SystemC modules, or want to add extra modules to the model generated by ArchC. In these cases they are free either to alter the `main` function to add code to set

```
int sc_main(int ac, char *av[])
{
  //Clock
  sc_clock clk("clk",20,0.5,true);

  //ISA simulator
  sparcv8_arch SPARCV8("sparcv8");

  SPARCV8(clk.signal());

#ifdef AC_DEBUG
  ac_trace("sparcv8.trace");
#endif

  ac_init(SPARCV8);

  sc_start(-1);


#ifdef AC_STATS
  SPARCV8.ac_sim_stats.time = sc_simulation_time();
  SPARCV8.ac_sim_stats.print();
#endif


#ifdef AC_DEBUG
  ac_close_trace();
#endif

  return 0;
}
```

Figure 4.3: Main function for the SPARC-V8 model

up their own extra features or instantiate the processor modules inside their own custom modules and other SystemC code. Figure 4.3 shows a typical main function, generated by `acsim` for our SPARC-V8 model. Names of the module and trace file may be altered at designer's will.

For building a simulator generated with `acsim` run: `make -f Makefile.archc`

This makefile may also receive one of the following three command-line arguments:

clean :
: Erases all binary files for this model.

model_clean :
: Erases all source files that are automatically generated by `acsim`.

sim_clean :
: Erases both source and binary files. Source files that are hand-written by the designer, like the `.ac` description files, are not erased.

distclean :

Erases both source and binary files. Also deletes the `main.cpp` and `Makefile.archc`. Use with care to not delete a modified `main.cpp` file.

If you already have generated a simulator in your current work directory, we strongly recommend the use of a *make -f Makefile.archc sim_clean* command before generating a new simulator through `acsim`. Specially if you are going to use different command-line options to generate the new simulator. Otherwise, you can get some strange compilation errors due to old binaries that were not rebuilt.

4.2.4 Loading and Running Applications

A compiler for the target architecture may or may not be available during model design. In order to cover both situations, ArchC simulators are capable of loading applications using two formats: hexadecimal and binary.

In order to load hexadecimal files into ArchC, the designer must respect a simple format convention:

- The file may be divided into sections like .text, .data, .rawdata etc;
- Application code is stored in the .text section;
- If no section is declared, ArchC will assume that the file contains only code;
- The file only contains lines storing section names or data;
- Data is stored as: <addr> <data1> [<data2> ... <data_n>], where addr is the address where data1 will be stored and the remaining data, in the same line, will be stored into contiguous addresses, incremented by the target architecture's word-size;
- Data is always in hexadecimal.

Figure 4.4 shows a typical hexadecimal file in the ArchC format.

The ELF binary accepted by ArchC also has a format convention:

- ArchC start the simulation at address 0, so the program needs to begin at this address;
- The block of addresses from 0x40 (64) to 0xFF (255) is reserved to the ABI emulation feature. If this feature is active, any instructions in this block will be substituted to internal functions for ABI/OS system call emulation. Please don't use the reserved addresses in your target program;
- The ArchC System Call Library should be linked with the application. This library can be downloaded in the ArchC site following the menu links Models→Compilers;

The simulator loads applications through the command-line argument load:

mips1.x –load=<ArchC hexa or ELF file> [arg1] [arg2] ... [argn]

where argn's represent optional arguments to be passed to the running application. Notice that this is only possible for models with ABI (Section 4.6) already implemented.

```
.text:
0000 3c1d0050 23bdfc00 3c1c0003 0c000020
0010 279cb348 00000000 3c1f0050 03e00008
0020 00000000 00000000 00000000 00000000
0030 00000000 00000000 00000000 00000000
0040 00000000 00000000 00000000 00000000
0050 00000000 00000000 00000000 00000000
0060 00000000 00000000 00000000 00000000
0070 00000000 00000000 00000000 00000000
0080 27bdffe0 24030003 afb00010 afbf0018
0090 afb10014 14830028 00a08021 8ca40004
00a0 0c0001ad 00000000 0c000642 00000000
00b0 8e040008 3c080002 0c005748 25053348
00c0 3c070002 3c040002 24f14990 2485feb8
.data:
22370 6d706567 32656e63 6f646520 56312e32
22380 2c203936 2f30372f 31390000 28432920
22390 31393936 2c204d50 45472053 6f667477
```

Figure 4.4: An ArchC Hexadecimal Application File

4.3 The ArchC Timed Simulator Generator

Starting with version 2.0, ArchC has two separate tools for generating interpreted simulators: `acsim`, covered on section Section 4.2, and `actsim`. The `acsim` tool generates only functional simulators, whereas `actsim` generates only timed (ie, cycle-accurate with single pipeline and multicycle) simulators.

Development of `actsim` isn't yet finished, although a beta version of such tool is included in version 2.0 of the core ArchC distribution. It still lacks some features, particularly TLM conectivity.

The more important aspects of `actsim`, notably usage, command-line options and structure and usage of the generated simulators, follow closely those of `acsim` and were covered on Section 4.2.

4.4 The ArchC Compiled Simulator Generator

The *ArchC Compiled Simulator Generator* (accsim) is composed by a simulator generator and a decoder generator. It uses the `acpp` to extract the information from the model description files, in order to create all C++ classes 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.

The `accsim` tool, however, has not been updated for ArchC 2.0. Users interested in `accsim` must use ArchC 1.6 for now. The `accsim` tool is, however, being revamped and will be included in later versions of the core ArchC distribution.

Group	Function	Host interaction
I/O	open	√
	creat	√
	close	√
	read	√
	write	√
	isatty	√
	lseek	√
	fstat	√
Control	_exit	√
	chmod	
	chown	
	stat	
	getpid	
	kill	
	unlink	
Time	time	
	times	
	gettimeofday	
Memory	sbrk	√

Table 4.1: Supported System Calls

4.5 The ArchC Binary Utilities Generator

Information regarding the ArchC binary utilities generator (`acbingen`) can be found in a separate manual, available at [5]. Since this tool is a major addition to the ArchC language, it is covered on its own manual with detailed information. Moreover, the binary utilities features are not required if one does not plan to generate binary utilities.

4.6 Operating System Call Emulation

ArchC generate simulators capable of being instrumented with an OS call emulation mechanism, which enables ArchC models to simulate applications containing I/O operations. This system enables calls to POSIX-compatible system routines in a simulated application, without requiring any change to the application code. Since file I/O is performed transparently, the input and output data for the application program are read/written directly from/to the host file system, and all console operations are redirected to the host console. Table 4.1 show all system calls supported by ArchC simulators.

Due to incompatibilities between the host and target OS, not all routines need host interaction. For example, a few programs in our benchmarks used time functions. These functions contain structures that are incompatible between different architectures, due to alignment problems in the structure fields (alignment for char and short types). We solved this problem, for now, by returning a zeroed structure, so that programs run correctly, but cannot count time. Another possible approach is to copy the structure field-by-field, correcting them as necessary. But this approach needs more user intervention than the current one.

Designers of new ArchC models have to tell from which storage elements (memory, register bank, etc) the arguments to system calls will come from. The way to do that is by writing interface functions that provide the required information to the ArchC simulator. Typical 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 more than that, how to distinguish the type of the arguments from integer numbers, pointers or strings (note that in some cases an endianness conversion may be required);
- How to save the return value as an integer or pointer (may also need conversion);
- How to return from the system call. Usually, this is just a jump instruction that uses the register which contains the return address;
- How to store strings so the command line arguments can be stored into memory before simulation begins.

All operating system functionality is implemented as a new class in ArchC, named `ac_syscall`, which has virtual methods that needs to be specialized for each new processor to provide the correct ABI implementation for the architecture.

Functions that know how to get parameters from the target system calls are an example of interface functions that must be implemented for each target architecture. 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. As an example, Figure 4.5 shows some functions for the MIPS architecture and how they are encoded using ArchC to access the register file and memory. Notice from that example, that the only information needed was the register numbers used by the architecture as arguments, the return value, and the return instruction.

The methods presented in Figure 4.5 are just specialization of some methods of the base ArchC system call class, called `ac_syscall`, which has 265 source code lines and do not require any change between architectures. The MIPS I system call methods required only 51 code lines to implement the specialized methods. The SPARC V8 also required the same number of extra lines since the difference between the MIPS-specific and SPARC-specific files are only the register numbers. The small number of lines in the specific files show that it is very easy to port this method to new architectures.

If designers are going to provide the ABI implementation, they need to run `acsim` with the `-abi` command-line option to generate a simulator prepared to do OS emulation. For examples of ABI implementation to be used with ArchC simulators, take a look at the `mips1_syscall.cpp` and `sparcv8_syscall.cpp` files contained in the respective model package at the ArchC website [5].

```
void mips1_syscall::get_buffer(int argn,
                               unsigned char* buf,
                               unsigned int size)
{
  unsigned int addr = RB.read(4+argn);

  for (unsigned int i = 0; i<size; i++, addr++) {
    buf[i] = MEM.read_byte(addr);
  }
}

int mips1_syscall::get_int(int argn)
{
   return RB.read(4+argn);
}

void mips1_syscall::set_int(int argn, int val)
{
   RB.write(2+argn, val);
}

void mips1_syscall::return_from_syscall()
{
  ac_pc = RB.read(31);
}
```

Figure 4.5: Application Binary Interface Functions for the MIPS Architecture

4.7 GDB Support for Simulators

Simulators built with ArchC can use the GDB protocol very easyly. Just implement a few processor dependent methods for the interface and the simulator will be able to respond to GDB, allowing users to debug software inside the simulator. This is a new functionality and is implemented for simulators generated for **functional** models, it is **not implemented for cycle-accurate simulators yet**. It cannot be used for simulators generated by `accsim` at this time. The SPARC-V8 and MIPS-I models available at the ArchC Site [5] already counts with the additional information to activate this functionality.

The developer should implement methods to interface the architecture and GDB, mapping registers and memory to the desired GDB format. These methods, explained in detail below, should be implemented in a file named after your processor: **PROC_gdb_funcs.cpp**, given your processor is implemented in **PROC.ac**. Then "`make distclean`" your simulator, run `acsim` with `-gdb`, and make it again. Acsim with `-gdb` will make your simulator class inherit from `AC_GDB_Interface` and use the functions defined in **PROC_gdb_funcs.cpp**.

When built, the simulator module will have the `enable_gdb()` method. It can be called, after the module instantiation, to make the particular simulator module enable its GDB support and listen to GDB connections in the port specified as the method's first argument.

If a port number is not specified, the default value is used, 5000.

4.7.1 Register Support

The designer must implement `nRegs()`, `reg_read()` and `reg_write()` from `AC_GDB_Interface`, so the simulator can send the read and write register packets to GDB. You must check GDB documentation to learn the order the registers should be provided, and map them in `reg_read()` and `reg_write()`. Then define the number of registers GDB expects to receive/send by using `nRegs()`. The order is defined by the `REGISTER_RAW_SIZE` and `REGISTER_NAME` macros. Please read the "`info gdb`", section "`Remote Protocol`", nodes "`Packets`" and "`Register Packet Format`" for more information.

Example: If you have one bank with general purpose registers (`RB_GP`), one with floating point registers (`RB_FP`) and one with status register (`RB_S`). If GDB expects the packets in the order: 32 general purpose registers, 32 floating point registers and 8 status register and PC, your functions should be like those in Figure 4.7.

4.7.2 Memory Support

The designer must implement `mem_read()` and `mem_write()` from `AC_GDB_Interface` to inform how to read and write memory regions. Example: If you use just one memory bank (no separated data and instruction memory), it's easy as shown in Figure 4.6:

```
unsigned char YOUR_CLASS::mem_read( unsigned int address ) {
  return ac_resources::IM->read_byte( address );
}

void YOUR_CLASS::mem_write( unsigned int address,
                            unsigned char byte) {
  ac_resources::IM->write_byte( address, byte );
}
```

Figure 4.6: Memory manipulation routines for GDB support

```
int YOUR_CLASS::nRegs( void ) {
  return 73;
}

ac_word YOUR_CLASS::reg_read( int reg ) {
  // General Purpose
  if ( ( reg >= 0 ) && ( reg < 32 ) )
    return RB_GP.read( reg );

  // Floating Point
  else if ( ( reg >= 32 ) && ( reg < 64 ) )
    return RB_FP.read( reg - 32 );

  // Status
  else if ( ( reg >= 64 ) && ( reg < 72 ) )
    return RB_SR.read( reg - 64 );

  // Program Counter
  else if ( reg == 72 )
    return ac_resources::ac_pc;

  return 0; // unmaped register? return 0
}

void YOUR_CLASS::reg_write( int reg, ac_word value ) {
  // General Purpose
  if ( ( reg >= 0 ) && ( reg < 32 ) )
    RB_GP.write( reg, value );

  // Floating Point
  else if ( ( reg >= 32 ) && ( reg < 64 ) )
    RB_FP.write( reg - 32, value );

  // Status
  else if ( ( reg >= 64 ) && ( reg < 72 ) )
    RB_SR.write( reg - 64, value );

  // Program Counter
  else if ( reg == 72 )
    ac_resources::ac_pc = value;
}
```

Figure 4.7: Register manipulation routines for GDB support

Chapter 5

ArchC TLM Connectivity

This appendix offers a brief overview on ArchC TLM and follows it with a detailed, step by step tutorial on how to make full use of the ArchC TLM capabilities.

5.1 Introduction

Transaction-Level Modeling (TLM) is a series of hardware project techniques and methodologies which aim to raise as much as possible the abstraction level of the descriptions and models used to develop, prototype and simulate both hardware and hardware/software systems.

TLM achieves that by creating several standardized abstraction levels between pure specifications and register transfer level (RTL) models. The fact that TLM specifies discrete abstraction levels for computation and communication, means either of them can be refined independently, considering a system-level model. Also, the fact abstraction levels are standardized means that components (which can be models in several different of those abstraction levels) can be shared and reused among projects.

OSCI offers a TLM library for SystemC. This header-only library defines a series of standardized interfaces that provide the necessary separation between computation and communication modeling, therefore enabling TLM in SystemC.

ArchC 2, among other benefits, has TLM support. This means simulators generated by the `acsim` tool are independent SystemC modules and can communicate with other user-created SystemC modules via SystemC TLM interfaces using the ArchC protocol.

For more on TLM and the SystemC TLM library, refer to the OSCI SystemC TLM white paper. It is distributed with the SystemC TLM 1.0 library, and is required read for the rest of this appendix.

5.2 ArchC TLM Basics

This section describes briefly the interfaces used for ArchC TLM and the ArchC TLM protocol, which are of great importance to the user.

5.2.1 ArchC TLM Interfaces

ArchC TLM was developed so that designers can integrate `acsim`-generated simulator modules to their PV (Programmer's View) system-level SystemC models. Having PV models in mind, the interface chosen for ArchC TLM was the `tlm_transport_if`, since it's bidirectional and has requests and responses tightly coupled, exactly like in a method call, therefore requiring the least effort of users.

5.2.2 ArchC TLM Protocol

The ArchC TLM protocol is defined in `ac_tlm_protocol.H`. It consists in a pair of structs modelling the request and response packets. Users are strongly encouraged to read this source file. It really is that simple.

The request packet, `ac_tlm_req`, has four fields:

`type` : Type of the transaction. It can be one of five values: `READ`, `WRITE`, `LOCK` (for locking a device or bus), `UNLOCK` and `REQUEST_COUNT` (for debugging reasons, requests a count of the transaction packets).

`dev_id` : Device ID. Every ArchC TLM initiator port gets automatically assigned a device ID, which might be useful information for multiport devices like buses.

`addr` : Address.

`data` : Data sent.

The response packet, `ac_tlm_rsp`, on the other hand, has only three fields:

`status` : Transaction status. Can be either `ERROR` or `SUCCESS`, indicating whether the transaction was successful or not.

`req_type` : Type of the transaction.

`data` : Data received.

5.3 ArchC TLM How-to

This section contains a very detailed, step-by-step explanation on how to use ArchC TLM ports and communication.

5.3.1 TLM Initiator Port

ArchC 2.0 provides a TLM initiator port to your processor, allowing it to communicate with external modules. In order to create a TLM initiator port, you have to declare it on the *project_name*`.ac` file. Here's an example made with the `mips1` model, replacing the DM element from `ac_cache` to `ac_tlm_port`:

```
AC_ARCH(mips1){
  //ac_mem    DM:5M;
  ac_tlm_port DM:5M;
  ac_regbank  RB:32;
//...
```

We substituted the `ac_mem` (main memory) with an `ac_tlm_port` because we want to use this model with an external TLM memory. Now all you have to do is run `acsim` and it will generate the simulator code with the `ac_tlm_port` already in place. Please note that a range must be declared with the `ac_tlm_port`, indicating the range of addresses (starting on 0x0) that will be accessible via the port (in this case, 5 megabytes).

One important thing about ArchC 2.0 is that the processor access to either `ac_tlm_ports` or internal memories (elements of type `ac_storage`) is always made via an `ac_memport`. In this case, the `ac_memport` correspondent to the `ac_tlm_port` we've declared will be called `DM`, and the port itself will be called `DM_port`. That is, the `actlmport` gets a '`_port`' suffix in its name.

Here's a brief illustration of how this works: whenever you want to access the port from the inside of the processor, that is, on instruction behaviors, for instance, you use the `ac_memport` called `DM`, like this (in the *project_name*`_isa.cpp` file):

```
value = DM.read(address);
```

However, all access to this port from outside the processor will have to use the `ac_tlm_port` object, which in this case is called `DM_port`. The main use of this will be in binding the port to an external module, like this:

```
proc_instance.DM_port(ext_memory);
```

Which is a binding of the port present in `proc_instance` to an external memory `ext_memory`.

After that, the most important to know is what to implement in the external modules so that communication is possible. And that is known as the ArchC TLM protocol. It works like this: an `ac_tlm_port` *is-a* (ie, extends) `sc_port<ac_tlm_transport_if>`, where `ac_tlm_transport_if` is the same as `tlm_transport_if<ac_tlm_req,ac_tlm_rsp>`.

Basically this means that you can only bind an `ac_tlm_port` to a module that extends `ac_tlm_transport_if` (which would be, technically, a channel) or to a `sc_export<ac_tlm_transport_if>` that is bound to such an object.

So the first requisite is that your external module responsible for communicating directly with the processor must inherit from `ac_tlm_transport_if`.

That obliges this module `ext_module` to implement the:

```
ac_tlm_rsp ext_module::transport(const ac_tlm_req& req);
```

method that is declared by `tlm_transport_if<>`. This method will receive a reference to an ArchC TLM transaction request packet (declared at the `ac_tlm_protocol.H` file), process it accordingly, and return an ArchC TLM transaction response packet (also declared at the `ac_tlm_protocol.H`) informing the transaction requester (that is, our processor module) of the transaction status, whether it failed or succeeded.

By doing this on the external slave module, everything is ready to work. Just bind the port from the processor instance to the module or module export and you can already access it via the read/write methods of the `ac_memport DM`, inside the instruction behaviors.

5.3.2 TLM Interrupt Port

ArchC 2.0 also features communication initiated from an external module, with the processor module as a slave, via its interrupt mechanism. To enable an interrupt mechanism on your model, first you have to declare an `ac_tlm_intr_port` on the *project_name*`.ac` file. As an example, you can add an interrupt port to our `mips1` model just by doing this:

```
AC_ARCH(mips1){

  ac_cache   DM:5M;
  ac_regbank RB:32;

  ac_tlm_intr_port inta; // Add this line
//...
```

Now we have an interrupt port on our `mips1` model, called `inta`. By running the `acsim` tool on it, we'll see that it generates some extra files related to interrupts. The one we'll modify is *project_name*`_intr_handlers.cpp`. First, we'll copy *project_name*`_intr_handlers.cpp.tmpl` to *project_name*`_intr_handlers.cpp`. Then we'll edit the file.

When you open the file, you'll see that it contains an `ac_behavior` definition, pretty much in the same way as the *project_name*`_isa.cpp` file:

```
// Interrupt handler behavior for interrupt port inta.
void ac_behavior(inta, value) {
}
```

Inside this `ac_behavior`, you can write pretty much standard behavior code, altering register values, the program counter, writing or reading the memory etc. Basically, this behavior code represents whatever the processor hardware does when it encounters an interrupt. Note that it receives a value from the interrupt, which can be used on the interrupt handling, and it's your choice whether to use it or not. Another possibility is declaring more than one `ac_tlm_intr_ports` on the *project_name*`.ac` file, which will lead to each of them having its separate interrupt handler method defined on this *project_name*`_intr_handlers.cpp` file. So there are lots of options to implement interrupt behavior on your model. The ArchC interrupt system is very flexible because we don't want to hinder the developers' options.

Now that we know how interrupts work inside the processor, what we have to understand is how an external module interrupts the processor. First, whenever you declare an `ac_tlm_intr_port` on your *project_name*`.ac` file it means that you will have an `ac_tlm_intr_port` as a member of your processor module.

This `ac_tlm_intr_port` will have the exact same name you declare on `project_name.ac`, which in this case is `inta`. The `ac_tlm_intr_port` *is-a* (inherits from) `sc_export<ac_tlm_transport_if>`, and this export is bound to itself, because `ac_tlm_intr_port` also implements the `transport()` method mentioned above, and this method has a fixed implementation, which calls an interrupt handler that is bound to a port (those are the ones you implement on the *project_name*`_intr_handlers.cpp`, one for each port).

Then, to give an external module the ability to interrupt a processor, all you have to do is binding an `sc_port<ac_tlm_transport_if>` of this module to the `ac_tlm_intr_port` of the processor, like this:

```
ext_module.out_port(proc_module.inta);
```

and then have the external module call the `transport()` method via this port, passing an `ac_tlm_req` transaction request packet. Whatever value there is on the data field of the packet will be passed as the 'value' parameter of the interrupt handler. Ideally, direct usage of the `transport()` function should be delegated to a user-layer method (with an appropriate name like `interrupt()`), and this user-layer method the one invoked by the module's behavior or process methods.

Appendix A

Porting a model from ArchC 1.6 to 2.0

This appendix presents nine steps on how to port your ArchC model from version 1.6 to 2.0, presented in a straightforward yet thorough manner:

1. Change *project_name*`-isa.cpp` to *project_name*`_isa.cpp`.

2. Edit *project_name*`_isa.cpp` so that it includes these headers:

   ```
   #include "project_name_isa.H"
   #include "project_name_isa_init.cpp"
   #include "project_name_bhv_macros.H"
   ```

3. Add:

   ```
   using namespace modelname_parms;
   ```

   to your *project_name*`_isa.cpp`, *project_name*`_gdb_funcs.cpp` and *project_name*`_syscall.cpp` files. This is necessary because all the processor-specific types like `ac_word` and `ac_Hword` etc are now contained on a namespace relating them to the processor model, avoiding type name clashes in projects including more than one different processor models.

4. In your *project_name*`_isa.cpp`, *project_name*`_gdb_funcs.cpp` and *project_name*`_syscall.cpp` files, there's a chance that some architectural elements from the model, like a data memory DM, are being accessed like this:

   ```
   ac_resources::DM.something();
   ```

If this is the case, you can safely remove every occurrence of `ac_resources::` from your files because ArchC 2.0 was constructed so that every implementation of instruction behaviors, GDB functions or syscall helper functions have direct visibility of all architectural elements. So the correct version of the DM example is much cleaner:

```
DM.something();
```

5. Your *project_name*_`isa.cpp`, *project_name*_`gdb_funcs.cpp` and *project_name*_`syscall.cpp` files might be including `"ac_resources.H"`. This will cause a compilation error, because `acsim` no longer generates an `ac_resources.H` file nor an `ac_resources` class. The architectural elements now appear as non-static members of the *project_name*_`arch` class), so you must remove the:

   ```
   #include "ac_resources.H"
   ```

   line from all those files.

6. In your *project_name*_`isa.cpp`, *project_name*_`gdb_funcs.cpp` and *project_name*_`syscall.cpp` files, replace every call to the `ac_stop()` function to `stop()`. There's no need to change the parameters, it was a simple function renaming.

7. If your *project_name*_`isa.cpp` file has global functions, you might have to modify them. Whatever element you want to access inside of them, that wasn't passed as a parameter, will be out of scope because architectural elements are not global anymore. The easiest way of solving the problem of access to the architectural elements is passing them by reference as a function parameter, and then changing all the calls to the function to pass those extra parameters you just added (which is easily done via the search-and-replace feature of your favourite editor). Let's take a look at how this was done on the SPARC V8 model:

   We changed:

   ```
   inline void update_pc(bool branch, bool taken, bool b_always, bool annul,
                         ac_word addr)
   ```

   To add an extra `ac_pc` reference parameter:

   ```
   inline void update_pc(bool branch, bool taken, bool b_always, bool annul,
                         ac_word addr, ac_reg<unsigned>& ac_pc)
   ```

   And every occurrence of:

   ```
   update_pc(a, b, c, d, e)
   ```

To pass the extra `ac_pc` parameter by reference:

```
update_pc(a, b, c, d, e, ac_pc)
```

To verify of what type those elements are (like, in this case, `ac_pc`, which is of type `ac_reg<unsigned>`), take a look at `ac_arch.H` (an ArchC library include file) or *project_name*`_arch.H` (generated by acsim), which contain the declarations for the elements you want (the elements common to all processor models are members of the `ac_arch` class, whether the specific ones, such as most of those you declared on the AC file, are members of *project_name*`_arch`).

If you're writing a new model from scratch, please remember that using global functions with side-effects in methods of a class which will be instantiated more than once is usually indicative of not-so-great programming. And it's the fact why we're supplying this step 7, which was entirely avoided in the porting of models that didn't use such global functions (like `mips1`, for example).

8. In your *project_name*`_isa.cpp`, *project_name*`_gdb_funcs.cpp` and *project_name*`_syscall.cpp` files, there's a chance some of the instruction behavior or helper methods declared on them might use variables or constants as `extern`. If those are either architectural resources or constants from the parms file (which comprehend all of the cases we've seen so far), you're absolutely free to remove the extern line completely, because such variables or constants are already visible to those methods.

   Here's one example from our SPARC V8 model. In the file `sparcv8_syscall.cpp`, we had the line:

   ```
   extern const unsigned AC_RAM_END;
   ```

   Inside the `sparcv8_syscall::set_prog_args()` method. We removed this `extern` line completely because `AC_RAM_END` is already visible to the method, so there's no need for `extern` (which indicates cleaner coding, since using lots of `extern`s is quite inelegant).

9. There's a chance your model has global variables, most probably defined on *project_name*`_isa.cpp` (like in our SPARC V8 model: they're defined on `sparcv8_isa.cpp`) or on one of the other `cpp` files. Now, global variables present two serious problems. The first of them is that they're usually bad programming practice. They shouldn't be used when they aren't necessary. The second problem is much more serious: they make it completely impossible to run more than a single instance of your processor model.

   Global variables alone won't keep your model from compiling, but since they're global, there will always be only one instance of each of them, which will be a major problem if you want to use multiple instances of the simulator generated by acsim in, for instance, an multiprocessor SoC model.

The solution we suggest is exactly the same we used in porting our SPARC V8 model: converting all those global variables to registers. As an example, in the `sparcv8_isa.cpp` file we had those:

```
unsigned char CWP = 0xF0;     //Current window pointer
unsigned char WIM = 0x00;     //Window invalid bit (points to the
                              //invalid reg window)
```

We removed those declarations from that file and added register declarations to `sparcv8.ac`, like this:

```
ac_reg<8> CWP;
ac_reg<8> WIM;
```

With this modification, no other line of the `sparcv8_isa.cpp` had to be modified, because those registers can be accessed or used in the exact same way the unsigned char variables were. Also, notice that `ac_regs` now can have their width specified inside angle brackets (which were previously used only to specify the format, when you wanted to use a formated register). This is a new feature of ArchC 2.0, because in 1.6.x every register you declared had the same width specified by `ac_wordsize`, meaning you couldn't declare a register with width different from the word size. But now you can, in 2.0.

On a side note, if you had global variables of type `bool`, for flags or something like that, and you want your `ac_regs` to behave exactly like the global variables they're substituting, you can do that, declaring them as `ac_reg<1>`.

Of course, the removal of global variables, converting them to registers, leads to the need of three other collateral modifications.

The first of them is removing the stray `extern`s: there's a chance that variables declared on one cpp file are being accessed on other file, so you'll have to remove the `extern` statements from those other files.

Another modification is needed when those global variables were initialized with a value different from zero. Such is the case of the SPARC V8 CWP register shown above. Since every `ac_reg` is initialized to zero at construction time, if you want them to have another value when the processor starts running, you should put such initialization at the **begin** `ac_behavior`. This is what we've done with the SPARC V8 model:

```
void ac_behavior(begin)
```

```
{
  dbg_printf("@@@ begin behavior @@@\n");
  REGS[0] = 0;  //writeReg can't initialize register 0
  npc = ac_pc + 4;

  CWP = 0xF0; // Added this line.
}
```

The third and perhaps more important modification is converting whatever global function that modified global variables as their side-effect to receive those variables (which have now became registers) as reference parameters. The procedure is exactly the same as detailed on step 7, and it's needed for pretty much the same reasons: what once were global variables now are fields of an object, so you need their reference to modify them.

For exemple, our SPARC V8 had an `npc` global variable that we modified to become a register (which is much saner), and since the global function `update_pc()` modified `npc`, we had to pass it as an extra reference parameter. So we essentially changed:

```
inline void update_pc(bool branch, bool taken, bool b_always, bool annul,
                      ac_word addr, ac_reg<unsigned>& ac_pc)
```

To include `npc` as one of the parameters:

```
inline void update_pc(bool branch, bool taken, bool b_always, bool annul,
                      ac_word addr, ac_reg<unsigned>& ac_pc,
                      ac_reg<ac_word>& npc)
```

Following this, we had to do a search-and-replace in the rest of the code to change every occurrence of:

```
update_pc(a, b, c, d, e, ac_pc)
```

to pass the extra npc parameter:

```
update_pc(a, b, c, d, e, ac_pc, npc)
```

If you've made all the modifications detailed in these 9 steps, then congratulations, you should have an ArchC 2.0 compliant model. Of course, some models are far easier to port to 2.0 than others. The `mips1` model, for instance, was incredibly easy to port, requiring only 15 minutes at most, and being guided only by the compiler error messages, instead of this guide. It didn't require the steps 7 to 9, which would definitely make the process trickier. Another model, `sparcv8`, on the other hand, required all of those steps, but even so was fast to port.

However, if you have thoroughly made all those modifications to your model and it still doesn't work in 2.0 (but it did in 1.6), feel free to contact us in the forums. If you realize a more complicated compatibility problem between 1.6 and 2.0 is missing from this walkthrough, please tell us so, as we'll be delighted to know and therefore use the good news to update this document.

Bibliography

[1] 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.

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

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

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

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

[6] http://www.cygwin.com. The Cygwin Environment Website.

[7] http://www.gnu.org/software/binutils. The GNU Binutils Website.

[8] http://www.gnu.org/software/bison/. The Bison Parser Generator.

[9] http://www.gnu.org/software/flex/. The Flex Lexical Analyser Generator.

[10] http://www.systemc.org. SystemC homepage.

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

[12] OSCI. *SystemC Version 2.0 Language Reference Manual*, 2003.

[13] Richard P. Paul. *SPARC Architecture, Assembly Language Programing, and C*. Prentice Hall, 2000.

[14] Roland H. Pesch and Jeffrey M. Osier. *The GNU binary utilities*. Free Software Foundation, Inc., May 1993. version 2.15.

[15] Richard Stallman. *Using the GNU compiler collection*. Free Software Foundation, Inc., May 2004. For GCC version 3.4.3.