

Chapter 2

Open-Source Languages

Sandro Rigo, Luiz Santos, Rodolfo Azevedo, and Guido Araujo

As stated in Chap. 1, the main goal of this book is to enable ESL research based on an open-source infrastructure. In order to make it possible to readers that are not familiar with SystemC and/or ArchC to follow the model and platform descriptions presented in the remaining chapters of this book, this chapter briefly reviews the main concepts related to these two open-source languages. Moreover, ArchC will be put in practical use in Chap. 4. Nevertheless, we strongly encourage such readers to refer to the literature referenced in the following sections for a more complete specification of both languages.

2.1 Basic SystemC Concepts

Since the launching of the Open SystemC Initiative (OSCI) in 1999, SystemC has evolved into one of the most important languages for ESL design. As a consequence, the SystemC literature comprises not only in-depth explanations on language constructs [5, 8], but also extensive overviews on how to put it to practical use [2–4, 7].

S. Rigo (✉) · R. Azevedo · G. Araujo

Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil
e-mail: sandro@ic.unicamp.br

R. Azevedo

e-mail: rodolfo@ic.unicamp.br

G. Araujo

e-mail: guido@ic.unicamp.br

L. Santos

System Design Automation Laboratory, Computer Science Department, Federal University of Santa Catarina, Florianópolis, Brazil
e-mail: santos@inf.ufsc.br

That is why this book focuses instead on how to generate and integrate processor models into SystemC platform descriptions. However, to pave the way towards platform descriptions, a few SystemC concepts and constructs will be reviewed by means of an illustrative example: a minimalist system consisting of a PowerPC processor tied to a generic read-write memory.

Figure 2.1 shows the description of such a system. The function `sc_main` is the starting point (its arguments have the same meaning as those in the C++ main function).

The actual description lies between Line 6 and Line 21. In Line 6 an instance `PPC` of a PowerPC processor of type `ppc` is declared. Line 8 describes an instance `MEM` of a memory of type `mem`. The name of an instance is passed as a parameter to the instance's constructor for inner storage (so that it could be used for debugging purposes).

Line 11 describes the connection between the processor's data memory port (`DM_port`) and the memory's (`target_export`).

Being essentially the source code for an executable representation, a SystemC description contains not only the actual instantiation of components and their connections, but also simulation control and post-simulation diagnosis.

That's why the code inside the `sc_main` function executes in three phases. In Phase 1, which is called elaboration, data structures to hold the described components are created, initialized, and their connections are built. Since descriptions are typically hierarchical, creation and initialization are propagated through the sub-components.

As it will be illustrated in a while, the functionality of each component is captured by means of one or several processes, which model concurrently executing behaviors. During the elaboration phase, all the processes attached to the described components are registered.

In Phase 2, the function `sc_start` (Line 15) launches the simulation of the described (sub)system by invoking the attached behaviors. Upon its return, `sc_start` marks the beginning of a new phase.

Phase 3 handles the outcome of simulation so that diagnosis reports can be issued. For instance, in Line 21 the inner state of processor `PPC` is printed (by invoking the method `PrintStat()` in Line 19). This phase ends by returning a status: zero meaning success; non-zero, failure. This example assumes that the processor has an attribute `exit_status`, which is returned.

To describe the functionality of a system's component, SystemC employs the notion of **module**, which is essentially an encapsulation of the component's state and behavior. Therefore, a system is described as a set of interconnected modules. To grant modules concurrent behaviors, SystemC relies on the notion of **process**, which is a special class method of a module that is registered in SystemC's underlying simulation kernel. Processes are invoked by SystemC's event-triggered scheduler, thereby appearing as concurrent behaviors. Although SystemC has three basic process types, this chapter reviews only the notion of **thread**, since it is the predominant process type used throughout this book. Once initiated by the simulation kernel scheduler, a thread executes until it terminates, although it can be suspended

```

1  int sc_main(int ac , char *av [])
2  {
3      // Phase 1: ELABORATION
4
5      // An instance of a PowerPC processor
6      ppc PPC("PPC");
7      // An instance of a read-write memory
8      mem MEM("MEM");
9
10     // Connection between PPC's and MEM's data ports
11     PPC.DM_port(MEM.target_export);
12
13     // Phase 2: SIMULATION
14
15     sc_start();
16
17     // PHASE 3: DIAGNOSIS
18
19     PPC.PrintStat();
20
21     return PPC.exit_status;
22 }

```

Fig. 2.1 SystemC top-description of a minimalist system

and resumed during its execution. For a given simulation, a thread is executed only once.

To illustrate those basic notions and review a few more concepts, let us show a sketch of a PowerPC description, which consists of two files, as depicted in Fig. 2.2 and Fig. 2.3.

Figure 2.2 describes the module encapsulating PowerPC's functionality. Observe the header files being included (Line 1 to Line 5) and note that the description assumes that the processor's architectural resources and instruction set were described elsewhere (`ppc_arch.H` and `ppc_isa.H`, respectively).

Although SystemC provides a macro (`SC_MODULE`) to ease module description, the example shows an alternative style (Line 7) where the module inherits from two pre-defined classes: `ac_module` (which itself implicitly inherits from SystemC's `sc_module` class) and `ppc_arch` (where architectural resources such as register file, endian, and wordsize are described).

Note that two main attributes of the processor are shown: its instruction pointer `pc` (Line 11) and its instruction-set architecture `ISA` (Line 13), which encapsulates the description of binary instruction formats and fields.

The class method in Line 16, represents the processor's global functionality (whose implementation will be shown later in terms of individual instruction behaviors). This method is turned into a process by a combination of SystemC constructs. First, the module `ppc` is registered as owner of a process (Line 18), then its method `behavior` is registered as a thread (Line 23).

```

1  #include "systemc.h"
2  #include "ac_module.H"
3  ...
4  #include "ppc_arch.H"
5  #include "ppc_isa.H"
6
7  class ppc: public ac_module, public ppc_arch {
8
9  public:
10
11     unsigned pc;
12
13     ppc_parms::ppc_isa isa;
14
15     /// Behavior execution method.
16     void behavior();
17
18     SC_HAS_PROCESS(ppc);
19
20     /// Constructor.
21     ppc(sc_module_name nm): ac_module(nm), ppc_arch(), isa(*
        this) {
22
23         SC_THREAD(behavior);
24         pc = 0;
25     }
26
27     virtual void PrintStat();
28
29     void load(char* program);
30
31     virtual ~ppc() {};
32 };

```

Fig. 2.2 A sketch of the processor's description (ppc.h)

Thread registration is an essential part of a module's constructor, whose scope is illustrated between Line 21 and Line 25. Note that the processor's module inherits from predefined constructors (Line 21), registers its thread and initializes its `pc`. The corresponding destructor is shown in Line 31.

Two other class methods are shown in Line 27 and Line 29. The first prints statistics for post-simulation diagnosis purposes and the second loads in memory the program to be executed by the processor. Notice that, as opposed to `behavior`, those auxiliary methods are not registered as processes. They are just ordinary C++ methods that do not represent concurrent behavior.

Let us now focus on Fig. 2.3, which describes the overall processor's behavior as a loop (between Line 7 and Line 24) where instructions are endlessly decoded and

```

1  #include  "ppc.H"
2
3  void ppc::behavior()
4  {
5      unsigned ins_id;
6
7      for (;;)
8      {
9          // Instruction decoding
10
11         ins_id = decode(pc);
12         switch (ins_id)
13         {
14             ...
15
16             case 10: // Decoding outcome for addi
17
18                 // Instruction execution
19
20                 ISA.behavior_addi( //Instruction fields ... );
21                 break;
22         }
23         break;
24     }
25 }

```

Fig. 2.3 A sketch of the processor’s description (ppc.cpp)

executed. Each instruction is assigned an identifier (declared in Line 5) to distinguish among instruction behaviors. For a given `pc` value, the respective instruction is decoded and its identifier is assigned (Line 11). As a result, one among the various instruction behaviors is selected (between Line 12 and Line 22). For simplicity, all behaviors are omitted, except the one corresponding to the `addi` instruction (Line 16), namely the method `behavior_addi` from the processor’s instruction-set architecture (Line 20).

Although the most important concepts required throughout this book were briefly summarized in this example, the reader should refer to SystemC manuals [5, 8] and specialized books [2–4, 7] for an in-depth study of SystemC concepts, syntax, and usage.

For the example in Fig. 2.1, we assumed that a SystemC model of a processor (PPC) was available, as sketched in Fig. 2.2 and Fig. 2.3. Since such a model is too complex to be written directly in SystemC, it is instead automatically generated from the ISA description of the processor. The next section introduces an appropriate language for the generation of SystemC processor models.

2.2 Introduction to ArchC

The rapidly increasing complexity of modern system architectures raised difficulties that end up delaying the whole design process and preventing designers from meeting their stringent time-to-market. Such difficulties have forced hardware architects and software engineers to reconsider how designs are specified, partitioned, and verified. As a consequence, designers are starting to move from *Register-Transfer Level* (RTL) design towards the so-called *Electronic System Level* (ESL) design and beyond the abstraction limitations of *Hardware Description Languages* (e.g. VHDL, Verilog) to ESL languages, which offer more levels and styles of abstraction (e.g. SystemC) and address not only the hardware view, but also its software counterpart. Therefore, when moving to ESL abstractions, designers have to face several new issues.

Among them, two issues are directly related to processor modeling at the ESL. The first is how to produce code for the *different* target processors under exploration. This issue is tackled through the automatic generation of a software toolkit (assembler, linker, compiler) for every distinct target processor. Such a generation tool is commonly based on an *Architecture Description Language* (ADL) [6]. The second issue is how to produce cooperative executable models for the *multiple* processors of a given platform. Although the generation of an *Instruction-Set Simulator* (ISS) has long been one of the goals of an ADL, the integration of multiple cooperating ISSs withing a platform is a recent aspect of ADL usage. This section introduces a SystemC-based architecture description language called ArchC, which addresses the above-mentioned issues.

Besides, their application and well-known suitability for designing and experimenting with new architectures in the industry, ADLs can be very useful for academic purposes, like teaching/researching computer architecture. On the one hand, at the undergraduate level, models of well-known architectures are the most appropriate to learn how a pipelined architecture works (e.g. 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, update policy, associativity, etc. On the other hand, at the graduate level, researchers can use ADLs to model modern architectures and experiment with their *Instruction-Set Architectures* (ISAs) and internal organizations with all the flexibility demanded in research projects.

ArchC [1, 10] is a simple language that follows a 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 or legacy) processor's architecture by automatically generating not only software tools for code generation and inspection (like assemblers, linkers, and debuggers), but also executable processor models (integrated ISSs) for platform representation, such as the PPC model invoked in Fig. 2.1 (Line 6).

Figure 2.4 illustrates a simplified, ADL-based design exploration flow. From the ADL description of a target processor (and a few auxiliary files) the tool generator

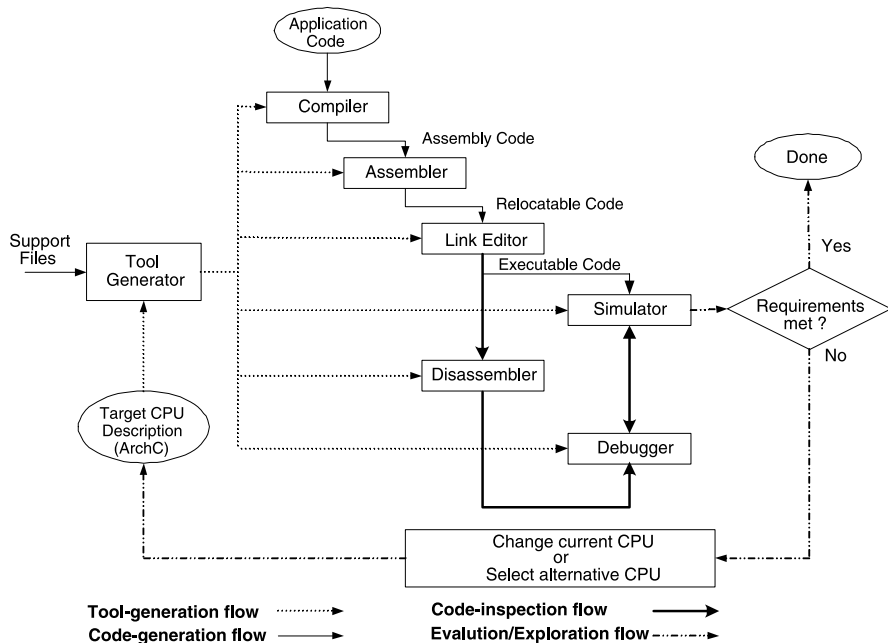


Fig. 2.4 ADL-based exploration flow

synthesizes a compiler's backend, an ISS, and a set of binary utilities. The application's source code is compiled, assembled and linked, resulting in an executable code that is run on a processor model (ISS). After code inspection, which may require disassembly and debugging, the outcome of the simulation is evaluated in face of the requirements and design criteria. If a requirement or a criterion is not met, another candidate processor may be selected and the process is repeated until all constraints and criteria are satisfied. (Since Fig. 2.4 depicts a complete, ADL-independent flow, a given ADL may not support the generation of the whole toolkit; e.g. the automatic generation of compilers is not yet available for ArchC.)

The flexibility provided by such design exploration capabilities is mandatory in an ESL design environment, where multiple processors have to be considered in the composition of complex virtual platforms. For example, to quickly produce and evaluate different platform instances based upon distinct processors, all that is required is the ADL description of each candidate processor. Therefore, the decision on which processors are best suited to a given application is more accurate when a broader set of alternatives is explored through efficient ADL-based automatic generation.

In ArchC, an architecture is represented by two separate descriptions: *Instruction Set Architecture* (AC_ISA) and *Architecture Resources* (AC_ARCH). Within the AC_ISA description, the designer provides information on instruction formats (lengths and fields), on instruction decoding and assembly (binary and symbolic encodings), along with the behavior of each instruction. In the AC_ARCH description,

```

1  AC_ARCH(powerpc) {
2      ac_wordsize 32;
3      ac_mem MEM:8M;
4      ac_regbank GPR:32;
5      ac_reg MSR;
6      ARCH_CTOR(powerpc) {
7          ac_isa ("powerpc_isa.ac");
8          set_endian ("big");
9      };
10 };

```

Fig. 2.5 Excerpt of the PowerPC AC_ARCH Description

the designer declares how the processor is organized in terms of storage devices, pipeline structure, endian etc. Based on these two descriptions, ArchC can generate interpreted simulators (using SystemC) and compiled simulators (using C++), along with assemblers, linkers, and debuggers (using the GNU Binutils framework [9]).

Section 2.2.1 and Sect. 2.2.2 discuss the descriptions of architecture resources and ISA by means of examples, whereas Sect. 2.2.3 summarizes the evolution of ArchC towards platform modeling.

Throughout this book, our illustrative examples rely mainly on samples of ArchC descriptions for the PowerPC architecture. However, when this choice may limit our discussion of ADL features, we sometimes also include samples from MIPS, SPARC-V8, and Intel 8051 ArchC models, which can be downloaded from the ArchC website [12].

2.2.1 Architecture Resources Description

The architecture resources represent the structural information about the target architecture, like register banks, memory, pipeline, special registers, endian, etc. ArchC collects such information from the so called AC_ARCH description.

Obviously, the degree of detail adopted for this description depends on the level of abstraction required for the desired executable model. Although ArchC allows (in its version 1.6) the declaration of pipelines and multicycle instructions, in this book we focus on *functional models*, which are high-level descriptions suitable for the building of virtual platform models using TLM and SystemC. (For a complete ArchC specification, the reader should refer to the language reference manual [11].)

An architecture description at the functional level needs little structural information, as shown in Fig. 2.5. This example illustrates the minimum amount of architecture resource information required to build a PowerPC functional model.

Let us explain the main keywords in the example of Fig. 2.5.

AC_ARCH: An architecture resource description always starts with this keyword.

The designer should provide the model's name (e.g. powerpc).

ac_wordsize: Declares the size of the processor's word in number of bits.

`ac_regbank`: Declares the register bank and its number of registers (e.g. the GPR bank has 32 registers).

`ac_reg`: Declares a single register (e.g. MSR).

`ac_mem`: Declares a memory of a given size (e.g. MEM has 8 megabytes). The size can be expressed in bytes (no unit abbreviation needed), in kilobytes (K or k), in megabytes (M or m), or in gigabytes (G or g).

`ac_tlm_port`: Declares an external TLM communication port. It is followed by the name of that port object, a colon, and the size of its address space. This size, just like in `ac_mem`, may be expressed either in bytes or its multiples. (Chapter 3 will explore TLM ports in detail).

`ac_tlm_intr_port`: Declares an interrupt TLM communication port, followed by the port object name.

`ARCH_CTOR`: Initializes the `AC_ARCH` constructor declaration.

`ac_isa`: Informs the name of the file containing the `AC_ISA` description (e.g. `powerpc_isa.ac`) attached to this architecture description.

`set_endian`: Defines the architecture's endianness as "big" or "little".

2.2.2 Instruction Set Architecture Description

The `AC_ISA` description provides the behavior of every instruction and all the information required to automatically synthesize a decoder for a given ISA. This description is divided in two files, one containing instruction and format declarations, another containing instruction behaviors.

ISA Specification

For simplicity, the main ArchC keywords appearing in an `AC_ISA` description are addressed by means of an example. Figure 2.6 shows a fragment of `AC_ISA` description extracted from a PowerPC model and Fig. 2.7 shows another from the SPARC model.

First, let us focus on the description of instructions, formats, and encodings by providing an overview of the main keywords. (Chapter 4 will explore their usage to build processor models).

`AC_ISA`: An ISA description always starts with this keyword. The designer should provide the model's name (e.g. `powerpc` in Fig. 2.6, Line 1; `sparcv8` in Fig. 2.7, Line 1).

`ISA_CTOR`: It merely initializes the `AC_ISA` constructor declaration.

`ac_format`: It declares an instruction format and its fields (e.g. in Fig. 2.6, Line 2, a format `I1` is defined as the concatenation of four fields; the first, named `opcd`, consists of 6 bits). ArchC provides an additional construct that allows fields to overlap. It can be used to facilitate the description of complex instruction sets, as illustrated by the example in Fig. 2.7, Line 8. The declaration of alternative

```

1  AC_ISA(powerpc){
2      ac_format I1 = "%opcd:6 %li:24:s %aa:1 %lk:1";
3      ac_format B1 = "%opcd:6 %bo:5 %bi:5 %bd:14:s %aa:1
        %lk:1";
4      ac_format XO1 = "%opcd:6 %rt:5 %ra:5 %rb:5 %oe:1 %xos:9
        %rc:1";
5      ac_format SC1 = "%opcd:6 0x00:5 0x00:5 0x00:4 %lev:7 0
        x00:3 0x01:1 0x00:1";
6
7      ac_instr<I1> b, ba, bl, bla;
8      ac_instr<B1> bc, bca, bcl, bcla;
9      ac_instr<XO1> add, add_, adc, mullw, divw, subf;
10     ac_instr<SC1> sc;
11
12     ac_asm_map reg {
13         /* default gas assembler uses numbers as register
            names */
14         "" [0..31] = [0..31];
15     }
16
17     ISA_CTOR(powerpc){
18
19         add.set_asm("add %reg, %reg, %reg", rt, ra, rb);
20         add.set_decoder(opcd=31, oe=0, xos=266, rc=0);
21
22         bca.set_asm("bca %imm, %exp, %addr(pcrel)", bo, bi, bd
            );
23         bca.set_decoder(opcd=16, aa=1, lk=0);
24
25         lmw.set_asm("lmw %reg, %imm (%reg)", rt, d, ra);
26         lmw.set_asm("lmw %reg, %exp@l(%reg)", rt, d, ra);
27         lmw.set_decoder(opcd=46);
28
29         sth.set_asm("sth %reg, %imm (%reg)", rs, d, ra);
30         sth.set_asm("sth %reg, %exp@l(%reg)", rs, d, ra);
31         sth.set_decoder(opcd=44);
32         ...
33         pseudo_instr("mr %reg, %reg") {
34             "or %0, %1, %1";
35         }
36     };
37 }

```

Fig. 2.6 Fragment of the PowerPC `ac_isa` description

field choices starts with a square bracket (“[”). Additional groups are given after a vertical bar (“|”). When all alternatives are declared, a closing square bracket (“]”) ends the declaration. It should be noted that ArchC decodes all

overlapping fields. However, although they can be accessed independently, not all are simultaneously valid. The designer has to define which alternative group is valid according to the value of some other field. For the example at Line 8, the field “is” plays the role of a selector. When `is=1`, fields “r2a” and “rs2” are valid; otherwise “r2b” and “imm7” are the valid fields.

`ac_instr<fmt>`: It declares an instruction and ties it to a predefined format. Formats are assigned to instructions using a syntax similar to C++ *templates*. In Fig. 2.6, for instance, the instruction `add` at Line 9 is tied to the instruction format `X01` declared at Line 4.

`set_decoder`: It initializes the instruction decoding sequence, which is a key element to the automatic generation of an instruction decoder for the executable processor model. The sequence is composed of pairs `<field_name = value>`. In Fig. 2.6 at Line 20, for instance, `add.set_decoder` states that a bit stream coming from memory is an `add` instruction if, and only if, fields `opcd`, `oe`, `xos`, and `rc` contain the values 31, 0, 266, and 0, respectively.

Now, let us address the keywords that define symbolic names for instructions, registers and groups of instructions, since they are the key to the automatic generation of binary utilities (e.g. assemblers). Chapter 6 will explore such keywords and binary utilities generation in more depth.

`ac_asm_map`: Specifies a mapping between assembly symbols and values (e.g. in Fig. 2.6, Lines 12–15) define the set of register names and their corresponding numbers in the register bank of the PowerPC architecture).

`set_asm`: Associates an assembly syntax string and operand encoding to an instruction. The syntax of this construct is similar to the `printf` family 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 associated instruction field, specifying the operand encoding (e.g. in Fig. 2.6, Line 19, the `add` instruction uses three operands of type `reg` with are associated, respectively, with the fields `rt`, `ra`, and `rb` of the format declared in Line 4).

`pseudo_instr`: Describes a pseudo instruction in terms of previously described instructions (e.g. in Fig. 2.6, Lines 33–35, the pseudo instruction `mr` is associated with the predeclared instruction `or`).

Instruction Behavior Description

The designer must also provide the operations executed by each instruction. This is done by means of the so-called behavior methods. ArchC allows three hierarchical levels for instruction behavior description: the generic behavior, which contains actions that must be executed for every instruction, the instruction-type behavior, which describes actions to be executed by the subset of instructions tied to a given

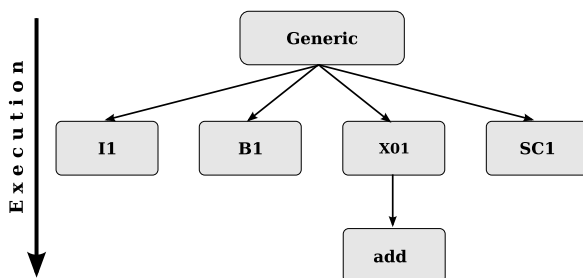
```

1  AC_ISA(sparcv8){
2    ac_format Type_F1="%op:2 %disp30:30";
3    ac_format Type_F2A="%op:2 %rd:5 %op2:3 %imm22:22";
4    ac_format Type_F3A="%op:2 %rd:5 %op3:6 %rs1:5 %is:1 %asi:8 %
      rs2:5";
5    ac_format Type_F3B="%op:2 %rd:5 %op3:6 %rs1:5 %is:1 %simm13
      :13:s";
6    /* format for trap instructions */
7    ac_format Type_FT="%op:2 %r1:1 %cond:4 %op2a:6 %rs1:5
8      %is:1 [%r2a:8 %rs2:5 | %r2b:6 %imm7:7]";
9    ac_instr<Type_F1> call;
10   ac_instr<Type_F2A> nop, sethi;
11   ac_instr<Type_F3A> add_reg, sub_reg;
12   ac_instr<Type_F3B> add_imm, sub_imm;
13   ac_instr<Type_FT> trap_reg, trap_imm;
14   ...
15 }

```

Fig. 2.7 Fragment of the SPARC `ac_isa` description

Fig. 2.8 The instruction behavior hierarchy in ArchC



instruction format, and the instruction-specific behavior that distinguishes it from all other instructions. The idea behind this hierarchy is that designers can factorize the instruction behavior to minimize the code size of the model. The automatically generated processor model will rely on such hierarchical behavioral description: every time a new instruction is fetched, the execution always starts at the generic behavior method and then goes down through the type and specific behaviors. Figure 2.8 illustrates this sequence for the PowerPC's `add` instruction, which was declared in Fig. 2.6.

In ArchC descriptions, instruction behavior methods are actually written in pure SystemC/C++ code, thereby avoiding that users should learn additional keywords (beyond those described so far). The designer can even declare helper functions to perform special actions and use them inside their behavior descriptions. For more details on how to design a processor model using ArchC, the reader should refer to Chap. 4, which presents a step-by-step model development process, through a richer set of illustrative examples.

2.2.3 The Evolution of ArchC Towards Platform Modeling

ArchC was first designed for processor architecture research, specially for experimenting with new instruction sets for application-specific architectures. The main goal was to generate simulators which were fast enough to run real software applications by means of a processor description that should be easy to write and maintain. From the very beginning, SystemC was the choice for the ArchC simulator generation tool.

The first open-source release of ArchC happened in February 2004. Its first versions evolved to support interpreted simulator generation in SystemC, compiled simulation generation and optimization using C++, and software tools like assemblers. But, as the number of ArchC users increased, it became clear that its most popular application was to generate processor models to be embedded into complex virtual platform models written in SystemC.

Since its 2.0 release in 2001, SystemC was gaining momentum as a language to enable design in higher abstraction levels. At that time, system-level design was attracting a lot of attention from the EDA community, and SystemC was one of the most suitable languages to support this new design paradigm (actually, the main goal of SystemC was to enable system-level modeling [4]).

In short, with the advent of System-on-Chip (SoC) designs, the whole design process where ADLs were applied changed. Processor models are not only used for processor architecture development anymore, but also became an important part of heterogenous platform models on a SoC design flow, aiding on the application of the new ESL design methodologies.

Aiming this new horizon for their languages, ADL designers have to increase not only the expression power of their languages, making them capable of modeling the modern complex architectures, but also the modularity and portability of their generated simulators, making them suitable for ESL-based design flows. The most important aspect on this matter is the communication capabilities of those simulators. An ADL will only be adopted by a platform designer if its simulators are easily integrated in their platform models, which implies in making them capable of communicating with other hardware modules.

Although it has always been possible to integrate ArchC models into SystemC platform models (since they were written in SystemC), this was not an easy task up to version 1.6. Users had to manually alter the automatically generated simulator to include a communication channel so as to connect the processor model to a wrapper or to another SystemC module directly.

As the demand from researchers interested in virtual platforms had grown substantially and since the so-called *Transaction Level Modeling* (TLM) style had risen as the most promising alternative for platform modeling in SystemC-based ESL environments, ArchC developers decided to add a feature to their automatically generated functional simulators from version 2.0 on: the capability of declaring TLM ports. As a result, it became pretty simple for platform designers to use those simulators as processor models within an ESL environment by modeling communication

in the TLM style. Besides, the new feature also granted the capability of modeling an interrupt system.

Chapter 3 will introduce the TLM concept and its application in SystemC, and Chap. 5 will present an in-depth discussion on the platform integration capabilities of ArchC-generated simulators by means of a few platform design examples.

References

1. R. Azevedo, S. Rigo, M. Bartholomeu, G. Arajo, C. Arajo, E. Barros, The archc architecture description language. *Int. J. Parallel Program.* **33**(5), 453–484 (2005)
2. J. Bhasker, *A SystemC Primer* (Star Galaxy Publishing, Allentown, 2002)
3. D.C. Black, J. Donovan, *SystemC: From the Ground Up* (Springer, Berlin, 2004)
4. T. Grötter, S. Liao, G. Martin, S. Swan, *System Design with SystemC* (Kluwer Academic, Dordrecht, 2002)
5. IEEE Standards Association, IEEE 1666™ Standard SystemC® Language Reference Manual. Available at <http://standards.ieee.org>, December 2005
6. P. Mishra, N. Dutt (eds.), *Processor Description Languages* (Morgan Kaufmann, San Mateo, 2008)
7. W. Müller, W. Rosenstiel, J. Ruf, *SystemC: Methodologies and Applications* (Springer, Berlin, 2003)
8. Open SystemC Initiative, SystemC® Language Reference Manual, Version 2.2. Available at <http://www.systemc.org>. Accessed in February 2009
9. R.H. Pesch, J.M. Osier, *The GNU Binary Utilities* (Free Software Foundation Inc, Boston, 1993)
10. S. Rigo, G. Araujo, M. Bartholomeu, R. Azevedo, ArchC: A SystemC-based architecture description language, in *16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)* (2004), pp. 66–73
11. The ArchC Team, The ArchC Architecture Description Language v 2.0 Reference Manual. Available at <http://www.archc.org>, August 2007
12. The ArchC website (March 2008). <http://www.archc.org>



<http://www.springer.com/978-1-4020-9939-7>

Electronic System Level Design

An Open-Source Approach

Rigo, S.; Azevedo, R.; Santos, L. (Eds.)

2011, IX, 146 p., Hardcover

ISBN: 978-1-4020-9939-7