Sandro Rigo · Rodolfo Azevedo
Luiz Santos *Editors*

Electronic System Level Design

An Open-Source Approach

Springer

Electronic System Level Design

Sandro Rigo • Rodolfo Azevedo • Luiz Santos
Editors

Electronic System Level Design

An Open-Source Approach

Springer

Editors
Sandro Rigo
Instituto de Computaçao
Universidade Estadual de Campinas
Av.Albert Einstein 1251
Campinas, São Paulo 13083-970
Brazil
sandro@ic.unicamp.br

Luiz Santos
Depto. Informática e Estatística
Universidade Federal de Santa Catarina
UFSC Campus
Florianópolis, Santa Catarina 88030-300
Brazil
santos@inf.ufsc.br

Rodolfo Azevedo
Instituto de Computaçao
Universidade Estadual de Campinas
Av.Albert Einstein 1251
Campinas, São Paulo 13083-970
Brazil
rodolfo@ic.unicamp.br

ISBN 978-1-4020-9939-7 e-ISBN 978-1-4020-9940-3
DOI 10.1007/978-1-4020-9940-3
Springer Dordrecht Heidelberg London New York

Library of Congress Control Number: 2011927759

© Springer Science+Business Media B.V. 2011
No part of this work may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, microfilming, recording or otherwise, without written permission from the Publisher, with the exception of any material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work.

Cover design: VTeX UAB, Lithuania

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Electronic System Level Design, or ESL Design, is generally understood as the set of tools, methodologies and design techniques applied to modern electronic systems design, from high-end chips and systems, to embedded devices, to integrated hardware and software systems. Given the complexity of current systems, advanced tools and methodologies have become absolutely essential to achieve the necessary productivity, quality, cost and performance expected in a design process. One of the important tenets in ESL Design is the need for early design analysis. This is done mainly through high-level modeling and simulation, performance and power analysis and functional verification, before committing the design to lower-levels of abstraction aimed at synthesis and optimization. This is especially true for complex systems involving different types of components such as processors, custom blocks and software. In fact, it may be totally impractical to simulate such systems at a low-level of representation such as register-transfer level, due to extremely long simulation times. High-level models are simpler to write, understand, optimize and debug than lower-level models, and they can simulate significantly faster. The more the design can be refined, optimized and verified at a high-level of abstraction, the higher the overall design productivity, the better the quality and consequently the lower the cost of the final result. However, high-level models and development environments are not without their own difficulties. It is not simple to write a high-level model at the appropriate abstraction level which will result in the best trade-off between architectural details and simulation speed. For this end, researchers have formalized different abstraction levels at different levels of architectural and timing accuracy. Depending on what types of design analysis need to be done, one abstraction level may be more suitable than another. This adds to the modeling complexity, and in the worst case, if multiple models need to be developed, it may start reducing the productivity advantage of a high-level modeling methodology. A high-level modeling and simulation framework is a complex software engineering challenge. Different types of models, such as a processor model, custom blocks and application software, need to be compiled, linked, executed and debugged together. This goes far beyond the correct individual modeling of a block using a high-level language. A successful high-level design methodology depends heavily on how well automated this framework is. SystemC is a system-level specification and design language (based on C++

classes) that has been widely adopted for high-levels of abstraction modeling. SystemC also emergence of transaction-level models (TLM) as an abstraction layer and modeling style capable of fully separating the computation part from the communication part of a model. Despite the widespread acceptance and use of SystemC and TLM, it is still very complex to create models which achieve a perfect balance between the required architectural detail for the desired types of analysis and simulation speed. This book addresses the very issues raised above. It presents a high-level design methodology, support tools and framework capable of full system modeling and design exploration, including components such as processors, custom blocks and application software. One of the main contributions of the book is the detailed description of ArchC. ArchC is an architectural description language (ADL) which allows designers to model instruction-set architectures and automatically map them into SystemC simulatable descriptions. ArchC is an ADL capable of detailed modeling of instructions and supporting processor architectures. Since ArchC generates an executable SystemC model, it can be fully integrated with other SystemC models using transaction-level interfaces, which allows the specification, modeling and simulation of complete systems. The book also explains in detail how the overall software environment works, including how the models are compiled, linked, executed together and debugged as a platform. Detailed examples using single and dual core platforms, coupled with custom accelerators and running real life applications are presented. Power modeling is also addressed in this book. Authors describe an interesting extension to SystemC called PowerSC, which allows switching and power information to be gathered and computed during and integrated with the functional simulation. Important to readers should the fact that the platform presented is Open-Source and available for download, whereas comparable systems available today are proprietary. This will certainly help researchers and developers alike to jumpstart their modeling efforts by using a readily available platform and development tools. Readers interested in a good overview of ESL methodologies as well as those interested in practical implementation details of architectural-description languages, platform modeling and support tools will be well served by this book.

Visiting Professor Reinaldo A. Bergamaschi
Odysci, Brazil/USA
University of Campinas, Campinas, Brazil

Contents

Part I System Design Representation

1 Electronic System Level Design 3
Luiz Santos, Sandro Rigo, Rodolfo Azevedo, and Guido Araujo

2 Open-Source Languages 11
Sandro Rigo, Luiz Santos, Rodolfo Azevedo, and Guido Araujo

3 Transaction Level Modeling 25
Sandro Rigo, Bruno Albertini, and Rodolfo Azevedo

Part II Open-Source Models and Tools

4 ArchC Model Design Handbook 39
Rodolfo Azevedo and Sandro Rigo

5 Building Platform Models with SystemC 71
Rodolfo Azevedo, Sandro Rigo, and Bruno Albertini

6 Retargetable Binary Tools 99
Alexandro Baldassin, Paulo Centoducatte, and Luiz Santos

Part III Advanced Topics

7 Debugging SystemC Platform Models 117
Bruno Albertini, Sandro Rigo, and Guido Araujo

8 SystemC-Based Power Evaluation with PowerSC 129
Felipe Klein, Rodolfo Azevedo, Luiz Santos, and Guido Araujo

Index 145

Contributors

Bruno Albertini Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil, balbertini@ic.unicamp.br

Guido Araujo Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil, guido@ic.unicamp.br

Rodolfo Azevedo Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil, rodolfo@ic.unicamp.br

Alexandro Baldassin Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil, alebal@ic.unicamp.br

Paulo Centoducatte Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil, ducatte@ic.unicamp.br

Felipe Klein Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil, klein@ic.unicamp.br

Sandro Rigo Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil, sandro@ic.unicamp.br

Luiz Santos System Design Automation Laboratory, Computer Science Department, Federal University of Santa Catarina, Florianópolis, Brazil, santos@inf.ufsc.br

Part I
System Design Representation

Chapter 1
Electronic System Level Design

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

1.1 The ESL Concept

Systems-on-chip (SoCs) became a reality in the mid-nineties, as a result of the long evolution of VLSI technology and the sheer growth of integrated circuit complexity. As it happens each time electronic design complexity impairs the expected time-to-market, the quest for higher productivity involves a combination of the following key notions:

Abstraction At design entry, the level of representation is raised to cope with the increased design complexity. As a consequence, the design flow goes through several levels and styles of representation. Hardware design representation has been raised through physical, circuit, gate, register-transfer, and functional/behavioral levels.

Reuse Pre-designed components are assembled within a new design. Reuse goes hand-in-hand with design paradigms, such as cell-based, IP-based, and platform-based.

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

S. Rigo · R. Azevedo · G. Araujo
Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil

S. Rigo
e-mail: sandro@ic.unicamp.br

R. Azevedo
e-mail: rodolfo@ic.unicamp.br

G. Araujo
e-mail: guido@ic.unicamp.br

S. Rigo et al. (eds.), *Electronic System Level Design*,
DOI 10.1007/978-1-4020-9940-3_1, © Springer Science+Business Media B.V. 2011

Automation To overcome the error-prone and time-consuming nature of manual refinement, designers rely on Electronic Design Automation (EDA) tools. To refine the design representation from higher to lower abstraction levels, (layout, logic, register transfer, and behavioral) synthesis tools can be used. To check for equivalent functionality across successive levels of representation, a solid infrastructure of automatic verification tools is available.

Exploration The analysis of alternative design solutions with respect to area, performance and power at a given abstraction level reduces the probability that, after refinement through lower levels, the design might turn out not to meet the specified requirements, leading to redesign.

Although integrated circuit design has already witnessed abstraction level raising from the circuit level to the gate level and then to the *Register-Transfer Level* (RTL), the so-called SoC Revolution [10] required a broader paradigm shift. Since the integrated circuit became an integrated *system*, the new abstraction should harmonize the representation of both hardware and software. Abstraction offers system-level design representations, such as executable hardware-software especifications [2], transaction-level modeling (TLM) [4] and UML modeling. The *platform-based design paradigm* [14] guides reuse from the perspective of a reference system architecture. Automation asks for software-toolkit generation [9], such as retargetable compilers and binary utilities (since the software embedded in the system may be run on possibly distinct multiple target processors). Exploration addresses the joint-evaluation of hardware and software components.

This leads to the concept of *Electronic System Level* (ESL) [1], a generic term for a set of abstractions (possibly at distinct levels), which are adequate for the design representation of SoCs. It complies with the need for hardware and software co-design, while building upon legacy hardware design representation.

1.2 Requirements of an ESL Representation

A representation suitable for ESL design should provide a unique representation of the platform architecture, capture parameters handled by hardware-dependent software, allow advance performance and power evaluation on the target platform, rely on encapsulation of processor models and non-programmable components, support platform debugging, and enable virtual prototyping. Let us consider each of such aspects individually:

Uniqueness of representation Actual hardware-software co-design should rely on a unique abstraction of the hardware platform so that changes performed by either the hardware design or the software development teams could be agreed upon a same reference. It should also play the role of golden reference model for the hardware verification team.

Hardware-dependent software The representation must provide a programmer's view of the platform that not only captures the *Instruction-Set Architecture* (ISA)

of each processor, but also includes bit-accurate descriptions of I/O device registers, memory maps for all devices with configurable registers and memory layout for data, application code, and initialization routines. This allows the early development of hardware-dependent software, such as drivers and boot code, enabling actual hardware-software co-design.

Advance performance evaluation Although it is possible to obtain a rough performance estimate by running the application software on a stand-alone instruction-set simulator, a much more accurate performance evaluation is obtained when not only the processors, but all the other platform components are captured in the design representation.

Advance power evaluation In spite of the fact that power estimation is much more accurate at lower design levels, the ability to assess power earlier in the design flow helps in the identification of power bottlenecks, thereby allowing low-power design from the very beginning.

Encapsulation of processor models Since processors are the most complex system components, the expected ESL productivity gain would be seriously impaired if processor models had to be developed from scratch. Therefore, the representation should encapsulate either reused processor models extracted from a library or automatically generated processor models.

Encapsulation of non-programmable components From the perspective of hardware-dependent software development, the behavior of a component is what matters, regardless of how it will be implemented later, as far as a bit-accurate view of its I/O registers is available. Therefore, the representation should allow the functional modeling of hardware components.

Platform debugging The software to be embedded into the system could be run on the host workstation and conventional debuggers could be used to pinpoint application code bugs. Besides, code inspection could be improved by porting a debugger so that it could run on an instruction-set simulator of a target processor of a platform. Although yet useful, this conventional approach is clearly limited to uncovering software bugs. Therefore, to expose architecture design bugs, the representation should support component I/O probing or even component inner probing.

Virtual prototyping The representation should support the co-simulation of hardware and software in such a way that a virtual prototype could be used to evaluate the functionality, the performance, and the power/energy consumption of the final product. Such a prototype would contain application software, hardware-dependent software and platform architecture, elements allowing sufficiently accurate pre-evaluation without the need to dive into more time-consuming RTL-based prototyping or emulation.

1.3 ESL Design Flow

As a consequence of platform-based design, ESL flows are typically a combination of bottom-up and top-down approaches. In spite of that, Fig. 1.1 shows an idealized

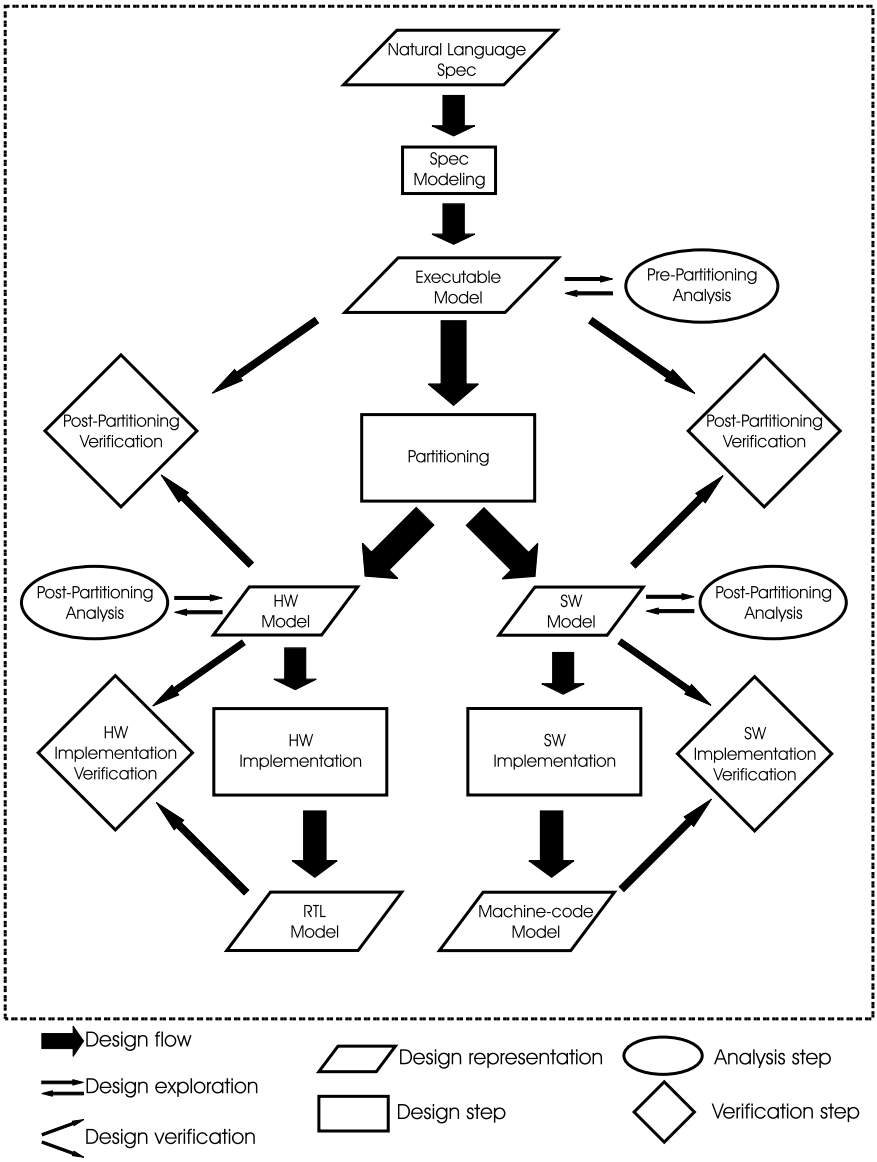


Fig. 1.1 An idealized top-down ESL design flow

top-down ESL design flow, which is used here as a frame to illustrate the main design steps rather than to advocate a particular design flow. Figure 1.1 adopts the terminology introduced in [1].

From a *natural language specification*, i.e. a document capturing the product requirements and constraints, the first design step consists in obtaining a specification

written in an executable or declarative language. (This book focuses on executable specifications, such as those obtained with SystemC [5]).

The resulting executable model, which captures the *functional specification* of a system, is submitted to a preliminary analysis step so as to provide grounds for future architectural decisions. By means of proper instrumentation, initial estimates for performance, storage space, power, and communication traffic can be obtained.

Based upon that preliminary analysis, the next design step leads to an *architectural specification* of a system, obtained from its functional specification through *hardware-software partitioning*, i.e. the mapping of algorithms to software running on target processors or to non-programmable hardware blocks. Although the resulting architecture specification captures hardware and software views, they should be built as interacting engines that execute cooperatively within a unified architectural model. To reach a threshold of simulation performance that enables hardware-software codesign, such architectural model may rely on the TLM style. (In Chap. 5, this book proposes an open-source infrastructure to build executable architectural models that comply with TLM, which is reviewed in Chap. 3).

Once an architectural specification is available after partitioning, it has to be verified with respect to the original functional specification. Essentially, such *post-partitioning verification* aims at demonstrating that the design satisfies the specified requirements.

A few building blocks are required for the architectural specification, such as processors, non-programmable hardware components, memories, and buses. On the one hand, a building block representing a non-programmable hardware component is easily modeled by its (single) behavior (e.g. the C++ implementation of its underlying algorithm). On the other hand, since processors exibit many complex behaviors, their executable models are usually generated from declarative processor models written in an *Architecture Description Language* (ADL). (This book describes an open-source ADL in Chap. 2).

The resulting architectural model also enables a *post-partitioning analysis* step, where more accurate estimates can be obtained. For instance, performance, storage space, and power can be evaluated at the light of the chosen target processors; communication traffic can be correlated with the number of transactions. This analysis leads to the exploration of alternative architectural solutions, for instance by experimenting with distinct target processors. For such an exploration, cross compilers and binary tools must be available for the target processors. (This book describes open-source infrastructure for automatically retargeting binary utilities in Chap. 6).

Post partitioning analyses of the hardware and software models provide grounds for proper *hardware and software implementation*, giving rise to an RTL model for the hardware and application binaries for the software.

The resulting implementations have to be verified with respect to the architectural specification. The ultimate goal of such *implementation verification* is to demonstrate that the implementation has no flaws. For this, the observability of a device under verification has to be increased through white-box techniques allowing the inspection of a component's inner structure. (A suitable infrastructure for white-box verification is proposed in Chap. 7).

Although, at the first glance, executable models may seem able to track behavior only (and thereby performance), they have to be extended to simultaneously track other important issues, like power consumption, for instance. (Chapter 8 shows how SystemC descriptions can be instrumented to account for power).

This book addresses a few important aspects of the ESL design flow. It focuses on languages and artifacts used as infrastructure for executable specification, hardware-software modeling, and design tools (implementation, analysis, and verification).

1.4 Target Audience, Scope and Organization

Although several ESL technologies are currently provided by EDA vendors, this book focuses on technologies available under open-source licenses, such as SystemC [5], OSCI TLM [12], ArchC [13], and PowerSC [8].

This book intends to provide grounds for further research on ESL, by means of open-source artifacts and tools, thereby stimulating the unconstrained deployment of new concepts, tools, and methodologies. It devises electronic system design from the pragmatic perspective of a SystemC-based ESL representation, by showing how to *build* and how to *use* ESL languages, models, and tools.

Other specification languages suitable for ESL design such as MATLAB [15], UML [11], SDL [7] and SystemVerilog [6] are outside the scope of this book. Proprietary ESL-compliant processor modeling, such as [3], are also beyond the intended scope.

Most of the requirements enumerated in Sect. 1.2 are covered in the next chapters, which are organized as follows:

- **ESL Specification**: SystemC is one of the most promising ESL languages. It is actually a class library that extends the C++ language in such a way that hardware and software components can be described within an executable specification. Some relevant aspects of SystemC are reviewed in Chap. 2 as a basis for the discussions in further chapters.
- **ESL-Compliant Processor Modeling**: Since describing processor models directly in SystemC would not be practical, most of processor modeling and software toolkit generation relies on ADLs. Since the ADLs designed in the mid-nineties preceded the rise of SystemC, the executable processor models generated by them were not thought to be encapsulated as SystemC modules. The ArchC language is an ADL born in the SystemC era. Instead of directly generating an executable processor model, a functional or cycle-accurate SystemC model is produced. The ArchC language is summarized in Chap. 2, while Chap. 4 and Chap. 5 explain how ArchC can be efficiently used to build ESL-compliant processor models. As an important complement, Chap. 6 shows how the ArchC ADL can be used as a starting point for the automatic generation of assemblers, linkers, and debuggers.
- **TLM-Based ESL Design**: A crucial concept to ESL is the separation between computation and communication of a system module. Since SystemC is already

built upon that concept, the definition of interfaces to encapsulate communication protocols within channels allows a convenient style of design representation: the system can be described as a set of modules (each exhibiting its concurrent behavior) that communicate by means of transactions through channels. The TLM style provides a programmer's view of the platform that allows early development of hardware-dependent software. Chapter 3 reviews the main assets of the TLM design representation.

- **ESL Design Verification**: Since the SystemC Verification Library is an already well-established verification infrastructure based on the conventional black-box approach, Chap. 7 focuses on a complementary verification aspect: the use of data introspection to enable white-box verification strategies.
- **ESL Power-Conscious Exploration**: Although SystemC can model hardware and software functionalities, thereby allowing performance-based exploration, it does not come with support for modeling power consumption. Chapter 8 shows how SystemC can be extended to capture signal transitions, to correlate them with dynamic power consumption, to add up the static power contribution from technology libraries, and to accommodate distinct power macromodels.

References

1. B. Bailey, G. Martin, A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology* (Morgan Kaufmann, San Mateo, 2007)
2. D.C. Black, J. Donovan, *SystemC: From the Ground Up* (Springer, Berlin, 2004)
3. Coware Inc., CoWare® Processor Designer. Available at http://www.coware.com. Accessed in February 2009
4. F. Ghenassia, *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems* (Springer, New York, 2006)
5. IEEE Standards Association, IEEE 1666™ Standard SystemC© Language Reference Manual. Available at http://standards.ieee.org, December 2005
6. IEEE Standards Association, IEEE 1800-2005™ Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language. Available at http://standards.ieee.org, 2005
7. International Telecommunication Union, Specification and Description Language (SDL), ITU-T Recommendation Z.100. Available at http://www.itu.int, August 2002
8. F. Klein, G. Araujo, R. Azevedo, R. Leao, L.C.V. dos Santos, An efficient framework for high-level power exploration, in *50th Midwest Symposium on Circuits and Systems (MWSCAS)* (2007), pp. 1046–1049
9. R. Leupers, P. Marwedel, *Retargetable Compiler Technology for Embedded Systems: Tools and Applications* (Kluwer Academic, Dordrecht, 2001)
10. G. Martin, H. Chang, *Winning the SoC Revolution: Experiences in Real Design* (Kluwer Academic, Dordrecht, 2003)
11. Object Management Group, OMG Unified Modeling Language™, Superstructure, V2.1.2. Available at http://www.omg.org. Accessed in February 2009
12. Open SystemC Initiative, TLM Transaction Level Modeling Library, Release 2.0 Draft 2. Available at http://www.systemc.org. Accessed in February 2009

13. 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
14. A. Sangiovanni-Vincentelli, G. Martin, Platform-based design and software design methodology for embedded systems. IEEE Des. Test Comput. 23–33 (2001)
15. The Mathworks Inc., MATLAB®. Available at http://www.mathworks.com. Accessed in February 2009

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

S. Rigo et al. (eds.), *Electronic System Level Design*,
DOI 10.1007/978-1-4020-9940-3_2, © Springer Science+Business Media B.V. 2011

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 subcomponents.

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

```
int sc_main(int ac, char *av[])
{
 // Phase 1: ELABORATION

 // An instance of a PowerPC processor
 ppc PPC("PPC");
 // An instance of a read-write memory
 mem MEM("MEM");

 // Connection between PPC's and MEM's data ports
 PPC.DM_port(MEM.target_export);

 // Phase 2: SIMULATION

 sc_start();

 // PHASE 3: DIAGNOSIS

 PPC.PrintStat();

 return PPC.exit_status;
 }
```

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 auxiliarly 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

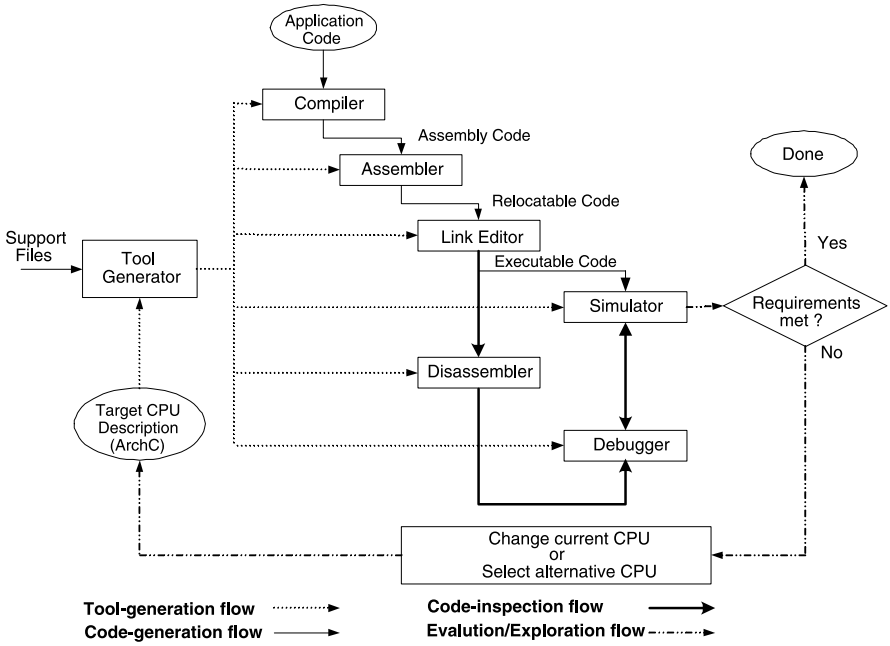


Fig. 2.4 ADL-based exploration flow

synthesizes a compiler's backend, an ISS, and a set of binary utilities. The application's source code is compiled, assembled and linkedited, 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,

```
AC_ARCH(powerpc) {
        ac_wordsize 32;
        ac_mem MEM:8M;
        ac_regbank GPR:32;
        ac_reg MSR;
        ARCH_CTOR(powerpc) {
                ac_isa("powerpc_isa.ac");
                set_endian("big");
    };
};
```

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

```
AC_ISA(powerpc){
  ac_format I1 = "%opcd:6 %li:24:s %aa:1 %lk:1";
  ac_format B1 = "%opcd:6 %bo:5 %bi:5 %bd:14:s %aa:1
      %lk:1";
  ac_format XO1 = "%opcd:6 %rt:5 %ra:5 %rb:5 %oe:1 %xos:9
      %rc:1";
  ac_format SC1 = "%opcd:6 0x00:5 0x00:5 0x00:4 %lev:7 0
      x00:3 0x01:1 0x00:1";

  ac_instr<I1> b, ba, bl, bla;
  ac_instr<B1> bc, bca, bcl, bcla;
  ac_instr<XO1> add, add_, adc, mullw, divw, subf;
  ac_instr<SC1> sc;

  ac_asm_map reg {
    /* default gas assembler uses numbers as register
        names */
    ""[0..31] = [0..31];
  }

  ISA_CTOR(powerpc){

    add.set_asm("add %reg, %reg, %reg", rt, ra, rb);
    add.set_decoder(opcd=31, oe=0, xos=266, rc=0);

    bca.set_asm("bca %imm, %exp, %addr(pcrel)", bo, bi, bd
        );
    bca.set_decoder(opcd=16, aa=1, lk=0);

    lmw.set_asm("lmw %reg, %imm (%reg)", rt, d, ra);
    lmw.set_asm("lmw %reg, %exp@l(%reg)", rt, d, ra);
    lmw.set_decoder(opcd=46);

    sth.set_asm("sth %reg, %imm (%reg)", rs, d, ra);
    sth.set_asm("sth %reg, %exp@l(%reg)", rs, d, ra);
    sth.set_decoder(opcd=44);
                          ...
    pseudo_instr("mr %reg, %reg") {
      "or %0, %1, %1";
    }
  };
}
```

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 `XO1` 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

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

Fig. 2.7 Fragment of the SPARC `ac_isa` description

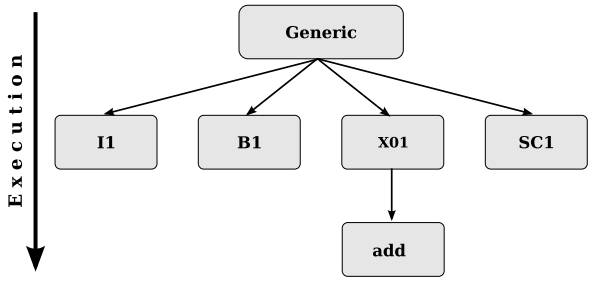


Fig. 2.8 The instruction behavior hierarchy in ArchC

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. Grötker, 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

Chapter 3
Transaction Level Modeling

Sandro Rigo, Bruno Albertini, and Rodolfo Azevedo

This chapter introduces the Transaction Level Modeling (TLM) design methodology. Its main goals are to explain the main concepts behind TLM and to show that it can work very well in SystemC-based designs. That is why we discuss the strong relation between SystemC and TLM and emphasize the main features in the SystemC TLM 2.0 standard. As in Chap. 2, the intention is not to serve as a substitute for the reference manual. Instead, our focus is to provide a good understanding on the underlying ideas and to show how to put TLM to work in SystemC-based platform designs. (For details, the interested reader should refer to the OSCI TLM 2.0 User Manual.) We finalize this chapter with a simple platform example using the SystemC TLM standard. Chapter 5 will present more elaborate platform examples by relying on the main concepts and elements introduced by Chaps. 2–4.

3.1 Introduction

As already discussed in Chap. 1, the rise of SoCs caused a paradigm shift on system design flow. The TLM methodology was created in the search for a new paradigm that could allow design representation at an intermediate level of abstraction be-

S. Rigo (✉) · B. Albertini · R. Azevedo
Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil
e-mail: sandro@ic.unicamp.br

B. Albertini
e-mail: balbertini@ic.unicamp.br

R. Azevedo
e-mail: rodolfo@ic.unicamp.br

S. Rigo et al. (eds.), *Electronic System Level Design*,
DOI 10.1007/978-1-4020-9940-3_3, © Springer Science+Business Media B.V. 2011

tween paper specification and RTL models. That is why TLM plays nowadays an important role in ESL technologies.

The first key concept behind TLM is that unnecessary details should be avoided at the early phases of the design flow. By using a high-level of abstraction, TLM leads to a large gain of simulation speed and modeling productivity, which enables new design methodologies. The main goal of a TLM model is to be a common reference model for teams working on software, hardware, architecture analysis, and verification. For this reason, it should appear very early in the design flow, preferably right after HW-SW partitioning.

The second key concept is the separation between computation and communication. In a TLM representation, modules contain concurrent processes that execute their behaviors, whereas communication is performed through packet exchanging among these modules, which are called *transactions*. Communication is implemented inside channels, hiding the protocols from the modules, but exposing their interfaces.

Although TLM is language independent, SystemC fits perfectly its style of representation by allowing adequate levels of abstraction and by providing elements for supporting separate computation and communication. The connection between SystemC and TLM has grown so strong in the recent years that OSCI created a special TLM Working Group to define the SystemC TLM standard [7].

A TLM-based representation fits well in an ESL design flow. As discussed in Chap. 1, there are some requirements that a design representation should meet to be suitable for ESL design, which are:

- provide a unique representation of the platform architecture;
- capture parameters handled by hardware-dependent software;
- allow advance performance evaluation on the target platform;
- allow advance power evaluation on the target platform;
- rely on encapsulation of processor models;
- rely on encapsulation of non-programmable components;
- support platform debugging;
- enable virtual prototyping.

All these requirements were considered in the principles of TLM [2], as we are going to further discuss in the following sections.

The key to successful TLM usage is a good architecture planning. Designers should partition hardware into key pieces and model every piece as a SystemC module. TLM is then used to model communication between those modules by using high-level function calls to replace the expensive and slow RTL signal-based communication (which models every event on the system). For a full functional model of the whole system, software is then integrated into the TLM platform. After proper validation, the resulting TLM representation can become a golden reference model for further phases of the design flow (e.g. RTL model validation).

3.2 The Evolution Towards the OSCI TLM 2.0 Standard

Since its version 2.0 was released in 2001, SystemC has been enabling a TLM approach for model development. At that time, all the key language constructs for enabling TLM, like channels, ports, and interfaces, were already available.

The basic hierarchical unit in SystemC is a *module*. Inside a module, the computation is implemented by *processes* (methods or threads). Communication is implemented through *channels*, which can appear in a variety of complexity levels. A channel can be as simple as a wire or be hierarchically designed to comprise sophisticated communication mechanisms. Modules are interconnected through channels. Channels implement *interfaces*, which expose to the modules the methods available to perform communication. *Ports* are the binding point between modules and channels. Therefore, every transaction between two modules can be reduced to the invocation of a function.

At certain point in the evolution of SystemC support for TLM, it became clear that a commonly accepted standard would be required for the whole SystemC community so that software, hardware, and verification engineers could be able to achieve the expected productivity levels. As a result, OSCI founded the so-called TLM Working group (TLMWG) [7], which became responsible for the definition of the envisaged standard. This group included representatives of several companies, such as STMicrolectronics, Mentor Graphics, Cadence, and ARM, among others. In 2005, the 1.0 version of the TLM Standard was approved by OSCI.

Interoperability is a key factor to ESL design because it improves the chances of in-house and third-party IP reuse for the building of virtual platform models. Unfortunately, the experience proved that TLM 1.0 was not able to strongly promote TLM model interoperability and did not offer a good balance between model development time, speed, and accuracy.

TLM 1.0 had a set of functions to standardize both blocking and non-blocking communication interfaces. However, there was no standard to define which data structures (or class), would carry the data on transactions. Each designer was responsible for defining his own data structures to compose the packets transmitted in each transaction. Each IP designer or provider ended-up creating its own data payload format, which had a clear impact on IP module reuse. The usual solution was to include a wrapper (or transactor) between two IPs in order to translate between the different payload formats, thereby establishing the communication. This makeshift not only increased the modeling effort, but also resulted in a penalty on simulation performance.

Although interoperability was the main missing point in TLM 1.0, two additional issues hampered its widespread adoption: there was no timing annotation mechanism and there were some performance limitators. Time delays had to be implemented by calls to the `wait()` method, which caused context switches and, as a consequence, slowed down the simulation. Every transaction needed to pass its data as value or `const` references, further increasing simulation times. To work around those issues, designers had to rely on non-standard solutions. Therefore, interoperability and model reuse were indirectly affected.

As virtual platforms and TLM continued to grow in importance, specially in performance analysis and design space exploration, the TLMWG kept working on developing the libraries for TLM in SystemC. Its main goal was to make interoperable TLM a reality for SystemC users. The 2.0 version of the SystemC TLM standard was finally released in June 2008. The next section introduces its main functionalities.

3.3 Main Features in the TLM 2.0 Standard

TLM 2.0 was developed with memory-mapped bus applications as its main target. This type of application simulates communication and demands concurrent execution, which is achieved by means of several SystemC processes. Therefore, TLM 2.0 developers made clear that pure algorithmic models, those which can be simulated by a single thread and require no inter-process communication, were not a main concern on the definition of the standard [5]. Since designers could rely on TLM 1.0 core interfaces to model purely untimed systems, TLM 2.0 focused on providing proper interfaces for two different coding styles: *loosely-timed* (LT) and *approximately-timed* (AT) [4].

As clearly stated in the OSCI's TLM 2.0 User Guide, users could come up with other coding styles, but these two must be enough to model any use case to which the development of TLM 2.0 was targeted for. The choice between LT and AT styles depends on what is expected from the models, as we are soon going to discuss in detail; a good rule of thumb is that, for starting a new design from scratch, it is advisable to first build a LT model, and then refine it towards an AT model if necessary.

LT descriptions include enough details to model the communication, but without considering timing issues. On the one hand, this level of detail is sufficient to model interrupts and to allow the booting of an OS over a multicore platform. On the other hand, it is not suitable for bus development due to the lack of timing. LT platform models are capable of modeling functionalities for components, debug, and memory maps. This makes them suitable for functional verification, and early software development. LT models were usually referred as Programmer's View (PV) models in former SystemC and TLM literature [1–3]. The LT coding style is better supported by the blocking transport interface in the TLM 2.0 standard.

Approximately-timed (AT) models present every process running locked to the simulation time. These models are suitable for architecture exploration and performance analysis, since they can include resource contention and arbitration. In the AT coding style, transactions are broken into multiple phases, marked by timing points. The TLM 2.0 standard supports timing annotation and multiple timing points by means of its non-blocking transport interface. It is important to mention that, although user-defined protocols may have multiple timing points, to be fully compatible with the generic payload, the designer must be aware that the base protocol has four points marking the beginning and end of both the request and the response.

Since time penalties increase with timing accuracy, designers usually start from a full LT model (no timing at all). As the design evolves, modules with some time accuracy are added, either via decomposition (model partition) or through refinement (adding details to make it more precise). To keep the simulation speed benefits offered by LT abstraction, different teams use different platforms and load time-accurate models only when and where needed. This generates an *hybrid model*, where parts with no notion of time are synchronized with accurate parts through the use of time wrappers or explicit synchronization.

Every SystemC model that uses TLM must assume the role of an initiator or a target. An *initiator* is responsible for assembling the payload and sending a request packet through a function call. A *target* receives the payload and acknowledges it, returning the call. After processing the data, the target sends a response packet, if applicable. Modules that are both an initiator and a target are modeled as they where one or another at a given time, even if the time between a send and a receive is null. An example of such a module is the so called interconnect modules, whose sole functionality is to transfer transactions packets from its initiator socket to its target socket, without modifying its contents.

To become an initiator or a target, a SystemC module only needs to supply a connection socket and implement one of the four interfaces, according to the type of communication desired. The four transport interfaces are the blocking and non-blocking versions for forward, initiator to target, and backward, target to initiator, communication. These interfaces may be specialized.

The blocking version is used with LT models. In this case, target modules receive a request packet and must return a response packet. Initiators stay blocked until the target returns from the call.

The non-blocking interface is more suitable to AT models. The protocol is divided into phases, where both initiator and target take actions. The initiator module begins a communication in the same way as it would do with a blocking transport interface, but the target module may return without a response, freeing the initiator to continue execution. The initiator should accept target packets, that should be the response to the initiator request. The modeling structure for both types are shared and unified. This allows the same model to be used as blocking or non-blocking without changes. Both methods allow timing annotation.

As stated before, all communication using TLM is composed by a payload that is transferred from one side to another by means of a function call. To promote full interoperability among IP modules, TLM 2.0 has a fixed guideline for payloads. It is called *generic payload*, and contains fields for a command, an address, data, byte enables, and response status. These attributes were intentionally chosen to suit memory-mapped bus models.

It could be argued that this packet structure is too restrictive. In practice, only a few transaction-level models are likely to need something different from a masked data/address request-response packet. Therefore, it is largely enough to model the use cases which the standard was targeted for. Nevertheless, it is still possible to design specific structures for protocols that demand specialized attributes, by extending the generic payload. However, the less specialized the structure, the higher the

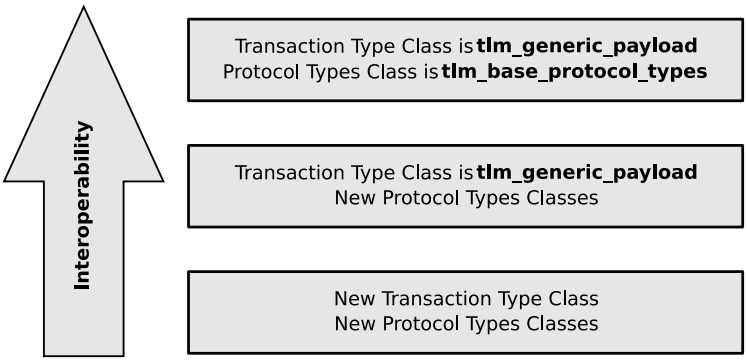


Fig. 3.1 Relation between interoperability and payload customization

amount of modules compatible with a given design, thereby promoting interoperability and reuse, as clearly stated in Chap. 6 of the OSCI TLM-2.0 User Manual [5]. This notion is illustrated in Fig. 3.1, which shows three alternatives recommended by OSCI (for designing transaction and protocol types classes for TLM models), and their increasing level of interoperability.

TLM 2.0 defines a generic payload and three kinds of interfaces: *transport*, *DMI*, and *debug*. Interfaces were developed to readily supply most, if not all, cases of communication. As discussed above, the *transport interface* consists of a request-reply protocol with blocking and non-blocking versions.

The *DMI interface* is also called by LT models. DMI returns a pointer to the requested memory location, thereby allowing a faster access, bypassing all communication stack, including function calls. Concurrency is solved by reserving memory chunks for exclusive access and invalidating those pointers after use.

The *debug interface* gives an initiator access to memory without delay or side-effect. It passes a normal payload to memory, which is instantly processed. It is not intended for regular usage, since a memory access seems as if it never happened. This interface can be called from both LT and AT models.

In order to better understand the interoperability issues involving the three alternatives shown in Fig. 3.1, let us first take a look at the so-called combined interfaces [5] in TLM 2.0. Figure 3.2 shows an excerpt of the declaration code that may be found in the `tlm_fw_bw_ifs.h` file in the TLM 2.0 distribution released by OSCI in 2008. First, the protocol types' structure is declared (Line 1). Notice that it uses the standard generic payload for data transfer and also the standard phase type for controlling transaction phases. This protocol type classes are passed to the TYPES argument in the interfaces declaration (Line 8). The combined interfaces group the transport, DMI, and debug interfaces. Figure 3.2 shows the forward combined interface declaration using the types defined in the structure above. The backward interface declaration follows the same pattern.

Temporal decoupling is allowing SystemC processes to run ahead of the simulation time, which consists in controlled by the SystemC scheduler. The idea is to speed up simulation by allowing processes to run up to the point where one needs

```
struct tlm_base_protocol_types
{
  typedef tlm_generic_payload tlm_payload_type;
  typedef tlm_phase tlm_phase_type;
};

// The forward interface:
template <typename TYPES = tlm_base_protocol_types>
class tlm_fw_transport_if :
  public virtual tlm_fw_nonblocking_transport_if<typename
      TYPES::tlm_payload_type, typename TYPES::
      tlm_phase_type>,
  public virtual tlm_blocking_transport_if<typename TYPES
      ::tlm_payload_type>,
  public virtual tlm_fw_direct_mem_if<typename TYPES::
      tlm_payload_type>,
  public virtual tlm_transport_dbg_if<typename TYPES::
      tlm_payload_type>
{};
```

Fig. 3.2 Combined interfaces declaration. (Extracted from OSCI TLM 2.0 distribution)

synchronization or interaction with another process, also reducing the scheduling overhead by decreasing the amount of context switches and events. Temporal decoupling only makes sense if associated to the LT coding style, since AT models have a much stronger dependency between timing and data.

A typical case that creates a synchronization barrier is a process reaching a point where there is a dependency on a value that comes from another process. This value may not have been updated yet. In a SystemC temporal decoupled implementation the process has two choices: accepting the current value or returning the control to the SystemC kernel. The first option allows the process to continue the simulation, but is up to the process to determine if no harm will be done to the correctness of the model.

3.4 A Small TLM Platform Example

To conclude this chapter, we exemplify TLM usage by means of a small platform example. This platform is composed by a PowerPC (PPC) processor model (downloaded from the ArchC website [6]), a memory, and a debug IP. The PPC processor runs a “hello world” program stored on a TLM memory. The debug IP does nothing until the end of the simulation, when it dumps the memory’s content to a file just before simulation ends. This platform can be viewed at Fig. 3.3.

Let us first analyze the processor side. The ArchC processor simulator can be generated with an internal memory or a TLM port to connect to external storage devices. In this example, we are going to connect it to a memory IP using

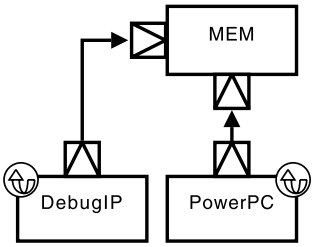


Fig. 3.3 Platform example: a PPC processor, a memory, and a debug IP

```
1  AC_ARCH(powerpc) {
2
3          ac_wordsize 32;
4
5          ac_tlm_port MEM:8M;
6
7          ac_regbank GPR:32;
8
9          ac_reg SPRG4;
10         ac_reg SPRG5;
11     ...
12 }
```

Fig. 3.4 Modified PowerPC architecture description in ArchC

the TLM 2.0 transport interface. In order to do that, we just have to change the line `ac_mem MEM:8M;` to `ac_tlm_port MEM:8M` on `powerpc.ac` file, as showed in Fig. 3.4 (Line 5). This will generate a simulator instrumented with a TLM communication port using the same name as the memory but with a `_port` suffix, becoming `MEM_port`. The value (8 MB) is used internally by ArchC to instantiate the memory. For TLM, it should still be specified for bound checking purposes. Also, the stack is supposed to be at the end of the memory.

This port is a generic payload initiator socket, as already discussed in this chapter. The ArchC simulator generator will translate the declaration into a generic payload initiator socket declaration included in the final PPC model. It can be seen among the storage device declarations in Fig. 3.5 (line 3), which is a piece of code extracted from the `powepc_arch.cpp` file generated by ArchC.

This initiator socket should be binded to any TLM 2.0 compatible target socket, in this case implemented by the (`simple_memory`) class depicted in Fig. 3.6 (line 19). The `simple_memory` memory class implements three TLM interfaces: blocking transport (line 20), DMI (line 21), and debug (line 22). We discuss how these interfaces are used later in this section.

Figure 3.7 shows a piece of code extracted from the (`sc_main`) function implementation for this platform. The platform components are instantiated in lines 7

```
1   //!Storage Devices.
2
3    tlm_utils::simple_initiator_socket<ppc> MEM_port;
4
5    ac_memport<powerpc_parms::ac_word, powerpc_parms::
         ac_Hword> MEM;
6    ac_regbank<32, powerpc_parms::ac_word, powerpc_parms::
         ac_Dword> GPR;
7    ac_reg<powerpc_parms::ac_word> SPRG4;
8    ac_reg<powerpc_parms::ac_word> SPRG5;
9    ac_reg<powerpc_parms::ac_word> SPRG6;
10
11      ...
```

Fig. 3.5 TLM port declaration inside the PPC architecture resources description

and 8. The connection between the processor and the memory is accomplished by the port binding in line 10.

Every time it is invoked, the PPC will issue a memory request using either the blocking transport protocol or DMI. The first access is always done through the blocking transport interface. The memory answers the request using the same packet assembled by the processor. Figure 3.8 shows the generic payload packet declaration in line 1 and assembling in lines 3 to 10. The package declaration follows the generic payload standard, without any customization. The packet assembling is used by the processor to assemble a read request, although a write request is very similar. The key points in this packet are:

- data lenght: it is always 4 bytes in ArchC;
- streaming width: must have the same size as the data (ArchC does not use burst requests);
- byte mask: set to zero to indicate non-usage;
- the last two fields (which are mandatory): set to `no DMI` and `no response` yet.

In our example, the memory IP supports DMI, so the field `set_dmi_allowed` that must be set to `false` by the processor will be set to `true` when the memory returns. This is a hint to the processor that the memory supports DMI and that it should be used to speed up simulation. The answer is expected to change to a `tlm::TLM_OK_RESPONSE` or an error pattern (there are a few patterns specified by OSCI and the user may generate his own error code).

After receiving the response, the processor verifies if DMI is supported and issues a request to a DMI pointer as shown in Fig. 3.9. The structure `dmi_packet` is declared as private to the processor class by means of a `tlm::tlm_dmi` variable. The method `get_direct_mem_ptr` is implemented by the memory and is supposed to return `true` if a valid pointer to the beginning of the memory is available at `dmi_packet.get_dmi_ptr()`. For now on, the processor can access the memory directly using `memcpy` instead of sending new TLM requests.

```
#ifndef SIMPLE_MEMORY
#define SIMPLE_MEMORY

#define SC_INCLUDE_DYNAMIC_PROCESSES

#include "systemc"
using namespace sc_core;
using namespace sc_dt;
using namespace std;

#include "tlm.h"
#include "tlm_utils/simple_initiator_socket.h"
#include "tlm_utils/simple_target_socket.h"

class simple_memory:
    public sc_module {

public:
    tlm_utils::simple_target_socket<simple_memory> socket;
    void b_transport( tlm::tlm_generic_payload& trans,
        sc_time& delay );
    bool get_direct_mem_ptr(tlm::tlm_generic_payload&
        trans, tlm::tlm_dmi& dmi_data);
    unsigned int transport_dbg(tlm::tlm_generic_payload&
        trans);
    void direct_write(uint32_t* d, uint32_t a);
    simple_memory( sc_module_name module_name , int k =
        5242880, int t = 0);
    ~simple_memory();
private:
    uint8_t *memory;
    sc_time latency;
    unsigned int size;
};

#endif /*SIMPLE_MEMORY*/
```

Fig. 3.6 A memory implementing transport, DMI, and debug interfaces

Moreover, the processor implements a method `invalidate_direct_mem_ptr`, which is used by the memory in the same way as the processor requested the DMI pointer, but to invalidate any pointer that the processor may be holding. In such platform, this cannot occur because the processor is the only master module. However, in a concurrent environment with more than one master, this invalidation could occur. DMI can be allowed with restrictions like read-only, or can be allowed only for a few address ranges, not for the full memory. In this example, the memory allows access to everything, as depicted in Fig. 3.10. If time accuracy is being used, the property latency hints the processor to annotate the corresponding timing.

```
int sc_main(int ac, char *av[])
{
    simple_memory*  MEM;
    powerpc*        ppc;

    // Instantiate components
    memory    = new simple_memory("mem",8688608);
    ppc       = new powerpc("processor");

    ppc->MEM_port.bind( MEM->socket );

    ...
}
```

Fig. 3.7 Module instantiation and TLM port binding

```
tlm::tlm_generic_payload* packet = new tlm::
    tlm_generic_payload;

packet->set_command( tlm::TLM_READ_COMMAND );
packet->set_address( address );
packet->set_data_ptr( reinterpret_cast<unsigned char*>(&
    data) );
packet->set_data_length( 4 );
packet->set_streaming_width( 4 );
packet->set_byte_enable_ptr( 0 );
packet->set_dmi_allowed( false );
packet->set_response_status( tlm::
    TLM_INCOMPLETE_RESPONSE );
```

Fig. 3.8 TLM generic payload usage

```
if ( packet->is_dmi_allowed() )
{
  dmi_suported = MEM_port->get_direct_mem_ptr( *packet,
      dmi_packet );
}
```

Fig. 3.9 TLM DMI request

The memory IP implements `b_transport`, which is the method that will answer the processor's read and write requests and `transport_dbg`, which implements the debug interface. The blocking transport is very simple, consisting only in verifying if the packet is a read or a write (using the same field set by the processor) and doing a proper `memcpy` to or from the internal vector that represents the memory structure itself.

```
bool simple_memory::get_direct_mem_ptr(tlm::
    tlm_generic_payload& packet, tlm::tlm_dmi& dmi_packet)
    {
    dmi_packet.allow_read_write();
    dmi_packet.set_dmi_ptr( reinterpret_cast<unsigned char
        *>( &memory[0] ) );
    dmi_packet.set_start_address( 0 );
    dmi_packet.set_end_address( size-1 );
    dmi_packet.set_read_latency( latency );
    dmi_packet.set_write_latency( 2*latency );
    return true;
  }
```

Fig. 3.10 TLM DMI response

The debug interface uses the same generic payload as `b_transport`, but takes no time at all. The debug IP at this platform issues a read request using the debug interface to a specific memory position every time it is scheduled. The PPC processor writes a true value at this position just before program ending, so that the debug IP monitors the memory position for a change and, afterwards, use the same debug interface to read every address from the memory and dump its value to a file, before exiting.

The reader is strongly encouraged to try to reproduce this small platform example using SystemC and TLM. This exercise will provide the proper background for a complete understanding of the more advanced platform-based design concepts to be discussed in depth throughout this book (especially in Chap. 5 and Chap. 7).

References

1. D.C. Black, J. Donovan, *SystemC: From the Ground Up* (Springer, Berlin, 2004)
2. F. Ghenassia, *Transaction-Level Modeling with Systemc: TLM Concepts and Applications for Embedded Systems* (Springer, New York, 2006)
3. T. Grötker, S. Liao, G. Martin, S. Swan, *System Design with SystemC* (Kluwer Academic, Dordrecht, 2002)
4. Open SystemC Initiative, Requirements Specification for TLM 2.0. Available at http://www.systemc.org/downloads/standards/, September 2007
5. Open SystemC Initiative, OSCI TLM 2.0 User Manual, June 2010
6. The ArchC website (2010). http://www.archc.org
7. The SystemC TLM Working Group website (2010). http://www.systemc.org/apps/group_public/workgroup.php?wg_abbrev=tlmwg

Part II
Open-Source Models and Tools

Chapter 4
ArchC Model Design Handbook

Rodolfo Azevedo and Sandro Rigo

4.1 What Is a Model?

Before starting a processor description, one must define the required abstraction level. In this section, we are going to briefly describe some of the possible abstraction levels and the implications of designing in such levels. The designer must also be concerned about his ability to refine the model from a higher to a lower abstraction level, that is the direction we are going to describe the levels:

Untimed Level In this level, all the instructions are executed correctly, but there is no time information. It is equivalent to take one cycle to execute each instruction, but even this single cycle is not taken into account, only the number of instructions executed which can be easily converted to time if the first assumption is correct. It is good to start coding at this level to tackle the processor complexity before you start inserting time to it. The ArchC models at this level are called functional models and they can run almost all application-level software. To run Operating Systems, you will need some external timers and a way to couple them to the untimed model, probably refining your processor to the Loosely Timed Level.

Loosely Timed Level Some time information is embedded into the model. Although the information is not very precise, you can get some time statistics and can also start coupling the processor to timed systems without much problems. To get such time information you can, usually, add some counters to your model and some `wait` clauses in your SystemC code to enable that. Usually, this counters are incremented by every instruction using the instruction average time, not

R. Azevedo (✉) · S. Rigo
Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil
e-mail: rodolfo@ic.unicamp.br

S. Rigo
e-mail: sandro@ic.unicamp.br

S. Rigo et al. (eds.), *Electronic System Level Design*,
DOI 10.1007/978-1-4020-9940-3_4, © Springer Science+Business Media B.V. 2011

the real time executed for every instance. This time information can be enough for most of the requirements unless you have a complex timing requirement external to your model that you need to follow (like a full timed bus). Notice that adding `wait` clauses slow down the model execution time. To reduce the slowdown, you can also use the TLM 2.0 `quantumkeeper` to call `wait` once every *N* cycles. If you want more precision, you need to go to the Timed Level.

Timed Level Although you cannot synthesize this model, it must behave exactly as a real circuit implementation regarding to time. Every instruction must have its precise time assigned and the model should advance the simulation time accordingly. You can couple this model with a full timed bus and control an external peripheral completely timed using software instructions to make the delays. The so called ArchC cycle-accurate model is in this abstraction level. The next step is to go to synthesis by using the RTL level.

RTL If you have your model at the RTL level, it means that you have a synthesizable representation of it. You can use EDA tools to transform your model into hardware.

ArchC can help designers in the first three mentioned levels and this chapter will guide them mostly in the first level, giving tips for the other two. As examples, we are going to use already available models that can be freely downloaded from the ArchC web site at http://www.archc.org.

ArchC defines versions for processor models taking into account their stability level. Table 4.1 shows the version number each model will pass and how to certify this versions. The first and second digits are reserved for the roadmap, the last one can be used to indicate intermediate development between two versions. This roadmap indicates that, to aquire stability, the model must execute a huge amount of instructions in an organized fashion, and generate the correct result. The minimal stability expected for a model to run big programs is version 0.5.0.

The next sections will guide the designer through the model development process following this roadmap.

4.2 Start Modeling—Architectural Information

The fist step to model a processor is to describe the architectural resources that it will use like register file, isolated registers, memories, and caches. In this part we will also describe some processor features like endian, wordsize, and the way to inform a second file that contains the instruction set description. All these informations are described inside the first ArchC file, in the `AC_ARCH` block. For more information about syntax, you should refer to Chap. 2 and the ArchC Language Reference Manual (LRM) [5].

The first step into modeling is to gather some information about the processor. Table 4.2 gives a brief overview of the information needed for the processors we are going to use in this Section. Figure 4.1 shows the code for the PowerPC `AC_ARCH` description. Notice that, `AC_ARCH` requires one parameter that is the processor name. This name will prefix several filenames and variables in the ArchC simulator.

Table 4.1 ArchC model development roadmap

Version	Stage	Benchmark	Certifies that
0.0.x	`ac_isa` and `AC_ARCH`	–	–
0.1.0	`ac_isa` and `AC_ARCH` declared	–	All instructions are correctly decoded
0.2.0	Instruction behavior	–	Individual behaviors are working properly
0.3.0	`ac_isa` and `AC_ARCH` completed	acStone	All programs passed successfully
0.4.0	ABI design finished	–	Individual system calls are working properly
0.5.0	Model description completed	Mediabench	All selected programs passes successfully
0.6.0	Testing...	MiBench (small version)	All selected programs passed successfully
0.7.0	Testing...	MiBench (large version)	All selected programs passed successfully
1.0.0	Final test	SPEC 2000	All selected programs passed successfully

Table 4.2 Architectural characteristics of the processors

Characteristic	PowerPC	MIPS	SPARC	8051	PIC16F84	ArchC keyword
Word Size	32	32	32	8	8	`ac_wordsize`
Register Bank Size	32	32	256/32	–	256	`ac_regbank`
Simulator Memory	8 MB	5 MB	5 MB	64 KB	5 MB	`ac_mem`
Endian	big	big	big	–	–	`set_endian`

```
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. 4.1 Excerpt of the PowerPC `AC_ARCH` description

From Table 4.2, one can see the PowerPC word size, which is the default size for all ArchC registers, operators, and memory operations. This is declared in Line 2 of

```
1  AC_ARCH(i8051){
2    ac_cache   IRAM:255;
3    ac_cache   IRAMX:64K;
4    ac_icache  IROM:64k;
5    ac_wordsize 8;
6    ARCH_CTOR(i8051) {
7      ac_isa("i8051_isa.ac");
8      set_endian("big");
9    };
10 };
```

Fig. 4.2 Excerpt of the 8051 AC_ARCH description

Fig. 4.1. This description is inside the AC_ARCH block (Line 1) as all the others in this section.

Although the memory is not expected to be inside the processor, ArchC allows the designer to declare a memory in AC_ARCH for standalone processor simulator. By doing so, it is possible to use the processor to run your code before integrating it to the platform and build a bigger simulator. The memory declaration is done by the ac_mem clause, as shown in Line 3 of Fig. 4.1. This is the typical way of declaring memories for Von-Neumman architectures, where both data and instructions reside in the same memory. The M suffix in the memory size specify MB. For Harward architectures, you have two options: You can declare two different memories using ac_mem and ArchC will use the first as the instruction memory. The second alternative is to use the ac_icache for instructions and the ac_dcache or ac_cache for data. You can use a syntax identical to the ac_mem as shown in Fig. 4.2 Lines 2–4.

TIP: *For ArchC, memory is always byte addressed, if the processor requires other kind of addressing, it is up to the designer to provide such interface in the instruction set implementation.*

The next step is to declare the register banks and the standalone architecture registers. PowerPC has 32 registers of 32 bits, since the ac_wordsize has already been defined, it is only necessary to declare the general purpose registers (GPR) as in Line 4 in Fig. 4.1. The 8051, showed in Fig. 4.2, does not have an internal register file; it declares only a small external memory that holds a few data registers together with peripheral registers. The SPARC processor (Fig. 4.3) is a little different because it has a big register file with a register window. In this case, the register bank (RB) was declared with 256 registers (Line 3 of Fig. 4.3) and a smaller version (REGS) was declared for processor optimization (Line 4).

The standalone architecture registers are declared using the ac_reg keyword. There are three different kind of ac_reg declarations, the first and simplest one is to declare a register using the architecture word size. Line 5 of Fig. 4.1 shows the declaration of the Machine State Register (MSR) as a 32-bit register. The second alternative can be found in the Fig. 4.3, Lines 6–8, where three registers are declared with different sizes, like the Current Window Pointer (CWP) declared with 8 bits, using the C++ template syntax. The third and last way consists in applying a specific

```
1  AC_ARCH(sparcv8){
2    ac_mem      DM:5M;
3    ac_regbank RB:256;
4    ac_regbank REGS:32;
5    ac_reg npc;
6    ac_reg<1> PSR_icc_n;
7    ac_reg<8> WIM;
8    ac_reg<8> CWP;
9    ac_wordsize 32;
10   ARCH_CTOR(sparcv8){
11     ac_isa("sparcv8_isa.ac");
12     set_endian("big");
13   };
14 };
```

Fig. 4.3 Excerpt of the SPARCv8 AC_ARCH description

field format to the register. In this case, the register will be broken in several fields to be accessed individually. To specify the format, replace the register size with the format's name. Formats can be declared with the keyword ac_format, which will be described in the Sect. 4.3.

One must notice that the Program Counter is not declared. ArchC automatically declares ac_pc for every design and also fetches the instruction pointed by this register when necessary. Since the SPARC architecture requires two program counters (npc is the other one) to handle branch delay-slots, npc is declared in Line 5 of Fig. 4.3.

To preserve the look & feel of SystemC, ArchC also requires a block that looks like a constructor for the architecture. This block is the ARCH_CTOR, presented in Line 6 of Fig. 4.1. It takes the processor name as a parameter. The name must match the AC_ARCH name.

Inside the ARCH_CTOR, there are, basically, two different commands. The first, ac_isa specifies the file that describes the Instruction Set (Line 7 of Fig. 4.1). The second specifies the architecture endian with the keyword set_endian. All the examples mentioned used big-endian. The 8051 has word size 8, so it could accept both endians.

After declaring the AC_ARCH section, the user must prepare the Instruction Set description in the file specified by ac_isa. The next section will cover the Instruction Set declaration.

4.3 Declaring Instructions

For now on, we will focus on the instruction declarations.

The Instruction Set must be described in the file specified in the ac_isa clause of the AC_ARCH file as mentioned before. Figure 4.4 shows an excerpt of the PowerPC ISA file. To better explain it, two formats are described in Table 4.3 and Table 4.4. Every field in the format must appear in the declaration. The format size is

Table 4.3 PowerPC instruction type I1 fields format

Field Name	opcd	li	aa	lk
Field Size	6	24	1	1
Signed/Unsigned	U	S	U	U
Syntax	%opcd:6	%li:24:s	%aa:1	%lk:1

Table 4.4 PowerPC instruction type X14 fields format

Field Name	opcd	rs	0x00	rb	xog	0x00
Field Size	6	5	5	5	10	1
Signed/Unsigned	U	U	U	U	U	U
Syntax	%opcd:6	%rs:5	0x00:5	%rb:5	%xog:10	0x00:1

```
1  AC_ISA(powerpc) {
2   ac_format I1="%opcd:6 %li:24:s %aa:1 %lk:1";
3   ac_format X14="%opcd:6 %rs:5 0x00:5 %rb:5 %xog:10 0x00:1";
4   ac_format XO1="%opcd:6 %rt:5 %ra:5 %rb:5 %oe:1 %xos:9 %rc:1";
5   ac_instr<I1> b, ba, bl, bla;
6   ac_instr<XO1> add, add_, adc, mullw, divw, subf;
7   add.set_decoder(opcd=31, oe=0, xos=266, rc=0);
8   add_.set_decoder(opcd=31, oe=0, xos=266, rc=1);
9   addc.set_decoder(opcd=31, oe=0, xos=10, rc=0);
10  ...
11 }
```

Fig. 4.4 Excerpt of the PowerPC `ac_isa` description

the sum of the sizes of all the fields. Both examples have 32-bit sizes. For architectures that have variable-size instructions, each different size must have, at least, one format associated to it, as shown in Fig. 4.5. The fields default to unsigned values. If a field is signed, as field `li` in Table 4.3, it must be explicitly specified.

Some instruction formats have constant fields, like X14 in Table 4.4, where `0x00:5` represents a 5-bit field with all bits 0. Non-zero values can be declared in the same way. Figure 4.4 shows the `ac_format` clauses for these formats.

TIP: *You can only repeat a field name in a different format if the field has the same size and is positioned in the same bit offset in the instruction. This means that field names must represent exactly the same piece of every instruction they appear. This restriction exists to simplify the instruction decoder description. In this way, it is common to have some fields, usually the immediates, suffixed with a number indicating its size (like `imm22` and `imm7` in Fig. 4.6).*

There are some situations where the format must be broken in two just because one or two different fields. ArchC provides an alternative that allows the creation of only one format with overlapping fields. One example is shown in Line 8 of Fig. 4.6. The alternative fields are placed inside brackets and separated by a vertical

```
1  AC_ISA(i8051)
2  {
3   ac_helper
4   {
5    sc_uint<17> pc;
6    int reg_indx;
7    unsigned long pc_stability;
8    unsigned long old_pc;
9    unsigned long curr_pc;
10  };
11  ac_format Type_3bytes = "%op:8 %byte2:8 %byte3:8";
12  ac_format Type_2bytes = "%op:8 %byte2:8";
13  ac_format Type_OP_R = "%op1:5 %reg:3";
14  ac_format Type_IBRCH = "%page:3 %op2:5 %addr0:8";
15  ac_format Type_1byte = "%op:8";
16  ...
17 }
```

Fig. 4.5 Excerpt of the i8051 `ac_isa` description

bar character. In the example, the format `Type_FT` can have the fields `r2a` and `rs2` or `r2b` and `imm7`.

After all the formats have been declared, it is time to declare the instructions. ArchC requires that instructions with different decoding bits or formats to be declared separately. Instructions are declared using a C++ template style as in Lines 5–6 of Fig. 4.4. The names and number of instructions per line are just related to organization issues. It is good to have instruction names related to the instruction mnemonic but you usually will not be able to always use the mnemonic, mainly when the processor has several instructions with the same mnemonic as in Lines 11–12 of Fig. 4.6.

TIP: *Beware of the C++ reserved words that can not be used as instruction names, mainly the logic operators* `and`, `bitand`, `compl`, `not_eq`, `or_eq`, `xor_eq`, `and_eq`, `bitor`, `not`, `or`, `xor`.

This file may also contains declarations required to automatically retarget the binary tools for this architecture. This is the focus of the Chap. 6 and will not be explained here.

Similar to `AC_ARCH`, the `ac_isa` block also has a constructor called `ISA_CTOR`. Inside it, there should be the decoding sequence for all the instructions. For ArchC, a decoding sequence is a sequence of fields and values that uniquely identifies each instruction. ArchC uses this information to create a decoding tree that will be embedded into the simulators. Lines 7–9 of Fig. 4.4 shows the way the `set_decoder` keyword must be used.

The decoding tree can be seen as a chainned sequence of `switch` statements in C, where the sequence of fields specified in the `set_decoder` clauses. Figure 4.7 gives an overview of how the decoding tree can be implemented. In the figure, every three capital letter values should be changed by the respective field

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

Fig. 4.6 Excerpt of the SPARC `ac_isa` description

```
switch (op) {
  case XXX: switch (op2) {
              case ZZZ:
              case WWW:
              case KKK:
              default: // unknown instruction
            }
  case JJJ: switch (op3) {
              case ZZZ:
              case WWW:
              case KKK:
              default: // unknown instruction
            }
  default: // unknown instruction
}
```

Fig. 4.7 Simplified version of a decoder implementation

value. This sort of decoding tree usually gets implemented as jump tables by the compilers improving the speed compared to a sequence of if statements.

To improve simulation performance, ArchC uses a *decoder cache* to handle the already decoded instructions. The decoder cache works as follows: everytime one instruction is accessed for the first time, it is decoded using the aforementioned steps. After that, the simulator stores the decoded data structures in a cache for future use. The next time the same memory address is requested, the decoded data structure will be available.

This method has a problem when dealing with self morphing code. In this case, it is recommended to use the `acsim` option `--no-dec-cache` or `-ndc` to disable the decoder cache. Unfortunatelly, the simulator speed will be slower in this case.

Since the designer writes the decoder sequences instruction by instruction, it is common to make some mistakes and, in that case, the simulator will run different instructions or complain about an undeclared instruction. The following are the most common bugs:

- The designer mistypes one of the field values in the `set_decoder` and two different fields get the same decoding sequence. In this case, although implementation dependant[1], the simulator will decode the instruction as the first declared one.
- The designer declares one instruction with a decoder sequence that is a subset of another instruction. Although this could not be technically described as a bug, the result can be different depending on the sequence of the instructions declaration. As can be seen in Fig. 4.7, the current implementation may select the instruction with the smaller decoding sequence instead of the longer one. To solve that, you should declare the instruction with the longer decoder sequence first.
- The designer declares two instructions with different decoder sequences with fields in different orders. This can be better explained in an example: suppose two instructions with 3 fields in the `set_decoder`, say field A, B, and C. Suppose the first instruction asks for a decoding sequence of A = 7, B = 10, C = 15 and the second one asks for A = 7, C = 15, B = 10. Although the decoding sequences are different, they refer to the same fields with the same values and only one of them will be decoded. Designers are recommended to use the fields in the same sequence as they appear in the format declarations.

`acsim` has a command line option called `--dumpdecoder`, or `-dd`, which dumps the decoder sequence the simulator will use. The output of this option can help the designer in detecting what is going on with the simulator.

4.4 Implementing Instructions

After the user has created the first two `.ac` files, it is now time to execute the ArchC simulator generator, called `acsim`. `acsim` is responsable for reading those files and creating a set of source files implementing a simulator. We will cover these files now. For all these examples, we are going to use the PowerPC processor. So, the two input files are named: `powerpc.ac` and `powerpc_isa.ac`.

`Makefile.archc` This file contains the rules to compile the simulator. After you created all the required files, the user needs only to execute `make -f Makefile.archc` and he will get the simulator compiled.

[1] When we say that a feature is implementation dependant, it is because there is no formal definition for such situation and we are referring to the current implementation as the common behavior. Particularly in the decoder sequence, a different approach to decode instructions could make this order infeasible.

```
1
2  //!Behavior executed before simulation begins.
3  void ac_behavior( begin ){};
4
5  //!Behavior executed after simulation ends.
6  void ac_behavior( end ){};
7
8  //!Generic instruction behavior method.
9  void ac_behavior( instruction ){};
10
11 //! Instruction Format behavior methods.
12 void ac_behavior( I1 ){}
13 void ac_behavior( D1 ){}
14 void ac_behavior( X1 ){}
15
16 //!Instruction lb behavior method.
17 void ac_behavior( add ){}
18
19 //!Instruction lbu behavior method.
20 void ac_behavior( addc ){}
```

Fig. 4.8 Excerpt of the generated powerpc_isa.cpp.tmpl template file

`main.cpp.tmpl` The ArchC simulator can be used both as a standalone application or as an instruction set simulator inside a platform. This is a template file for using ArchC as a standalone simulator. It is inside this file that you will find the `sc_main` function implementation. The Makefile described above will make a copy of this file to `main.cpp` the first time it is executed so that the user remains with the template file and can change anything he wants in the copy.

`powerpc_isa.cpp.tmpl` This is another template file, the main focus of this section. In this file the user must fill the behavior for all the instructions. ArchC does not write over any user files, that is why this file ends with `.tmpl`. To start coding, first rename it to `powerpc_isa.cpp`. We will cover more of this file contents latter on this section.

`powerpc.cpp` This is the main PowerPC simulator file. This is where the instruction decoder and the main execution loop are declared.

powerpc_arch.cpp This is where the architectural resource elements, like register bank, memories, etc., are declared.

`powerpc.H` This is the main simulator header file. Inside this file the main simulator class is declared, with exactly the same name as the processor.

`powerpc_isa.H` Declares all methods related to the instruction behaviors.

`.H files` The other .H files are header files for their respective .cpp containing glue code that can be moved to other places in the future.

The next step, after the first call to `acsim`, is to rename the `powerpc_isa.cpp.tmpl` to `powerpc_isa.cpp` and start editing it. Figure 4.8 shows a piece of this file.

For every instruction declared in the `ac_isa`, there must be a behavior implementation here. The behavior is the sequence of code required to execute the in-

```
1 void ac_behavior(instruction)
2 {
3   ac_pc += 4;
4 }
```

Fig. 4.9 Global behavior for the PowerPC model

struction, reading data from the register bank, calculating the values, and writing the results as an example.

TIP: *It is recommended that you declare all the instructions before start writing the behaviors so that you can get the full template file. In the case you forget any instruction or decide to include new instructions latter, the template file will be generated again but you should not rename it over the modified file. Instead, just copy & paste the necessary lines from the template to your final file.*

As mentioned before, ArchC has a hierarchical view for the instruction implementation. So, for each instruction, three behavior methods are executed: the global behavior, that is the same for every instruction, the type behavior, that is the same for every instruction of the same type, and the instruction behavior itself.

The global behavior for PowerPC is shown in Fig. 4.9. There is only one line that increments the Program Counter (`ac_pc`) by 4 units every instruction. Usually this method has a very simple implementation, most of the time comprising of only incrementing the `ac_pc`. But there are cases where more code is necessary, like in the ARM model, shown in Fig. 4.10.

The ARM processor has a common field in every instruction, named `cond`, which must be checked in order to execute the instruction. As an example, a value 14 in this field means that the instruction must be always executed. If the value is 0, the instruction will be executed only if the `Z` (zero) flag is set. In order to allow decisions in these situations, ArchC has a method named `ac_annul` which cancel the instruction being executed, not calling the other two behavior methods. In Fig. 4.10, the `ac_annul` is shown in Line 29. This same construction can be used in the Type behavior to cancel instruction behavior.

The Type behavior is a sequence of code that is executed for all instructions that have the same format. Most of the processors do not require such behavior. It is common on cycle-accurate models, which are out of the scope of this book.

The instruction behavior contains all the code required to complete the instruction functionality. The user is free to use any valid SystemC code inside the method. Function calls and external (global) variables are also allowed although we recommend the usage of helper functions, described latter in this section.

TIP: *The user should be advised to prefer using native C types instead the SystemC ones whenever possible since the C types simulate faster.*

Figure 4.11 shows a simple implementation of an `add` instruction in the PowerPC. Figure 4.12 shows the same code with an alternative syntax. To execute an `add`, it is necessary to read registers `ra` and `rb`, add their values and write the result to register `rt`. The second syntax is easier to understand since the register bank is accessed as a vector. The first syntax is more similar to the SystemC coding style.

```
1  void ac_behavior( instruction ) {
2    execute = false;
3    switch(cond) {
4      case  0: if (flags.Z == true) execute = true; break;
5      case  1: if (flags.Z == false) execute = true; break;
6      case  2: if (flags.C == true) execute = true; break;
7      case  3: if (flags.C == false) execute = true; break;
8      case  4: if (flags.N == true) execute = true; break;
9      case  5: if (flags.N == false) execute = true; break;
10     case  6: if (flags.V == true) execute = true; break;
11     case  7: if (flags.V == false) execute = true; break;
12     case  8: if ((flags.C == true)&&(flags.Z == false))
13                execute = true; break;
14     case  9: if ((flags.C == false)||(flags.Z == true))
15                execute = true; break;
16     case 10: if (flags.N == flags.V) execute = true; break;
17     case 11: if (flags.N != flags.V) execute = true; break;
18     case 12: if ((flags.Z == false)&&(flags.N == flags.V))
19                execute = true; break;
20     case 13: if ((flags.Z == true)||(flags.N != flags.V))
21                execute = true;  break;
22     case 14: execute = true; break;
23     default: execute = false;
24   }
25   ac_pc += 4;
26   RB.write(PC, ac_pc);
27
28   if (!execute)
29     ac_annul();
30   }
31 }
```

Fig. 4.10 Global behavior for the ARM model

```
1 void ac_behavior(add)
2 {
3   GPR.write(rt, GPR.read(ra) + GPR.read(rb));
4 }
```

Fig. 4.11 Behavior for the PowerPC add instruction (version 1)

Both alternatives are equivalent. The PowerPC `add` instruction will be executed as this sequence of three method calls:

1. `ac_behavior(instruction)`
2. `ac_behavior(D1)`
3. `ac_behavior(add)`

Figure 4.13 shows how to load data from memory to implement a *load byte* instruction. In this case, the `MEM` is accessed with the method `read_byte` to read a byte from an effective address. Notice that the behavior checks to see if the source

```
1 void ac_behavior(add)
2 {
3   GPR[rt] = GPR[ra] + GPR[rb];
4 }
```

Fig. 4.12 Behavior for the PowerPC add instruction (version 2)

```
1 void ac_behavior(lbz)
2 {
3   int ea;
4   if(ra != 0)
5     ea = GPR[ra] + (short int) d;
6   else
7     ea = (short int) d;
8   GPR[rt] = (unsigned int) MEM.read_byte(ea);
9 };
```

Fig. 4.13 Behavior for the PowerPC lbz instruction

```
1 void ac_behavior(begin)
2 {
3   REGS[0] = 0;  //writeReg can't initialize register 0
4   npc = ac_pc + 4;
5
6   CWP = 0xF0;
7 }
```

Fig. 4.14 SPARC behavior for begin of simulation

register is zero and, case positive, the register is not added to the `ea` variable. This happens because the PowerPC specifies that register 0 must have always the value 0. This check is usually not necessary for compiled code. Another alternative implementation is to not allow the `GPR[0]` to be written.

There are two other behavior methods that are called at special times, the `begin` and `end` behaviors. The PowerPC model uses the first to setup the stack pointer. The SPARC model (Fig. 4.14) uses it to clear register 0 (Line 3), to setup the `npc` register (Line 4), and to setup the `CWP` (Line 6).

The `end` behavior is usually used to print statistics or final results from the model for debugging purposes.

To implement some instructions, the user may find it useful to declare other functions or variables external to the models. The main problem with such external declarations is that they may be misused when more than one processor model is executed at the same time, as in a multicore simulator. To solve this problem, ArchC has an alternative declaration way, using helper functions and variables.

The helper block is declared inside the `ac_helper` construct in the `ac_isa` file. Since every processor in ArchC is a class, the `ac_helper` makes everything inside the block as members of the processor class. This solves the problem men-

```
1 ac_helper {
2  sc_uint<17> pc;
3  int reg_indx;
4  unsigned long pc_stability;
5  unsigned long old_pc;
6  unsigned long curr_pc;
7 };
```

Fig. 4.15 Helper variables declared in the i8051 processor

```
1 acsim powerpc.ac -abi
2 make -f Makefile.archc
3 powerpc.x --load=hello
```

Fig. 4.16 Command lines required to build a PowerPC simulator and run a binary program

tioned before, allowing more than one instance of the same processor to be instantiated at the same time. Figure 4.15 shows a sample code from the i8051 processor using `ac_helper` to declare some auxiliary variables necessary to the processor model.

Most of the time, the variables declared in the `ac_helper` clause can also be declared as registers inside the processor with the `ac_reg` clause. They are, usually, necessary when they need to be of a specific type that is not going to be explicitly declared by ArchC.

Since helper functions are members of the processor class, they can directly access all processor resources like memory, register bank, etc., without having to receive them as parameters.

4.5 Running the Simulator

After the previous steps, it is now time to create and generate the simulator for the first time. The simplest command lines to generate and run a simulator for PowerPC are shown in Fig. 4.16. Line 1 shows how to compile the PowerPC model and create the SystemC simulator. The `-abi` option indicates that the simulator will emulate the operating system ABI, that will be detailed in Sect. 4.7. The result will be a set of SystemC files and a Makefile. Line 2 shows how to build the simulator and Line 3 shows how to run it passing a binary program called `hello` and, optionally, a few arguments to this program after the program name.

TIP: *Since the* `make` *command requires a* `Makefile` *and not* `Makefile.archc`*, the user will have to use the* `-f` *option to specify the correct file. It is possible to rename* `Makefile.archc` *to* `Makefile` *to simplify the command line. But, every time the user runs* `acsim`*, it will recreate* `Makefile.archc`*. A common solution to this is to create a soft link from* `Makefile.archc` *to* `Makefile` *using the Linux* `ln` *command (*`ln -s Makefile.archc Makefile`*).*

Table 4.5 Command line options

Long	Short	Description
- -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

```
0 01bce789 0 0 019203819
10 0 0 0 0
```

Fig. 4.17 Sample HEX file

The ArchC simulator accepts several command line options, each of them with a long and small format. Table 4.5 shows the available options to generate the simulator and their meanings.

ArchC simulators implement two different program loaders. The most common is the ELF [6] loader, with which the user can load files directly from the compiler output. ELF is, most of the time, the best option for ArchC simulators. But, sometimes, there is no software toolset for the target architecture. In this case, the HEX file must be used. HEX is a very simple file format, composed of lines with addresses and values. Figure 4.17 shows a sample HEX file as an example. Every line starts with the address, in hexadecimal, followed by a set of processor words to be loaded in that and in the subsequent addresses. Notice that the values are processor words, if the processor has 32-bit words, then these values will be considered 32-bit words. There is no limit in the number of values in each line. In case of address overlap from different lines, the loader will write all the values in sequence. This can be useful to configure a peripheral in a platform. The initial Program Counter value is 0 (zero), so the simulator will start at that address.

When the user does not have a toolset, he probably will not have an operating system call emulator to stop the simulator. The simplest and easiest way to stop the simulator in this situation is to jump to outside the memory declared in the `AC_ARCH` section. If the Program Counter points to an address outside the system memory, the simulator will stop warning about it.

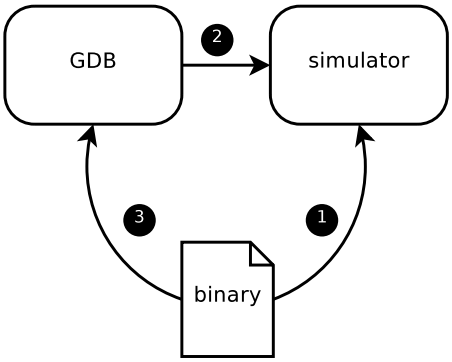


Fig. 4.18 GDB remote simulator interface

4.6 Debugging the Model—First Steps

After describing all the instructions, or a set of instructions big enough to allow program execution, the model should be ready to run some code, as explained in the previous section. Sometimes there will be bugs spread through the code that need to be detected and solved. This section will describe how to model an interface from the generated simulator with the GNU GDB [3] debugger. This interface allows the designer to use an already available GDB and connect it to the simulator to debug the program running on it. Usually, the first set of programs to be run will be the `acStone` suite, a benchmark used to debug models, which will be described at the end of the section.

4.6.1 Using the GDB Interface

Figure 4.18 shows how the GDB interface must be used with the ArchC simulator. Four steps are required, the first three are shown in the figure and the fourth is a command to be issued to GDB:

1. Run the simulator with the target binary. The simulator must be created with `-gdb` option. After startup, the simulator will pause waiting for GDB.
2. Open GDB and connect to the simulator. Use the command `target remote localhost:5000` to make the default connection. The first two keyworks say that GDB should consider a remote simulator and the `localhost:5000` is the socket port to connect.
3. Load the program into GDB. Although the program is already loaded in the simulator, you need to load it again into the GDB. The program should be compiled with debug information.
4. Command the simulator from inside the debugger with GDB commands. Usually, you should first set a breakpoint in `main()` and then go on simulating your program step-by-step.

```
int powerpc::nRegs(void) {
  return 140;
}
```

Fig. 4.19 GDB-interface method to return the number of registers in the architecture

4.6.2 Implementing the GDB Interface

The simulator–GDB interface is a socket connection where the commands come from GDB and the simulator executes and returns the answers. Instead of implementing all the lower level interface, the processor designer needs only to give some information about the processor model and the ArchC will generate all the underlying infrastructure. The required information should be implemented in a file called `<processor_name_gdb_funcs.cpp>` (for PowerPC, it is called `powerpc_gdb_funcs.cpp`). In this file the user must implement methods that are included in the processor main class, to:

- Inform the number of registers available in the architecture.
- Return a register value given a register number.
- Set a register value given a register number and the value.
- Return a memory word given a memory address.
- Set a memory word given a memory address and the word value.

The following source code snippets are based on the PowerPC processor but the code is very similar between architectures.

Figure 4.19 shows the `nRegs` method implementation. This method must return the number of registers available in the architecture based on the GDB point of view. GDB sees all the registers as a single big register file. Each register number is a different register, no matter if one can be an integer and another can be a floating point register.

For PowerPC, this information can be obtained from the GDB header file called `ppc-tdep.h` inside the `gdb` directory of the GDB source tree. An excerpt of this file is shown in Fig. 4.20. Notice that the constant `PPC_NUM_REGS` contains the total number of registers, which is 140 as shown in Fig. 4.19. In the PowerPC specific case, this number has been increased recently after the inclusion of the Cell SPE support.

Figure 4.21 shows the `read_reg` method implementation. This method receives a register number, using the GDB single register file convention, and must return the register value. The `write_reg` method, shown in Fig. 4.22 is very similar, except for writing instead of reading a register.

The `mem_read` (Fig. 4.23) and `mem_write` (Fig. 4.24) methods are very simple to implement. They must read and write a memory word from the processor memory structure. Usually, for processors with only one global memory, these methods have only one line each. Although some processors require the memory address to be aligned, it is usually not necessary to check for alignment here because GDB does it, if necessary, before asking for a memory word.

```
1  /* Register number constants.  These are GDB internal
       register
2     numbers; they are not used for the simulator or remote
          targets.
3     Extra SPRs (those other than MQ, CTR, LR, XER, SPEFSCR)
          are given
4     numbers above PPC_NUM_REGS.  So are segment registers and
          other
5     target-defined registers.  */
6  enum {
7    PPC_R0_REGNUM = 0,
8    PPC_F0_REGNUM = 32,
9    PPC_PC_REGNUM = 64,
10   PPC_MSR_REGNUM = 65,
11   PPC_CR_REGNUM = 66,
12   PPC_LR_REGNUM = 67,
13   PPC_CTR_REGNUM = 68,
14   PPC_XER_REGNUM = 69,
15   PPC_FPSCR_REGNUM = 70,
16   PPC_MQ_REGNUM = 71,
17   PPC_SPE_UPPER_GP0_REGNUM = 72,
18   PPC_SPE_ACC_REGNUM = 104,
19   PPC_SPE_FSCR_REGNUM = 105,
20   PPC_VR0_REGNUM = 106,
21   PPC_VSCR_REGNUM = 138,
22   PPC_VRSAVE_REGNUM = 139,
23   PPC_NUM_REGS
24 };
```

Fig. 4.20 Excerpt from GDB source code file ppc-tdep.h defining the register numbers

```
1  ac_word powerpc::reg_read(int reg) {
2    unsigned int n;
3    if ((reg >= 0) && (reg < 32))
4      return GPR.read(reg);
5    else {
6      switch (reg) {
7        case 96:
8          n = ac_resources::ac_pc;
9          break;
10       ...
11       default:
12         n = 0;
13     }
14     return n;
15   }
16 }
```

Fig. 4.21 GDB-interface method to read a register

```
1  void powerpc::reg_write(int reg, ac_word value) {
2    /* general purpose registers */
3    if ((reg >= 0) && (reg < 32))
4      GPR.write(reg,value);
5    else {
6      switch (reg) {
7      case 96:
8        ac_resources::ac_pc = value;
9        break;
10     ...
11     default:
12       /* Register not implemented */
13       break;
14     }
15   }
16 }
```

Fig. 4.22 GDB-interface method to write a register

```
1  unsigned char powerpc::mem_read(unsigned int address) {
2    return ac_resources::IM->read_byte(address);
3  }
```

Fig. 4.23 GDB-interface method to read a memory word

```
1  void powerpc::mem_write(unsigned int address, unsigned char
       byte) {
2    ac_resources::IM->write_byte(address, byte);
3  }
```

Fig. 4.24 GDB-interface method to write a memory word

With the GDB interface ready, it is time to compile the model and use the GDB. For now on, we are assuming that you have a pre-build GDB for your architecture. The next step is to embed the simulator hook in your simulator. To do so, you must edit the `main.cpp` and include a call to the `enable_gdb()` method right after the `init()` method call. Figure 4.25 shows how the code should be after the inclusion of Line 9. Figure 4.26 shows how to create a simulator with the GDB methods.

It is now time to run the simulator and connect the GDB to it. Before doing so, we will first describe the acStone benchmark, which is designed to check processor models for correctness by using a set of 75 C source code files. ArchC is shipped with acStone in its base directory. Every acStone program is numbered and the first two digits represent the program category, while the other digit represents the program sequence inside the category. Each category checks for some basic processor funcionality, by testing a small set of instructions, as shown in Table 4.6.

Because acStone is written in C, it is necessary to have a C compiler to run it. If you are designing a new architecture, you will probably not have the required infrastructure at this moment. Nevertheless, you should look the acStone files trying

```
int sc_main(int ac, char *av[])
{
  //!  ISA simulator
  powerpc powerpc_proc1("powerpc");
#ifdef AC_DEBUG
  ac_trace("powerpc_proc1.trace");
#endif
  powerpc_proc1.init(ac, av);
  powerpc_proc1.enable_gdb();
  cerr << endl;

  sc_start(-1);
  ...
}
```

Fig. 4.25 Change to the `main.cpp` file to support remote GDB connection

```
acsim powerpc.ac -abi -gdb
make -f Makefile.archc
```

Fig. 4.26 Compiling the model together with the GDB interface

Table 4.6 acStone program categories

Category	Programs	Description
00x	1	Empty program that returns 0
01x	8	Check constants with different data types
02x	7	Check type casts from/to different data types
03x	4	Check additions of different data types
04x	4	Check subtractions of different data types
05x	8	Check multiplications of different data types
06x	8	Check divisions of different data types
07x	5	Check boolean operators
08x	5	Check shifts of different data types
11x	9	Check if clauses
12x	6	Check loops
13x	4	Check function calls
14x	6	Check arrays with different data types

to mimic their contents in assembly in order to check your processor. You will notice that, by using these simple set of files, you can clean the first big set of bugs in your processor design.

For those who have a C compiler, acStone can help even more, because there are also scripts to run the simulator, GDB, and check the results, as we will show latter.

```
1  int main() {
2    return 0;
3    /* Return 0 only */
4  }
```

Fig. 4.27 First program checked by acStone: `000.main.c`. This program only returns 0

```
1  int main() {
2    signed char c;
3    c=0x55;   /* After c is 85 */
4    /* Before c is 85   */   c=0xAA;   /* After c is -86 */
5    /* Before c is -86  */   c=0x00;   /* After c is 0 */
6    /* Before c is 0    */   c=0xFF;   /* After c is -1   */
7    /* Before c is -1   */   c=0x80;   /* After c is -128 */
8    /* Before c is -128 */   c=0x01;   /* After c is 1 */
9    /* Before c is 1    */   c=0x7F;   /* After c is 127 */
10   /* Before c is 127  */   c=0xFE;   /* After c is -2 */
11   /* Before c is -2   */   return 0; /* Return 0 */
12 }
```

Fig. 4.28 Second program checked by acStone: `011.const.c`. This program checks `signed char` constants

Figure 4.27 shows the simplest acStone program, `000.main.c`, aimed at only checking the basic runtime infrastructure required by programs in the architecture. This simple program uses 89 instructions in total for PowerPC, from which 9 are distinct instructions (`bctr`, `bl`, `blr`, `li`, `lwz`, `mr`, `mtctr`, `stw`, `stwu`). If different parameters are considered different instructions, then there are 26 instructions. If any of these 9 instructions has a serious bug, no PowerPC program can be run. That is the philosophy behind acStone. If you find a bug running this program, you will only need to check these 9 instructions.

Going further, let us consider the second acStone file is `011.const.c`, shown in Fig. 4.28. This source file is a little bigger, but you should notice a lot of comments explaining what should happen before and after each instruction. By compiling `011.const.c`, you generate 105 assembly instructions, from those 10 are unique (the same 9 from `000.main.c` plus `stb`) and are explored in a total of 34 different parameters. Notice that, by going incrementally, you will probably find your bugs in the new instructions.

TIP: *You can check the instructions in a program by using `objdump` or ask for ArchC to dump the instruction statistics by creating your model with `-s` parameter.*

We will use the `011.const.c` file as an example for GDB execution together with the acStone capabilities. The same approach can be used for any other compiled program. After compiling it with `powerpc-elf-gcc` and `-ggdb` option (Line 1 of Fig. 4.29), you can run the program using the simulator (Line 2), which will start and pause at Line 10. On another terminal, you should start GDB (Line 1

```
1  $ powerpc-elf-gcc -ggdb 011.const.c -o 011.const.powerpc
2  $ powerpc.x --load=011.const.powerpc
3
4                SystemC 2.2.0 --- Jan 15 2009 14:41:31
5          Copyright (c) 1996-2006 by all Contributors
6                       ALL RIGHTS RESERVED
7  ArchC: Reading ELF application file: 011.const.powerpc
8
9  ArchC: -------------------- Starting Simulation -------------
10 AC_GDB: listening to port 5000
11 AC_GDB: connected to port 5000
12 ArchC: -------------------- Simulation Finished -------------
13 SystemC: simulation stopped by user.
14 ArchC: Simulation statistics
15     Times: 0.01 user, 0.05 system, 621.92 real
16     Number of instructions executed: 34
17     Simulation speed: (too fast to be precise)
```

Fig. 4.29 Running and debugging `011.const.c` through the simulator and GDB

of Fig. 4.30), ask it to load the binary program symbols[2] (Line 9), connect to a remote target (Line 11) and start debugging. As a result, on the other terminal, ArchC should show that you have been connected (Line 11). On the GDB terminal, you will, usually, start setting a breakpoint in your `main` function (Line 15) and ask the debugger to continue running (Line 17). A set of step-by-step execution is shown (starting on Line 22) and a display command (Line 24) to show the `c` variable after each step will allow the user to compare the expected value, stated by comments on acStone, with the real one. If any problem is found, you can check the referring assembly instructions. When the simulator reaches the end of the program, it will finish and return to the terminal console. If you want to debug again, you must restart the simulator.

To simplify even more the program checking task, acStone can help you to compile it. To do so, you should use the *Makefile.archc* contained in its directory. Figure 4.31 shows the command and how to use it. Line 1 shows how to compile all acStone programs to the PowerPC architecture by using the *build* rule of `Makefile.archc`. The other two lines of Fig. 4.31 (Lines 2–3) must be run in different terminals, the first will start one simulator for each acStone program and the second will open a GDB for each program to control its simulator. The results will be saved for future check.

When designing another processor model, it is recommended that you copy and change those two scripts to point to your simulator. Since acStone uses C files, you will only need to check for the versions of your compiler, debugger and simulator

[2] Although ArchC simulator has already load the binary file, it has not loaded the symbols that will allow for source file debugging.

```
$ powerpc-elf-gdb
GNU gdb 6.1
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=powerpc-elf".
(gdb) file 011.const.powerpc
Reading symbols from 011.const.powerpc...done.
(gdb) target remote localhost:5000
Remote debugging using localhost:5000
0x00000000 in _start ()
warning: no shared library support for this OS / ABI
(gdb) break main
Breakpoint 1 at 0x10c: file 011.const.c, line 40.
(gdb) c
Continuing.

Breakpoint 1, main () at 011.const.c:40
40        c=0x55;
(gdb) n
43        /* Before c is 85 */ c=0xAA;
(gdb) display c
1: c = 85 'U'
(gdb) n
46        /* Before c is -86 */ c=0x00;
1: c = -86 ' '
(gdb) n
49        /* Before c is 0 */ c=0xFF;
1: c = 0 '\0'
(gdb) c
Continuing.

Program exited normally.
$
```

Fig. 4.30 Controlling the simulator execution from inside GDB

```
make -f Makefile.archc ARCH=powerpc build
run_simulator_powerpc.sh powerpc
run_gdb_powerpc.sh powerpc
check.sh powerpc
```

Fig. 4.31 Using acStone scripts

```
1  Files 000.main.powerpc.out and data/000.main.data are identical
2  Files 011.const.powerpc.out and data/011.const.data are identical
3  Files 012.const.powerpc.out and data/012.const.data are identical
4  Files 013.const.powerpc.out and data/013.const.data are identical
5  Files 014.const.powerpc.out and data/014.const.data differ
6  ...
```

Fig. 4.32 Result of acStone scripts

which, in our case, are `powerpc-elf-gcc`, `powerpc-elf-gdb`, and `powerpc.x`.

We will continue using `011.const.c` as our example, the same actions happen to all other programs. First, Line 1 of Fig. 4.31 compiles it and creates `011.const.powerpc`. The next line (2) will execute it through the `powerpc.x` that we created previously. In another terminal, the next command (Line 3) will open one GDB instance and use the commands in the file `gdb/011.const.gdb` to control it. The output will be saved in the file `011.const.powerpc.out` and must be equal to `data/011.const.data`, which is the reference answer. The Line 4 of Fig. 4.31 shows how to use the script to check the result. Figure 4.32 shows an excerpt of the result with correct results for the first 4 programs and an error detected on the execution of `014.const.powerpc`.

Besides acStone and its scripts, there are also other ArchC funcionalities that may help to debug a model:

Dump the decoder This is the first step if you think that the instructions are not being correctly executed. This can be done by using the `-dd` option from `acsim`.

Detect correct instruction sequence Use the `-g` to generate program traces and update logs for your processor. This helps to see the instruction sequence and also what is happening after each instruction.

Debug inside the simulator Since your code is just a small piece of the simulator, the best way to debug it is through the simulator source code. One very important method to set breakpoint is the processor behavior class (for PowerPC, it is `powerpc::behavior` inside the `powerpc.cpp` file). This method contains the main simulator loop, which will fetch, decode and execute every instruction in the program.

Debug your instructions After detecting a bug in a specific instruction, you can set a breakpoint inside it and try to discover what is happening in its execution. Be particularly careful with signed and unsigned values, usually this kind of mistake takes a lot of debugging time.

Completely clean and recompile your model After reexecuting `acsim` with new parameters, it is often recommended to clean and recompile the model so as to avoid any problem with remaining files. The most affected file is the

main.cpp, because ArchC does not create it directly but change a template file called main.cpp.tmpl and maintain the other file if it exists. To fully clean acsim generated files, use the distclean option to make.

4.7 Compiler and Operating System Support

Up to now, we have been using a pre-build gcc compiler for our architecture. This is the simplest way of using it but there are times where you will need to create your own compiler binary. If it is not your case, you can skip this section. In this section we will only consider architectures supported by GCC. This means that if you are designing a whole new architecture, you will have much more work to put GCC to work for you.

There are several guides on making cross-compilers based on GCC[3]. We will focus on the differences between those cross-compilers and the one required by ArchC. While ArchC tries to maintain most of the processor ABI[4], a few specific architecture rules may be relaxed. Unless otherwise controlled, ArchC simulators start executing from address 0 (the memory beginning), this is the first difference noticed. Fortunately, there is a GCC option, -Ttext=0, to ask the linker to start the program at address 0. In the beginning of your cross-compiler design, you will also find useful the -nostdinc and -nostdlib options to ask GCC not to use the standard include files nor the standard libraries. By using these options, you are also skipping the _start function, which is where the program starts its execution.

To emulate the operating system, ArchC intercepts some function calls[5] inside one of the GCC standard C libraries, named Newlib. In this way, you can emulate one system call even if you are not using the same operating system and, for designers, they only need to implement a few small methods in a similar way to the GDB interface. But there are other methods to use operating system facilities inside your simulator: you can emulate them intercepting the system call instruction, or you can run a real operating system inside your simulator. We will not cover these last two alternatives.

To make the ABI emulator, the user must:

1. Create the helper methods so that the simulator is able to read and write data accordingly to the processor's ABI;
2. Compile the ABI stub library for your architecture, and
3. Provide the startup file for the architecture.

[3]The ArchC website has one such guide.

[4]Application Binary Interface—this is the set of rules that controls how processor resources like registers, memories should be used by software to implement function calls, stacks, etc.

[5]To use this feature, the -abi option must be passed to acsim.

```
int powerpc_syscall::get_int(int argn)
{
  return GPR.read(3+argn);
}
```

Fig. 4.33 ABI emulator method to read an integer passed as a parameter to a function

```
void powerpc_syscall::set_int(int argn, int val)
{
  GPR.write(3+argn, val);
}
```

Fig. 4.34 ABI emulator method to write an integer passed as a parameter to a function

4.7.1 The Helper Methods

To write the helper methods, the recommended procedure is to copy a sample file for any other processor and change it accordingly to yours. For PowerPC, this file is called `powerpc_syscall.cpp`. The first five methods are related to the way your processor pass/receive parameters to/from functions. The next does the fixup to return from system call and the last one moves the command line parameters from the outside world to the simulated environment. We will cover each one in the following paragraphs.

We are going to use the `write` function as example to explain the helper methods. It has 3 parameters, the first is the file descriptor to write data, the second is a buffer containing the data to be written and the third is an integer representing the buffer size. This function prototype is `ssize_t write(int fildes, const void *buf, size_t nbyte)`.

To use the ABI emulator, a function stub for the `write` should be included instead of the original function. This stub will only call the ArchC simulator that will handle the write operation. To do that, it must know how to get the parameters from the simulated environment. To read the first parameter, it is necessary to call the `get_int` method. This function will receive 0 as parameter, meaning that the simulator wants the first function parameter. Figure 4.33 shows how simple is to implement it on PowerPC. It is just a matter of reading a register since PowerPC passes parameters to function as registers[6]. Reading the third parameter is similar to the first since it is also an integer. This method can be used to read integers, pointers and characters. The opposite of `get_int` is `set_int` (Fig. 4.34), which puts a value in the corresponding parameter. The string version of these functions are shown in Fig. 4.35 and 4.36.

There is another function called `set_buffer_noinvert` (Fig. 4.37) that does not consider the endian while copying the data.

[6]In a processor that parameters are passed in the stack, this function would need to calculate the correct address, fetch the value and return it. This is why ArchC needs these helper functions, to allow a wide range of processors implementations.

```
1 void powerpc_syscall::get_buffer(int argn, unsigned char*
      buf, unsigned int size)
2 {
3   for (unsigned int i = 0; i<size; i++, addr++) {
4     buf[i] = MEM.read_byte(addr);
5   }
6 }
```

Fig. 4.35 ABI emulator method to read a block of data passed as a parameter to a function

```
1 void powerpc_syscall::set_buffer(int argn, unsigned char*
      buf, unsigned int size)
2 {
3   unsigned int addr = GPR.read(3+argn);
4   for (unsigned int i = 0; i<size; i++, addr++) {
5     MEM.write_byte(addr, buf[i]);
6   }
7 }
```

Fig. 4.36 ABI emulator method to write a block of data passed as a parameter to a function

```
1 void powerpc_syscall::set_buffer_noinvert(int argn, unsigned
      char* buf, unsigned int size)
2 {
3   unsigned int addr = GPR.read(3+argn);
4   for (unsigned int i = 0; i<size; i+=4, addr+=4) {
5     MEM.write(addr, *(unsigned int *) &buf[i]);
6   }
7 }
```

Fig. 4.37 ABI emulator method to write a block of data passed as a parameter to a function, ignoring endianess issues

The next method is `return_from_syscall` (Fig. 4.38) that does the fixup code to allow a clean function return. By clean function return we mean that there will not be any return from the function stub so that you must clean any necessary registers, set the stack (if necessary), and set the program counter to the correct place.

The last method to implement is related to program command line parameters. This method, shown in Fig. 4.39, is responsable to receive all the simulator parameters and pass it to the simulation environment so that the program being executed has access to it. The `set_prog_args` is called after the simulator starts its execution and receives `argc` and `argv` as parameters. The PowerPC model approach is to reserve 512 bytes at the end of the simulation memory space and copy the command line parameters to there. After that it sets registers `r3` and `r4` to the parameters received so that `main` receives the right parameters.

```
void powerpc_syscall::return_from_syscall()
{
  unsigned int oldr1;
  unsigned int oldr31;
  oldr1=MEM.read(GPR.read(1));
  oldr31=MEM.read(GPR.read(1)+28);
  GPR.write(1,oldr1);
  GPR.write(31,oldr31);
  ac_pc=LR.read();
}
```

Fig. 4.38 ABI emulator method to do the fixup required to return from a system call

```
void powerpc_syscall::set_prog_args(int argc, char **argv)
{
  int i, j, base;
  unsigned int ac_argv[30];
  char ac_argstr[512];
  base = AC_RAM_END - 512;
  for (i=0, j=0; i<argc; i++) {
    int len = strlen(argv[i]) + 1;
    ac_argv[i] = base + j;
    memcpy(&ac_argstr[j], argv[i], len);
    j += len;
  }
          //Write argument string
  GPR.write(3, AC_RAM_END-512);
  set_buffer(0, (unsigned char*) ac_argstr, 512);
          //Write string pointers
  GPR.write(3, AC_RAM_END-512-120);
  set_buffer_noinvert(0, (unsigned char*) ac_argv, 120);
          //Set r3 to the argument count
  GPR.write(3, argc);
          //Set r4 to the string pointers
  GPR.write(4, AC_RAM_END-512-120);
}
```

Fig. 4.39 ABI emulator method to put the command line parameters inside the processor memory space to be read in the future

4.7.2 ABI Stub Library

ArchC already provides the ABI stub library so that you will only need to compile. The library, named `libac_sysc` is available to download on the ArchC website. Inside the library there is one file called `ac_real_sysc` where most of the stubs are placed. Some other files contains functions that only return 0, they are not being emulated. By compiling this file with your program you enable it to emulate the operating system.

The best method to merge the programs is by creating a linker script and a new spec file for it. Since the linker scripts vary too much from architecture to archi-

```
1          .text
2          .align  2
3          .globl  _start
4  _start:
5          bl      main
6          bl      _exit
7          .end    _start
8          .align 6
```

Fig. 4.40 ac_start.s content

tecture, we are not covering it here. We encourage the designers to see the simple linker version we provide with our models and check if it is possible to convert the usually complex linker scripts in this way.

4.7.3 Startup File

This is, usually, the simplest part to implement. The startup file, also called `ac_start.s` must be written in assembly code but has only to call `main` as shown in Line 5 of Fig. 4.40 for the PowerPC processor. Remember that there is no need to setup parameters since the simulator will call the `set_prog_args` method before starting executing. Sometimes you may want to setup stack pointer, the ArchC PowerPC model does this on the initial behavior.

One question that usually arises is the order in which ABI support and GDB interface are designed. Certainly, if you could have ABI implementation right at the beginning, you would easily pass the acStone benchmark. Unfortunatelly, this is exactly the motivation to have the GDB interface and run acStone at the beginning. To run a program with any sort of I/O operation, you will need lots of instructions that must be validated prior to execution. So, we recommend the order presented here.

4.8 Refining the Model

Now that the basic infrastructure is already done, it is time to run big programs like those found in Mediabench [2] and MiBench [1].

The selected Mediabench programs are `adpcm`, `gsm`, `jpeg`, `mpeg`, and `pegwit`. All those programs should be run with their original input and should generate their expected output[7]. Both Mediabench and MiBench have `README` files to explain their usage. Each program may have some different command line parameters. We will cover an example here.

[7] It is possible that a few programs in Mediabench and MiBench generate the output with different endian compared to the standard one. This should be considered a valid result.

```
1 $ powerpc.x --load=bin/cjpeg -dct int -progressive -opt -
      outfile output/testout.jpeg input/testimg.ppm
2 $ powerpc.x -load=bin/djpeg -dct int -ppm -outfile output/
      testout.ppm input/testimg.jpg
3 $ diff --brief --report-identical-file data/testout.jpeg
      output/testout.jpeg
4 $ diff --brief --report-identical-file data/testout.ppm
      output/testout.ppm
```

Fig. 4.41 Commands to execute and check Medibench `jpeg` program

Lines 1–2 of Fig. 4.41 show how to execute the jpeg encoder and decoder, respectively, while Lines 3–4 show how to check if the result is correct. You must notice that the execution is done by using the PowerPC simulator (`powerpc.x`); the program is loaded as already mentioned and the rest of the command line is passed directly to the simulated environment to be used by the encoder and decoder.

You must first run and check all selected Mediabench programs. After passing this level, your model is considered as being at version 0.5, which represents a good stability level.

The next step is to check the MiBench benchmark. Each program in MiBench has two input sets: the small input set and the large input set. Version 0.6 considers all the selected programs using the small input set and version 0.7 considers the large input set.

The selected MiBench programs are divided into categories, as shown bellow:

Automotive: `basicmath`, `bitcount`, `qsort`, `susan`
Consumer: `jpeg`, `lame`
Network: `dijkstra`, `patricia`
Office: `stringsearch`,
Security: `rijndael`, `sha`
Telecomm: `crc32`, `fft`, `adpcm`, `gsm`

By passing all those programs using both small and large input sets, your model can be considered stable enough to run any kind of program. Usually, at this moment, you will have run more than 100 billion instructions and checked for the program results.

The next and last step is to run SPEC 2000 [4]. All SPEC programs must be run. SPEC 2000 programs are very big and require a huge amount of simulation time so, for the time being, no attempt was made to pass the processor models available on the ArchC website through this benchmark.

4.9 Going Faster—How to Improve Your Simulator Performance

As already mentioned in Sect. 4.3, decoding all the instructions to be executed can take a significative amount of simulation time. This overhead is highly minimized

by using the decoder cache provided by ArchC. Whenever possible, you should use the decoder cache, which is enabled by default on all ArchC simulators. Unfortunatelly, there are situations where the decoder cache cannot be used. The main restriction arrives from the fact that the decoder cache, acts as a instruction cache without any cache coherence mechanism, even with its own data memory. The two most common situations you are going to face inconsistency are shown below. To avoid inconsistency, you should create your simulator passing the `-ndc` option to `acsim`.

Self-modifying code If your program write instructions to memory and try to execute them, it will work the first time for each memory address. After that, the instruction will be cached and will not be redecoded again.

Shared memory Any kind of external use of the processor memory can lead to a inconsistency in the decoder cache. Usually, this happens when you are building a platform, which will be shown in the next chapter.

There are also several compiler optimizations that can increase simulators performance. Usually you may want to optimize both the program that will run on the simulator to reduce the number of simulated instructions and the simulator itself to execute more instructions per second. The ArchC simulator is designed to take advantage of *inline* on its main simulator inside the processor *behavior* method (for PowerPC, this method is implemented in the `powerpc.cpp` file). Some compilers ask for the direct inclusion of the ISA implementation file (for PowerPC, `powerpc_isa.cpp`) to inline all instructions behavior. This can be done easily by using a `#include` in the top of the main processor file (in our case, `powerpc.cpp`, and removing the same file from the `Makefile`.

The instruction selection is also written in a switch so that the compiler can, easily, use a branch table to select the correct option. These two optimizations can significantly reduce simulation time. Unfortunatelly, not all compilers do this by default. If this is your problem, you should try to find an optimization flag that ask for *branch tables* and use it while compiling your model.

When running platforms with external devices, the `set_instr_batch_size` method of all ArchC generated processor models can be used to tradeoff between speed and accuracy. Chapter 5 will show this method in more details.

Almost all ArchC features take time even if they are not being used. If you want a really fast simulator, avoid any command line option passed to `acsim` that is not strictly necessary.

References

1. M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, R.B. Brown, MiBench: A free, commercially representative embedded benchmark suite, in *International Workshop on Workload Characterization*, December 2001, pp. 3–14. IEEE Xplore
2. C. Lee, M. Potkonjak, W.H. Mangione-Smith, MediaBench: A tool for evaluating and synthesizing multimedia and communications systems, in *Proceedings of the 30th Annual International Symposium on Microarchitecture (Micro-30)*, December (1997)

3. R. Stallman, et al., *Debugging with GDB, the GNU Source-level Debugger*, 9th edn. (Free Software Foundation, Boston, 2004)
4. Standard Performance Evaluation Corporation, http://www.specbench.org
5. The ArchC Team, The Archc Architecture Description Language V 2.0 Reference Manual (2007). Available at http://www.archc.org
6. TIS Committee, *Executable and Linking Format Specification* (1995). May

Chapter 5
Building Platform Models with SystemC

Rodolfo Azevedo, Sandro Rigo, and Bruno Albertini

With the advent of highly complex designs in the embedded market in the last decade, the available methodologies evolved so that development platforms are key to design success. Since they are easier to set up and modify, virtual development platforms are gaining ground over physical platforms, as a starting point for system design.

This chapter focuses on building virtual platforms and shows examples on how to do it with the help of languages, models, and tools presented in the previous chapters. When building a virtual platform, some characteristics must be known in advance, like memory mapping and interconnection structure, although they can be changed later if needed.

On the one hand, modifying the address mapping of some peripherals is usually a simple task, although it may require a few software changes. On the other hand, changing the interconnection structure typically requires a big effort in reworking the hardware representation of the platform but, usually, requires no changes in the software counterpart, although it is likely to affect system performance. In any case, some of the changes are made as the platform evolves and may include moving pieces of software to hardware, which is much easier to implement in a virtual platform than in a physical one (since the executable representation of a non-programmable hardware component is usually a behavioral model that closely resembles the piece of software to be implemented in hardware).

R. Azevedo (✉) · S. Rigo · B. Albertini
Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil
e-mail: rodolfo@ic.unicamp.br

S. Rigo
e-mail: sandro@ic.unicamp.br

B. Albertini
e-mail: balbertini@ic.unicamp.br

S. Rigo et al. (eds.), *Electronic System Level Design*,
DOI 10.1007/978-1-4020-9940-3_5, © Springer Science+Business Media B.V. 2011

```
1 AC_ARCH(myproc) {
2  ac_wordsize 32;
3  ac_mem MEM:4M;
4  ac_regbank GPR:32;
5  ARCH_CTOR(myproc) {
6    ac_isa("myproc_isa.ac");
7  };
8 };
```

```
1 AC_ARCH(myproc) {
2  ac_wordsize 32;
3  ac_tlm_port MEM:4M;
4  ac_regbank GPR:32;
5  ARCH_CTOR(myproc) {
6    ac_isa("myproc_isa.ac");
7  };
8 };
```

Fig. 5.1 ArchC 2.0 TLM declaration for standalone and TLM memory model

We focus on processor-centric platforms by exploring designs with one or more processors in a shared memory environment with a few extra peripherals. Our main example is an MP3 player, which is organized by splitting its functionality into two processors and two custom non-programmable hardware components. This chapter helps the designer to describe the whole system, from the processor to every peripheral. The memory address mapping is described and the difference between the memory view of a processor within a platform is distinguished from the memory view of processor model used as a standalone instruction-set simulator (ISS), since such difference can affect the compiler's and the software view of the memory. We also address endianness as a system issue and show how it can affect processor modeling.

5.1 ArchC and TLM Interface

ArchC2.0 originally relies on the TLM version 1.0, which is being replaced by the fresh new TLM 2.0. As TLM 2.0 is backward compatible with TLM 1.x, there is no problem in using it while ArchC itself is under migration towards the new version.

Every ArchC processor model has a storage element connected to it, which represents an internal memory. This is done through the `ac_storage` class, which is used by every processor model. When declaring such storage element in the ArchC model file, the user can choose between `ac_mem`, `ac_cache`, or `ac_tlm_port`. The first two allow the representation of inner memory when the processor model is used as a standalone ISS. As a consequence, neither the memory representation can be shared by another instance of a processor model, nor can it be used for memory mapping I/O device registers.

Since it implements the same methods for read and write, despite the memory model used, the TLM port can be viewed by a processor model as an ordinary storage element. By employing the TLM port declaration, the processor model redirects all memory accesses to that port through a TLM socket, which is instantiated into the model as a blocking initiator transport interface.

Figure 5.1 shows the code for a generic standalone ISS (left side) and for a TLM memory model ISS (right side). Line 3 is the key to enabling the ArchC TLM. For standalone processor models, whose focus is on the ISS itself, developers can use `ac_mem`, telling ArchC to generate an internal memory. To connect the ISS to

```
/// ArchC TLM request type.
enum ac_tlm_req_type {
  READ, WRITE, LOCK, UNLOCK, REQUEST_COUNT
};
/// ArchC TLM response status.
enum ac_tlm_rsp_status {
   ERROR, SUCCESS
};
/// ArchC TLM request packet.
struct ac_tlm_req {
  ac_tlm_req_type type;
  int dev_id;
  uint32_t addr;
  uint32_t data;
};
/// ArchC TLM response packet.
struct ac_tlm_rsp {
  ac_tlm_rsp_status status;
  ac_tlm_req_type req_type;
  uint32_t data;
};
```

Fig. 5.2 ArchC 2.0 TLM protocol

external modules, developers should simply change that line to `ac_tlm_port`. As a result, ArchC generates an instrumented model that redirects all memory accesses to the TLM port. However, it is necessary to preserve the name `MEM`, since it is employed inside the module (e.g. ISA declaration). Developers should connect the resulting processor model to the representation of a communication block, which can be a simple TLM memory or a complex bus. The definition of a TLM boundary (e.g. 4 MB in the example) is used to allocate the stack when the application does not provide one and to detect segmentation faults if the ISS tries to access invalid memory locations. Example of connections are provided in Sect. 5.2.

All processor read and write requests directed to the TLM socket are associated with a payload, which consists of a request or response packet. The available TLM interface is the blocking transport interface, which only allows a request-response protocol. When the ISA simulator does a memory request, ArchC creates a request packet containing the respective address and calls the slave transport function through the TLM channel. The slave module assembles and returns a response packet containing the requested data.

The format of the ArchC packet is shown in Fig. 5.2. The request types can be an ordinary read/write access, a bus locking/unlocking action, or simply a counting request to emulate bus timing. Response packets admit only two types: success or error. Since request and response packets are different, the packet structure is clearly not homogeneous.

In the request packet, `dev_id` is required by some buses and `data` is used for interrupt purposes. With TLM 2.0, these structures should change to merge both packets within a single structure.

```
1  class ac_tlm_mem :
2    public sc_module,
3    public ac_tlm_transport_if {
4  public:
5    sc_export <ac_tlm_transport_if> target_export;
6      ...
7    ac_tlm_rsp transport(const ac_tlm_req &request) {
8      ac_tlm_rsp response;
9      switch(request.type) {
10       case READ:     // Packet is a READ one
11         response.status =  readm(request.addr,
               response.data);
12         break;
13       case WRITE:     // Packet is a WRITE
14         response.status = writem(request.addr, request.data);
15         break;
16       default:
17         response.status = ERROR;
18         break;
19     }
20     return response;
21   }
22   ...
23 };
```

Fig. 5.3 ArchC 2.0 TLM slave example

Figure 5.3 shows part of an ArchC 2.0 TLM-compatible memory representation. Key implementation aspects include the specialization of the interface at Line 3, the TLM port declaration at Line 5, and the transport function at Line 7. That function is invoked when the processor performs a request. It opens the request packet, assembles a response packet (declared at Line 8) and returns it to caller. The actual work of accessing the data structure that emulates a memory (in this case a plain vector omitted for simplicity) is performed by a pair of internal functions, `readm` and `writem` (which are also not shown for simplicity). Those functions can be used to annotate time or to wait for some simulation time, so as to emulate a real memory behavior.

5.2 Platforms with ArchC

To simplify the ArchC usage on platforms, it is recommended to start using the ArchC Reference Platform (ARP). ARP is nothing more than a way of organizing all the software required to build the platform in a directory tree with global configuration where needed. The list below shows the directories available:

- **bin:** It contains the available ARP utilities (to be described latter).
- **ip:** Here are kept the available non-programmable platform components.
- **is:** It contains the available interconnection structures (e.g. buses, NoCs).

- **platforms:** This is the place keeping the main platform files which interconnect all the required modules.
- **processors:** Here lay all the available platform processor models (not only ArchC models, but also processor models developed with other tools).
- **sw:** It holds all the software to be run on the platform.
- **wrappers:** Platform wrappers (if needed) are stored in this directory.

Although at the first glance such a splitting directory structure may seem unnecessary, it was deliberately designed to allow a same component to be reused across several distinct platforms. For instance, a single copy of a given processor model, say PowerPC, is kept in the processors directory, regardless of the number of derived platforms relying on that model and regardless of how many instances of it are actually used within a given platform.

To simplify the management of all the modules needed for a platform, the ARP package comes with the `arp.py` utility. This utility has three usage modes represented by one of the following options:

--pack It packs all components required to build a platform in a single file that can be shipped to other workstation.

--unpack It unpacks the single file created with the previous option, thereby allowing it to be executed in the current ARP directory tree.

--list It lists the components inside a given package.

By relying on the ARP utility, developers do not need to worry about the location of the required files, allowing a more reusable organization. The ARP engine will locate all the required files and use them.

5.3 Platform Examples

One of the main consequences of providing ArchC processor models with an external connection is that it allows modular platform construction. By using various kinds of modules and a default interconnection methodology, developers can simulate distinct platforms ranging from a simple processor-bus-memory system to complex heterogeneous architectures. This section illustrates, by means of a few examples, how to use ARP to describe a platform by showing how to attach ArchC processor models and how to design software for them.

5.3.1 A Processor-Memory Platform

The first platform has a single processor and an external memory (both components will be reused in more elaborate examples later on). The chosen processor is a PowerPC, whose processor model was addressed in the previous chapters. In this introductory example, the platform software is simply a “Hello World”-like program.

```
# Variable that points to ArchC installation path
PLATFORM := single_ppc

# Variable that points to SystemC installation path
export SYSTEMC := /opt/esl/systemc

# Variable that points to TLM installation path
export TLM_PATH := /opt/esl/tlm

# Variable that points to ArchC installation path
export ARCHC_PATH = /opt/esl/archc
```

Fig. 5.4 Makefile.arp configuration options

```
IP := ac_tlm_mem
IS :=
PROCESSOR := powerpc
SW := hello_ppc
WRAPPER :=
```

Fig. 5.5 defs.arp configuration file for the single PowerPC platform

As a starting point, appropriate files must be created or moved within ARP's directory structure, as follows:

arp/processors/powerpc	The files associated with the PowerPC ArchC processor model.
arp/sw/hello_world	The source file of the software to be executed (this name was used to make it different from the dual core example we are going to see latter).
arp/platform/single_ppc	The file containing SystemC code to connect all modules and execute the platform.
arp/ip/ac_tlm_mem	Contains the implementation of the TLM Memory.

Figure 5.4 shows the configuration variables available in the `Makefile.arp` file. The first variable, which indicates the platform name, points to the `single_ppc` directory. This is the only variable that needs to be changed when switching between platforms. The other three variables point, respectively, to the SystemC, TLM and ArchC paths.

Another configuration file, called `defs.arp`, which is placed inside each platform, is located at `arp/platform/single_ppc/defs.arp` for this example. Figure 5.5 shows the file content. It consists of one line for each component type. If more than one component of any type is used, their names must be placed in the same line (separated by spaces). If the same component is used more than once, it is not necessary to place its name twice in this file.

These are the only two files required for configuration. The PowerPC processor model needs only the change made on Fig. 5.1 to convert the local memory to an external TLM interface. The `ac_tlm_mem` is the same shown in Fig. 5.3.

```
#include <stdio.h>

int main(int ac, char *av[]){
  int i;
  for(i=0;i<10;i++)
    printf("Hi from processor PowerPC!\n");
}
```

Fig. 5.6 Software to be executed in the single-core PowerPC platform

```
1  const char *project_name="powerpc";
2  const char *project_file="powerpc.ac";
3  const char *archc_version="2.0beta1";
4  const char *archc_options="-abi -dy ";
5
6  #include  <systemc.h>
7  #include  "powerpc.H"
8  #include  "ac_tlm_mem.h"
9
10 using user::ac_tlm_mem;
11
12 int sc_main(int ac, char *av[])
13 {
14   powerpc powerpc_proc1("powerpc");
15   ac_tlm_mem mem("mem", 8*1024*1024);
16
17   powerpc_proc1.MEM_port(mem.target_export);
18
19   powerpc_proc1.init(ac, av);
20   cerr << endl;
21
22   sc_start();
23
24   powerpc_proc1.PrintStat();
25   return powerpc_proc1.ac_exit_status;
26 }
```

Fig. 5.7 Single-core platform description in SystemC

For this platform, the software is exactly the same as can be executed in the standalone PowerPC. Figure 5.6 shows the program that will be executed in the platform. Notice that this software can be easily executed as an example of Chap. 4.

To finish the example, it is necessary to connect the components within the platform main file. Figure 5.7 contains the `sc_main` declaration. Lines 1–4 show a few ArchC parameters that can be used for debugging purposes. Platform components are declared as SystemC regular components. Line 14 shows how to declare one PowerPC processor. The memory is declared in a similar way at Line 15, except for the fact that it requires and extra parameter to specify its size.

After declaration, it is time to bind the PowerPC memory port (`MEM_port`) to the memory itself, which is done in Line 17. The ArchC processor model needs to be

```
1  $ make
2  $ make run
```

Fig. 5.8 Executing the platform

initialized by calling its `init` method (Line 19). The simulation is launched as usual in Line 22. After the platform simulation is concluded, the PowerPC's `PrintStat` method is called (Line 24) to show statistics on the screen, before the simulation process is terminated (Line 25).

To run the executable representation of the platform, two commands are needed: one to build the representation, another to execute it, as shown in Fig. 5.8. The resulting screen, exhibited at simulation completion, is shown in Fig. 5.9. Note that the welcome messaged is printed 10 times, as expected from Fig. 5.6.

Although some refinements could be done on this platform, it is kept as is for simplicity. We leave such improvements to the next section, where a dual-core platform will be built through iterative refinements.

5.3.2 Dual Core Platform

A pretty simple platform, derived from the previous one, is described in Fig. 5.10. It consists of a PowerPC (Line 3), a TLM memory (Line 4), and a simple router

```
1                SystemC 2.2.0
2          Copyright (c) 1996-2006 by all Contributors
3                       ALL RIGHTS RESERVED
4  ArchC: Reading ELF application file: hello_ppc.x
5
6  ArchC: ------- Starting Simulation ----------
7
8  Hi from processor PowerPC!
9  Hi from processor PowerPC!
10 Hi from processor PowerPC!
11 Hi from processor PowerPC!
12 Hi from processor PowerPC!
13 Hi from processor PowerPC!
14 Hi from processor PowerPC!
15 Hi from processor PowerPC!
16 Hi from processor PowerPC!
17 Hi from processor PowerPC!
18 ArchC: ------- Simulation Finished ----------
19 SystemC: simulation stopped by user.
20 ArchC: Simulation statistics
21     Times: 0.01 user, 0.03 system, 0.00 real
22     Number of instructions executed: 17571
23     Simulation speed: (too fast to be precise)
```

Fig. 5.9 Screen capture—Hello World

```
int sc_main(int ac, char *av[])
{
  powerpc ppc_proc1("ppc1");
  ac_tlm_mem mem("mem", 8 * 1024 * 1024);
  ac_tlm_router router("router");

  ppc_proc1.MEM_port(router.target_export);
  router.MEM_port(mem.target_export);

  ppc_proc1.init(ac, av);
  sc_start(-1);

  ppc_proc1.PrintStat();
  cerr << endl;

  return ppc_proc1.ac_exit_status;
}
```

Fig. 5.10 ArchC 2.0 TLM simple single core processor-router-memory platform

```
class ac_tlm_router :
  public sc_module,
  public ac_tlm_transport_if
{
public:
  ac_tlm_port MEM_port;
  sc_export< ac_tlm_transport_if > target_export;

  ac_tlm_rsp transport(const ac_tlm_req &request)
  {
    ac_tlm_rsp response;
    response = MEM_port->transport(request);
    return response;
  }
  ac_tlm_router(sc_module_name module_name);
  ~ac_tlm_router() {};
};
```

Fig. 5.11 TLM router, first version

(Line 5). Note that the main difference between this platform and the previous one is the presence of a router. The platform's components are connected through TLM binding at Lines 7–8. The router was built such that all TLM requests are simply passed from the processor to the memory and all memory responses are merely returned to the processor. As a result of full router transparency, this platform can run exactly the same code as the previous one.

As shown in Fig. 5.11, the router is a very simple module, which essentially consists of two TLM ports. Note that the router is very similar to the memory described in Fig. 5.3, except from the fact that it passes all incoming requests to the output port without processing (as opposed to the memory functionality), as can be seen in the

```
1  int sc_main(int ac, char *av[])
2  {
3    sc_report_handler::set_actions("/IEEE_Std_1666/deprecated",
         SC_DO_NOTHING);
4    //!  ISA simulator
5    powerpc ppc_proc1("ppc1"), ppc_proc2("ppc2");
6    ac_tlm_mem mem("mem", 8 * 1024 * 1024);
7    ac_tlm_router router("router");
8
9    char *av1[] = {"dual_ppc.x", "--load=hello_dual.x", ""};
10   int ac1 = 3;
11   char *av2[] = {"dual_ppc.x", "--load=hello_dual.x", ""};
12   int ac2 = 3;
13
14   ppc_proc1.MEM_port(router.target_export1);
15   ppc_proc2.MEM_port(router.target_export2);
16   router.MEM_port(mem.target_export);
17
18   ppc_proc1.init(ac1, av1);
19   ppc_proc2.init(ac2, av2);
20   ppc_proc1.set_instr_batch_size(1);
21   ppc_proc2.set_instr_batch_size(1);
22
23   sc_start(-1);
24
25   ppc_proc1.PrintStat();
26   ppc_proc2.PrintStat();
27   return ppc_proc1.ac_exit_status + ppc_proc2.ac_exit_status;
28 }
```

Fig. 5.12 Dual core platform, with two PowerPC processors

`transport` method (Line 9), more specifically at Line 12. To run this platform, it is necessary to add its description to ARP's `is` directory and to include the line `IS := ac_tlm_router` in the file `defs.arp` (described in Fig. 5.5).

Now it is time to add a second processor to the derived platform so as to build a dual-core system. The first modification must be done at the platform `main.cpp` file, as shown in Fig. 5.12. First, the two processors (`ppc1` and `ppc2`) are declared at Line 5. Then, we included two command line parameters, one for each of the processors (Lines 9–11). For the time being, the parameters are exactly the same: they are serving as placeholders to put two programs to run, one for each processor (to actually run two different programs, they must be placed in distinct memory addresses). Then, the previous version of the router was extended (Lines 14–15) with an extra port (`target_export1` and `target_export2`). After processor model initialization (Lines 18–19) and immediately before simulation launching, a same parameter was changed in both processor models (Lines 20–21): `set_instr_batch_size` was set to 1 so as to allow each underlying ISS to alternately run one instruction at time, before calling a `wait` to switch to the other ISS (usually, each ArchC ISS runs 200 instructions before switching; by setting that parameter to 1, the simulation more closely resembles the actual parallel execution

```
//!Generic begin behavior method.
void ac_behavior(begin)
{
  /* Here the stack is started */
  GPR.write(1, AC_RAM_END - 1024);
  ...
}
```

Fig. 5.13 PowerPC original `ac_behavior(begin)` method

at the expense of slowing down the simulation; feel free to play with this parameter for a tradeoff between speed and precision).

After running this platform, the welcome message *"Hi from processor PowerPC!"* appears 20 times on the screen, as expected.

At this point, there are two pending issues to be solved for this platform: the two processors have exactly the same stack and are running exactly the same code. Let us address each of them at a time.

The initial value for the stack pointer can be set in two different places: in the startup assembly code or in the processor model description. If the stack pointer value is set in both places, the startup assembly code will prevail, as expected in a real platform. To make platform initialization flexible enough for further potential improvements, we prefer to make changes in the processor model. In the processor description, the place to set the initial value for the stack pointer is in the `ac_behavior(begin)` method, as illustrated in Fig. 5.13, where register 1 (the stack pointer) is written with the address `AC_RAM_END - 1024`. The 1024-byte offset represents reserved memory space through which the simulator can pass command-line parameters to the platform.

Let us now show how we can provide multiple stacks by modifying the code in Fig. 5.13 by defining a stack size and allocating successive stack blocks for each processor. Fig. 5.14 illustrates the required changes. First, the default stack size is defined (256 KB) and the number of active processors is initialized. When a processor model is activated, its the stack pointer is assigned the address of a 256 KB block depending on the number of processors activated so far. In this way, we can automatically handle an arbitrary amount of processors up to the declared memory limit. Since all the processors share the same memory, they can pass pointers from the stack to each other.

Now it is time to make the processors run different code. Although this could be done by loading different codes to different address spaces, we are going to solve this problem in a straightforward way, by loading exactly the same code but splitting the processors execution flow on the fly. Like when solving the previous issue, this is a generic solution that can be applied to an arbitrary number of processors.

We start by creating a new component (`ac_tlm_lock`), which acts as a hardware lock. We reuse exactly the same code as `ac_tlm_mem` by changing only the `readm` and `writem` methods. Figure 5.15 shows the modified method (it is even simpler than the memory's). The `writem` method only assign the `value` private variable and the `readm` returns its value and change it to 1. By so doing, we can

```
1  #define DEFAULT_STACK_SIZE (256 * 1024)
2  static int processors_started = 0;
3  ...
4  //!Generic begin behavior method.
5  void ac_behavior( begin )
6  {
7    /* Here the stack is started */
8    GPR.write(1, AC_RAM_END - 1024 - processors_started++ *
         DEFAULT_STACK_SIZE);
9  }
```

Fig. 5.14 PowerPC original `ac_behavior(begin)` method

```
1  ac_tlm_rsp_status ac_tlm_lock::writem( const uint32_t &a ,
       const uint32_t &d )
2  {
3    value = d;
4    return SUCCESS;
5  }
6
7  ac_tlm_rsp_status ac_tlm_lock::readm( const uint32_t &a ,
       uint32_t &d )
8  {
9    d = value;
10   value = 1;
11   return SUCCESS;
12 }
```

Fig. 5.15 Lock module `readm` and `writem` methods

use this component as a simple hardware lock, which can be checked by the same software code to split the program flow.

After creating that component, the next step is to connect it to the platform. This can be easily performed inside our router, by creating a new `ac_tlm_port` called `LOCK_port` (after Line 6 of Fig. 5.11) and changing the `transport` method as in Fig. 5.16. To distinguish between ports, an `if` statement is included in Line 5. Selection is based on the request address. Note that the address 0x800000 is adopted for the lock component. When a request is issued with that address, the lock component receives the packet. Note that, every time a new component is needed, it is only a matter of including a new (nested) `if` statement for selection purposes and adding a new output port in the router. This make this router easily scalable.

To show that both pending issues were solved by the described mechanisms, we are now going to change the "Hello World" program. Figure 5.17 shows the changes. The first change is the inclusion of a `RecursiveHello` (Line 3) function, which recursively calls itself several times. We do so to show that the processors actually have different stacks. To use the lock hardware, we add a pointer to the 0x800000 address. The `AcquireLock` just has to wait until this pointer has a zero value (remember that for every hardware read, the value is changed to one). The `Release-`

```
ac_tlm_rsp transport( const ac_tlm_req &request ) {

  ac_tlm_rsp response;

  if (request.addr != 0x800000)
      response = MEM_port->transport(request);
  else
      response = LOCK_port->transport(request);

  return response;
}
```

Fig. 5.16 New transport method for the router

```
1  #define STARTUP_ADDRESS 0x800000;
2  volatile int procCounter = 0;
3  void RecursiveHello(int n, int procNumber)
4  {
5      if (n) {
6          printf("Hi from processor PowerPC %d!\n",
               procNumber);
7          RecursiveHello(n - 1, procNumber);
8      }
9  }
10
11 int main(int ac, char *av[]){
12   int procNumber;
13
14   AcquireLock();
15   procNumber = procCounter;
16   procCounter ++;
17   ReleaseLock();
18
19   if (procNumber % 2) {
20       for (i = 0; i < 100000; i ++);
21   }
22   RecursiveHello(10, procNumber);
23   exit(0);
24   return 0;
25 }
```

Fig. 5.17 New Hello World program for a dual core example

`Lock` is as simple as assigning a zero to the lock pointer. These two functions are shown in Fig. 5.18. At Fig. 5.17, Line 15, each processor gets a different number by reading the incrementing the global `procCounter` variable. Notice that this line is protected by the lock. In Lines 19–21, we had to implement a delay loop because the `printf` function we used is not reentrant[1] (if we remove the delay loop, the

[1] A reentrant function is a function that can be called more than once at the same time, providing the expected behavior. Usually, this is accomplishing by not using any global buffer.

```
1  #define LOCK_ADDRESS 0x800000;
2  volatile int *lock = (volatile int *) LOCK_ADDRESS;
3
4  void AcquireLock()
5  {
6      while (*lock);
7  }
8
9  void ReleaseLock()
10 {
11     *lock = 0;
12 }
```

Fig. 5.18 `AcquireLock` and `ReleaseLock` implementations

welcome message will be printed 20 times, but the messages from each processor will be mixed in the output screen).

The screen resulting from running the executable platform representation is shown in Fig. 5.19. Notice that, due to the delay loop, one processor first prints the welcome message 10 times and, only after that, the other processor prints yet another 10 times.

Although this solves the reentrance problem, it is not what we expected by running two processors at the same time. Let us change the code to have the opportunity of interleaving the `printf` functions. To do so, we need to have some concurrence control: we added a new function (`PrintHello`) to print the welcome message and changed the Line 6 of Fig. 5.17 to call it. The `PrintHello` implementation is shown in Fig. 5.20. Notice that we have just called `AcquireLock()` before and `ReleaseLock()` after the `printf`. By running this new code, the 20 welcome messages (10 from processor 0 and 10 from processor 1) will result interleaved.

5.4 The MP3 Example

Up to now, we have seen just a "Hello World" platform. Now we will put together one full example, an MP3 decoder based on a dual core platform. Although decoding a MP3 is not a processor-hungry task, it is a very good example of software that can be partitioned between hardware and software to serve as an example. We will use the MP3 implementation called `dist10`[2].

5.4.1 Profiling

The first required step is to profile the MP3 software decoder to do a hardware-software partitioning. We will use the `gprof`. To use profiling with gprof, the de-

[2]This implementation is considered a base implementation with focus on readability instead of speed.

```
                SystemC 2.2.0
          Copyright (c) 1996-2006 by all Contributors
                      ALL RIGHTS RESERVED
ArchC: Reading ELF application file: hello_dual.x

ArchC: --------- Starting Simulation ---------
ArchC: Reading ELF application file: hello_dual.x

ArchC: --------- Starting Simulation ---------
Hi from processor PowerPC 0!
Hi from processor PowerPC 0!
Hi from processor PowerPC 0!
Hi from processor PowerPC 0!
Hi from processor PowerPC 0!
Hi from processor PowerPC 0!
Hi from processor PowerPC 0!
Hi from processor PowerPC 0!
Hi from processor PowerPC 0!
Hi from processor PowerPC 0!
ArchC: --------- Simulation Finished ---------
Hi from processor PowerPC 1!
Hi from processor PowerPC 1!
Hi from processor PowerPC 1!
Hi from processor PowerPC 1!
Hi from processor PowerPC 1!
Hi from processor PowerPC 1!
Hi from processor PowerPC 1!
Hi from processor PowerPC 1!
Hi from processor PowerPC 1!
Hi from processor PowerPC 1!
ArchC: --------- Simulation Finished ---------
SystemC: simulation stopped by user.
ArchC: Simulation statistics
    Times: 0.18 user, 0.04 system, 0.19 real
    Number of instructions executed: 22563
    Simulation speed: 125.35 K instr/s
ArchC: Simulation statistics
    Times: 0.18 user, 0.04 system, 0.17 real
    Number of instructions executed: 922019
    Simulation speed: 5122.33 K instr/s
```

Fig. 5.19 Hello World output for the Dual Core Platform

```
volatile int canPrint[2] = {0,0};

void PrintHello(int procNumber)
{
  AcquireLock();
  printf("Hi from processor PowerPC %d!\n", procNumber);
  ReleaseLock();
}
```

Fig. 5.20 `PrintHello` function with concurrence control

```
1            1.53 0.00 596520/596520     main [1]
2   [2] 43.2 1.53 0.00 596520        SubBandSynthesis [2]
3            0.00 0.00      3/8          mem_alloc [34]
4            0.00 0.00      1/1          read_syn_window [46]
5   -----------------------------------------------
6            0.03 1.26 1060480/1060480    main [1]
7   [3] 36.4 0.03 1.26 1060480        III_hybrid [3]
8            1.26 0.00 1060480/1060480    inv_mdct [4]
9   -----------------------------------------------
10           1.26 0.00 1060480/1060480    III_hybrid [3]
11  [4] 35.6 1.26 0.00 1060480        inv_mdct [4]
```

Fig. 5.21 Snippet from `gprof` output for `dist10` decoder

coder must be compiled with the `-pg` option passed to gcc (we also recommend the inclusion of the `-g` option for detailed line-by-line profiling). Execution of decoder[3] shall generate a `gmon.out` file. Execute `gprof` and it will output the profiling results. We show some snippets of this file in Fig. 5.21. It is out of the scope of this book to fully interpret the `gprof` output, but it is important to know that this snippet shows 3 of the 46 functions that were called to decode the MP3 file. The functions are: `SubBandSynthesis`, `III_hybrid`, and `inv_mdct`. This information can be easily detected by looking at the Lines 2, 7, and 11 which represent the numbers associated to these three functions. After the number, comes the percentage of the total times spent in these functions, which are 43.2% for `SubBandSynthesis`, 36.4% for `III_hybrid`, and 35.6% for `inv_mdct`, followed by the total time spent inside each function without considering any function called. It is also important to mention the other lines that are used to represent the call graph of each module. For the `SubBandSynthesis` block (Lines 1–4), the report mentions that this function is called only by `main`, which is placed before `SubBandSynthesis` in the block, and two functions are called: `mem_alloc`, and `read_syn_window`.

Based on this very brief description, it is easy to see that if we can move one function to hardware, this function should be `SubBandSynthesis`, because it takes 43.2% of the time (both `mem_alloc` and `read_syn_window` take 0% of the time). If we had enough resources, the second candidate would be `inv_mdct`, because it takes 35.6%. Notice that, although `III_hybrid` takes a larger percentage of time (36.4%), it takes only 0.03 s of the total time. Although the profile data is based on running the application in a specific machine, the results will look very similar when running on the real hardware.

The platform layout will look like Fig. 5.22 with two IPs and two PowerPC processors. In the next section, we will move the original code to the platform.

[3] Please, use a big input file to avoid interference from your environment.

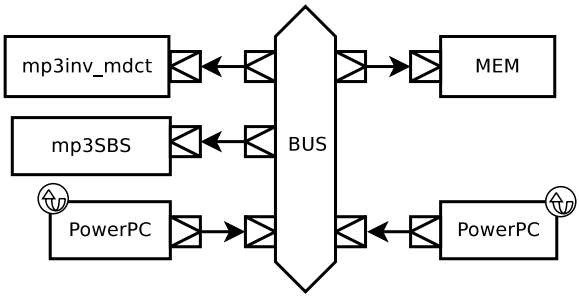


Fig. 5.22 Proposed MP3 Platform with 2 processors and 2 IPs (`SubBandSynthesis` and `inv_mdct`)

5.4.2 Moving the Code to the Platform

Before we continue building the platform, it is necessary to have the `dist10` code running inside our simulator. To do that, we have to make a few changes. The first one is that our platform is dual-core and the `dist10` is not. We have to split the code into two cores. Indeed, we need to split the code into 4 pieces, two processors and two IPs. Before we do that, let us first make the code run inside our platform using only one core. To do that, we change the previous section code (Fig. 5.17) to split the execution flow into two main functions: `main0` and `main1`. Our main function only detects the processor number and executes one of the functions depending on the value read. We have chosen this option because this code snippet can be used into any kind of program you need in the future. We also renamed the `main` function from `dist10` (file `musicout.c`) to `main_mp3` and included a prototype in Line 6 of Fig. 5.23. Giving the same command line parameters, the platform is now capable of decoding an MP3 file.

As you can guess, this software takes a reasonable time to execute, even for small MP3 files, but the results are the same, which is our goal. It is not our goal to run the profiling again because, at this time, the profile would be affected by the simulator, invalidating the reported timing information.

5.4.3 Building the Hardware/Software Interface

The next step is to look into the two functions to see if it is possible to build a platform, by implementing them as non-programmable hardware components. They will only require a few small changes to transfer the parameters and the results. For now on, we are going to call `SubBandSynthesis` as `SBS` and `inv_mdct` as `imdct` for simplicity.

The `SBS` code starts as in Fig. 5.24. The first issue to be solved are the two pointers received as parameters (`bandPtr` and `samples`) but looking at the code, we can see that both `bandPtr` and `samples` are bound by 32, so a memory map will do the job. The second issue in the conversion are the `static` variables, mostly the pointers. Since we will be using a behavior model, some of the internal variables

```
1  #define LOCK_ADDRESS 0x800000;
2  volatile int *lock = (volatile int *) LOCK_ADDRESS;
3  volatile int id = 0;
4  void AcquireLock();
5  void ReleaseLock();
6  int main_mp3(int ac, char *av[]);
7
8  int main0(int procNumber, int ac, char *av[])
9  {
10 }
11 int main1(int procNumber, int ac, char *av[])
12 {
13     main_mp3(ac, av);
14 }
15
16 int main(int ac, char *av[]){
17   register int i, procNumber;
18
19   AcquireLock();
20   procNumber = id++;
21   ReleaseLock();
22   if (procNumber == 0)
23     main0(procNumber, ac, av);
24   else
25     main1(procNumber, ac, av);
26   exit(0);
27   return 0;
28 }
```

Fig. 5.23 New `main()` for the MP3 software

may still be maintained as pointers, but we should avoid those that point to global values outside the module.

Let us start a new IP design named `ac_tlm_sbs` based on the `ac_tlm_mem` as we did to the `ac_tlm_router` (see Sect. 5.3.2 for more information). The goal now is to isolate this function and all the required dependencies. You can do that by creating a new file and moving the function and all its dependencies to this file. Notice that you may have to leave two copies of the same function if it is required by both the IP and the software part. This happened with the `mem_alloc` function while the function `create_syn_filter` could be removed completely.

It is now necessary to build the hardware and the software interfaces for this function. As already mentioned, both `bandPtr` and `samples` are bound by 32 values in the source code loop. So, they can be easily memory-mapped. It is now time to map them to the memory. The router needs to be changed as shown in Fig. 5.25. We just divided the address space into 1 MB blocks and assigned them, as shown in Table 5.1. Notice that the minimum reserved slot is 1 MB, although the lock IP requires only one word (4 bytes). We do that to simplify address decoding in the future hardware implementation.

```
int SubBandSynthesis (bandPtr, channel, samples)
double *bandPtr;
int channel;
short *samples;
{
  register int i,j,k;
  register double *bufOffsetPtr, sum;
  register long foo;
  static int init = 1;
  typedef double NN[64][32];
  static NN FAR *filter;
  typedef double BB[2][2*HAN_SIZE];
  static BB FAR *buf;
  static int bufOffset[2] = {64,64};
  static double FAR *window;
  int clip = 0;
  ...
```

Fig. 5.24 Beginning of the SBS function

```
ac_tlm_rsp transport( const ac_tlm_req &request ) {
  ac_tlm_rsp response;
  int addr = request.addr >> 20;
  switch (addr) {
    case 0x8: response = LOCK_port->transport(request);
              break;
    case 0x9: response = SBS_port->transport(request);
              break;
    case 0xA: response = IMDCT_port->transport(request);
              break;
    default:  response = MEM_port->transport(request);
              break;
  }
  return response;
}
```

Fig. 5.25 New router to support the 2 new IPs

Table 5.1 MP3 platform address space

From	To	Component
0x0000000	0x07FFFFF	System memory
0x0800000	0x08FFFFF	Lock IP
0x0900000	0x09FFFFF	SBS IP
0x0A00000	0x0AFFFFF	IMDCT IP

Let us now map the SBS IP to the memory space. Table 5.2 shows the address mapping of the three input parameters and the output value. We have aligned the parameters in a 256-byte boundary for didactic purposes.

Table 5.2 SBS memory addresses

From	To	Parameter
0x0900000	0x09000FF	`double bandPtr[32]`
0x0900100	0x0900103	`int channel`
0x0900200	0x090027F	`short samples[32]`
0x0900300	0x0900303	`int return_value`

This address map will be needed both in the hardware part and in the software. Figure 5.26 shows the code snippet that we will include in the software whenever accessing the peripheral. We also include it in the peripheral itself to help decoding the address space.

The address mapping is only one part of the interface. Now we know how to send and receive data to our IP. The next step is to define its programming model. From a function call view, we first must pass all parameters and only after that we can read the result. We also want to allow the program to read the values recently written to the parameters, although it is not necessary in this module. To do that, we will maintain the `memory` variable inside the `ac_tlm_mem` we copied as `ac_tlm_sbs`. We then update the `writem` and `readm` methods to reflect the SBS behavior.

The `writem` method, shown in Fig. 5.27, acts in a very similar way to the `ac_tlm_mem`, except from the fact that the memory address must have its base address removed (Line 3[4]). We use byte copying so as to make easy to change the endian if necessary. However, it is a very good optimization to use word copying whenever possible.

Figure 5.28 shows the `readm` method. We implemented the SBS call whenever the module receives a read request to the return value address (Line 9). To do so, we assign the pointers with the correct values and call the SBS function (Line 15), storing the result in the return value so that the next 4 lines always return the correct values.

The last step is to call the SBS from our program. Instead of changing Fig. 5.23, we will change the current SBS software implementation (Fig. 5.29). This function can be divided into three parts: variable declaration, data transfer, and read the result.

```
1  #define SBS_BASE_ADDRESS 0x0900000
2  #define SBS_TOTAL_SIZE 0x400
3  #define SBS_bandPtr (0)
4  #define SBS_channel (0x100)
5  #define SBS_samples (0x200)
6  #define SBS_return_value (0x300)
```

Fig. 5.26 SBS Address map definition in the C/SystemC code

[4]The base address removal could be done in the router also. Sometimes it is easier to do there and also make the platform more flexible.

```
ac_tlm_rsp_status ac_tlm_sbs::writem( const uint32_t &a ,
    const uint32_t &d )
{
  int addr = a - SBS_BASE_ADDRESS;

  memory[addr]   = (((uint8_t*)&d)[0]);
  memory[addr+1] = (((uint8_t*)&d)[1]);
  memory[addr+2] = (((uint8_t*)&d)[2]);
  memory[addr+3] = (((uint8_t*)&d)[3]);

  return SUCCESS;
}
```

Fig. 5.27 The `writem` method to send parameters to the SBS module

```
ac_tlm_rsp_status ac_tlm_sbs::readm( const uint32_t &a ,
    uint32_t &d )
{
  int *return_value;
  double *bandPtr;
  int *channel;
  short *samples;
  int addr = a - SBS_BASE_ADDRESS;

  if (addr == SBS_return_value) {
    bandPtr = (double *) &(memory[SBS_bandPtr]);
    channel = (int *) &(memory[SBS_channel]);
    samples = (short *) &(memory[SBS_samples]);
    return_value = (int *) &(memory[SBS_return_value]);

    *return_value = SubBandSynthesis(bandPtr, *channel,
        samples);
  }
  (((uint8_t*)&d)[0]) = memory[addr];
  (((uint8_t*)&d)[1]) = memory[addr+1];
  (((uint8_t*)&d)[2]) = memory[addr+2];
  (((uint8_t*)&d)[3]) = memory[addr+3];

  return SUCCESS;
}
```

Fig. 5.28 The `readm` method to receive results from the SBS module

Variable declaration where we initialize the pointers to the correct position in memory (the values were defined in Fig. 5.26).

Data transfer Transfer the `bandPtr`, `samples`, and `channel` to the hardware device using pointers. Notice that it is not necessary to maintain any order, since the processing will be done only when the return value is read.

```
int SubBandSynthesis (double *bandPtr, int channel, short *
    samples)
{
    int i;
    double *target_bandPtr = (double *) (SBS_BASE_ADDRESS +
        SBS_bandPtr);
    int *target_channel = (int *) (SBS_BASE_ADDRESS +
        SBS_channel);
    short *target_samples = (short *) (SBS_BASE_ADDRESS +
        SBS_samples);
    int *return_value = (int *) (SBS_BASE_ADDRESS +
        SBS_return_value);

    for (i = 0; i < 32; i ++) {
      target_bandPtr[i] = bandPtr[i];
      target_samples[i] = samples[i];
    }
    *target_channel = channel;
    return *return_value;
}
```

Fig. 5.29 The new SBS software implementation

```
void inv_mdct(in, out, block_type)
double in[18];
double out[36];
int block_type;
{
  int     k,i,m,N,p;
  double  tmp[12],sum;
  static  double  win[4][36];
  static  int init=0;
  static  double COS[4*36];
  ...
```

Fig. 5.30 Beginning of the `imdct` function

Read the result — This is done in the last line, by returning the `return_value` variable.

This code could be improved by using a technique called *double buffer*. In this technique, the hardware and the software work in parallel, by duplicating the address space so that the hardware can work on one set of data while the software is generating the next set. This technique can also be generalized as `multi-buffer`, when the number of buffers is higher than 2.

The `imdct` function can be transfered to hardware using the same technique mentioned before. Figure 5.30 shows the beginning of the function. As in the SBS, the parameters are passed as pointers but, in this case, the return value is also one of the parameters (the `out` vector).

```
1  #define IMDCT_BASE_ADDRESS 0x0A00000
2  #define IMDCT_TOTAL_SIZE 0x400
3  #define IMDCT_in (0)
4  #define IMDCT_out (0x100)
5  #define IMDCT_block_type (0x300)
6  #define IMDCT_answer_ready (0x304)
```

Fig. 5.31 IMDCT Address map definition in the C/SystemC code

```
1  ac_tlm_rsp_status ac_tlm_imdct::writem( const uint32_t &a ,
       const uint32_t &d )
2  {
3    int addr = a - IMDCT_BASE_ADDRESS;
4
5    memory[addr]   = (((uint8_t*)&d)[0]);
6    memory[addr+1] = (((uint8_t*)&d)[1]);
7    memory[addr+2] = (((uint8_t*)&d)[2]);
8    memory[addr+3] = (((uint8_t*)&d)[3]);
9
10   *answer_ready = 0;
11
12   return SUCCESS;
13 }
```

Fig. 5.32 The `writem` method to send parameters to the IMDCT module

To model the implementation of `imdct` as a non-programmable hardware component, we must follow the same steps as we did for the `sbs`, resulting in the code in Figs. 5.31, 5.32, 5.33, and 5.34. In the following, only the differences will be described.

Instead of having one return value, the IMDCT has the parameter `out` which is used to return 36 doubles. So, we trigger the `inv_mdct` function execution by a read in any element of the `out` vector, but we do not want to execute this function for every element, since they are calculated all at once. We declared a new class property called `answer_ready`, initialized with `false` (0). This property is mapped into the internal memory. Every time we get a write operation in our device, we assign `false` (0) to `answer_ready` (Fig. 5.32, Line 10), indicating that a read in the `out` must first recalculate the outputs. After we recalculate the outputs, we assign `true` (1) to `answer_ready` (Fig. 5.33, Line 14) to indicate that the next read can be done without recalculation.

To execute the IP implementation (Fig. 5.34), we first send `in`, and `block_type` and then we read `out` to get the output values. Notice that we do not send the `out` values to the IP because we looked into `inv_mdct` and found that these values are not used as input.

So far we have finished the parallel IP design. But it is worth to mentioning an alternative approach to implement such devices that are really useful in some platforms. We designed the SBS to fire its execution when the user read the output value `return_value`. This will make the IP to recalculate the output if we read

```
ac_tlm_rsp_status ac_tlm_imdct::readm( const uint32_t &a ,
    uint32_t &d )
{
  double *in, *out;
  int *block_type;

  int addr = a - IMDCT_BASE_ADDRESS;

  if ((addr == IMDCT_out) && (addr < IMDCT_block_type) &&
     (! *answer_ready)) {
    in = (double *) &(memory[IMDCT_in]);
    out = (double *) &(memory[IMDCT_out]);
    block_type = (int *) &(memory[IMDCT_block_type]);

    inv_mdct(in, out, *block_type);
    *answer_ready = 1;
  }
  (((uint8_t*)&d)[0]) = memory[addr];
  (((uint8_t*)&d)[1]) = memory[addr+1];
  (((uint8_t*)&d)[2]) = memory[addr+2];
  (((uint8_t*)&d)[3]) = memory[addr+3];

  return SUCCESS;
}
```

Fig. 5.33 The `readm` method to receive results from the IMDCT module

```
void inv_mdct(double in[18], double out[36], int block_type)
{
    int i;
    double *target_in = (double *) (IMDCT_BASE_ADDRESS +
        IMDCT_in);
    double *target_out = (double *) (IMDCT_BASE_ADDRESS +
        IMDCT_out);
    int *target_block_type = (int *) (IMDCT_BASE_ADDRESS +
        IMDCT_block_type);

    for (i = 0; i < 18; i ++)
      target_in[i] = in[i];

    *target_block_type = block_type;

    for (i = 0; i < 36; i ++)
        out[i] = target_out[i];
}
```

Fig. 5.34 The new IMDCT software implementation

the `return_value` twice in sequence, without changing the input. To solve this, we may also include a control register to fire the execution at a software command. Usually, when we do that, we also include a second register to show the IP status, if

Table 5.3 SBS memory addresses

From	To	Parameter
0x0900000	0x09000FF	`double bandPtr[32]`
0x0900100	0x0900103	`int channel`
0x0900200	0x090027F	`short samples[32]`
0x0900300	0x0900303	`int return_value`
0x0900304	0x0900307	`int fire_execution`
0x0900308	0x090030B	`int answer_ready`

it has the output value already calculated. This second register may not be necessary this time since we are designing a functional module which does not follow the correct timing. But we are going to include both new variables in this example to serve as a basis for your future designs. Notice also that these two registers could be unified. For the sake of simplicity, we are not going to merge them. Table 5.3 presented a revised version of Table 5.2 including two new registers. The #define's of Fig. 5.26 must also be changed accordingly.

A new change is also required in the `readm` and `writem` behaviors. Basically, we need to move the `SubBandSynthesis` call from `readm` to `writem` to use the new `fire_execution` register. The new versions are in Fig. 5.35 and Fig. 5.36.

Now that we have the two IPs implemented in our platform, the next step is to create a parallel version of the software to run on our two processors, which is the focus of the next section.

5.4.4 The Next Steps Towards Parallel Software

To make a parallel version of the MP3 code, the following steps must be executed:

1. Find the places to split it into two or more different parts;
2. Move the variables declaration so as to share them with all the threads;
3. Define the communication protocol;
4. Define the end-of-program flag and use it in all the threads.

The fist step can be made easy if we use the IP calls as boundaries. In this way, since we have two IPs, it would be easier to divide the code into 3 parts and use 3 processors. However, it is not necessarily the best alternative, because we may end with a very small piece of code to handle one of the parts. In spite of that, let us look first to this alternative.

The MP3 decoder is essentially a loop after the initialization code. The two IPs are called inside `III_hybrid` and `SubBandSynthesis`. So the first processor should do the initialization part (everything prior to the `III_hybrid`), and then flag the second one to send the data to execute `III_hybrid`, sending the data to `inv_mdct` and doing everything up to `SubBandSynthesis`, where it should

```
ac_tlm_rsp_status ac_tlm_sbs::writem( const uint32_t &a ,
    const uint32_t &d )
{
  double *bandPtr = (double *) &(memory[SBS_bandPtr]);
  int *channel = (int *) &(memory[SBS_channel]);
  short *samples = (short *) &(memory[SBS_samples]);
  int *return_value = (int *) &(memory[SBS_return_value]);
  int *fire_execution = (int *) &(memory[SBS_fire_execution])
      ;
  int *answer_ready = (int *) &(memory[SBS_ready]);

  int addr = a - SBS_BASE_ADDRESS;

  memory[addr]   = (((uint8_t*)&d)[0]);
  memory[addr+1] = (((uint8_t*)&d)[1]);
  memory[addr+2] = (((uint8_t*)&d)[2]);
  memory[addr+3] = (((uint8_t*)&d)[3]);
  if ((addr == SBS_fire_execution) && (*fire_execution != 0))
       {
      *return_value = SubBandSynthesis (bandPtr, *channel,
          samples);
      *answer_ready = 1;
  }
  else {
      *answer_ready = 0;
      *fire_execution = 0;
  }
 return SUCCESS;
}
```

Fig. 5.35 The new `writem` method to send parameters to the SBS module

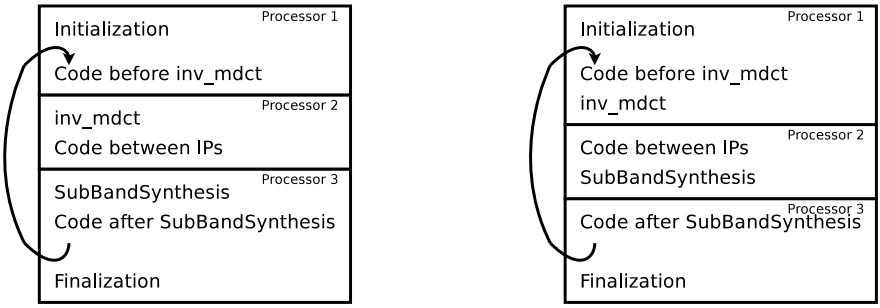
```
ac_tlm_rsp_status ac_tlm_sbs::readm( const uint32_t &a ,
    uint32_t &d )
{
  int addr = a - SBS_BASE_ADDRESS;

  (((uint8_t*)&d)[0]) = memory[addr];
  (((uint8_t*)&d)[1]) = memory[addr+1];
  (((uint8_t*)&d)[2]) = memory[addr+2];
  (((uint8_t*)&d)[3]) = memory[addr+3];

  return SUCCESS;
}
```

Fig. 5.36 The new `readm` method to receive results from the SBS module

flag the third processor do execute the `SubBandSynthesis` and end the loop. At the end of the processing, the last processor should also finalize the execution. This sequence is demonstrated in Fig. 5.37a. It is worth mentioning the alternative implementation of Fig. 5.37b, where the IP calls are moved to the first and second



(a) Firts alternative (b) Second alternative

Fig. 5.37 Parallel implementation with three processors

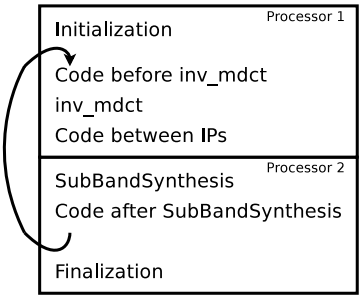


Fig. 5.38 Parallel implementation with two processors

processors. The ideal solution will be the one with the small overhead and also the smallest execution time (to be evaluated with a new profiling).

When considering only two processors, the division of the code between them will also require a new profile. We show one alternative implementation in Fig. 5.38.

The second step, moving the variables declaration is a straightforward one. It is only necessary to check for each variable in `main_mp3` (the old `main` function we renamed), which variable crosses the boundaries between two processors. For each of them, its declaration must be made global and the first processor should give it the default value.

The third step, defining the communication protocol, is a little tricky because the synchronization issues not only from the IPs but also from shared variable usage. Indeed, the IP calls and returns should be safe since they are called by only one of the processors. So, for every variable that will need a specific control, you must declare the required auxiliary variables and patch the code accordingly. Figure 5.20 shows one alternative implementation we did in Sect. 5.3.2. To speedup the software execution, you may use a technique called double-buffer, that interchanges two buffers between the communicating modules so that while the first module is writing in one buffer the second one is reading from the previous buffer. This technique allows for a very parallel execution and can be applied to this code.

The last step is to synchronize the end of execution. In our platform, the MP3 file is opened by the first processor and the output file should be written and closed by the last processor. In a real hardware platform, you probably will not want to end the programs, but put the processors in a low power mode, which may also need some synchronization. The synchronization can be done similarly to the previous step, with each processor using one variable to flag its end and checking for the other flags before waiting for new data to process.

This platform will be used in Chap. 7 to illustrate the debug infrastructure.

Chapter 6
Retargetable Binary Tools

Alexandro Baldassin, Paulo Centoducatte, and Luiz Santos

6.1 Introduction

Binary tools is the name usually given to the set of tools which manipulates binary files. They can be roughly divided into code generation and code inspection tools. For instance, the assembler and linker are employed to generate object code whereas the debugger is used to inspect the generated code.

Due to their nature, binary tools are heavily tied to a particular processor architecture and, to some degree, to the operating system. It is the responsibility of the application binary interface (ABI) to set the rules that allow an object code to run on a given operating system. The same object code can be executed, without any modification, on any system employing a compatible ABI. The format in which the binary code is stored also plays an important role. Such object code formats come in different flavours depending on the underlying system. The ELF [3] (Executable and Linking Format) is worth mentioning due to its widespread use in embedded systems. ELF is also the current object file format for binaries in the Linux operating system.

Binary tools are essential in a platform-based design, specially in the early development of hardware-dependent software (HDS) since it enables actual hardware-software co-design. Given a platform, design exploration is crucial to fulfilling func-

A. Baldassin (✉) · P. Centoducatte
Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil
e-mail: alebal@ic.unicamp.br

P. Centoducatte
e-mail: ducatte@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

S. Rigo et al. (eds.), *Electronic System Level Design*,
DOI 10.1007/978-1-4020-9940-3_6, © Springer Science+Business Media B.V. 2011

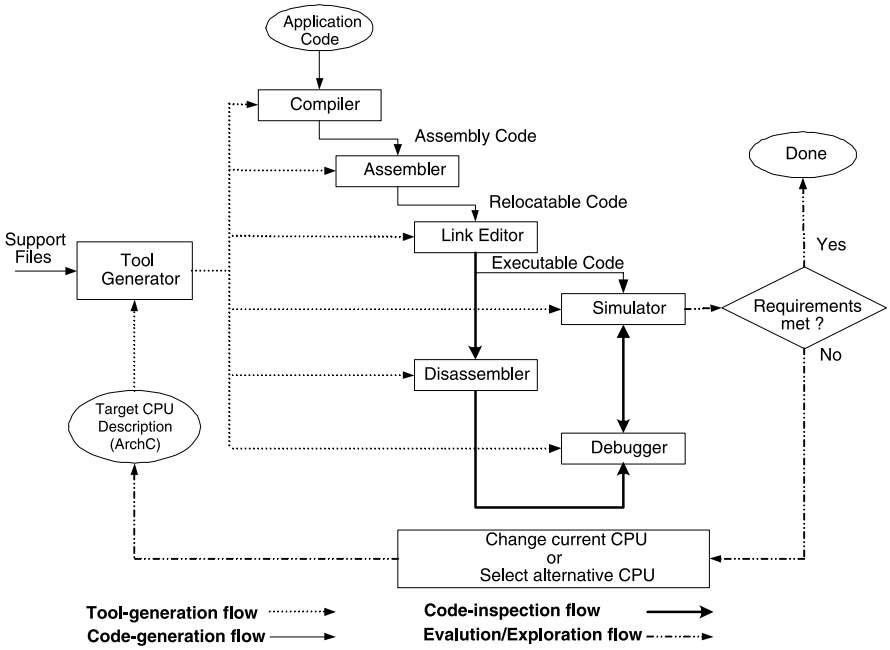


Fig. 6.1 CPU exploration flow with ArchC

tional requirements, as well as real-time, low-power and code-size constraints. Design exploration may require code generation and inspection for several alternative CPUs, possibly including application-specific instruction-set processors (ASIPs). In this context, the availability of binary tools for the most popular general purpose processors does not help much. Besides, the task of manually retargeting binary tools for each new explored CPU would not be affordable under the time-to-market pressure.

The ArchC package provides a tool capable of automatically generating the most important binary utilities, namely an assembler, linker, disassembler and debugger. The generation process is guided by a high-level description of the target architecture and a set of support files. Figure 6.1 shows the overall exploration flow of CPUs in a platform-based design employing ArchC and its generated tools.

First of all, notice in Fig. 6.1 the existence of four different flows. The tool-generation flow starts off with the description of the target CPU and a set of support files. These files are used as input to the tool generator which in turn generates the binary tools. It should be noticed that compiler generation is not currently supported. A simulator can be generated as previously seen in Chap. 4. The code-generation flow translates source files (written in a high-level programming language) into an executable file. Several tools are employed during this process. A compiler usually translates the high-level source file into a set of assembly source code files. These files are in turn transformed into relocatable object files by the assembler. Finally, the linker creates an executable file using the previously generated relocatable object

Table 6.1 Retargeting effort

	ArchC	Generated	Gain
MIPS	284	2226	7.8
SPARC	568	2701	4.7
PowerPC	559	5767	10.3
i8051	424	2508	5.9
PIC16F84	138	2137	15.5

code. The executable code can be run in the instruction-set simulator and its output checked to see if requirements are met. If that is not the case, then the current CPU might be changed or even replaced by another one. This last flow is described in the figure as the evaluation/exploration flow. At last, the code-inspection flow is commonly used to check the correctness of the executable code. The debugger can be plugged into the instruction-set simulator and the code can be inspected as the simulation runs.

To highlight the importance of automatic generation of binary tools, Table 6.1 shows the productivity gain achieved by using the ArchC generation tool when compared to generated code. The numbers represent lines of source code for five well-known architectures. For instance, for the MIPS architecture, the table shows that the user has written 284 lines of source code and the ArchC tool has generated 2226 lines of code, representing a gain of 7.8. The table suggests that, on average, manual retargeting would require 8.8 times more effort.

The rest of this chapter is organized as follows. Section 6.2 introduces the available language constructs related to binary tools. Section 6.3 describes the technique developed to retarget the GNU binutils tools. Finally, Sect. 6.4 presents a quick guide to get the tools up and running.

6.2 Language Support for Binary Tools

This section describes which information should be made available so that automatic generation of binary tools can be carried out. The generator requires a few new language constructs and support files to be created. It is important to bear in mind that the additional information is not mandatory if only simulators are to be generated.

Overall, there are 7 language constructs required for binary tools generation: `set_endian`, `ac_format`, `ac_instr`, `set_decoder`, `ac_asm_map`, `set_asm`, `pseudo_instr`. The first four of them (which have already been described in Chap. 2 and Chap. 4) are also used by the simulator generator. On the other hand, the remaining three constructs are specific to binary tools and must be specified in the `AC_ISA` part of the processor model. These constructs along with two support files hold all the necessary information for binary tools generation, providing details about processor-specific assembly language symbols (such as register

```
1   ac_asm_map reg {
2     "%r"[0..31] = [0..31];
3     "%g"[0..7] = [0..7];
4     "%o"[0..7] = [8..15];
5     "%l"[0..7] = [16..23];
6     "%i"[0..7] = [24..31];
7     "%fp" = 30;
8     "%sp" = 14;
9   }
```

Fig. 6.2 SPARC-V8 register names and encoding values

names), instruction syntax, operand encoding/decoding as well as synthetic instructions.

6.2.1 Assembly Language Symbols

The `ac_asm_map` language construct allows one to specify the assembly language symbols and their corresponding encoding values. A set of symbols can be grouped and given a unique identifier. The identifier can be referenced later when instruction operands are described with `set_asm` , allowing the assembler to recognize only the specific set of symbols as valid.

As an example, consider Fig. 6.2, which specifies the SPARC register names and their corresponding encoding values. In this particular example the mapping identifier is `reg` (Line 1). This identifier can be used later when the instruction operands are described (Sect. 6.2.2) in order to constrain the set of valid symbols recognized by the assembler.

6.2.2 Instruction Syntax and Operand Encoding

The instruction syntax and encoding information are specified by means of two special methods of an `ac_instr` object: `set_asm` and `set_decoder`. The former specifies both assembly language syntax and operand encoding, while the latter specifies the opcode. Notice that `set_decoder` has already being described in Chap. 4 and therefore this section only explains how `set_asm` works.

Figure 6.3 shows the general form of `set_asm`. Here, `insn` is an ArchC instruction whose assembly syntax and operand encoding are being defined. The construct is split into a *syntax string* and an optional *field list*. The syntax string (`"mno %op1,%op2"`) is always specified between double-quotes and is made up of literal characters (`mno`, `,`) and operand identifiers (`%op1`, `%op2`). The set of characters up to the first whitespace constitutes the instruction mnemonic (`mno`). Operand identifiers always start with the special character `%` and act as placeholders for binary values assigned at assembling and/or linking time. The place in the instruction where

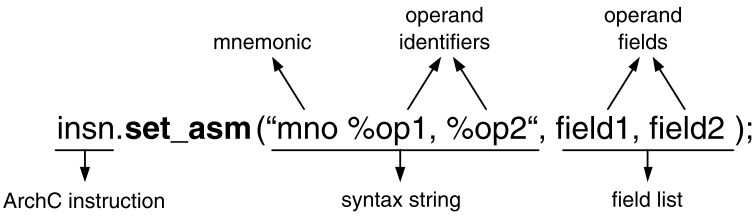


Fig. 6.3 Description of an arbitrary instruction

these values are encoded is specified in the field list. For each operand identifier there must be a corresponding operand field. In the given example, the value hold by `op1` will be placed in instruction field `field1`, while the value hold by `op2` will be placed in instruction field `field2`.

The `set_asm` construct specifies how the generated assembler will parse an assembly source code file and emit the binary code. The purpose of an operand identifier is twofold: first, it tells the assembler which set of symbols should be regarded as valid; second, it provides the corresponding encoding value. For instance, consider the following declaration:

```
<insn>.setasm("insn %reg", f1);
```

Assume we are using the register mapping declared in Fig. 6.2. The operand `reg` forces the assembler to accept only the symbols defined in the aforementioned mapping. For instance, the instruction `toy %l7` will be accepted by the assembler and the corresponding value (23) will be encoded in the instruction field `f1`. Now consider the instruction `toy label1`. It will cause the assembler to emit an error since the symbol `label1` is not defined in the `reg` mapping.

To simplify the description of operands, the ArchC language provides the following built-in operand identifiers:

- **imm**: used to represent integer-like immediate operands.
- **addr**: used for symbolic address operands. In this case, the assembler will not try to extract the symbol value at assembling time but rather emit a relocation entry for the symbol.
- **exp**: used for expressions (a combination of immediate and symbolic operands). For instance, the operand `label1 + 10` is regarded as an expression operand (`label1` is a symbolic address whereas `10` is an immediate value).

6.2.3 Instruction Encoding and Modifiers

The instruction encoding process is performed primarily by the assembler and optionally by the linker (if relocation is present). As explained previously, at assembling time the assembler extracts the value from the operand identifier and encodes it in the corresponding instruction field specified in the field list. If the value is

larger than the field's capacity (i.e. the value requires more bits than available) the extra bits are discarded. Similarly, if the value requires less bits than available in the instruction field, then the extra bits will be cleared.

This aforementioned scheme is the default encoding behavior. It handles the commom case, but it may not suffice if a transformation should be applied to an operand value before encoding takes place. Such a case happens with pc-relative operands, where the encoding value is the result of the subtraction of the instruction address (probably added to an offset) from the symbol value. To deal with non-conventional cases, ArchC introduces the notion of *modifiers*. A modifier is a function that transforms a given operand value. If a modifier is specified, the assembler and/or linker first executes the modifier code using the original operand value as input. The modifier output is then used as the final encoding value.

In ArchC, a modifier can be attached to any operand identifier. All one has to do is to specify a modifier name and an optional addend after the operand identifier between parentheses. For instance, the following description:

```
1  ba.set_asm("ba %exp(pcrel)", disp22, an=0);
```

specifies the SPARC instruction `ba` (branch always) with an operand of type `exp`. A modifier named `pcrel` is assigned to this operand, meaning that the operand value must be transformed by the `pcrel` modifier. The modifier code is specified outside the ArchC model, in a file named `modifiers` living in the same directory as the ArchC source files. Two versions need to be specified: one for encoding (used by the assembler and linker) and another for decoding (used by the disassembler and debugger). The code should be written in the C language.

Figure 6.4 shows the description of the `pcrel` modifier. The keywords `ac_modifier_encode` and `ac_modifer_decode` are used to specify the encoding and decoding modifiers, respectively. The name of the modifier must follow the keyworks inside parentheses (Lines 1 and 6). At least 4 special variables are defined within the modifier context, accessed through the `reloc` pointer: `input` contains the operand value; `address` contains the instruction address at assembling or linking time; `addend` contains an optional value defined as part of the modifier (not used in the SPARC example); and `output` contains the modifier's result. In Line 3 of Fig. 6.4 one can see the C code for the encoding modifier `pcrel`. The encoding value (`reloc->output`) is computed by subtracting the current instruction address (`reloc->address`) from the symbol value (`reloc->input`). Since the value is stored in words (4 bytes), an additional shift to the right by 2 bits must be performed (Line 3). The decoding modifier is similarly defined in Line 8.

Modifiers can represent complex encoding schemes. It is also possible to have direct access to the instruction formats and fields (declared with `ac_format`) inside a modifier. This will come in handy if multiple fields must have their values set, since a single `output` variable will not suffice. As an example, consider the immediate data processing operands of the ARM architecture. One single operand may have multiples encoding values and must be encoded into two different instruction fields. The declaration of such an instruction will be as follows:

```
1  ac_modifier_encode(pcrel)
2  {
3    reloc->output = (reloc->input - reloc->address) >> 2;
4  }
5
6  ac_modifier_decode(pcrel)
7  {
8    reloc->output = (reloc->input << 2) + reloc->address;
9  }
```

Fig. 6.4 Modifier code (SPARC)

```
1  ac_modifier_encode(aimm)
2  {
3    unsigned int a;
4    unsigned int i;
5
6  #define rotate_left(v, n) (v << n | v >> (32 - n))
7
8    for (i = 0; i < 32; i += 2)
9      if ((a = rotate_left (reloc->input, i)) <= 0xff) {
10       reloc->Type_DPI3.rotate = i >> 1;
11       reloc->Type_DPI3.imm8 = a;
12       return;
13     }
14
15   reloc->error = 1;
16 }
```

Fig. 6.5 Complex modifier code (ARM)

```
1  and3.set_asm("and %reg, %reg, #%imm(aimm)", rd, rn, rotate
       +imm8);
```

Note that the third operand (`imm`) is bound to the pre-defined fields `rotate` and `imm8` (the symbol + is used here for field concatenation). The modifier `aimm` is attached to the operand identifier and its code is presented in Fig. 6.5. Note that variables and common C structures such as loops (Line 8) can be used inside the modifier. Since the encoding affects 2 fields, a single `output` variable is not sufficient. The code hence accesses the instruction formats and fields directly (Lines 10 and 11).

```
1   pseudo_instr("ble %reg, %reg, %exp") {
2     "slt $at, %1, %0";
3     "beq $at, $zero, %2";
4   }
5
6   pseudo_instr("mul %reg, %reg, %imm") {
7     "addiu $at, $zero, %2";
8     "mult  %1, $at";
9     "mflo  %0";
10  }
```

Fig. 6.6 Defining synthetic instructions (MIPS)

6.2.4 Pseudo Instructions

Synthetic instructions, or pseudo instructions, are created based on other previously defined native instructions. ArchC provides the `pseudo_instr` construct for the purpose of defining the corresponding native instruction(s).

The first step in describing a synthetic instruction is to declare its syntax. Note that only the syntax string is necessary. The operand field is not specified since the pseudo instruction does not have *real* fields. Following the syntax string, a list of native instructions (those defined with `set_asm`) is specified. Parameters from the pseudo instruction syntax can be used by the native ones by employing the `%` character and indicating which parameter from the pseudo must be replaced (similar to the `macro` construct used by GNU assemblers).

Figure 6.6 shows two definitions of synthetic instructions used in the MIPS model. The first one, Lines 1 to 4, create the pseudo instruction `ble`, which uses three operands. It is defined as a sequence of two native instructions (Lines 2 and 3): `slt` and `beq`. The character `%` indicates a substitution of parameters. For example, the instruction `slt` in Line 2 uses the literal `$at` as the first operand, the second (`%1`) is the string associated with the second `%reg` in the pseudo instruction definition, and the third operand (`%0`) is associated with the first pseudo instruction operand.

The second synthetic instruction definition, Lines 6 to 10, creates the instruction `mul` with 3 operands. When an instruction such as `mul $2, $3, 10` is found by the generated assembler, it will be expanded into the following three:

```
1 addiu $at, $zero, 10
2 mult $3, $at
3 mflo $2
```

6.3 Binary Tools Retargeting

This section presents the retargeting technique developed to generate the binary tools. The GNU binutils framework is first introduced in order to give the reader an overview of the main components of this important package. After that the text delves into the retargeting technique itself.

6.3.1 GNU Binutils Package

The GNU binutils package [1] is a collection of tools aimed at binary file manipulation. Its main tools comprise an assembler (gas) and a link editor (ld). In addition, the package contains library managers (ar and ranlib), object file inspectors (objdump and readelf) and some other minor tools.

In general, a GNU binutils tool has a machine-independent module (the core), which provides the main tool operation and control flow, and a machine-dependent module, which implements processor-specific operations. Current processor support in the package includes RISC, and CISC architectures (AMD 29k, Alpha, ARM, Intel i386, Motorola 680x0, PowerPC, SPARC and TMS320C54x).

The package's tools rely on two main libraries known as *Opcodes* and *Binary File Descriptor* (BFD). In order to retarget a tool to a new processor, the machine-dependent code must be implemented and the main libraries may require extensions.

The purpose of the Opcodes library is to store information related to a processor's ISA. Since there is no standard format for this library, every processor port usually creates its own. In general, the library is described as a large table where each line represents an instruction, and the columns contain information such as opcodes, operand and instruction types, decoding information, and so on. The library also provides an optional application programming interface (API) for decoding purposes. The GNU debugger (gdb) and the GNU disassembler (objdump) are among the tools relying on this API. For them, the API routines must be retargeted.

The BFD library provides a set of generic routines to operate on object files, regardless of the adopted binary format. Figure 6.7 shows how the library is organized. Application programs interface with the library's front-end, which provides a set of format-independent routines to manipulate object files. The front-end is responsible for calling the proper back-end routine which in turn executes format-dependent operations on the object file.

The main advantage of the framework provided by the BFD library is that, once an object file format is implemented in the backend, it can be reused by all processor ports. Therefore, although processor-specific information (such as relocation) is still required, the overall porting effort is substantially reduced.

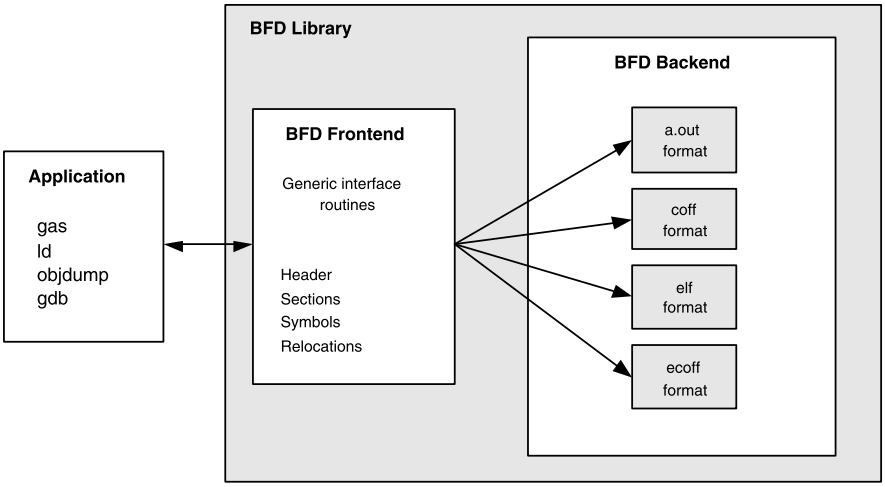


Fig. 6.7 BFD library organization

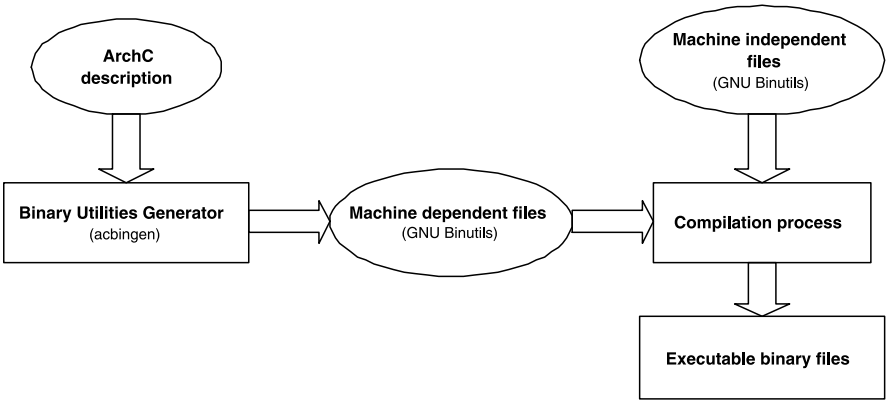


Fig. 6.8 Generation flow of binary utilities

6.3.2 Automatic Binutils Retargeting

The overall generation flow of the binary utilities is summarized in Fig. 6.8. Given an ADL description of the target CPU, the binary utility generation tool, henceforth called `acbingen`, outputs a set of machine-dependent files for the GNU binutils framework. These files are merged into the original binutils source tree and compiled, thereby generating the executable binary utilities for the target processor.

Sections 6.3.3 and 6.3.4 describe in detail the automated retargeting process for the Opcodes and BFD libraries. The generation of the main binary utilities, assembler and link editor, is discussed in Sects. 6.3.5 and 6.3.6, respectively.

6.3.3 Opcodes Library Generation

The implementation of this library is done in an *ad hoc* way by current CPU ports. We have created a standard format for this library consisting of three tables. The tables and the ArchC language constructs used to generate them are:

- `Opcodes`: stores instruction mnemonic, operand types and syntax, base binary image and instruction format. If the instruction is synthetic, it also stores a pointer to the native instructions in the `Synthetic Instructions` table (described next). This table is built from the following ArchC constructs: `set_asm`, `set_decoder`, `pseudo_instr` and `ac_format`;
- `Synthetic Instructions`: for each synthetic instruction declared with the `pseudo_instr` keyword, this table stores all the native instructions assigned to it;
- `Assembly Symbols`: a table representing the mapping between symbols and values, as declared with the `ac_asm_map` ArchC construct.

6.3.4 BFD Library Generation

Throughout this Section the ELF object file format is used to illustrate the retargeting concepts. It should be noticed that the same principles could be applied to any other object file format without loss of generality.

The format-independent routines of the ELF format are already implemented and used by a large number of CPU ports. The format-dependent routines are automatically generated by `acbingen` and deal mainly with relocation issues. Relocation information consists essentially in describing which instruction fields must be edited and how they should be changed. This information is collect from the ArchC source files where they are described by means of modifiers. The BFD library provides a *relocation table* which contains parameters describing the relocation resolution mechanism. These parameters are generated by `acbingen` from the information provided by the ArchC construct `set_asm`. Given an instruction operand requiring link edition, the resolution mechanism is derived according to the rules encoded by the ADL modifiers.

The steps performed by `acbingen` to automatically generate the relocation table entries are the following:

- Step 1: For each instruction found in the ADL description, check if there are relocatable operands (identified by the built-in types `addr` or `exp` in the assembly syntax);
- Step 2: For each relocatable operand, generate a relocation directive according to the modifiers described in the support file. Check if there exists an equivalent relocation directive in the relocation table. If not, insert the new directive in that table.

The relocation table is completely abstracted by the retargeting technique, i.e., the user does not even know it exists. When describing an operand field of an instruction, the user can rely on the ADL modifiers to compactly encode its relocation mechanism.

6.3.5 Assembler Generation

The assembler generated by `acbingen` is able to output relocatable object files, which are then merged into a single executable object file by the link editor. The whole process is shown in Fig. 6.9.

Figure 6.9(a) shows the structure of the assembler. Modules marked with an asterisk (`*`) are automatically generated by `acbingen`. The assembler core module provides machine-independent functionality (such as parsing of labels, directives and expressions). Whenever an architecture-specific operation has to be performed (such as instruction parsing and encoding), the core module invokes a so-called *hook* function, whose implementation lies in the target specific assembler module.

A generic parsing algorithm was implemented and is generated for each target CPU. The parsing of an assembly source file is guided by the information stored in the Opcodes library. Once an instruction is syntactically validated, binary code is emitted and a relocation entry is created for each relocatable operand (according to its type) using the BFD library. To give an idea of the work involved in retargeting this module, there are around 25 definitions, 10 global variables and 40 functions which are machine-dependent and are generated by `acbingen`.

6.3.6 Link Editor Generation

Figure 6.9(b) shows the link editor structure. Its main purpose is to transform one or more relocatable object files into an executable object for the target CPU.

The core module starts by reading each input object file and creates two main tables: a segment table (with information of the sections defined in the input files) and a global symbol table (with exported and imported symbols). Using the data contained in these tables and the relocation information, the linker once again pass through all the input files resolving symbols, assigning addresses to the segments and adjusting instruction fields according to the relocation entries. It then outputs an executable object file.

The BFD library is used to read and write the object files. It also contains the relocation resolution mechanism, which is machine dependent and is generated by `acbingen`. The retargeting process of the link editor is done through the *linker emulation* module. Each target system must have a distinct linker emulation module to indicate how certain target-specific issues must be resolved by the core. The emulation module includes a default link editor script and several hook functions which might be called by the core code in certain stages of the linking process.

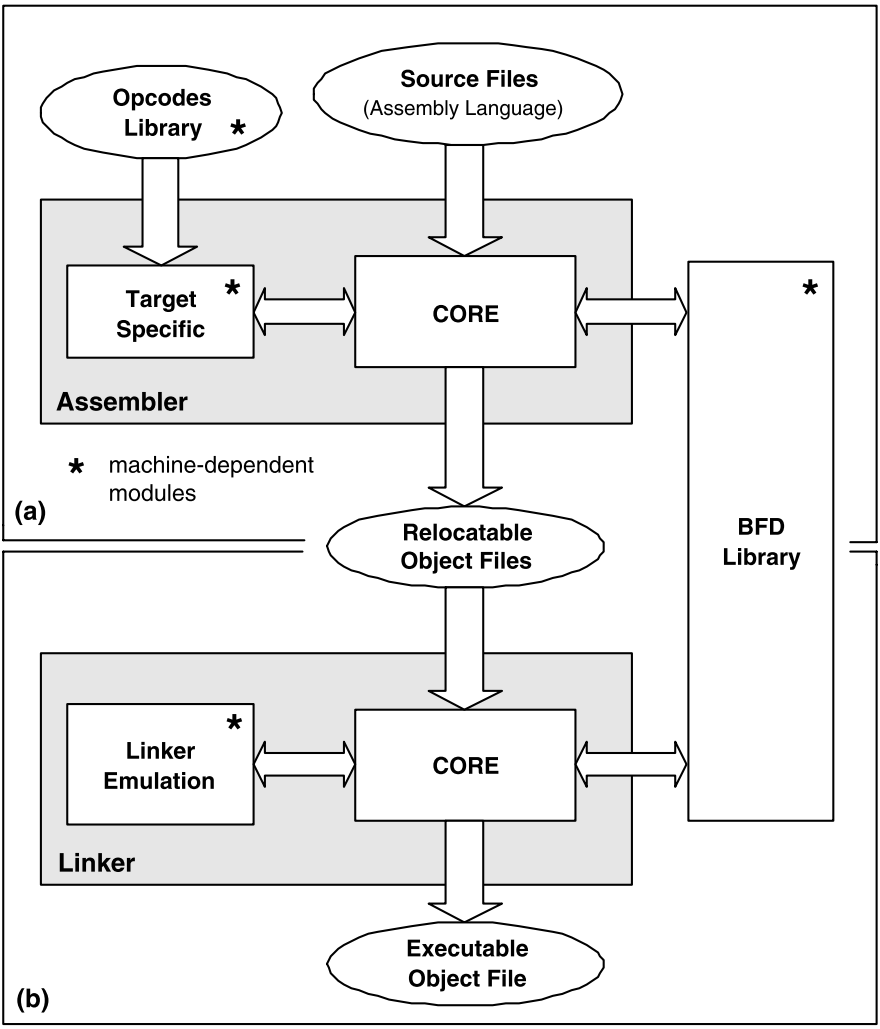


Fig. 6.9 Assembler and link editor structure

A link editor script is written using the linker command language. It describes how sections in the input files are mapped into the output file and the underlying memory layout. Figure 6.10 shows a very small example of a link editor script specifying the memory layout of the output file. The `.text` sections of the input files are combined and relocated relative to the base address 0x10000 in the output file. The same happens for `.data` (base address 0x200000) and `.bss` sections (base address 0x500000) of the input files.

The generation of the link editor emulation module is driven by a shell script executed during the binutils building process. Three target configuration files are

```
SECTIONS
{
  . = 0x10000;
  .text : { *(.text) }
  . = 0x200000;
  .data : { *(.data) }
  . = 0x500000;
  .bss : { *(.bss) }
}
```

Fig. 6.10 A simple linker script example

read by the script which in turn generates the default linker scripts and the hook functions. `acbingen` automatically generates a configuration file containing the start symbol for the linker and the start address of each object file section (`.text`, `.data` and `.bss`).

The memory layout used by the automatically generated link editor follows the memory requirements of ArchC simulators. This choice guarantees that executable object files generated by the link editor can be loaded and ran on ArchC simulators during design space exploration. Note, however, that this default memory layout does not constrain in any way the use of the generated link editor in other environments, since a suitable memory layout can be provided to the link editor at linking time.

6.3.7 Summary of Generated Files

Figure 6.11 shows the files automatically generated by `acbingen`, where `[arch]` represents the architecture name.

In addition to the files presented in Fig. 6.11, some auxiliary files (such as `Makefile.in` and `configure`) are automatically patched by our generation tool in order to correctly build the libraries and tools.

6.4 Putting it to Work

Before starting, make sure you have the binutils and gdb source tarballs (versions 2.16 and 6.4, respectively, are recommended). Uncompress the packages into directories of your choice, such as `BINUTILSDIR` for binutils and `GDBDIR` for gdb. After that, proceed to install the ArchC package as explained in [2] and pass two additional options to the `configure` script:

```
--with-binutils=$BINUTILSDIR --with-gdb=$GDBDIR
```

```
- binutils
 - bfd
  . cpu-[arch].c
  . elf32-[arch].c
 - opcodes
  . [arch]-opc.c
 - gas
  - config
   . tc-[arch].h
   . tc-[arch].c
 - ld
  - emulparams
   . [arch]elf.sh
 - include
  - elf
   . [arch].h
  - opcode
   . [arch].h
```

Fig. 6.11 File tree generated by the `acbingen` tool

Download the processor model of your choice from the ArchC website [2] and uncompress it on your hard disk. Let's assume the PowerPC model is being used and has been uncompressed into directory `PPCMODEL`. To generate the machine-dependent binary tools source code for the PowerPC you will use the `acbingen.sh` script (installed with the ArchC installation). Get inside the `PPCMODEL` directory and issue the following command (assuming the path to the script has been correctly set):

```
acbingen.sh -amyppc -i bin powerpc.ac
```

This process may take from several minutes to hours depending on your host machine. The binary utilities will be created and placed into the directory `bin` under the `PPCMODEL` directory.

The tools generated by ArchC are standard binutils and gdb tools. This means that the machine independent command line options supported by conventional tools are still supported by the generated tools. The generated assembler also extends the command line options with the following:

`-i`, `-insensitive-syms`
: the assembler considers symbolic names as being case insensitive;

`-s`, `-sensitive-mno`
: the assembler considers mnemonic strings as being case sensitive.

These options change the default behavior of conventional binutils assembler. There is a third command line option called `-archc` which displays the ArchC version used to generate the tool and the architecture name.

References

1. R.H. Pesch, J.M. Osier, *The GNU Binary Utilities* (Free Software Foundation, Boston, 1993)
2. The ArchC Website (March 2008). http://www.archc.org
3. TIS Committee, *Executable and Linking Format Specification* (1995)

Part III
Advanced Topics

Chapter 7
Debugging SystemC Platform Models

Bruno Albertini, Sandro Rigo, and Guido Araujo

7.1 Introduction

As it was already stated in the previous chapters of this book, new design methodologies and tools are being developed to cope with the ever increasing system complexity of modern designs, culminating with the so-called Electronic System Level (ESL). In this environment, the exploration of several SoC architectural models is key to achieve application tuning and improved performance. This demands a platform simulation infrastructure capable of doing fast simulation of both software and hardware, compelling designers to work at higher levels of abstraction.

Besides that, another key factor for a good ESL methodology is the early integration of verification into the design flow. As stated in Ghenassia's book on TLM techniques [5], one of the most appealing aspects of TLM is the possibility of reusing the whole platform infrastructure for both hardware and software development and verification, which means that the same high-level platform becomes the reference model on a functional verification flow for hardware IPs, but also is applied by the software team as their executable system specification.

The lower the abstraction level, the harder it gets to get good debug and analysis capabilities on it. It is well known that when a system goes to silicon the only points that can be observed are the external interface pins of the design. This should not be the case on ESL technologies. Higher levels of abstraction must not be appealing just for their fast simulation speeds, but also for their flexibility and capabilities when

B. Albertini (✉) · S. Rigo · G. Araujo
Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil
e-mail: balbertini@ic.unicamp.br

S. Rigo
e-mail: sandro@ic.unicamp.br

G. Araujo
e-mail: guido@ic.unicamp.br

S. Rigo et al. (eds.), *Electronic System Level Design*,
DOI 10.1007/978-1-4020-9940-3_7, © Springer Science+Business Media B.V. 2011

debug and analysis come into play. This creates a demand for new functionalities related to verification in TLM platform models and/or simulation environments. The main goals are: to increase verification coverage; increase the designer's ability to interact with the platform, for example, by guiding the simulation flow through certain corner situations by means of specific stimuli injection; and to make simulation environments more friendly during platform debugging, by enabling breakpoints on the simulation and data introspection. Bailey et al. [2] divide debug and analysis capabilities in three primary groups. Observability is simply the ability to see what is going on into the system. Controllability is to provide means of placing values into certain registers in order to increase control over the system. Correctability is to assist the designer in correcting the discovered bugs, which is far easier to provide in ESL than it is on silicon, but still needs to be addressed by ESL tools.

In this chapter, we discuss a mechanism that relies on computational reflection to enable designers to, on the fly, interact with the platform simulation. This allows them to monitor and change signals or even IP internal register values, so as to explore, in high level of details, what is really happening within each platform module. This mechanism was first introduced in [1].

7.2 Model Debugging and Verification in SystemC

As discussed in Chap. 2, SystemC is basically composed of a simulation kernel and a library with common hardware structures and data types, like signals, ports, bit vectors, etc. Those are standard C++ libraries, with object oriented constructions commonly found in any structured code. Based on that, SystemC users write their own models, which are normally organized as a hierarchy of several SystemC modules.

A problem arises exactly when one needs to debug a SystemC module with standard C++ debug tools, like GDB, while developing his/her own hardware model. Hardware designers are not concerned about SystemC internals, but with the model description they are working on. Unfortunately, ordinary C++ debugging tools do not understand the isolation between the hardware description and SystemC internal code, turning debugging into a painful task. That is why specialized tools for SystemC code debugging become mandatory in order to speed up the development cycle for SystemC designers. It gets worse in platform-based design, since there may be several complex modules to put together, demanding to debug not only their functionality but also the communication capabilities among them.

ARM has a tool suite called RealView Development that they claim to do be able—despite other things—to do real time SystemC debug, including platform exploration, low level debug and remote debug (for ARM devices). Vista Design is a tool from a toolset called Vista Architect, from Mentor Graphics. This is the most complete debugger tool aiming SystemC designs, allowing projectist to trace transactions, sequence of events and process execution. Platform Architect from CoWare is on the same track, with a focus on platform exploration and data monitors. XModelink SystemC debugger is another nice tool to trace thread executions in real-time, but is more limited than their competitors.

The main problem of those commercial tools is that often they use proprietary SystemC solutions that were especially developed for debugging purposes. Most of the time, those kernels comply with the standards, but specific designs and portability between IPs can be degraded, not to mention that it prevents the integration with legacy designs.

SystemCXML and LusSy are nice open-source tools for visualizing SystemC hierarchies, but no real-time debugging support is provided. For real-time debugging, we could only find GDB and GDB-based tools. GDB is hard to use for SystemC debugging, mainly because it does not distinguish between SystemC internal code and user code. If the designer is not careful, he/she may end up inside code from internal SystemC classes.

The technique presented below fulfill this gap without any SystemC or user intervention. Although it is possible to attach graphical debuggers to the interface, we do not aim at graphical visualization but at data introspection.

7.3 Why Computational Reflection?

Computational reflection is the ability of a system to observe and to modify itself. In the software domain, reflection is used to achieve performance optimization, by means of self-modification performed by the program during runtime, or to dynamically adapt a system for specific situations. Reflection can be classified as static or dynamic, depending on the mechanism used to generate the additional data structure needed for achieving the introspection [4]. Static reflection may be used by compilers to do memory optimizations. Reflection is most common on its dynamic form and in high-level managed programming languages like Java, C#, or Smaltalk.

So, why are we interested in computational reflection in a TLM hardware platform design environment?

One of the features we wanted in our methodology is that the IP designers should not need to be aware that the IP will be inspected, meaning that designers will develop their SystemC IPs in exactly the same way they already do. Moreover, it should also be possible to inspect, through the reflection mechanism, IPs whose source code are not even available. These requisites make static reflection or user-annotated strategies (where the reflection mechanism or the source code must be rewritten for every single IP) unsuitable for our purposes. One can claim that aspect-oriented programming is an interesting alternative to enable such data introspection [3]. In fact, it is well suited for this task. But its main drawback is that it requires a modification on the programming paradigm in which designers develop their IPs. Each SystemC module where data introspection would be applied should be developed following the aspect-oriented programming paradigm. This goes totally against the feature we wanted to provide. Moreover, hardware designers may not be willing to spend a reasonable amount of time to learn and to change between different programming paradigms, seriously restricting the adoption of this technique. Finally, all existent IPs would have to be rewritten in order to make the data introspection technique applicable.

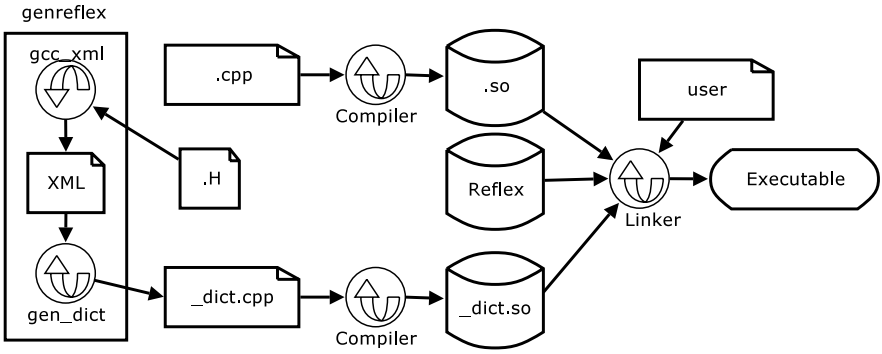


Fig. 7.1 Reflection flow using the Reflex-Seal library

7.4 Enabling Data Introspection in SystemC

Let us see how to provide data introspection on SystemC modules by means of an external reflection library, in our case, the Reflex-SEAL [6] library, designed by CERN as a part of the ROOT project [7]. The main aspects that make this library suitable to this task are: it is non-intrusive, provides a semi-automatic information gathering, imposes no external dependencies (except for the library itself), and there is no need to modify or replace the compiler.

Let us take a look at the whole software infrastructure used to build this reflection-enabled framework:

- GCC 4.4.x
- SystemC 2.2.0 (TLM 2.0)
- Reflex SVN r32289

Figure 7.1 shows the generation flow for a reflection-enabled SystemC module using the Reflex-Seal library. Basically it comprises three steps: information gathering, compilation, and linking.

The information gathering is performed through a script called `genreflex`, and comprises two phases. First, GCCXML is used to parse the source code header file and to generate an object equivalent structure in XML. Second, the XML file is parsed, generating a compilable dictionary that contains information about the object structure, like offset and type of the attributes and methods. All information gathering is done before the compilation and should be done once for every module to be reflected. Notice that the flow depicted in Fig. 7.1 makes clear that no source code file (.cpp) is demanded. Moreover, no source code annotation is required.

The dictionary is important to programs which use the reflected data. Through methods of the Reflex-Seal library, all the internal structure representation of the reflected module can be inspected. Usually, a pointer to the reflected object is enough for performing any reflection operation. Considering the platform design scenario, we are specially interested in attribute inspection for reading and writing, but calling methods and non virtual functions are also possible.

After information gathering, compiling and linking are very straightforward. In Fig. 7.1, the program which will perform the introspection on the SystemC module is included into the `user` input with all the remaining of the platform. This program must be linked, as long as the module itself, to the reflection library, the dictionary, and the SystemC library. The produced executable is a SystemC simulator of the platform capable of performing data introspection to aid during the debugging process.

SPIRIT [8] is a consortium created with the main objective of establishing a standard way to exchange IP information among major EDA companies. Their meta-descriptions allow any tool able to understand the notation to automatically import, export, and even compile any hardware module described either in a low-level language or in SystemC. One of the standard XML description notations defines ReflexBoxes as an introspection tool with inspection, watching, and debugging capabilities. Although ReflexBoxes are covered by the SPIRIT standard specification, the SPIRIT consortium does not specify the way in which WhiteBox modules should be implemented, but only their functionality. This SystemC WhiteBox implementation described using reflection is called *ReflexBox*.

SystemC hardware modules descriptions are C++ classes. Register contents, input, output, and any other data that may be significant for data introspection are stored in object attributes or variables. This is exactly the kind of information that reflection can give us. The ReflexBox implementation consists of a SystemC module that has zero delay and communicates to other modules by means of a simple protocol. It acts like any SystemC module, but it cannot be accessed by another module because it has no interface. When the ReflexBox process is scheduled by the SystemC kernel, it uses the dictionary for walking through an instance of the reflected object (another SystemC module) and gather information.

The ReflexBox is generic and can inspect any kind of SystemC module. The connection between the ReflexBox and the inspected module is created by the ReflexBox's constructor. It receives a pointer to the object during ReflexBox's instantiation. Each ReflexBox can handle just one IP instance, but the platform can have as many ReflexBox instances as required.

At the first execution of their main thread, RefleBoxes generate an internal list of the attributes and allow the user to set conditions, specifying the attributes of interest and what to do with them. Moreover, it is also possible to set breakpoints, used to stop the simulation when one of the conditions is satisfied or just log any changes suffered by any attribute. When a breakpoint condition is satisfied, the user is capable of observing the value, of changing it, or of just continuing the simulation. Remember that the ReflexBox has zero delay, so stopping the simulation does not advance SystemC's simulation time. ReflexBox scheduling must be performed in such a way that it inspects the reflected module on every simulation cycle, when the module could change its state. A carefully designed sensitivity list for the ReflexBox should make it sensible to the same events that can trigger the reflected module.

In Sect. 7.5 we present a detailed platform case study, illustrating all the steps necessary to include the ReflexBox module in order to enable the reflection-based debugging mechanism.

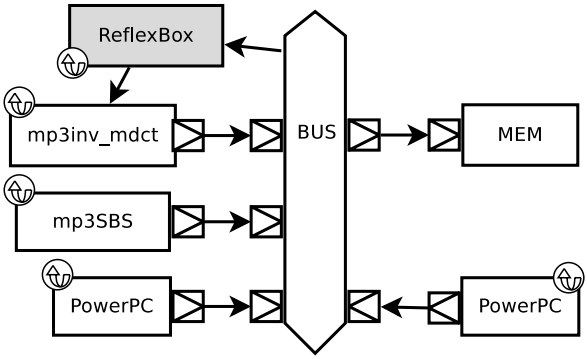


Fig. 7.2 MP3 platform

7.5 Debugging a Platform Simulation Model

In this section we use the same MP3 decoder platform that was constructed in Sect. 5.4, which is composed by two PPC processors, two specialized IPs, and memory. This platform runs a modified version of the MP3 decoder (dist10) and is depicted in Fig. 7.2. Here, our main goal is to understand how to achieve module introspection by applying the reflection mechanism to debugging purposes on this platform.

As it is detailed in Sect. 5.4, the two functions selected to be implemented in hardware IPs were SBS (Side band synthesizer) and IMDCT (Inverse cosine transform), which together account for 64% of the processing time in the decoding process. For demonstration purposes, we are going to reflect the IMDCT IP. In order to do that, there are three steps to be taken:

1. Generate a dictionary file for the IP;
2. Instantiate a ReflexBox, which is the module that executes data introspection;
3. Compile the platform to produce an executable model.

The three-step process described above assumes that the software infrastructure listed in the beginning of this chapter (GCC, SystemC, and the Reflex library) is already installed. The first step is easily accomplished by means of the `genreflex` application, which is included in the Reflex package, by running the following command:

```
genreflex ac_tlm_imdct.h -I <path_to_includes>
```

It is important to notice that this process does not require the source code, just the module's header file `ac_tlm_imdct.h`. The expected result is a file called `ac_tlm_imdct_rflx.cpp`, which represents the dictionary for this IP. This process needs to be performed once per reflected module and can be done by the IP provider itself. In Fig. 7.3, we provide an excerpt of one of the classes inside the `ac_tlm_imdct_rflx.cpp` file, where the members of the `ac_tlm_imdct` class, like the bus port `target_export`, `answer_ready`, and `memory` are identified (Lines 3–8).

```
...
//------Delayed data member builder for class ac_tlm_imdct
void __user__ac_tlm_imdct_db_datamem(Reflex::Class* cl) {
  ::Reflex::ClassBuilder(cl)
  .AddDataMember(type_3877, Reflex::Literal("target_export"),
       OffsetOf(__shadow__::__user__ac_tlm_imdct,
      target_export), ::Reflex::PUBLIC)
  .AddDataMember(type_14947, Reflex::Literal("memory"),
      OffsetOf(__shadow__::__user__ac_tlm_imdct, memory), ::
      Reflex::PRIVATE)
  .AddDataMember(type_2510, Reflex::Literal("answer_ready"),
      OffsetOf(__shadow__::__user__ac_tlm_imdct, answer_ready)
      , ::Reflex::PRIVATE);
}
...
namespace {
  struct Dictionaries {
    Dictionaries() {
      Reflex::Instance initialize_reflex;
      __user__ac_tlm_imdct_dict();
    }
    ~Dictionaries() {
      type_5923.Unload(); // class user::ac_tlm_imdct
    }
  };
  static Dictionaries instance;
}
```

Fig. 7.3 A dictionary file example

The second step is done in the main file of the platform description (`main.cpp`), instantiating the ReflexBox responsible for the runtime debugging and the target IP, like illustrated in Fig. 7.4.

Line 10 is the IP instantiation. For the purpose of this chapter, it is enough to know that this IP is a SystemC module. At Line 14, we declare a ReflexBox that is tied to this IP by its name (`ac_tlm_imdct`) and the pointer to the object instantiated in Line 10. The last paramenter is a number representing the port which the ReflexBox shall connect to accomplish the introspection.

By the time the designer compile and run the platform, the SystemC kernel will instantiate and run each thread, including the ReflexBox. Once all threads are initialized, the ReflexBox will stop the kernel and wait for the connection on the desired port (in this example, 6000). The designer shall connect to this port either by using a telnet application or a compatible debugging interface. In this example we use a direct telnet connection. In Fig. 7.5, Lines 1–5 show the connection to the ReflexBox being stabilished.

This means that the platform is running and it is successfully connected to the ReflexBox. At this point, simulation is on hold, waiting for instructions to continue. The platform simulation terminal will show some messages from the ReflexBox. In order to explore the platform, let's try some of the ReflexBox's features.

```
1
2  int sc_main(int ac, char *av[])
3  {
4    sc_clock mclock;
5
6    powerpc ppc_proc1("ppc1"), ppc_proc2("ppc2");
7    ac_tlm_mem mem("mem", 8 * 1024 * 1024);
8    ac_tlm_router router("router");
9    ac_tlm_startup startup("startup");
10   ac_tlm_sbs sbs("sbs");
11   ac_tlm_imdct imdct("imdct");
12
13  //ReflexBox instantiation
14   whitebox<ac_tlm_sbs> reflexbox("reflexbox","ac_tlm_sbs",&sbs,6000);
15
16   char *av1[] = {"dual_ppc.x", "--load=dual_mp3.x", "-A", "source.mp3", "dest.aiff"
        , ""};
17   int ac1 = 5;
18   char *av2[] = {"dual_ppc.x", "--load=dual_mp3.x", "-A", "source.mp3", "dest.aiff"
        , ""};
19   int ac2 = 5;
20
21 #ifdef AC_DEBUG
22   ac_trace("ppc_proc1.trace");
23 #endif
24
25   wbmp3sbs.clock(mclock);
26   ppc_proc1.MEM_port(router.target_export1);
27   ppc_proc2.MEM_port(router.target_export2);
28   router.MEM_port(mem.target_export);
29   router.STARTUP_port(startup.target_export);
30   router.SBS_port(sbs.target_export);
31   router.IMDCT_port(imdct.target_export);
32
33   ppc_proc1.init(ac1, av1);
34   ppc_proc2.init(ac2, av2);
35
36   ppc_proc1.set_instr_batch_size(1);
37   ppc_proc2.set_instr_batch_size(1);
38
39   cerr << endl;
40
41   sc_start(-1);
42
43   ppc_proc1.PrintStat();
44   ppc_proc2.PrintStat();
45   cerr << endl;
46
47 #ifdef AC_STATS
48   ppc_proc1.ac_sim_stats.time = sc_simulation_time();
49   ppc_proc1.ac_sim_stats.print();
50   ppc_proc2.ac_sim_stats.time = sc_simulation_time();
51   ppc_proc2.ac_sim_stats.print();
52 #endif
53
54 #ifdef AC_DEBUG
55   ac_close_trace();
56 #endif
57
58   return ppc_proc1.ac_exit_status + ppc_proc2.ac_exit_status;
59 }
```

Fig. 7.4 The main.cpp file for the MP3 platform

```
1  telnet localhost 6000}
2
3  Trying 127.0.0.1...
4  Connected to localhost.
5  Escape character is '^]'.
6
7  G
8
9  user::ac_tlm_imdct::answer_ready;
10 user::ac_tlm_imdct::memory;
```

Fig. 7.5 Interacting with to ReflexBox

```
1  Iuser::ac_tlm_imdct::answer_ready;;OnChange;;TRUE;
```

Fig. 7.6 Interacting with the ReflexBox

Let's type the `G` command, like showed in Fig. 7.5 (Line 7), resulting in a semicolon-separated list containing all SystemC elements that the ReflexBox could reflect (and you can play with), like in Line 9.

The ReflexBox can also reflect methods, giving to the user the possibility to call them. This is not useful into architecture exploration, so we will not show here. The last lines in Fig. 7.5 show all reflectable attributes of the class `ac_tlm_sbs`. The useful ones are the attributes representing IP registers, like `answer_ready`. This register should indicate that the IP is idle or processing (values false and true). An interesting intropection should be a watch over this register, to stop the simulation when the IP is called (it starts with 0, or idle). In order to do that, type the command displayed in Fig. 7.6 on the telnet window.

The ReflexBox is supposed to answer "Ok". If the answer is anything else (including nothing), please check for misspelling. Now we must say to the ReflexBox that we want to continue simulation. If you type the command below, the ReflexBox will finish the command input phase and start the simulation phase. Typing this at the beginning (after telnet connection) will execute the platform as no ReflexBox was connected:

```
1  Handshake_Finished
```

The output screen should be similar to that depicted in Fig. 7.7. The first field is a simulation timestamp, showed as picoseconds from the beginning of the simulation. At the first line, the control register changed its value to one. This indicates us that the processor took this amount of time to prepare the data, send it to the `ac_tlm_imdct` IP, and set the `answer_ready` register to 1. Consequently, the IP starts processing. We have the `answer_ready` register changing again at `9821365500ps`. Since the `answer_ready` changed its value from one to zero, we conclude that the IP is signaling to the processor that computation is over and data is ready. The difference between these timestamps represents the amount of

```
1  I9819471500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;
       FALSE;
2  I9821365500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;
       FALSE;
3  I9821447500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;
       FALSE;
4  I9823341500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;
       FALSE;
5  I9823423500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;
       FALSE;
6  I9825317500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;
       FALSE;
7  I9825399500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;
       FALSE;
8  I9827293500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;
       FALSE;
9  I9827375500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;
       FALSE;
10 I9829269500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;
       FALSE;
11 I9829351500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;
       FALSE;
12 I9831245500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;
       FALSE;
13 I9831327500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;
       FALSE;
14 I9833221500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;
       FALSE;
15 I9833303500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;
       FALSE;
16 I9835197500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;
       FALSE;
17 I9835279500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;
       FALSE;
18 I9837173500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;
       FALSE;
19 I9837255500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;
       FALSE;
20 I9839149500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;
       FALSE;
21 I9839231500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;
       FALSE;
22 I9841125500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;
       FALSE;
23 I9841207500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;
       FALSE;
24 I9843101500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;
       FALSE;
25 I9843183500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;
       FALSE;
26 ...
27 Connection closed by foreign host.
```

Fig. 7.7 The reflected attributes accessible through the ReflexBox module

```
1 Iuser::ac_tlm_imdct::answer_ready;;ConditionEqual;1;TRUE;
2 Handshake_Finished
```

Fig. 7.8 Interacting with the ReflexBox

```
1 Iuser::ac_tlm_imdct::answer_ready;0;OnChange;;TRUE;
```

Fig. 7.9 Interacting with the ReflexBox

```
1 Iuser::ac_tlm_imdct::answer_ready;;OnChange;;TRUE;
2 Iuser::ac_tlm_imdct::memory;;OnChange;;TRUE;
```

Fig. 7.10 Interacting with the ReflexBox

time taken by the IP to process this chunk of data. Using this watch, a designer can take a lot of information, like: how many times the processor invoked the IP for this input, if the time spent by both the IP and the processor (between two consecutive IP executions) is linear or data dependent etc.

Watches are really useful, but we need some more control over the simulation. Start simulation and connect telnet again. Type the commands depicted in Fig. 7.8 to the ReflexBox.

We are telling the ReflexBox that we want to stop simulation when `answer_ready` register is equal to '1' and then we finish the preparation phase. The execution will start and the simulation will hold at the very same point it did in the last execution, Line 1 in Fig. 7.7.

Simulation is on hold, so we can interact with the ReflexBox again. Suppose we want to modify the register contents to 0 (telling the processor that the IP is not processing) and continue the simulation until the end, but as a watch (just printing values as in the previous example). We can do that by typing the command showed in Fig. 7.9.

The output is pretty much like in the previous example, but the output file is wrong. The processor just understood that the IP finished processing and recorded another chunk of data into the buffer while the IP was still processing the old data, leading to a complete wrong output. The main point in this example is to illustrate that the user is capable of modifying the data during the simulation. This can be a very useful feature to force simulation into corner cases during platform verification.

Now, let's try to identify an error. Suppose the platform is running fine, but the output file is not passing the checksum. We can repeat the simulation again, but this time passing the two commands in Fig. 7.10 on the preparation phase.

These commands define a watch over each one of the registers, `answer_ready` and `memory` (it is a pointer to where the chunk of data to be processed resides). The output is showed in Fig. 7.11 (the remaining lines are the same as in the first example):

As showed, the `memory` changed on `9819471500ps` with the `answer_ready register`, because the processor had just programmed the IP to the first

```
I9819471500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;FALSE
    ;
I9819471500 ps;user::ac_tlm_imdct::memory;0x2003EF;FALSE;
    FALSE;
I9821365500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;FALSE
    ;
I9821447500 ps;user::ac_tlm_imdct::answer_ready;1;FALSE;FALSE
    ;
I9823341500 ps;user::ac_tlm_imdct::answer_ready;0;FALSE;FALSE
    ;
```

Fig. 7.11 Introspection output

execution. The problem is that it never changed anymore, meaning that on the second IP execution (at `9821447500ps` when the control register changed to one again), the IP processed the same chunk of data, which is not the expected behavior. This platform is implemented using a double buffering scheme, so this address should be different between two different IP executions.

Obviously the ReflexBox cannot identify errors by itself, but it proved to be a very handful tool to aid bug discovering. In this last example, using ReflexBox to manually change the `memory` value to the correct one is enough to get a correct simulation, saving a lot of time in searching the bug and correcting the program.

References

1. B.C. Albertini, S. Rigo, G. Araujo, C. Araujo, E. Barros, W. Azevedo, A computational reflection mechanism to support platform debugging in systemC, in *Proceedings of the International Conference on Hardware–Software Codesign and System Synthesis (CODES+ISSS)* (2007), pp. 81–86
2. B. Bailey, G. Martin, A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology* (Morgan Kaufmann, San Mateo, 2007)
3. D. Déharbe, S. Medeiros, Aspect-oriented design in SystemC: implementation and applications, in *SBCCI '06: Proceedings of the 19th Annual Symposium on Integrated Circuits and Systems Design* (ACM Press, New York, 2006), pp. 119–124
4. J. Ferber, Computational reflection in class based object-oriented languages, in *OOPSLA '89: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* (ACM Press, New York, 1989), pp. 317–326
5. F. Ghenassia, *Transaction-level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems* (Springer, New York, 2006)
6. S. Roiser, P. Mato, The SEAL C++ Reflection System, in *CHEP '04: Computing in High Energy and Nuclear Physics Congress*, Interlaken, Switzerland, September 2004 (CERN, Geneva, 2004)
7. Root Project (2009). http://root.cern.ch
8. SPIRIT Consortium (2007). http://www.spiritconsortium.org

Chapter 8
SystemC-Based Power Evaluation with PowerSC

Felipe Klein, Rodolfo Azevedo, Luiz Santos, and Guido Araujo

Although SystemC is considered the most promising language for SoC functional modeling, it does not come with built-in power modeling capabilities. This chapter presents PowerSC, a power estimation framework which instruments SystemC for power characterization, modeling and estimation. Since it is entirely based on SystemC, PowerSC allows consistent power modeling from ESL to lower abstraction levels. Section 8.1 shows how SystemC is extended to handle power, Sect. 8.2 describes how to instrument SystemC descriptions for power estimation, Sect. 8.3 illustrates how PowerSC allows the user to perform estimations either at RTL or gate-level with the same instrumentation. Finally, Sect. 8.4 explains how to put PowerSC to work.

8.1 SystemC Extensions for Power Modeling

In this section, we first show an extended design flow, which is fully based on SystemC, and then describe its supporting framework.

F. Klein (✉) · R. Azevedo · G. Araujo
Computer Systems Laboratory, Institute of Computing, University of Campinas, Campinas, Brazil
e-mail: klein@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

S. Rigo et al. (eds.), *Electronic System Level Design*,
DOI 10.1007/978-1-4020-9940-3_8, © Springer Science+Business Media B.V. 2011

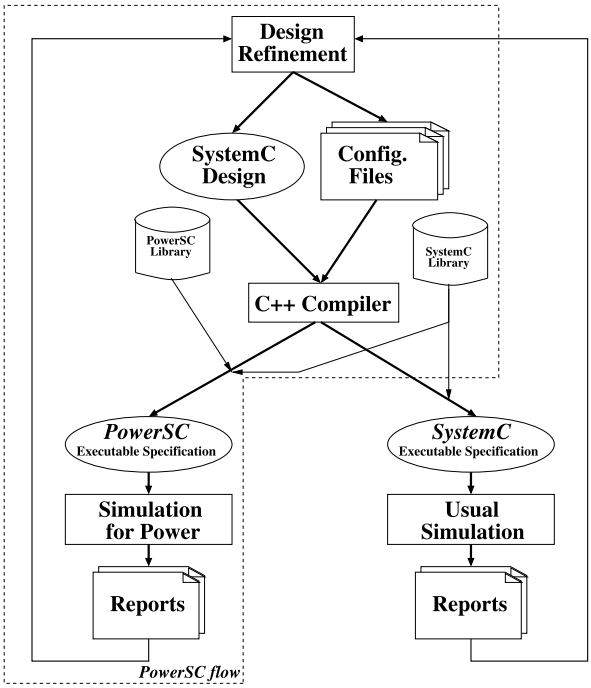


Fig. 8.1 PowerSC Design Flow

8.1.1 The Extended Design Flow

Figure 8.1 shows two complementary and orthogonal flows. At the right side, the usual purely-functional SystemC flow is sketched. At the left side, the power-aware PowerSC flow is illustrated.

No matter the adopted flow, the starting point is a SystemC description. The flow to be followed is determined by some configuration files (e.g. Makefiles), which instruct the C++ compiler to generate either a conventional executable specification (by linking the SystemC library only) or to produce an augmented executable specification (by linking both, the SystemC and PowerSC libraries).

The first step in the power-aware flow is to compile the PowerSC executable specification, which is instrumented to gather signal statistics during simulation. In the next step, simulation is launched and, as consequence of proper instrumentation, design elements are monitored and power information is dynamically recorded. At simulation completion, the resulting information is summarized in power reports. Reported bottlenecks can be employed to prompt design refinements. Refined designs should undergo as many iterations through the PowerSC flow as required to reach adequate power values.

Notice that the user can switch from one flow to another as many times as needed to optimize power and satisfy design constraints.

```
1        ...
2     template <int W, class T>
3     class psc_objinfo : public psc_objinfo_base,
4                         public psc_objinfo_if
5     {
6        ...
7     };
8        ...
9     template <int W>
10    class psc_bv : public sc_bv<W>,
11                   public psc_objinfo<W, sc_bv<W> >
12    {
13       ...
14    }
15       ...
```

Fig. 8.2 Extending SystemC classes

8.1.2 The SystemC Extensions

In order to enable the power analysis capabilities in SystemC, some extra classes were created in order to capture useful information for power calculations (e.g. switching activity). The main class responsible for such task is named `psc_objinfo`, as sketched in Fig. 8.2 (Line 3). Using multiple inheritance, this class was derived from `psc_objinfo_base` and `psc_objinfo_if` classes. The former implements the base mechanisms for capturing the switching activity and the latter actually provides the interface with methods for handling the statistics gathered throughout the simulation.

Moreover, C++ meta-programming has been used to make this class easily adaptable, according to two parameters: *W* (the bit-width of the object being monitored) and *T* (the base class which is being extended to be power-aware).

Assisted with this, SystemC signal and data type classes have been extended so as to inherit properties and behavior from both the original SystemC class and the `psc_objinfo` class. An example of such extension is given for the SystemC bit-vector class in Fig. 8.2 (Line 10). Notice that the prefix used to label PowerSC classes is "`psc_`", following a convention similar to the one employed in SystemC ("`sc_`").

Every PowerSC object is capable of automatically tracking its state throughout simulation, thereby capturing the switching activity of the associated design element. Each object can provide information such as its toggle count (number of $1 \rightarrow 0$ and $0 \rightarrow 1$ transitions), spatial correlation (relation between different signals) and static probability (probability of a signal being at a certain logic value), which are essential statistics for obtaining accurate power estimates [4].

To capture the toggle count during simulation, the assignment operator was overloaded for all relevant classes. Figure 8.3 shows an example of the underlying mechanism when monitoring the switching activity on a signal of type `psc_uint<2>` (2-bit unsigned integer).

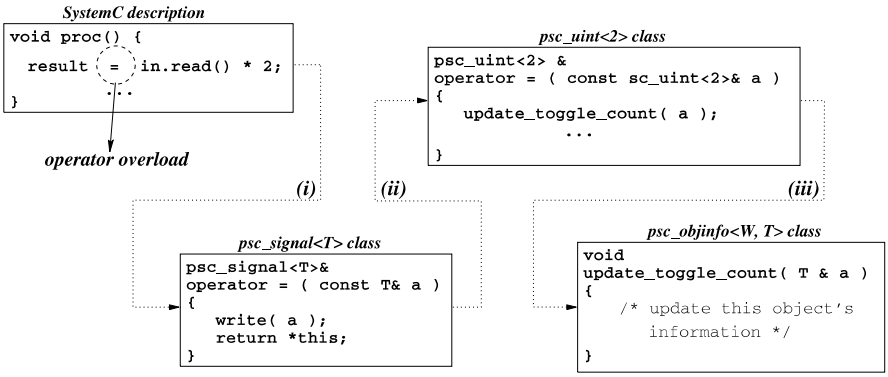


Fig. 8.3 Overloading operators for activity monitoring

```
1   #include <systemc.h>
2   #include <powersc.h> // <-- mandatory modification --
3   #include "muls32.h"
4
5   ...
6   int sc_main(int argc, char **argv) {
7     ... // modules instantiation
8     sc_start( /* simulation time */ );
9     PSC_REPORT_POWER; // <-- this is also necessary --
10    return( 0 );
11  }
```

Fig. 8.4 A PowerSC-enabled SystemC model

Consider the signal named *result* within *proc*(). Whenever a value is assigned to this signal, this operation will be handled by the power-aware SystemC class (i), which in turn calls the appropriate overloaded operator of the PowerSC data type class (ii). Then, this method takes care of computing the switching activity statistics accordingly (iii). It is important to remark that PowerSC is aware of the fact that the assignment to a signal does not change its value immediately, thus avoiding an unwanted loss of accuracy.

8.2 Instrumentation of a SystemC Description

Only two modifications are needed in the SystemC description to enable PowerSC usage. A mandatory change is the inclusion of the main PowerSC header file (`powersc.h`) within the model files. Another compulsory change is to invoke a PowerSC macro at the end of the simulator's main function (`sc_main`), in order to print out the results. These modifications are shown in Fig. 8.4, at Lines 2 and 9, respectively.

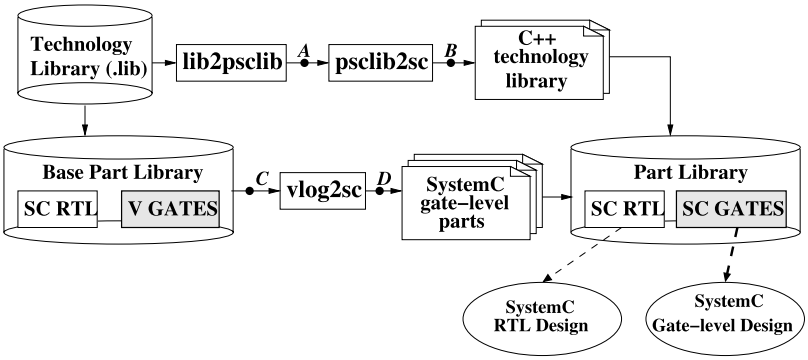


Fig. 8.5 SystemC gate-level generation

As it can be seen, the effort to enable the power estimation mechanism is minimal, leaving a small footprint in the original SystemC description. It should be noted that SystemC data types, signals and modules do not need to be manually changed. PowerSC modifies them automatically.

8.3 Support for Characterization at the Gate Level

To allow the more accurate power figures obtained at the gate level to be employed for the building of higher-level models, PowerSC provides a mechanism to generate a gate-level part library for a given technology library. The required generation steps are depicted in Fig. 8.5.

The base part library (containing components like adders and multipliers) is built from a technology library written in the so-called Liberty format [6] with a tool from Forte's Cynthesizer package [2]. Components are described in SystemC at the RT level (*SC_RTL*) and in Verilog at the gate-level (*V_GATES*).

To enable a unified representation within the PowerSC framework, two converters were implemented from the external formats into SystemC, as follows:

- the **vlog2sc** converter translates each netlist found in *V_GATES* to an equivalent SystemC gate-level description;
- the **lib2psclib** and **psclib2sc** converters translate the technology library from the Liberty format into a C++ representation.

The files produced by the converters result in the so-called SystemC gate-level part library (*SC_GATES*), which replaces the Verilog library.

Let us illustrate the conversion by showing samples of input and output formats. First, consider the conversion implied from point *A* to *B* in Fig. 8.5. A sample description of a 0.13 μm technology library is given in Fig. 8.6. This sample is described in the so-called PSCLib format, which is a simplified format as compared to Liberty. Note that it first defines the units to be adopted and then describes an existing library cell (xor2v0x1).

That description, converted in the form of a C++ technology library, results in two files: techlibrary.h and techlibrary.cpp. A sample of the latter is shown in Fig. 8.7. (These files are later used by the components in the SystemC gate-level part library *SC_GATES*).

Let us now consider the conversion implied from point *C* to *D* in Fig. 8.5. A sample of the Verilog description of a 4-bit adder at the gate-level is given in Fig. 8.8.

That description is converted to the SystemC model sketched in Fig. 8.9. Notice (at the first line) that the above-mentioned technology library is included. Observe that such SystemC model simply instantiates and binds cells from the technology library (e.g. the *I22* instance of a *XOR* gate), much in the same way as an ordinary SystemC design.

Thanks to the described converters, the PowerSC user can select the abstraction-level at which a design component should be simulated (either RTL or gate-level), by simply selecting the appropriate part library (either *SC_RTL* or *SC_GATES*). Regardless of the chosen abstract level, the PowerSC aspects visible to the user are exactly the same. The instrumentation of SystemC descriptions, as depicted in Fig. 8.4, is therefore abstraction-level independent.

8.3.1 Integration of Macromodels in PowerSC

One of the keystones of the PowerSC framework is the support for modeling power at different levels of abstraction. Power models can be tied to SystemC modules encapsulating functional, RTL or even gate-level models. To support the integration of pre-defined power models, PowerSC provides a macromodeling API, which consists of a set of C++ classes, as described in the following.

In order to add a new part to the library, two classes must be derived: `psc_macromodel` and `psc_macromodel_parms`. The latter encapsulates the parameters of a given macromodel as attributes (e.g. signal statistics). The former encapsulates the actual power computation, as sketched in Fig. 8.10, and essentially consists of two functions:

- `init_power_map`: it initializes the internal structure of the macromodel (defined by the user) with the power information from the characterization phase.
- `get_power`: it computes the dissipated power according to the parameters captured in the derived `psc_macromodel_parms` class. (It is internally invoked to generate the power reports from a specific power model.)

It should be noted that the inner details of the derived `psc_macromodel` class are hidden from the user. For integration purposes, the user must only create the code for those virtual functions.

A distinct `psc_macromodel` class is automatically created for each library component, since macromodeling techniques usually require a specific behavior for each component (tables initialized with different values, different power equations, and so on).

```
units {
   voltage = 1V;
   capacitive_load = 1.00 ff;
   time = 1ps;
   leakage_power = 1pW;
};
   ...
xor2v0x1 {
   leakage_power = 1074;

   input a {
      capacitance = 5.34;
      fanout_load = 1;
   }

   input b {
      capacitance = 6.7;
      fanout_load = 1;
   }

   output z {
      max_fanout = 4;
      max_capacitance = 119;
      function = (a^b);

      internal_power a_z_n:a {
         template = pwr_x1_195_5x10;
         transition_type = both;

         values = {
            {11.41, 11.475, 11.535, 11.56, 11.555},
            {11.245, 11.33, 11.415, 11.47, 11.475},
            {11.285, 11.36, 11.455, 11.52, 11.535},
            {11.425, 11.49, 11.585, 11.65, 11.68},
            {11.84, 11.86, 11.925, 11.98, 12},
            {12.61, 12.58, 12.56, 12.56, 12.56},
            {14.01, 13.875, 13.715, 13.6, 13.53},
            {16.34, 16.075, 15.69, 15.35, 15.135},
            {20.195, 19.75, 19.035, 18.305, 17.805},
            {26.445, 25.78, 24.615, 23.285, 22.26}
         }
      }
   }
   ...
};
```

Fig. 8.6 Sample of the psclib2sc input format

```
const double LibUnits::voltage = 1;
const double LibUnits::capacitive = 1e-15;
const double LibUnits::time = 1e-12;
const double LibUnits::leakage_power = 1e-12;
const double LibUnits::dynamic_power = 0.001;
   ...
void xor2v0x1::proc()
{
   z.write(
      ( a.read() ^ b.read() )
   );
}

xor2v0x1::~xor2v0x1()
{
#ifdef POWER_SIM

   sc_interface *_if_a = a.get_interface();
   psc_objinfo_if *_p_a = dynamic_cast<psc_objinfo_if*>(_if_a);

   sc_interface *_if_b = b.get_interface();
   psc_objinfo_if *_p_b = dynamic_cast<psc_objinfo_if*>(_if_b);

   sc_interface *_if_z = z.get_interface();
   psc_objinfo_if *_p_z = dynamic_cast<psc_objinfo_if*>(_if_z);
   ...
   double E_b_z_n =
     psc_geometric_centroid(pts_b_z_n, _p_b->get_net_delay(),
                            _p_z->get_net_load());

   psc_cell_power_info info(name(), celltype);

   info.set_level(PSC_GATE_LEVEL);
   info.set_leak_power(leakage_power);
   psc_pin_power_info pwr_a_z_n(_p_z->get_id(), E_a_z_n);
   pwr_a_z_n.set_rel_pins(1, _p_a->get_id().c_str());
   info.add(pwr_a_z_n);
   psc_pin_power_info pwr_b_z_n(_p_z->get_id(), E_b_z_n);
   pwr_b_z_n.set_rel_pins(1, _p_b->get_id().c_str());
   info.add(pwr_b_z_n);
   PSC_INSERT_CELL(info);
#endif
}
```

Fig. 8.7 Sample of the psclib2sc output format

Besides, such distinct `psc_macromodel` classes can also be useful when the designer wants to explore different macromodel techniques for each component (either to improve estimation accuracy or speed).

Actually, several macromodels have already been successfully integrated into the PowerSC framework, as described in [3]. Most macromodels rely on the following

```
module Add4 (a,b,out1);
   input [3:0] a, b;
   output [3:0] out1;
   ha2v0x2 I0 (.co(N53), .so(out1[0]), .a(b[0]), .b(a[0]));
   xor2v0x1 I22 (.z(N74), .a(a[1]), .b(b[1]));
   mxi2v0x05 I23 (.z(N70), .a0(a[1]), .a1(N53), .s(N74));
   iv1v0x1 I24 (.z(N57), .a(N70));
   xor2v0x05 I25 (.z(out1[1]), .a(N74), .b(N53));
   xor2v0x05 I26 (.z(N85), .a(a[2]), .b(b[2]));
   mxi2v0x05 I27 (.z(N81), .a0(a[2]), .a1(N57), .s(N85));
   iv1v0x05 I28 (.z(N59), .a(N81));
   xor2v0x05 I29 (.z(out1[2]), .a(N85), .b(N57));
   xor3v1x05 I3 (.z(out1[3]), .a(a[3]), .b(b[3]), .c(N59));
endmodule
```

Fig. 8.8 Sample of the vlog2sc input format

signal properties: the input signal probability P_{in}, the input transition density D_{in} and the output transition density D_{out}.

Let us take the so-called *EqTab* power macromodeling technique [1] as an example to illustrate the integration process. That macromodel considers the individual contribution of each input/output bit position. Let $D_{in}(x)$ and $D_{out}(x)$ be transition densities (bit toggles per time unit) measured at the x-th bit position for a stream of input and output vectors, respectively. For simplicity, Let us generically call them *bit-wise transition densities*. Let n and m be, respectively, input and output vector bit widths. Given a component, its power consumption is modeled by the following equation, where c_i denotes a coefficient:

$$Power = c_0 + c_1 * D_{in}(0) + c_2 * D_{in}(1) + \cdots + c_{n+m-1} * D_{out}(m-2) + c_{n+m} * D_{out}(m-1).$$

Actually, the *EqTab* technique relies on a look-up table which is indexed with (P_{in}, D_{in}). For each entry in this table, instead of directly storing a power value, the corresponding entry actually stores the coefficients of the equation above.

As a result, *EqTab* estimation consists of three steps: first, an RTL simulation is run and the bit-wise densities are collected, along with the properties P_{in}, D_{in} and D_{out}; then, the coefficients stored at entry (P_{in}, D_{in}) are returned (if (P_{in}, D_{in}) does not represent a valid entry, the closest valid point is used instead); finally, the returned coefficients and the collected bit-wise densities are combined according to the above equation.

EqTab characterization employs a single input vector stream for each pair (P_{in}, D_{in}) and consists in determining the respective set of coefficients. To find a proper set of coefficients, a system of equations is built as follows.

Let SW be a matrix with as many rows as the number of successive pairs of vectors in the stream (say, S pairs) and with as many columns as the compound vector bit-width $(n+m)$. A row of matrix SW stores the bit-wise transition densities taken between a pair of successive vectors. A column stores the transition density

```
#include "techlibrary.h"
   ...
SC_MODULE(Add4)
{
   sc_in<bool> a[4];
   sc_in<bool> b[4];
   sc_out<bool> out1[4];

   xor2v0x1 *I22;
   ...

   psc_signal_bool N74;
   ...

   SC_CTOR(Add4)
   {
      I22 = new xor2v0x1( "I22" );
      I22->z( N74 );
      I22->a( a[1] );
      I22->b( b[1] );
      ...
      PSC_OBJ_ALIAS(N74, "N74");
   }
};

SC_MODULE(Add4_wrapper)
{
   sc_in<sc_uint<4> > a;
   sc_in<sc_uint<4> > b;
   sc_out<sc_uint<4> > out1;

   Add4 *add4_0;
   ...
}

#define  Add4  Add4_wrapper
```

Fig. 8.9 Sample of the vlog2sc output format

of a given bit position along the input stream. Let P be a $S \times 1$ matrix, where each entry p_i stores the power consumed by the i-th pair of input vectors.

Characterization consists in first calculating the bit-wise transition densities for every successive pair of input vectors (storing them in a row of matrix SW) and measuring their resulting power consumption (storing it in an entry of matrix P).

```
class psc_macromodel {
    ...
public:
    virtual void init_power_map();
    virtual double get_power(const psc_macromodel_parms&);
    ...
};
```

Fig. 8.10 PowerSC macromodeling support

```
1  SC_MODULE(MulS16) {
2      sc_in<sc_int<16> > in1;
3      sc_in<sc_int<16> > in2;
4      sc_out<sc_int<32> > out1;
5      ...
6      void some_process();
7      ...
8      PSC_MACROMODEL_MulS16;
9      ...
10     SC_CTOR(MulS16);
11 };
```

Fig. 8.11 Part of a possible implementation for a multiplier in SystemC

Then, the set of fitting coefficients C is obtained by solving the system of equations $SW * C = P$ with standard regression techniques (e.g., least mean squares). Finally, such coefficients are stored at entry (P_{in}, D_{in}).

As a practical example for the technique described above, let us consider a component such as a 16-bit input multiplier. The code for this component is outlined in Fig. 8.11.

This sample code shows the main components of the multiplier, such as inputs/outputs and a process to perform the computation, given the values provided to inputs *in1* and *in2*. Also, it shows how the component could be instrumented regarding the usage of the macromodel created using the *EqTab* power macromodeling technique (Line 8).

This macro could be defined elsewhere with the appropriate commands so as to convert the component into a power-aware multiplier. Figure 8.12 sketches how this could be accomplished.

As mentioned previously in this section, the two classes `psc_macromodel` and `psc_macromodel_parms` must be derived according to the specific details of each technique. In this case, for example, the class `psc_macromodel_MulS16_EqTab` declares a table (`model_coeffs`), which keeps the values of the coefficients computed during characterization of the model. Also, notice that the power model is instantiated conditionally, depending on the value of the compiler directive `POWER_SIM`: if set, the multiplier will instantiate it; otherwise, the component behaves as the ordinary multiplier component.

```
...
#ifdef POWER_SIM
#define PSC_MACROMODEL_MulS16\
   static const char *celltype;\
   static psc_macromodel_MulS16_EqTab power_model;\
   ~MulS16()
#else
#define PSC_MACROMODEL_MulS16
#endif
...
#define TOTAL_POINTS  59
...
class psc_macromodel_parms_MulS16_EqTab :
                                public psc_macromodel_parms
                                    {
public:
   psc_macromodel_parms_MulS16_EqTab();
   ...
};

class psc_macromodel_MulS16_EqTab : public psc_macromodel
    {
private:
   static const double model_coeffs[TOTAL_POINTS+1][64+1];

public:
   void init_power_map();
   double get_power(const psc_macromodel_parms & p);
   ...
};
```

Fig. 8.12 Macromodel implementation for a multiplier

8.4 Putting PowerSC to Work

PowerSC reduces power evaluation infrastructure to a minimum. It only requires the standard SystemC library, the PowerSC library itself and a C++ compiler. In this section, we first describe the basic steps to obtain the required packages and to install them. Then we show how to use PowerSC by means of a few practical examples.

8.4.1 Downloading

PowerSC is composed of two different package types: (1) the framework itself and (2) related tools. Both were built using the GNU autotools, so as to make the installation process easier. If you are familiar with the procedure used with “autotools-like” packages, you will probably not have difficulties using it.

Besides, the Liberty parser provided by Synopsys was modified to convert technology libraries from the Liberty format to our simplified PSCLib format.

Package files are named using the following convention: NAME–VERSION.tar.gz. The required files are powersc-X.X.X.tar.gz, psclibtools-X.X.X.tar.gz and lib2psclib-X.X.X.tar.gz, where X.X.X denotes the version of each package.

At the time of writing, those files are not distributed on the web. To obtain them, please contact the authors via e-mail (addresses are available in the first page of this chapter).

8.4.2 Checking Requirements

Before installation, please, make sure that the following libraries and tools are installed in your system:

- SystemC and GCC: their release versions should be at least *2.1* and *3.0*, respectively. (Although other version combinations are likely to work, they may require adaptations in the installation files).
- Synopsis Liberty parser: it is required by the translator from the Liberty format to the PSCLib format. Synopsys provides this tool for free through its so-called TAP-in program. More information on how to join this program and obtain this parser is available at http://www.synopsys.com/partners/tapin.
- ANTLR parser generator: the parsers in the psclibtools package were created using ANTLR [5], which is capable of automatically generating parsers from grammatical descriptions to a variety of target languages, such as C, C++, Java, C#, Python and others. In our case, we target parsers in C++, for both the Verilog and PSCLib grammars.

8.4.3 Building the Packages

Once you made sure that the required tools and libraries are installed, you can build the packages as described in the following.

Building PowerSC

1. Choose an appropriate destination for the installation and unpack the PowerSC code, as follows:

   ```
   you@somewhere:~$ tar zxf powersc-X.X.X.tar.gz
   ```

   (As a result, a new directory named `powersc-X.X.X` will be created, containing all PowerSC files, including source code, examples and the scripts for installation.)

2. Prepare the installation by executing the configuration script, as follows:

```
you@somewhere:~$ cd powersc-X.X.X/
you@somewhere:~$ ./configure --with-systemc=DIR
```

(Notice that it is mandatory to specify where SystemC is installed using the option `--with-systemc`. This command will create the necessary makefiles to build the library. Among the many optional parameters, you may want to use the flag `--with-examples`, which causes useful examples to be built together with the library).

3. Build the library, by invoking the make command:

```
you@somewhere:~$ make all
```

(After the compilation process is finished, you will have the file `libpowersc.a` under the directory `powersc-X.X.X/src`. If you chose to build the examples, they will be ready for use under the directory `powersc-X.X.X/examples`.)

Building PSCLibTools

1. Unpack the release file of the tools, as follows:

```
you@somewhere:~$ tar zxf psclibtools-X.X.X.tar.gz
```

(As a consequence, the directory `psclibtools-X.X.X` will be created under the current directory.)

2. Prepare the installation by executing the configuration script, as follows:

```
you@somewhere:~$ cd psclibtools-X.X.X/
you@somewhere:~$ ./configure
```

3. Compile the package and/or install it, as follows:

```
you@somewhere:~$ make all install
```

(As a result, the tools `vlog2sc` and `psclib2sc` will be ready for use under the directory `psclibtools-X.X.X/src`; if you chose to install them in your system, they should have been copied by default to the system's application binaries directory).

Building Lib2PSCLib

1. Unpack the release file of the tool, as follows:

```
you@somewhere:~$ tar zxf lib2psclib-X.X.X.tar.gz
```

2. Prepare the installation by executing the configuration script, as follows:

```
you@somewhere:~$ cd lib2psclib-X.X.X/
you@somewhere:~$ ./configure [--enable-debug]
```

3. Compile the package and/or install it, as follows:

```
you@somewhere:~$ make all install
```

8.4.4 Running PowerSC

In order to illustrate the usage of PowerSC, we pick two designs from the examples provided with the library:

1. a 32-bit adder (gate-level);
2. a 16-bit input signed multiplier (RTL).

If the preceding steps of installing the required packages and building the PowerSC library were executed correctly, you should have these designs' executable specifications ready to run.

For the first example, the following commands should be entered in your shell prompt:

```
you@somewhere:~$ cd powersc-X.X.X/examples/Add32-gatelevel
you@somewhere:~$ ./run_x -i <input vector>
```

There are several files within each design directory including the design itself and some other files related to the testbench which instantiates it.

After entering this directory, simply type in the name of the executable (named `run_x`) and provide, as input, a file with the stimuli to be applied to the adder's inputs. Many examples are already available in the directory `input`, and can be used as a template for creating your own stimuli files.

Part of the resulting power report generated upon simulation completion is outlined in Fig. 8.13. Each line in the report represents an object (or cell) monitored during simulation. Columns 1–3 show the abstraction-level at which the object is described, the name of the cell within the design's hierarchy, and the type of the cell, respectively.

For instance, the first line of the report contains an object named **I106**, described at the gate level (**G**), which is an instance of type **xor2_1** from the technology library. The last two columns in this report show, respectively, the leakage and internal power, in Watts. At the end of the report, a summary of the power consumption of all objects is presented.

The steps to run the second design are quite similar to the example presented above and the commands to be typed in are as follows:

```
you@somewhere:~$ cd powersc-X.X.X/examples/MulS16-rtl-macromodel
you@somewhere:~$ ./run_x -i <input vector>
```

Figure 8.14 shows a sample power report for this design. Notice, that the first column contains the value **RT**, indicating that the object is described at the register-transfer level. In addition, instead of having the columns *Leakage Power* and *Internal Power*, there is a single column labeled **Aggregate Power**. This points out that the power value comes from a power macromodel, and not from values stored in a technology library.

```
  L  -      Cell  Name      -   Cell Type   -  Leakage Power  -  Internal Power
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 G   - tb0.dut.add32_0.I106 -  xor2_1        - 7.881970e-11    -  5.573337e-06
 G   - tb0.dut.add32_0.I107 -  and2_2        - 4.385880e-11    -  7.515706e-06
 G   - tb0.dut.add32_0.I1   -  fulladder     - 1.132440e-10    -  1.335455e-05
 G   - tb0.dut.add32_0.I171 -  xor2_1        - 7.881970e-11    -  5.662287e-06
              ...
 G   - tb0.dut.add32_0.I172 -  mux2_2        - 6.996270e-11    -  3.816533e-06
 G   - tb0.dut.add32_0.I173 -  xor2_1        - 7.881970e-11    -  4.930259e-06
 G   - tb0.dut.add32_0.I175 -  mux2_2        - 6.996270e-11    -  3.020991e-06
 G   - tb0.dut.add32_0.I176 -  xor2_1        - 7.881970e-11    -  2.743796e-06
 G   - tb0.dut.add32_0.I177 -  xor2_1        - 7.881970e-11    -  2.767456e-06
 G   - tb0.dut.add32_0.I178 -  mux2_1        - 6.680430e-11    -  2.285534e-06
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Summary:
  Switching power :    1.889796e-04W
  Internal power  :    3.113640e-04W
  Leakage power   :    6.419074e-09W
  Aggregate power :    0.000000e+00W
                     ----------------
  TOTALS          :    5.003500e-04W
```

Fig. 8.13 Power report sample for a 32-bit adder (gate-level)

```
  L  -      Cell  Name      -   Cell Type   -  Aggregate Power
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 RT  -  tb0.dut             -  MulS16       -  1.209492e-03
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Summary:
  Switching power :    0.000000e+00W
  Internal power  :    0.000000e+00W
  Leakage power   :    0.000000e+00W
  Aggregate power :    1.209492e-03W
                     ----------------
  TOTALS          :    1.209492e-03W
```

Fig. 8.14 Power report sample for a 16-bit input signed multiplier (RTL)

References

1. M. Anton, I. Colonescu, E. Macii, M. Poncino, Fast characterization of rtl power macromodels, in *ICECS'01: International Conference on Electronics, Circuits and Systems*, 2001, pp. 1591–1594
2. Forte Design Systems, Cynthesizer User's Guide, v3.3 edition, May 2007. http://www.forteds.com
3. F. Klein, R. Leao, G. Araujo, L. Santos, R. Azevedo, On the limitations of power macromodeling techniques, in *ISVLSI '07: IEEE Computer Society Annual Symposium on VLSI* (IEEE Computer Society, Los Alamitos, 2007), pp. 395–400
4. X. Liu, M.C. Papaefthymiou, A Markov chain sequence generator for power macromodeling. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **23**(7), 1048–1062 (2004)
5. T. Parr et al., Antlr—another tool for language recognition. http://www.antlr.org
6. Synopsys, Inc., Library Compiler User's Guide, v-2003-12 edition, 2003

Index

A
ABI, 41, 52, 53, 61, 63–67, 99
Abstraction, v, vi, 3, 4, 16, 18, 23, 25, 26, 29, 39, 40, 117, 129, 134, 143
acbingen, 108–110, 112, 113
acsim, 46–48, 52, 53, 58, 62, 63, 69
acStone, 41, 54, 57–62, 67
ADL, vi, 7, 8, 16–18, 23, 108–110
Approximately-timed, 28
ArchC, vi, 8, 11, 16–24, 31–33, 39–49, 51–55, 57, 59, 60, 62–64, 66–69, 72–80, 85, 100–104, 109, 112, 113
Architectural, v, vi, 7, 13, 40, 41, 48, 117
Architecture Description Language, 7, 16
Assembler, 8, 16, 18, 21, 23, 99, 100, 102–104, 106–108, 110, 111, 113
Assembly, 17, 21, 58–60, 67, 81, 100–103, 109, 110

B
BFD, 107–110
Binary, 4, 7, 13, 17, 21, 45, 52, 54, 60, 63, 99–103, 107, 108, 110, 113
Binutils, 18, 101, 107, 108, 111–113
Bit-vector, 118, 131

C
Cache, 16, 40, 46, 53, 69
Compiler, 4, 7, 16, 17, 46, 53, 57, 58, 60, 63, 69, 72, 100, 119, 120, 130, 139, 140

D
Debug, v, 28, 30–32, 34–36, 53, 54, 62, 98, 117, 118
Decoder, 19, 21, 44, 46–48, 53, 62, 68, 69, 84, 86, 95, 122
Decoding, 15, 17, 21, 45–47, 68, 84, 87, 88, 90, 102, 104, 107, 122
Decoupling, 30, 31
Design flow, 3, 5, 6, 8, 23, 25, 26, 117, 129, 130
DMI, 30, 32–36

E
Electronic System Level, v, 3, 4, 16, 117
ELF, 53, 60, 78, 85, 99, 109
Encoding, 19, 21, 102–105, 110
end behavior, 51
ESL, v, vi, 3–6, 8, 9, 11, 16, 17, 23, 26, 27, 117, 118, 129
Evaluation, 4, 5, 26, 101, 129, 140

F
Flow, 3, 5, 6, 8, 16, 17, 23, 25, 26, 81, 82, 87, 100, 101, 107, 108, 117, 118, 120, 129, 130

G
GCC, 63, 120, 122, 141
GDB, 54–62, 118, 119
GDB interface, 54, 55, 57, 58, 63, 67
genreflex, 120, 122

H
Helper, 22, 49, 51, 52, 63, 64
HEX, 53
Hierarchical, 12, 21, 22, 27, 49
Hierarchy, 22, 118, 143

I
Initiator, 29, 30, 32, 72
Instruction behavior, 13, 15, 19, 21, 22, 41, 48, 49
Instruction format, 13, 17, 19, 21, 22, 44, 48, 104, 105, 109

S. Rigo et al. (eds.), *Electronic System Level Design*,
DOI 10.1007/978-1-4020-9940-3, © Springer Science+Business Media B.V. 2011

Instruction set, 13, 17, 19, 23, 40, 42, 43, 48
Interface, vi, 9, 26–36, 42, 54, 55, 63, 72–74, 76, 87, 88, 90, 99, 107, 117, 119, 121, 123
Introspection, 9, 118–123, 128

L
Link editor, 107, 108, 110–112
Linker, 8, 16, 18, 63, 66, 67, 99, 100, 103, 104, 110–112
Loader, 53
Loosely, 28, 39
Loosely-timed, 28, 39

M
Macromodel, 9, 134, 136, 137, 139, 140, 143
Modifier, 103–105, 109, 110
Module, 8, 9, 12–14, 23, 26, 27, 29, 30, 34, 35, 73, 75, 76, 79, 82, 90, 91, 93–97, 107, 110, 111, 118–123, 132–134
MP3, 72, 84, 86–89, 95, 98, 122, 124

O
Opcode, 102, 107–110
Operating system, 39, 52, 53, 63, 66, 99
Overloading, 132

P
Payload, 27–30, 32, 33, 35, 36, 73
Performance, v, 4, 5, 7–9, 16, 26–28, 46, 68, 69, 71, 117, 119
Platform-based design, 4, 5, 36, 99, 100, 118
Port, 12, 19, 23, 27, 31–33, 35, 54, 60, 72–74, 77, 79, 80, 82, 107, 109, 118, 122, 123
Power, v, vi, 4, 5, 7–9, 23, 26, 98, 100, 129–131, 133, 134, 137–140, 143, 144
PowerPC, 12, 13, 18–22, 31, 32, 40–44, 47–52, 55, 59, 60, 62, 64, 65, 67–69, 75–78, 80–83, 85, 86, 101, 107, 113
PowerSC, vi, 8, 129–134, 136, 139–141, 143
PPC, 12, 13, 15, 16, 31–33, 36, 122
Process, v, 12–14, 16, 17, 22, 23, 26–28, 30, 31, 40, 78, 98, 100, 103, 108, 110, 111, 113, 121, 122, 139, 140
Profiling, 84, 86, 87, 97
Program loader, 53
Pseudo, 21, 106

R
Reflection, 118–122
Reflex, 120, 122
ReflexBox, 121–128
Register-Transfer Level, 4, 16
Reserved words, 45
Retargetable, 4, 99
RTL, 4, 5, 7, 16, 26, 40, 129, 134, 137, 143, 144

S
Simulator, 5, 16, 18, 23, 24, 31, 32, 40–42, 45–48, 51–65, 67–69, 72, 73, 80, 81, 87, 100, 101, 112, 121, 132
Switching, vi, 76, 80, 131, 132, 144
Synchronization, 29, 31, 97, 98
Syntax, 15, 16, 21, 40, 42, 44, 49, 102, 106, 109
SystemC, v, vi, 7–9, 11–13, 15, 16, 18, 23–28, 43, 49, 60, 76–78, 85, 118–124, 129, 131, 133, 141, 142

T
Target, 4, 5, 7, 8, 16, 18, 26, 28, 29, 32, 53, 54, 56, 60, 100, 108, 110, 111, 123, 141
Template file, 48, 49, 63
Thread, 12–14, 27, 28, 53, 95, 118, 121, 123
Timed, 28, 39, 40
TLM, vi, 4, 7–9, 18, 19, 23–33, 35, 36, 40, 72–74, 76, 78, 79, 117–120
Tools, v, vi, 4, 7, 8, 16, 23, 40, 45, 71, 75, 99–101, 107, 112, 113, 117–119, 140–142
Transaction Level Modeling, 23, 25
Transport, 28–30, 32–35, 72–74, 80, 82, 83
Type behavior, 21, 49

U
Untimed, 28, 39

V
Verification, v, 4, 7–9, 26–28, 117, 118, 127

W
WhiteBox, 121