An Object-Oriented Methodology for Modeling the Precise Behavior of Processor Architectures

João Cláudio Otero and Flávio Rech Wagner

Instituto de Informática - Universidade Federal do Rio Grande do Sul - Porto Alegre, Brazil

E-mail: {jcotero,flavio}@inf.ufrgs.br

Abstract: This paper presents SimPL, an object-oriented methodology for modeling processor behavior with precise timing, which may be used as a basis for teaching and design environments. Starting from a specialized class library, SimPL supports a very easy and flexible definition or modification of processor models of different architectural classes.

Key words: object-oriented hardware modeling, processor architectures, teaching environments, design environments

1. INTRODUCTION

Processors are becoming very complex. Pipelines, memory hierarchies, and superscalar features are implemented in very different ways. A wide range of architectural classes is available (RISC processors, DSPs, VLIW processors, microcontrollers), each of them with their particular solutions. This situation imposes severe requirements to computational environments intended for analyzing or exploring the processor design space.

Teaching environments are intended to understand concepts and to experiment alternative architectural solutions. In *design environments*, in turn, processor design space exploration is associated to performance evaluation, as in the design of embedded systems, where the most adequate processor must be found or designed for a given set of design requirements.

These two different goals have been addressed by distinct classes of environments. Teaching environments [1-5] are usually restricted to a given

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: [10.1007/978-0-387-35597-9_40](https://doi.org/10.1007/978-0-387-35597-9_40)

M. Robert et al. (eds.), *SOC Design Methodologies*

© IFIP International Federation for Information Processing 2002

processor architecture, where a particular configuration may be defined by setting parameter options before the simulation starts. A rich user interface, specialized for the range of processor configurations, is usually supported.

Processor design environments [7,8], in turn, usually offer some specialized language, which gives great flexibility for the definition of the processor architecture. These environments are also aimed at a retargetable generation of supporting tools, such as simulators, compilers, and debuggers.

This paper presents SimPL, an object-oriented methodology for modeling processor architectures that supports features addressing the modeling needs of both teaching and processor design environments. It is based on the SIMOO simulation framework [9], which offers a rich environment for object-oriented modeling of discrete systems and is being applied to the design of electronic embedded systems [10]. SimPL allows a very flexible modeling of various processor architecture classes. The precise timing behavior of the processor instructions may be modeled, if desired, but more abstract models may be also developed. Object-orientation supports an easy reuse, extension, and modification of processor models. The user interface gives specialized support for the modeling methodology.

This paper is organized as follows. Section 2 discusses related work and put SimPL in perspective. The SIMOO framework is briefly introduced in Section 3. The SimPL modeling methodology is presented in Section 4, and its strategy for modeling processor control is discussed in detail in Section 5. Section 6 illustrates the application of the modeling methodology to the well known DLX processor [11]. Section 7 briefly presents the SimPL supporting environment. Finally, Section 8 concludes and discusses future work.

2. COMPARISON WITH RELATED WORK

There are various simulation tools oriented to education in processor architecture, such as PCSpin [1], WinDLX [2], DLXview [3], and ESCAPE [4]. These tools offer a pre-defined set of design alternatives that can be explored by an easy, mostly interactive setting of parameters. Visualization tools allow the user to analyze the impact of these alternatives on instruction execution and processor performance. These tools are dedicated to a specific base processor, which can be modified in a very limited way.

Other simulation tools are more intended for design exploration, offering a richer set of parameters and design alternatives. SimpleScalar [5] is a well known tool of this class. Its parameters allow the design of very different pipelined and superscalar architectures. This feature is enhanced by the fact that the simulator source code is open, so that design alternatives that have not been initially considered can be also implemented. As a drawback, the

great flexibility makes difficult the construction of a specialized built-in user interface for interaction with all possible architectural models.

Tools that allow the specification of processor architectures and organizations are usually based on Architectural Description Languages (ADLs). Many of them automatically generate a retargetable tool suite, containing at least a simulator and a compiler. Some of them, such as ISDL [6], are only intended for the description of the instruction set behavior and cannot generate cycle-accurate simulators for current processors, because they do not support pipeline specifications. Other languages, such as EXPRESSION [7] and LISA [8], describe both behavior and structure.

In EXPRESSION, an RT-level netlist is described, together with the specification of instructions as lists of slots filled with operations corresponding to functional units. The RT-level netlist is composed of primitive components, such as *functional units*, *storage elements*, *ports*, and *buses*. A *pipeline description* specifies components in each pipeline stage, while a *data-transfer path description* specifies valid data transfers in the pipeline. The language has primitives for the detailed description of the memory subsystem. Many language features are intended for the automatic generation of a compiler producing code of high quality.

LISA also supports the description of pipelines. Its definition, however, is mainly intended for the generation of a very fast, cycle-accurate simulator using *compiled code* (as opposed to conventional *interpreted* simulators).

It must be noticed that, although languages such as EXPRESSION and LISA are powerful tools for the development of new processors, for instance in the context of embedded systems design, they may become cumbersome if used in a teaching environment, where the main requirements are easy modification of the processor configuration and good interaction resources.

SimPL is a processor design *methodology* (and not a *language* with some fixed set of constructs), which uses object-orientation as a basic paradigm for a flexible modeling of processor behavior and may serve well both application domains - teaching and processor design environments.

Since SimPL is built on top of a modeling and simulation framework, the processor modeling process automatically results in a simulator with built-in interactive resources for controlling experiments. On the other hand, the methodology does not generate other tools, such as compilers or assemblers, and the simulator code generation is not aimed at high performance.

3. THE SIMOO FRAMEWORK

SIMOO [9] is an object-oriented framework for multi-paradigm, multi-threaded, discrete simulation. Simulation entities in SIMOO are mapped to

autonomous elements, which are objects with their own execution threads. Each object in a given system may be modeled by a different simulation paradigm - e.g. event or process-oriented behavior, communication by ports or messages. The simulation engine manages communication and synchronization between objects, as well as simulation time advancement.

The model structure is graphically specified by hierarchical class and instance diagrams, while the object behavior is described in C++ and uses a specialized simulation library. From the specified diagrams and entities' behavior, an executable model is automatically generated. It already contains default resources for visualization, interaction, and experiment control.

The class diagram defines various types of relationships between objects, such as *knowledge* (connections for exchanging messages), *aggregation*, and *creation* (so that an object may create a new instance of another class in the model). Knowledge relationships may be dynamically created. The aggregation relationship implicitly defines a knowledge relationship between an aggregating object and all its sub-objects.

Visual interaction facilities are implemented by *interface elements* that belong to a domain that is independent from the model logic. The *monitor* is a special entity that receives copies of all messages sent to an autonomous element *AE* whose behavior is being tracked and redirects them to the interface element that is associated to *AE*. The mapping between autonomous and interface elements is managed by the monitor, so that it may be dynamically modified during the experiments.

4. SIMPL MODELING METHODOLOGY

The SimPL modeling methodology explicitly divides a processor into MainControl and Datapath blocks. MainControl aggregates two other types of control elements: InstructionSet and Execution. The datapath is an aggregation of *functional elements*. Furthermore, *information elements* are used for building a bridge between the control and datapath blocks.

The datapath block does not model physical connections between the functional elements, which are not aware of each other. Connections are implicitly defined by means of the processor behavior specified by the control block. This approach allows a more flexible architecture definition, since functional elements may be more easily replaced or introduced.

Each functional element is parameterized and may execute different *micro-operations*, which are defined as methods of the corresponding class. Control elements define the processor behavior by executing these micro-operations. However, they do not directly call methods of the functional

classes. Instead, they communicate only with the aggregating datapath element, which redirects the messages to its aggregated functional elements.

Therefore, the class diagram for the processor does not contain direct connections between a control element and all functional elements it may control. If these connections were necessary, the diagram would be much more complex. The designer, however, must be aware of the operations offered by functional elements. A window in the supporting environment displays all available operations for the definition of a control element.

Micro-operations typically define interactions between functional elements. Suppose the contents of register REG must be transferred to input Inp1 of functional element ALU. A control element would execute the following micro-operation, implemented as one of its local methods:

```
ALU_SET (Inp1, REG_CONTENTS ( ) )
```

As a result, a message is sent to Datapath. ALU_SET (par1,par2) is a micro-operation that requests Datapath to execute method SET of ALU. Its parameter *par2* is another micro-operation REG_CONTENTS, requesting Datapath to execute method CONTENTS for reading the contents of REG. These micro-operations implicitly create a path from the register to the ALU.

Figure 1 shows micro-operation REG_CONTENTS. This method is automatically created by the environment, from the element definition.

```
// Send message ACCESS to Datapath. Its parameters define which functional
// element must be accessed and which of its methods must be called.
```

```
int REG_CONTENTS (void) {
    Parameters par;
    MSGEA msg;

    par.Add (0);           // number of parameters
    par.Add ("CONTENTS");  // name of the method
    par.Add ("#REG");      // name of the functional element

    // create a message containing the above parameters
    msg.Set (Id (), InstSet_DatapathBlock, "ACCESS", NORMAL_PR, par);

    // send a message and wait for response in the current simulation time
    SendReceiveNow(&msg);

    return msg.GetDados ().GetParAsInt (0);    // return the value
}
```

Figure 1. Code for micro-operation REG_CONTENTS

Decode and Interpret are information elements. Decode can be used to check the contents of the Instruction Register to obtain an instruction opcode, for instance. Interpret may be used to interpret an expression written in an assembly language, for instance in order to calculate a memory address. These information elements are linked to control and functional elements by means of the SIMOO Monitor object, so that they can inspect the contents of other elements without having to send them messages.

5. DEFINING PROCESSOR CONTROL

Control elements, from types `MainControl`, `InstructionSet`, and `Execution`, interact with each other to define the overall control behavior.

`InstructionSet` defines each instruction as a set of *instruction steps*. Each step is a set of micro-operations to be executed in parallel. An instruction step is executed each time the instruction is *fired*. The `InstructionSet` element, however, does not enforce a timing relationship between the steps.

The complete set of instructions of a processor may be defined within a single `InstructionSet` element or may be divided into various such elements, for better model management purposes. Instructions may be easily included or removed, because they do not depend on each other.

An `InstructionSet` element may define complete instructions as well as *instruction fragments*, to be hierarchically used within other instructions or instruction fragments. Fragments model sets of instruction steps that are common to various instructions, such as a fetch operation or a memory access using a particular addressing mode.

Figure 2 illustrates the definition of an `InstructionSet` element that implements a fragment for a fetch operation in the DLX processor [11]. This fragment is defined as a single instruction step containing 5 micro-operations: read instruction memory position addressed by the program counter (PC); add 4 to the PC; assign PC+4 to the PC; assign PC+4 to the NewPC field of the IF/ID pipeline register; and assign the contents read from memory to the IR field of the IF/ID pipeline register. These micro-operations are implemented as methods that are local to the `InstructionSet` class, as shown in the methods windows (upper left in Figure 2).

`Execution` is the element that orders the firing of all instructions steps, according to a selected *execution mode*. Depending on the mode, several *firings* may be necessary to complete the execution of all steps. Each firing may be executed, for instance, at a consecutive clock cycle, but other timing schemes are possible. The following execution modes are available:

- a) *single-cycle* mode - all instruction steps are executed in a single firing;
- b) *multi-stepped* mode - each firing executes a certain number of steps; and

c) *multi-cycle* mode - each firing executes a single instruction step.

Figure 2. InstructionSet defining a fetch operation

Suppose an instruction composed by 6 micro-operations grouped into 3 steps. Now suppose that an element Execution fires this instruction using a multi-cycle mode. The code shown in Figure 3 is automatically generated for the method that implements the execution of the instruction.

```
// the incoming message specifies the stage to execute
stage = m.GetDados ().GetParAsInt (0)
switch(stage){
    case 1: {    micro-op_1 (); // micro-operations of stage 1
                micro-op_2 ();
            }break;
    case 2: {    micro-op_3 (); // micro-operations of stage 2
                micro-op_4 ();
            }break;
    case 3: {    micro-op_5 (); // micro-operations of stage 3
                micro-op_6 ();
            }break;
}
```

Figure 3. Method for executing an instruction in a multi-cycle mode

The Execution element offers micro-operations, corresponding to the step firings. They have one of the formats below:

EX (instruction, stage) - for multi-stepped and multi-cycle modes;

EX (instruction) - for the single-cycle mode.

The execution of instructions or fragments is ordered by MainControl, which is the topmost element of the control block hierarchy and defines the overall processor control, as well as a concrete timing behavior. MainControl uses the micro-operations offered by Execution. In a synchronous processor, firings will be synchronized with the clock cycles. Several cycles, however, may be associated to each firing, so that the latency of a functional element may be easily modeled.

6. MODELING THE DLX PROCESSOR

DLX is a 32-bit RISC processor, defined by Patterson and Hennessy [11]. It has four main instruction types: load / store, arithmetic / logical, branches and jumps, and floating point. The processor has a 5-stage pipeline, where stages are identified as Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM), and Write Back to Register or Memory (WB). The first two stages are identical for all instructions. Register values are read from the register file during the second stage.

The topmost level of the DLX model hierarchy contains three elements: MainControl, Datapath, and an information element Decode. MainControl aggregates an Execution element, which in turn instantiates a single InstructionSet element, containing the definition of all processor instructions. InstructionSet is related to Datapath by a *connection* (a *knowledge* relationship, as introduced in Section 3).

The DLX Datapath element aggregates all functional elements and memories. Multiplexers do not need to be included, since connections are not explicitly modeled in SimPL. The basic SimPL library already contains most classes that implement datapath elements of the DLX processor. They need to be only instantiated and parameterized. Datapath elements make a large set of micro-operations available to the control elements, such as:

int PC_CONTENTS ()

DATA_MEM_WRITE (address, value)

int INST_MEM_READ (address)

EX_REGISTER (subregister, operation)

IF_REGISTER (subregister, operation)

The InstructionSet element defines the four DLX instruction types - data transfer, arithmetic and logic, control, floating point - in a hierarchical way, using the modeling features of the SIMOO environment.

For each instruction, the `InstructionSet` element defines three instruction steps. They correspond to the pipeline stages EX, MEM, and WB, which are distinct for each instruction. Each of these stages is defined as a set of micro-operations. The pipeline stages IF and ID, which are identical for all instructions, are modeled by two instruction fragments `InstrFetch` and `InstrDecode` and invoked directly from the `MainControl` element.

The `Execution` element invokes instructions (in fact: invokes micro-operations that are made available by `InstructionSet`) in a multi-cycle mode, where each processor cycle executes a single instruction step. Besides that, the `Execution` element makes micro-operations of the form

`EX (par1, par2)`

available, where *par1* represents an instruction and *par2* an instruction stage. These micro-operations are invoked by `MainControl`.

The overall control behavior and timing is defined within `MainControl`, as shown in Figure 4. Instruction fragments `InstrFetch` and `InstrDecode` are invoked first, to execute micro-operations corresponding to stages IF and ID.

```
instr4 = NOP;    // pipeline is initially empty
instr5 = NOP;
while (true) {
    EX (InstrFetch);
    EX (InstrDecode);
    // execute stage 3 (EX) of instruction decoded in instr3
    EX (instr3 = DECODE(),3);
    // execute stage 4 (MEM) of instruction instr4
    EX (instr4,4);
    // execute stage 5 (WB) of instruction instr5
    EX (instr5,5);
    // instructions advance through the pipeline...
    instr5 = instr4;
    instr4 = instr3;
    // simulation advances to the next clock cycle
    wait(1);
}
```

Figure 4. `MainControl` for the DLX processor

The information element `Decode` monitors all updates to `InstructionRegister`, an element defined within `Datapath`, and returns an instruction *inst* to be executed by micro-operation `EX(instr3=DECODE(),3)`. This information element is necessary because the control block, according to the SimPL approach, does not have a direct association to each functional

element within the datapath block. Decode, using a SIMOO monitor, may obtain the current instruction and pass it to the control block.

The movement of instructions through the pipeline is modeled by auxiliary variables *instr3* thru *instr5* in Figure 4. Variables *instr1* and *instr2* are not necessary because the first two pipeline stages are implemented by instruction fragments InstrFetch and InstrDecode that are identical for all instructions. All 5 pipeline stages are executed in parallel, and instructions move to the next stage also in parallel. The SIMOO function *wait(1)* advances the simulation clock and leads the processor to the next cycle.

Specialized information elements, that monitor the state of the pipeline, inform MainControl about the occurrence of branch and data dependencies. When one of these dependencies is detected, MainControl invokes a method that implements a desired penalty procedure. The invocation of these information elements is not shown in Figure 4.

New instructions may be easily introduced, and existing instructions may be easily modified. This can be done by implementing these changes in the control element InstructionSet. The new micro-operations that are thus created may be used by Execution. If needed, new functional elements may be created within Datapath. The micro-operations that are made available by these new elements may be invoked from InstructionSet.

7. THE SUPPORTING ENVIRONMENT

A dedicated environment, built on top of SIMOO, offers resources that give additional support to the SimPL modeling methodology. SIMOO already offers various modeling resources, as introduced in Section 3. It also builds a default user interface for each simulation model, including read and write access to all model variables and full control over the simulation execution.

For helping the modeling process, this environment offers windows for each control element. These windows display all micro-operations that are available for the definition of the corresponding control element. As an example, the large set of micro-operations defined in the datapath block, which are available for the specification of instructions or instruction fragments, is shown in the window corresponding to InstructionSet, so that the programmer is aware of their existence. The same happens for Execution, so that the micro-operations that are made available by InstructionSet can be browsed by the programmer.

As already mentioned in Section 5, the environment automatically generates the code of the method belonging to class InstructionSet, which executes instruction steps according to the execution mode that has been chosen in the control element Execution.

The SimPL supporting environment also gives the user the possibility of accessing and modifying values of parameters that have been defined for the various functional elements.

When a model is built and element parameters are defined for a given processor configuration, SIMOO classes are automatically generated. These classes may be edited by means of all object-oriented features of the basic SIMOO framework. By polymorphism and inheritance, for instance, elements may be specialized, the behavior of methods may be changed, or entirely new elements may be built. If these new classes follow the SimPL modeling paradigm, all resources and advantages of the SimPL environment become available.

8. CONCLUSIONS AND FUTURE WORK

This paper presented the SimPL methodology for modeling the precise behavior of processor architectures. SimPL is built on top of SIMOO, a framework for object-oriented modeling and simulation of discrete systems. Benefiting from object orientation, the methodology aims at a very flexible modeling approach. This feature is extremely useful for a fast exploration of the processor architecture design space and also for teaching purposes. The methodology is based on the following main concepts:

- datapath and control are explicitly separated, and each of them is modeled as an hierarchy of elements;
- data paths between functional elements are not explicitly modeled; instead, they are implicitly defined by the control behavior;
- a basic library of highly parameterized functional elements is available;
- micro-operations are offered both by datapath and control elements, corresponding to the functions they may execute;
- the control behavior is modeled by three different control elements - InstructionSet defines micro-operations that must be executed for each instruction; Execution organizes micro-operations of the instructions into steps; and MainControl defines the overall timing by invoking instruction steps and associating them to clock cycles; and
- instructions may be hierarchically defined as combinations of instruction fragments.

The separation of the control behavior into three different types of elements, together with the hierarchical definition of instructions, allows an easy modeling of a large variety of micro-architectures and timing behaviors.

From a given processor model, the methodology makes it extremely easy to obtain a new one, with a different timing behavior, or with a different set

of instructions. Full exploitation of object orientation greatly enhances the capabilities for deriving new processor models from old ones.

As a validation strategy, a complete model of the DLX processor has been developed. Other architectural classes, such as VLIW and DSP, will be modeled later on. The SimPL supporting environment is being implemented. Through this environment, classes and methods that are necessary for the methodology will be automatically generated. SimPL is being linked to CSDSim [12], an environment offering a specialized user interaction with a processor model. CSDSim emulates a classroom, following a client-server approach, where the server runs the processor model and clients implement the interaction of the instructor and students with the model.

9. REFERENCES

- [1] J.R.Larus. "SPIM S20: A MIPS R2000 Simulator". Available at ftp://ftp.cs.wisc.edu/pub/spim/spim_documentation.ps
- [2] H. Grünbacher. "WinDLX Tutorial - A First Example". Available by e-mail at maziar@vlsivie.tuwien.ac.at
- [3] G.Adams. "DLXview - (Preliminary) User's Manual". Available at <http://yara.ecn.purdue.edu/~teamaaa/dlxview>
- [4] P.Verplaetse. "ESCAPE v1.1 Manual". Available at <http://www.elis.rug.ac.be/escape>
- [5] D.Burger and T.M.Austin. "The SimpleScalar Tool Set, Version 2.0". Available at <http://www.cs.wisc.edu/~mscalar/simplescalar.html>
- [6] G.Hadjiyiannis et al. "ISDL: An Instruction Set Description Language for Retargetability". In: 34th Design Automation Conference, 1997.
- [7] A.Halambi et al. "EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability". In: DATE'99 - Design, Automation and Test in Europe. Munich, Germany, March 1999.
- [8] S.Pees, A.Hoffmann, V.Zivojnovic, and H.Meyr: "LISA Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures". In: 36th Design Automation Conference. New Orleans, USA, June 1999.
- [9] B.Copstein, F.R.Wagner, and C.E.Pereira. "SIMOO - An Environment for the Object-Oriented Discrete Simulation". In: ESS'97 - 9th European Simulation Symposium. Passau, Germany, October 1997.
- [10] F.R.Wagner, M.Oyamada, L.Carro, and M.Kreutz. "Object-Oriented Modeling and Co-simulation of Embedded Electronic Systems". In: L.M.Silveira, S.Devadas, and R.Reis (eds.), VLSI: Systems on a Chip. Kluwer Academic Publishers, 2000.
- [11] D.A.Patterson and J.L.Hennessy. "Computer Architecture: A Quantitative Approach". Morgan Kaufmann Publishers, Inc, 1996.
- [12] G.Rodrigues, D.M.Becker, and F.R.Wagner. "CSDSim: A Didactic Processor Simulation Environment Based on the Client / Server Architecture". In: 12th Symp. on Computer Architecture and High Performance Computing. São Pedro, Brazil, October 2000.