EXPERIENCES IN MODELING AND SIMULATION OF COMPUTER ARCHITECTURES IN DEVS

Gabriel Wainer

Dept. of Systems and Computer Engineering
Carleton University
4456 Mackenzie Building
1125 Colonel By Drive
Ottawa, ON. K1S 5B6. Canada.

Sergio Daicz Alejandro Troccoli

Departamento de Computación
FCEN – Universidad de Buenos Aires
Planta Baja. Pabellón I.
Ciudad Universitaria (1428)
Buenos Aires. Argentina.

E-mail: gwainer@sce.carleton.ca

Abstract

The use of traditional approaches to teach Computer Organization usually generates misconceptions in the students. The simulated computer ALFA-1 was designed to fill this gap. DEVS was used to attack this complex design, allowing the definition and integration of individual components. DEVS also provided a formal specification framework, which allowed reducing testing time and improving the development process. Using ALFA-1, the students acquired some practice in the design and implementation of hardware components, which is not usually achievable in Computer Organization courses.

Keywords: Applications of DEVS methodology, DEVS models, Simulation in education, Computer organization.

1. INTRODUCTION

Educators in Operating Systems and Computer Organization courses usually face several problems derived from the learning process in the area. Computer Architecture concepts are usually analyzed theoretically, leaving the students with incomplete and sometime erroneous views of how a computer works. These misconceptions remain in higher level courses making difficult a thorough learning in the area.

Computer organization literature [1, 2, 3, 4, 5] usually attacks the complexity of computer systems by using several layers to describe them. Each layer describes one abstraction level, providing higher insight when analyzing a given subsystem. These levels usually include assembly language, instruction sets, microprogramming and digital logic. Lower levels (such as transistor, electron or atomic levels) are usually not described. The layers are studied using different modeling techniques. For instance, many existing books express assembly language syntax using state machines, while circuits are described using Boolean logic. This diversity contributes to loose comprehension of system operation as a whole. Likewise, detailed behavior of the subsystems and their interaction are complex to be attacked. The introduction of higher levels (Programming Languages, Operating Systems) makes the task even more complex.

Even practice helps to make clear theory, experimental tasks are difficult to accomplish in the area of architectural design. The construction of computer architectures requires expensive laboratories and expertise in some areas not widely known in early career courses. Likewise, there are very few software tools that can be used with educational purposes. In either case, it is also difficult to provide experimental assignments to

be fulfilled in the schedule of a standard course. At present most practical experience is achieved at the assembly language level, where the available tools are well known. Nevertheless, assembly programming does not provide experience in instruction set designing, microprogramming or digital logic. With this view in mind, we proposed to build a simulated computer to be used as educational tool. We have faced this project with several objectives in mind, which include:

1. The ability to describe the multiple abstraction levels studied in Computer Organization courses;
2. The possibility to be programmed by students in early stages of their careers (considering that students usually take courses on programming before studying Computer Organization);
3. The capacity of defining different components using a unique approach;
4. Extensibility of the components;
5. Modifiability of the architectures;
6. Good testing facilities;
7. Pedagogical values: the chosen tools should have a fast learning curve, due to the lack of time available in one term courses; and
8. The availability in public domain, to be used in any existing courses without restrictions.

We have faced this project with the goal of meeting these requirements. We will explain now several existing ways of attacking the problem of modeling and simulating computer architectures, and how we faced the task to meet with our goals.

1.1. Overview of related efforts

Since simulations of computer architectures have been around since the 1970's, this problem has been attacked using several points of view. Nevertheless, none of the available tools meets our requirements, as it is explained following. In this section we tried to cover the entire spectrum in the area, but at present there are much more tools available. We include a few examples of each available type of environment, as other existing tools differ slightly of those here included.

A. General purpose tools

Many of the existing tools are general purpose, and can be applied to build any kind of processor by defining an instruction set, the computer organization and its components. Most of these tools are devoted to analyze the performance of architectural properties. For instance, SimpleScalar [6] allows the flexible simulation of modern processors. The environment defines its own architecture, and it is provided with a GCC compiler. It allows a complete architecture to be defined as building blocks, and can include advanced architectural aspects (non-blocking caches, speculative and out-of-order execution). HASE (Hierarchical Architecture design and Simulation Environment) was built for the rapid development and exploration of computer architectures with multiple abstraction levels [7]. The environment includes a design editor, object libraries for each abstraction level, and validation facilities. SimOS [8] is a complete machine simulation environment designed to study uniprocessor and multiprocessor systems. It defines different details of

an architecture by providing different CPU models. It includes a high level description of the architecture, and different components can be included: caches, multiprocessor memory buses, disk drives, consoles, and other devices.

These tools (and many other similar) allow defining the main building blocks of an architecture and their interaction, but none meets our educational goals. They are devoted to analyze processor performance under architectural changes and to do research on Architectures and Operating Systems. Therefore, they are complex to be used in early courses. In addition, several of the levels needed (for instance, digital logic or assembly language) are not supported. Often, the building blocks cannot be extended or modified. Many are unavailable for public domain, or use in large courses. Nevertheless, extensibility and modifiability can be achieved when higher level constructions are considered; also good testing facilities are available.

B. Specific purpose tools

Many existing tools are built to emulate existing architectures. This is a good approach from a pedagogical perspective, but, in general, the goals of extensibility and modifiability are constrained by the underlying architecture. All the platforms described in this section lack these facilities, and most of them cannot describe all of the abstraction levels needed.

Several tools focus on the ***Intel*** 80x86 architectures. For instance, the p86 [9] defines the Instruction Set level and the Assembly Language level of 8086-based computers. It includes an assembler and debugger, allowing the students to experience a reduced version of the 8086 processor in a simulated environment. A similar set of tools is included in the Simx86 environment [10]. This set of tools include a family of simulators for the Intel 80x86 family, including a simulator for the 8088, 80286 and a partial simulator of the 80386. SimpleScalar [6] was used to build the functional simulation of the x86 instruction set, providing a specific purpose tool tailored to Intel architectures.

Other environments use the ***MIPS*** architecture. For instance, the previously discussed SimpleScalar [6] and SimOS [8] were used to define complete architectures based on different MIPS processors. The MPS simulator [11] is based on the MIPS R3000 processor. It includes the definition of RAM, ROM, processor, disks, tapes, printer and terminal. This simulator provides good understanding of the general computer organization and instruction set levels. It also provides good facilities for teaching in early undergraduate courses. Nevertheless, it does not allow the definition or other levels, and the goals of extensibility and modifiability goals cannot be achieved.

A tool almost adequate for our purpose is Spim, a simulator of MIPS R2000/R3000 assembly language programs [12]. It implements the assembler-extended instruction set, omitting some of the complex details. A good feature of this tool is that it is associated with a renowned book on Computer Architecture. Nevertheless, this simulator does not define many of the multiple abstraction levels described in other literature, and therefore is not useful for many existing courses in the area. Also, extensibility and modifiability are limited.

Several other currently used architectures have been built using simulation. For instance, SimOS [8] also was used to model ***Silicon Graphics*** and ***Digital Alpha*** processors. Alpha processor simulators were also presented in [16]. In the latter case, the authors have used simulation to find an architectural solution that satisfies some of the product goals for the Alpha architecture. They have used the tools to analyze pipelining levels and instruction-level parallelism. As we can see, several modern architectures have been modeled, and can be analyzed using simulation. However, none of them are suitable to be used in undergraduate courses. They do not meet any of our pedagogical needs, are difficult to be extended, or to be used to model other architectures.

We also could see that several other architectures are used to study architectures. The CHIP (Cornell Hypothetical Instructional Processor) is a simulated computer emulating a ***PDP-11*** processor. It was designed as educational tool for undergraduate courses, including dynamic memory mapping, two modes of processor operation and eight interrupt priority levels. It also supports emulated I/O devices, a debugger and a C compiler. PROVIR [14] is a virtual processor based on the ***IBM 360*** architecture. It includes the instruction level definition, an assembler, a debugger, and the kernel of an operating system. In [15], the authors describe a method for simulating the ***Z80*** processor using spreadsheets. The user can write assembly language programs, which are assembled and executed using a spreadsheet. In all these cases, the authors focused on defining the instruction set. No detailed specification of lower levels was included. Worse yet, the environments are based on processors that are not used nowadays, and the students cannot gain experience with current architectures.

Many other simulators are designed to analyze ***multiprocessor*** systems. For instance, Limes [17] simulates N processors running a parallel application. The tool implements the assembly language level, and can be used to evaluate architectures or parallel algorithms. PROTEUS is a high performance simulator for MIMD multiprocessors [18]. It was developed to simulate a wide range of architectures, trying to improve accuracy and performance. Several processors are connected via a bus or a network, but it is devoted to execute an application in multiple CPUs. In [19] a tool for the modeling and simulation of clustered computers was presented. The goal was to construct architectures of symmetric multiprocessors, clusters of uniprocessors, and to evaluate their performance through benchmarking. Several other examples of multiprocessor simulation can be found in [20, 21, 22, 23, 24, 25, 26, 27 28 and 29]. We do not describe these in detail, because none of them are adequate in the educational sense, nor meet our requirements. They could be used in higher level courses to support computer architecture lectures, but they are not adequate for our goals due to their complexity.

1.2. Development Approaches

After concluding that existing simulation tools were not appropriate, we decided to build a toolkit to meet all of our requirements. The next step was to choose which kind of development environment to use. Any simulation language could have been applied: Modelica [30], Maisie [31], Simulink [32], ACLS [33], ModSim [34], Simscript [35], etc. Many of the simulators included in section 1.1. were built using this ap-

proach. For instance, in [7] the tools were built using Sim++ [36]. In [19] BONeS, a Block-oriented Network simulator was used as the building tool [21]. Another possibility included the use of a Hardware Description Language (HDL).

HDLs are indispensable for computer and digital design. Presently, VHDL, Verilog and SDL are three of the most widely used description languages. VHDL [37] was developed by IBM, TI and Intermetrics in 1983, and became an IEEE standard in 1987 (and 1993). VHDL can be used for documentation, verification, and synthesis of large digital designs. Three different approaches can be used to describe hardware using this language: structural, data flow, and behavioral methods. To make designs more understandable and maintainable, a design is typically separated into several blocks. This might be done with a block diagram editor or the use of hierarchical drawings to represent a block diagram. Once the basic building blocks of a design are defined, they can be interconnected to create a larger design.

Verilog [38] is less sophisticated than VHDL and it was developed in 1983, becoming IEEE standard in 1995. Verilog is easier to learn than VHDL, but lacks constructs to support system level design. Structural models are built from gate primitives and other modules. Structural models describe a circuit using logic gates, letting the user to specify the function and delay for a gate. Test modules can be associated with designs. Once the structural models are defined, behavioral models can be included to define submodels in terms of inputs and outputs. A behavioral model can be used to test structural designs. Logic synthesis can be achieved from the model specifications, providing alternative implementations.

SDL [39] is a Specification and Description Language, standardized as ITU recommendation Z.100 (in 1980, latest version in 2000). It is a wide spectrum language to specify from requirements to implementation. It was developed as a description language for reactive systems, which allows the presentation in a graphical form as extended finite states. The basic theoretical model of an SDL system consists of a set of state machines that run in parallel. These machines are independent of each other and communicate with discrete signals. An SDL system consists of structure (including system, block, process, and procedure hierarchy), communication (signals with optional signal parameters and channels), behavior, data (in the form of abstract data types), and inheritance. It is a language widely used in telecommunications, but it has also been applied to other areas.

The use of a simulation language or a HDL could have provided good results for many of our goals. We would have prefered HDLs to simulation languages due to a variety of reasons. A proposed architecture can be extended or modified easily. Most HDLs include verification and validation tools. Multiple abstraction levels can be described in detail. Nevertheless, we face several educational problems if we intend to use a HDL or a simulation language. The main problem is that learning any of these languages could take most of the term before starting to use them in a simulated architecture. In this sense, some Computer Engineering careers include early courses on HDLs, and they could be a prerequisite for Computer Organization. But this is not the case of many Computer Science degrees, where Computer Organization is taught only to support future courses. Moreover, in several cases, the language constructions constrain the components that

can be simulated, suffering from limitations (for instance, VHDL is inadequate to represent mixed analog and digital processing [40]).

Due to these reasons, we preferred to develop models using a general purpose language (especially if a public domain compiler is available). Standard programming languages are flexible enough to describe multiple levels, to extend or modify a given architecture, and programming courses are prerequisite for Computer Organization subjects. This approach was used in several of the simulation tools presented in section 1.1. Some of these simulators were built using standard languages (C, C++ or Java; even in [15], the tool was developed using an Excel spreadsheet).

As a first stage of this project we built a simple computer called Alfa-0 [41] using C++ to develop each of the system's levels. This set of tools let the students to better understand the complete behavior in each layer. We built an environment similar to Spim, but emulating a SPARC processor [42], and a complete emulator of the ATARI processor [43]. The emulator allowed standard ATARI games to run on in any Intel processor, by defining the behavior of the processor and input/output subsystem.

Assembly language, microarchitecture, and digital logic levels were simulated individually, providing a complete outlook of the system organization. Unfortunately, the models were too complex to be integrated. Likewise, the use of a standard programming language caused the students to confuse the models developed with their simulators. This also lead to difficulties in extending or modifying the architecture. To avoid these problems, the simulated computer was completely redesigned using DEVS [44] as the modelling framework. This paradigm was chosen due to the hierarchical and discrete event nature of the problem to study. The following section will explain some basic aspects about this decision.

1.3. Overview of the DEVS modelling paradigm

DEVS provides a systems theoretic framework for describing discrete event systems as composites of sub-models. Each submodel can be behavioral (called *atomic*) or structural (called *coupled*), consisting of a time base, inputs and states that are used to compute the next states and outputs. Every model can be integrated into a hierarchy, allowing the reuse of tested models. A DEVS atomic model is described by:

$$M = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D >$$

X: input events set;

S: state set;

Y: output events set;

δ_{int}: $S \rightarrow S$, internal transition function;

δ_{ext}: $Q \times X \rightarrow S$, external transition function, with $Q = \{ (s, e) / s \in S, \text{ and } e \in [0, D(s)]\}$;

λ: $S \rightarrow Y$, output function; and

D: $S \rightarrow \boldsymbol{R}_0^+$, duration function.

Models use input/output ports to communicate. Each state in a model has a given lifetime, defined by the duration function. Once the lifetime of a given state finishes, the internal transition function is activated to

produce an internal state change. Before this change, the present state of the model can be spread through the output ports. These ports allow events to be sent to other models. The values are sent by the output function, which must execute before activating the internal transition. At any moment, a model can receive input external events from other models through its input ports. When an external event arrives, the external transition function is activated. The external transition function computes a new state for the model using the present state, the input values, and the elapsed time for the model (defined by the duration function). Every time a transition function is activated, a new lifetime must be associated with the new state.

DEVS atomic models can be used to build coupled models, defined by:

$$CM = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, \text{select} >$$

X is the set of input events;

Y is the set of output events;

D is an index of components, and $\forall i \in D$,

$\mathbf{M_i}$ is a basic DEVS model;

$\mathbf{I_i}$ are the influencees of model i, and $\forall j \in I_i$

$\mathbf{Z_{ij}}$: $Y_i \rightarrow X_j$ is the i to j translation function, and

select is the tiebreak selector.

Each coupled model consists of a set of basic models (atomic or coupled) connected through the input/output ports of the interfaces. Each component is identified by an index number. The influencees of each model define other models where output values must be sent. The translation function uses an index of influencees, created for each model (I_i). The function defines which outputs of model M_i are connected to inputs in model M_j. When two submodels have simultaneous events, the *select* function defines which of them should be activated first.

Unlike the other approaches presented earlier, DEVS meets all of our goals:

1. DEVS is a hierarchical and modular technique that allows the description of the multiple levels of an architecture. The SES/SB technique [45] even lets the user define different architectures in the same class hierarchy, choosing between different versions as needed.
2. There are different DEVS development environments that can be adapted to different teaching programs. According to the programming paradigm taught in the first year courses, different existing DEVS toolkits can be used: those using the procedural paradigm (mainly written in C/C++), those object-oriented (written in Java or C++), or the functional versions (DEVS/Scheme [45]).
3. DEVS supports the definition of models specified in different paradigms, allowing definition of multi-components, each defined using a different technique.
4. DEVS allows any existing model to be extended easily.
5. Coupled or atomic models can be modified.

6. Each model can be associated with an Experimental Framework (a set of DEVS atomic models that can be coupled with other DEVS models, providing an environment for conducting experiments) used as a testing module. This approach improves testing facilities.
7. The learning curve for DEVS is fast enough to be applied in undergraduate courses. Our students learned the basic aspects of the methodology and related tools in approximately 16 man/hours (taking only 2 man/hours to teach the basic aspects, and having a 2 hour training session) [46].
8. Many of the existing DEVS environments are public domain.

Besides meeting our individual goals, DEVS provides several advantages over the other approaches:

a) DEVS provides the advantages of being a formal approach. Formal specification mechanisms are useful to improve the security and development costs of a simulation. A formal conceptual model can be validated, improving the error detection process and reducing testing time. DEVS models are closed under coupling, therefore, a coupled model is equivalent to an atomic one, improving reuse. DEVS supplies facilities to translate the formal specifications into executable models. In this way, the behavior of a conceptual model can be validated against the real system, and the response of the executable model can be verified against the conceptual specification.

b) The existence of an internal transition function is a unique feature that eases the definition of certain properties. Internal state changes can be captured, describing complex internal interactions in a simple and natural way. For instance, if we intend to model a timer with different skews from a unique clock, we can use one signal generator, and the internal state of the clock could define the different output signals. Certain circuits (for instance, synchronous buses) react according to their internal state, which can be modeled straightforwardly using internal transition functions. Modeling of these phenomena is difficult under other methodologies.

c) DEVS is a complete modeling and simulation technique. It provides a way to specify models that can be coupled into higher level ones, which are later simulated by independent abstract entities (in centralized or parallel fashions). Each model can be associated with an experimental framework, allowing the individual testing of components and making integration testing easier.

d) DEVS, as a discrete event paradigm, uses a continuous time base, which allows accurate timing representation. Precision of the conceptual models can be improved, and CPU time requirements reduced. Higher timing precision can be obtained without using small discrete time segments (that would increase the number of simulation cycles).

e) Recently, a theory of DEVS quantized models was developed [47]. The theory has been verified when applied to predictive quantization of arbitrary ordinary differential equation models. Quantized models reduce substantially the frequency of message updates. As the information interchange is reduced, the models potentially incur into error. In this way, DEVS can be used to express hybrid digital/analog systems. GDEVS [40] also enables the definition of hybrid models, which are expressed in a combined discrete

event/differential equation formalism approximated by DEVS. In GDEVS, the accuracy of an analog subsystem is preserved using piecewise polynomial segments. The error introduced in this approximation can be controlled by increasing the order of the polynomials that represent analog signals between successive digital events.

Considering these advantages, we have designed and implemented our simulated computer, called Alfa-1 using the CD++ development environment [48]. This toolkit implements the theoretical concepts defined by the DEVS formalism. Atomic models can be programmed in C++, and can be later incorporated to a model class hierarchy. A specification language allows the definition of coupled models.

Alfa-1 was developed to model the architecture of a SPARC processor, also including memory, a bus and an input/output subsystem. If we compare our proposal with the ones defined in section 1.1., Alfa-1 was developed as a specific purpose architecture. Nevertheless, as all the components of the architecture have been developed independently, they can be used to define new components or other architectures. Several versions of each element have been developed using different abstraction levels. As the models have been developed using DEVS, they can be reused without further complication.

We have chosen the SPARC architecture, because these processors include several interesting features (for instance, multiple registers organized as overlapping windows) that cannot be found in other architectures. Many existing workstations are based in this processor, which are usually expensive. As we have reproduced the complete architecture, we have provided a way of running SPARC applications in other platforms provided with a GNU C++ compiler.

In the following sections we present some of the results obtained. We first include a definition of the underlying architecture. Then, the specification of some of the DEVS submodels is presented, exemplifying the definition of each model using CD++. Finally, we show some execution results.

As explained in the conclusions, we encountered none of the problems associated with other tools, and all of our goals were achieved. An important remark is that the approach proved to accomplish our educational goals, as undergraduate students developed the whole architecture and its components. They have taken a previous course on computer programming in C++, and have background in Mathematics. The definition of the formal models was done by 3rd year students of a Discrete Event Simulation course, and these formal specifications were used by students in the 2nd year Computer Organization course to build all the models that are described in the following sections.

2. A MODEL OF THE PROCESSOR ARCHITECTURE.

Alfa-1 uses a processor organization based in the specification of the Integer Unit of the SPARC processor (Sun Microsystems). Figure 1 shows a sketch of this architecture with the goal of understanding the main components of the model developed. This figure presents the main subcomponents of the Integer Unit,

which were defined as the components of a DEVS coupled model. Each of the component was either defined as an atomic or coupled model, specifying them using DEVS.

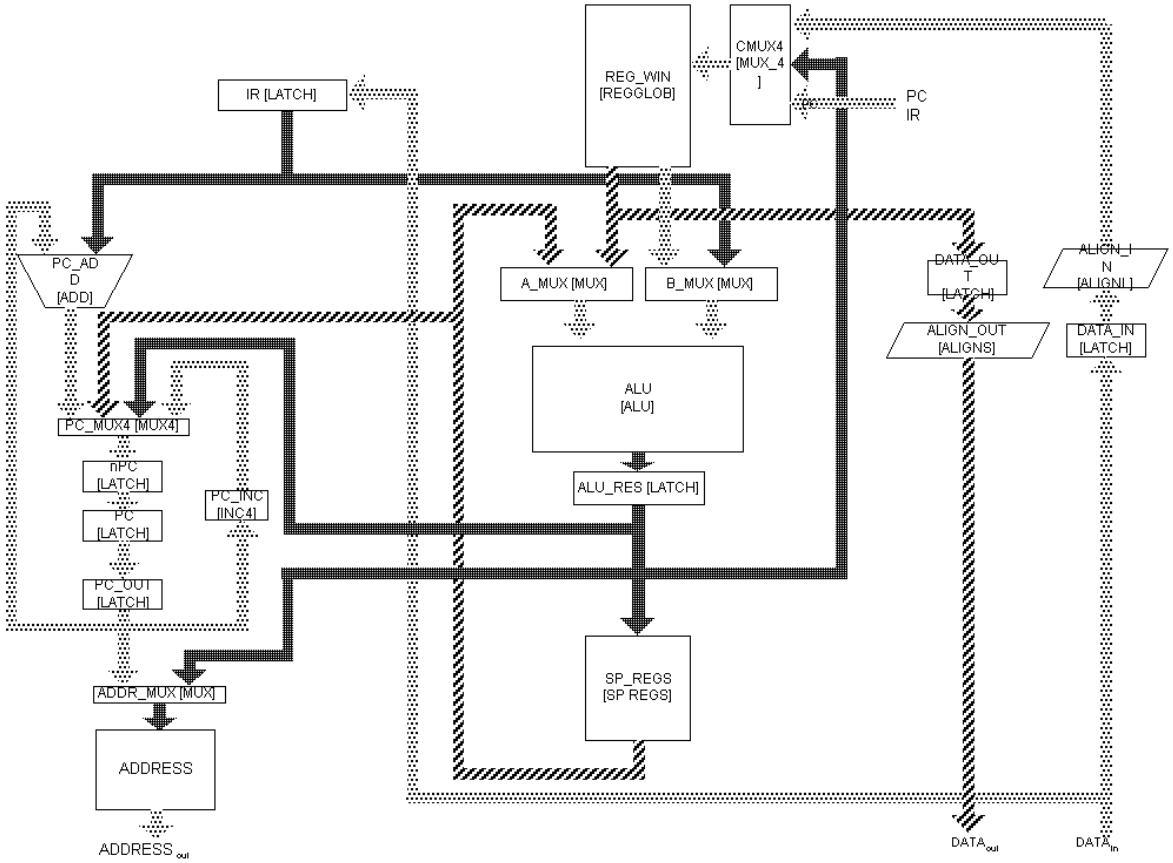

Figure 1. Organization of the Integer Unit.

This RISC processor is provided with 520 integer registers. Eight of them are global (*RegGlob*, shared by every procedure), and the remaining 512 are divided in windows of 24 registers each (*RegBlock*).

Each window includes input, output and local registers for every procedure that has been executed recently. When a routine begins, 16 new registers are reserved (8 local and 8 output), and the 8 output records of the calling procedure are used as inputs. A specialized 5-bit register, called *CWP* (Circular Window Pointer) marks the active window. Every time a new procedure starts, *CWP* is decremented.

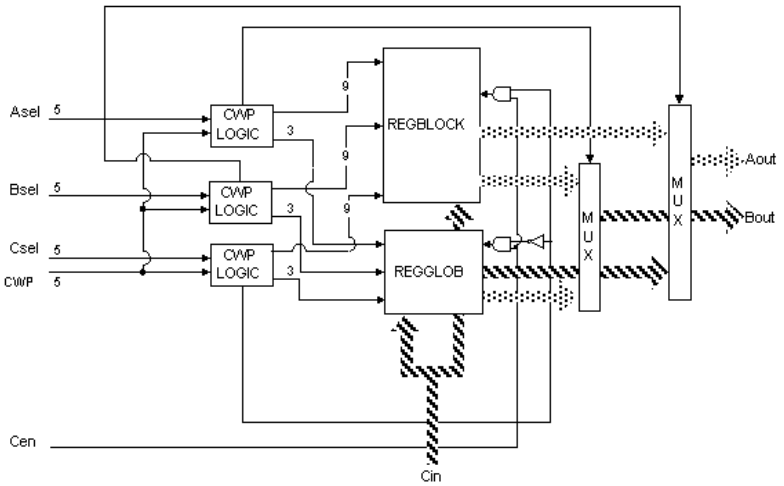

Figure 2. Organization of the processor's registers.

Besides these general purpose registers, the architecture includes:

- **PCs**: the processor has two program counters. The PC contains the address of the next instruction. The nPC (Next PC) stores the address of the PC after the execution of the present instruction. Each instruction cycle finishes by copying the nPC to the PC, and adding 4 bytes (one word) to the nPC. If the instruction is a conditional branch, nPC is assigned to PC, and nPC is updated with the jump address (if the jump condition is valid).
- **Y**: is used by the product and division operations;
- **BASE** and **SIZE:** The memory is considered flat (that is, neither segmentation nor pagination mechanisms are included). Likewise, multiprogramming is not supported. The BASE register points to the lowest address a program can access. The SIZE stores the maximum size available for the program.
- **PSR** (Processor Status Register): stores the current status for the program. It is interpreted as follows.

Bits	Content	Description
31..24	Reserved	
23	N – Negative	1 when the result of the last operation is negative
22	Z – Zero	1 when the result of the last operation is zero
21	V – Overflow	1 when the result of the last operation is overflow
20	C – Carry	1 when the result of the last operation carried one bit
19..12	Reserved	
11..8	PIL – Processor Interrupt Level	Lowest interrupt number to be serviced.
7	S – State	1= Kernel mode; 0=User mode.
6	PS – Previous State	Last mode.
5	ET – Enable Trap	1=Traps enabled; 0=Traps disabled.
4..0	CWP – Current Window Pointer	Points to the current register window.

Table 1. Contents of the Process Status Register

- **WIM** (Window Invalid Mask): this 32-bit register (one bit per window) is used to avoid overwriting a window in use by another procedure. When *CWP* is decremented, these circuits verify if the *WIM* bit is active for the new window. In that case, an interrupt is raised and the interrupt service routine stores the content of the window in memory. Usually, *WIM* only has one bit in 1 marking the oldest window.
- **TBR** (Trap Base Register): it points to the memory address storing the position of a trap routine.

Bits	Content	Description
31..12	Trap base address	Base address of the Trap table
11..4	Trap Type	Trap to be serviced
3..0	Constant (0000)	

Table 2. Contents of the Trap Base Register

The first 20 bits (Trap Base Address) store the base address of the trap table. When an interrupt request is received, the number of the trap to be serviced is stored in the bits 11..4. Therefore, the TBR points to the table position containing the address of the service routine. The last 4 bits in 0 guarantees at least 16 bytes to store each routine.

When the **instruction set** level of the SPARC architecture is analyzed, we see that each instruction has a fixed size of 32 bits. Memory operands may be 8, 16 or 32 bits. There are basic *Load*/*Store* operations, classified according to the size and sign of their operands.

Arithmetic and Boolean operations include *add*, *and*, *or*, *div*, *mul*, *xor*, *xnor*, and *shift*. These are able to change the PSR, according to the operation code used. Several *jump* instructions are available, including

relative jumps, *absolute* jumps, *traps*, *calls*, and *return* from traps. Other instructions include the movement of the register window, NOPs, and read/write operations on the PSR.

Multiplication uses 32-bit operands, producing 64-bit results. The most significant 32 bits are stored in the Y register, and the remaining in the ALU-RES register. Integer *division* operations take a 64-bit dividend and a 32-bit divisor, producing a 32-bit result. The Y register stores the 32 most significant bits of the dividend. One ALU input register stores the least significant bits of the dividend, and the other, the divisor. The integer result is stored in the ALU-RES register, and the remainder in the Y register. Most instructions are carried out by the ALU, whose structure is depicted in the following figure. It includes two multiplexers connected to the ALU, Multiplier/Divider unit and shifter.

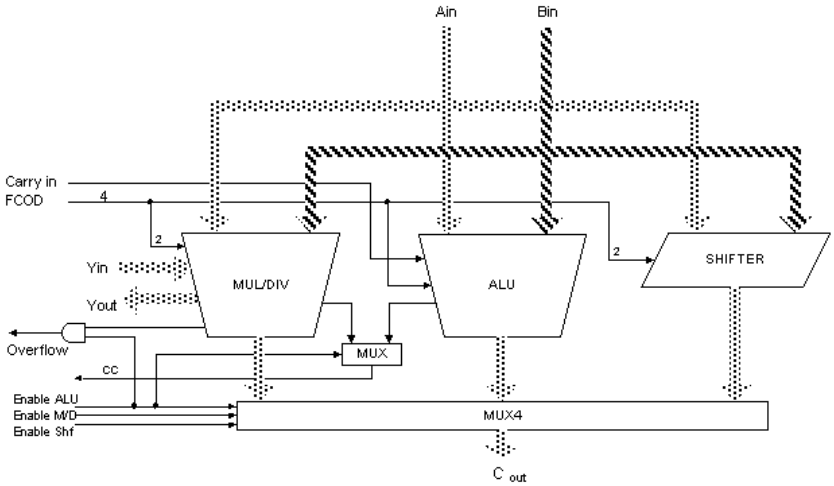

Figure 3. Organization of the ALU.

There are two execution modes: *User* and *Kernel*. Certain instructions can only be executed in Kernel mode. Also, the Base and Size registers are used only when the program is running in User mode.

The CPU executes under the supervision of the **Control Unit**. It receives signals from the rest of the processor using 64 input bits (organized in 5 groups: the Instruction Register, the PSR, BUS_BUSY_IN, BUS_DACK_IN, and BUS_ERR). Its outputs are sent using 70 lines organized in 59 groups. Some of them include reading/writing internal registers, activating lines for the ALU or multiplexers. Also, connections with the PC, nPC, Trap controller and PSR registers are included. Finally, the Data, Address and Control buses can be accessed.

The **memory** is organized using byte addressing and Little-Endian to store words. The processor issues a memory access operation by writing an address (and data, if needed) in the bus. Then, it turns on the AS (Address Strobe) signal, interpreted by the memory as an order to start the operation. The memory uses the address available and analyzes the RD_WR line to see which operation was asked. If a *read* was issued, one word (4 bytes) is taken from the specified address and sent through the data lines. In a *write* operation the address stored in the Byte Select register (lines BSEL0..3) defines the byte to be accessed in the word pointed to by the Address register. If an address is wrong, the ERR line is turned on. A Data Acknowledge (DTACK) is sent when the operation finished.

The system components are interconnected using a Bus (see Figure 4). The bus Masters use the BGRANT (Bus Grant) and IACK (IRQ Acknowledgment) lines to be connected to the two devices with the following lower and upper priorities. The device with the highest priority is connected to a constant "1" signal in the BGRANT line. The BGRANT signal is sent to the lower priority devices up to the arrival to a device that requested the bus. When the device finishes the transfer, a IACK is transmitted. Input/output operations are memory mapped. Each device have a fixed set of addresses. Data written in those addresses are interpreted as an instructions for a device. Fifteen IRQ lines (IRQ1..IRQ15) are provided, and devices are connected to these lines. Higher priority devices are connected to lower IRQs.

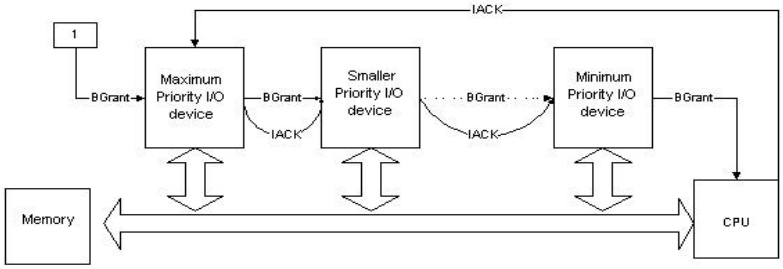

Figure 4. Organization of the Bus.

Finally, an external cache memory was defined. The generic structure for the cache controller is defined in figure 5. The design and implementation of these modules were not included in the original version of Alfa-1. They were defined as an assignment done by undergraduate students, following the procedures that will be presented in the following sections. In a first stage, the circuits were tested separately, different algorithms were implemented, and finally the device was integrated into the architecture. Each model was developed as a DEVS model that was integrated into a coupled model. This extension to the original architecture (which not will be explained in detail) shows some of the capabilities for extensibility and modifiability of Alfa-1.

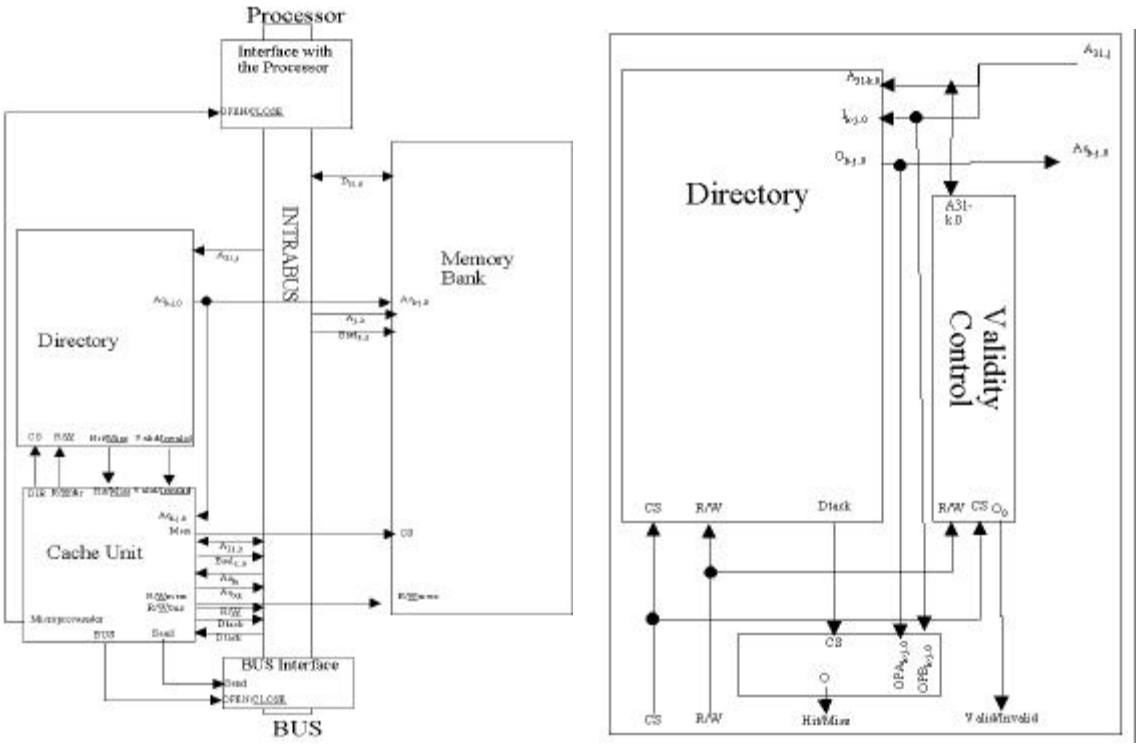

Figure 5. Organization of the Cache memory.

3. IMPLEMENTING THE ARCHITECTURE AS DEVS MODELS

The architecture presented in the previous section was completely implemented using CD++. First, the behavior of each component was carefully specified, analyzing inputs, outputs and timing for each element. The specification also provided test cases. Then, each component was defined as a DEVS following the specification. After, each model was implemented in CD++, including an experimental framework following the test cases defined in the specification. Finally, the main model was built as a coupled model connecting all the submodels previously defined. This model follows the design presented in the figure 1, and its detailed definition can be found in [49].

Two implementations were considered. First, we reproduced the basic behavior of each circuit, coded as transition functions. Then, some of them were implemented in detail using Boolean logic. The basic building blocks were developed as atomic models, coupling them using digital logic concepts. In this way, two different abstraction levels were provided. Depending on the interest, each of them can be used. Once thoroughly tested, the basic models were integrated into higher level modules up to completing the definition of the architecture. The following sections will be devoted to present some of the components implemented as assignments done by our students. We show how different abstraction levels can be modeled, and present examples of modifiability of Alfa-1.

3.1. Inc/Dec

As explained earlier, we use 520 general purpose registers organized as overlapped windows. In a given time, only one window can be active. The *Inc/Dec* model is the component that chooses the active window using a 5-bit *CWP* register. The models that are part of the CWP logic are shown in the figure 2. The *CWP* is incremented or decremented, and its value (stored in a d-latch represented as another DEVS) is received through the lines OP0-OP4. The outputs are transmitted through the lines RES0-RES4. This atomic model can be defined as:

$$\textbf{INC/DEC} = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D >.$$

$\mathbf{X} = OP \in \{0,...,2^5-1\} \cup FCOD \in \{0,1\}$;

$\mathbf{S} = OP, OLD \in \{0,...,2^5-1\}, delay\_time\_ID \in \boldsymbol{R}_0^+$; $\mathbf{Y} = RES \in \{0,...,2^5-1\}$;

The behavior for the transition functions can be informally defined as follows:

```
δext() {
   When (x is received in the port y) _y = x;
   If (FCOD = 1)            RES0..4 = ( _OP0..4 + 1 ) mod 10000h;      // increment
  else                      RES0..4 = ( _OP0..4 – 1 ) mod 10000h;      //  decrement
   hold_in(delay_time_ID)
}

δint() { passivate; }

λ() {
   If (RES0..4 is different to the last output) send RESP0..4 through the ports OP0..4;
  continue;
}
```

Figure 6. Behavior of the transition functions for the INC/DEC model [50].

The FCOD value is used to tell if the value must be incremented or decremented. The ALU model is used to do this operation. Here we can see that, when an external event arrives, the *hold_in* function is activated. This macro represents the behavior of the DEVS time advance function (D), and it is in charge of manipulating the *sigma* variable. This is a state variable predefined for every DEVS model, which represents the remaining time up to the next scheduled internal event. The model will remain in the current state during this time, after which an output and internal transition functions are activated. The *hold_in* macro makes this timing definition easier. *Passivate* is another macro, which uses an infinite *sigma*¸ and puts the model in *passive* phase (*hold_in(passive, infinite)*).

The following figure shows the implementation of these functions using CD++. As we can see, the *external transition* function (δ_{ext}) receives five operands as inputs, together with a function code. According to this code, the parameter is incremented or decremented. After, the model keeps the present value during a delay related with the circuit operation. The *output* function (λ) is activated, and if the circuit changed its state the present value is transmitted. Then, the *internal transition* function (δ_{int}) passivates the model (that is, an internal event with infinite delay is scheduled, waiting for the next input). The constructor allows to specify the model's name, input/output ports, and parameters.

As we can see, the definition of a DEVS atomic model is simpler than the use of any standard programming language. We have explained some of the advantages of using DEVS in section 1, but in this case, we can see how to apply it to build our models. DEVS provides a interface, consisting of only four functions to be programmed. This modular definition is independent of the simulator, and it is repeated for every model. Therefore, one can focus in the model development. The user only concentrate in the behavior under external events, the outputs that must be sent to other submodels, and the occurrence of internal events. Behavior for every model is encapsulated in these functions, together with the elapsed time definition. Testing patterns can be easily created, as the model can only activate these functions.

```
IncDec::IncDec( const string &name ):
 Atomic( name )
, OP0( this->addInputPort( "OP0" ) ), OP1( this->addInputPort( "OP1" ) )
, OP2( this->addInputPort( "OP2" ) ), OP3( this->addInputPort( "OP3" ) )
, OP4( this->addInputPort( "OP4" ) ), FCOD( this->addInputPort( "FCOD" ) )
, RES0( this->addOutputPort( "RES0" ) ), RES1( this->addOutputPort( "RES1" ) )
, RES2( this->addOutputPort( "RES2" ) ), RES3( this->addOutputPort( "RES3" ) )
, RES4( this->addOutputPort( "RES4" ) ), preparationTime( 0, 0, 10, 0 )   {
       string time( MainSimulator::Instance().getParameter(
           this->description(), "preparation" ) ) ;
   if( time != "" ) preparationTime = time ;
}
Model &IncDec::externalFunction( const ExternalMessage &msg ) {

// Check the input ports, assigning the input values.

    if( msg.port() == OP0 ) _OP[0] = (int) msg.value();
    if( msg.port() == OP1 ) _OP[1] = (int) msg.value();
    if( msg.port() == OP2 ) _OP[2] = (int) msg.value();
    if( msg.port() == OP3 ) _OP[3] = (int) msg.value();
    if( msg.port() == OP4 ) _OP[4] = (int) msg.value();
    if( msg.port() == FCOD ) _FCOD = (int) msg.value();

    if (_FCOD == 1) {  // Increment
        for (int i=0; i<=4; i++) v[4-i] = _OP[i];

        alu.activate(v,"00000","11",'1');     // Increment the va value useing the ALU
        alu.output(res);

        for (int i = 0; i<=4; i++)
               _RES[i].activate(res[i]);
    }
    else
    {                  // Decrement
        for (int i=0; i<=4; i++) v[4-i] = _OP[i];
```

```
        alu.activate(v,"11111","11",'0');     // Decrement the v value useing the ALU
        alu.output(res);

        for (int i = 0; i<=4; i++)
                _RES[i].activate(res[i]);
     }
    this->holdIn(active, preparationTime);   // Schedule a delay for the circuit
  return *this;
}


Model &IncDec::internalFunction( const InternalMessage & ) {
    this->passivate();        // When the delay is consumed, activate the output
        return *this ;
}

Model &IncDec::outputFunction( const InternalMessage &msg )  {
    if (_RES[0]!=_OLD[0] || _RES[1]!=_OLD[1] || _RES[2]!=_OLD[2] ||
        _RES[3]!=_OLD[3] || _RES[4]!=_OLD[4]) {
         sendOutput(msg.time(), RES0, _RES[0] );
         sendOutput(msg.time(), RES1, _RES[1] );
         sendOutput(msg.time(), RES2, _RES[2] );
         sendOutput(msg.time(), RES3, _RES[3] );
         sendOutput(msg.time(), RES4, _RES[4] );
            _OLD[0]=_RES[0];      _OLD[1]=_RES[1];
            _OLD[2]=_RES[2];       _OLD[3]=_RES[3];
            _OLD[4]=_RES[4];
    }
        return *this ;
}
```

Figure 7. INC/DEC model definition: transition functions[50].

Once we have defined the atomic model, we can test it by injecting input values and inspecting the outputs. An experimental frame can be built, including pairs of input/output values to test the model automatically. In any case, we have to build a coupled model including the model to be tested. This is defined as follows:

```
[top]
components : I_D@IncDec
out : RES0 RES1 RES2 RES3 RES4
in : OP0 OP1 OP2 OP3 OP4 FCOD
Link : OP0@top OP0@I_D
Link : OP1@top OP1@I_D
Link : OP2@top OP2@I_D
Link : OP3@top OP3@I_D
Link : OP4@top OP4@I_D
Link : FCOD@top FCOD@I_D
Link : RES0@I_D RES0@top
Link : RES1@I_D RES1@top
Link : RES2@I_D RES2@top
Link : RES3@I_D RES3@top
Link : RES4@I_D RES4@top

[I_D]
preparation : 0:0:5:0
```

Figure 8. INC/DEC coupled model definition [50].

These definitions follow the DEVS specifications. They are defined by its *components* (in this case, I_D, an instance of the IncDec model) and external parameters. Then, the *links* define the influencees and translation functions including the input/output ports for the model. In this case, the I_D model is related with the Top model, using the input/output ports defined earlier.

3.2. RegGlob

This model defines the behavior of the global registers. It keeps the contents of the 8 global registers, allowing read/write operations on them. Two auxiliary state variables, *olda* and *oldb*, store the last outputs, and output signals are transmitted only for the bits that changed. This model is defined by:

RegGlob = < X, S, Y, δ_{int}, δ_{ext}, λ, D >

X = ASEL $\in$ {0,..., 2^3–1 } $\cup$ BSEL $\in$ {0,..., 2^3–1} $\cup$ CSEL $\in$ {0,..., 2^3–1} $\cup$ CEN $\in$ {0,1} $\cup$ RESET $\in$ {0,1} $\cup$ CIN $\in$ {0,..., 2^{32}–1 };

Y = AOUT $\in$ {0,..., 2^{32}–1 } $\cup$ BOUT $\in$ {0,..., 2^{32}–1 }.

S = OLDA, OLDB, INPUT $\in$ {0,..., 2^{32}–1}, IN $\in$ {0,..., 2^{32}–1}^{8}, BCEN, BRESET $\in$ {0,1}, SELECTA, SELECTB, SELECTC $\in$ {0,..., 2^{32}–1}, delay_time_RG $\in$ $\boldsymbol{R}_0^+$;

A sketch of this model was shown in the figure 2. As we can see, it uses three select lines (*asel*, *bsel*, and *csel*) to choose two output registers and a register to be modified. An array of 32 integers (*IN*) keeps the present values of the registers. The Boolean line *cen* (C enable line) is used to allow write operations. The external transition function models the reception of an input. The function stores the desired operation according to the signal received. Also, we store an input value in the number of register to be activated. A new internal event is scheduled with a predefined delay, which models the circuit delay. If an external event arrives before the end of the delay, the operation is cancelled.

```
Model &Regglob::externalFunction( const ExternalMessage &msg ) {
   switch (msg.port()) {
        case cen:      bcen = (int)msg.value();      // C enable line turned on
        case reset:    breset = (int)msg.value();    // Reset
   }

   if( msg.port() == "cin+i" ) input[i]=(int)msg.value();  // Store the input lines

   if( msg.port() == "asel+i" ) {     // The i-eth line of the A input was enabled
     selecta= msg.value();            // Store the register number received
   }

   if( msg.port() == "bsel+i" ) {     // The i-eth line of the B input was enabled
     selectb= msg.value();            // Store the register number received
   }

   if( msg.port() == "csel+i" ) {     // The i-eth line of the C input was enabled
     selectc= msg.value();            // Store the register number received
   }
 this->holdIn ( active, delay );
 return *this;
}


Model &Regglob::internalFunction( const InternalMessage &msg )  {

   if (breset)                        // A reset signal was issued
     for (int i=0; i<255; i++) in[i]=0;      // The 8 register (32 bit each) are deleted
   if (bcen)                                 // The write line was enabled
    for (int i=0; i<32; i++)                 // Update the desired register
        in[(selectc*32)+i]=input[i];

  this->passivate();                         // Wait the next internal event
  return *this ;
}

Model &Regglob::outputFunction( const InternalMessage &msg )  {

   if (olda[i] != in[selecta*32+i]) {        // The register has changed
        this->sendOutput(msg.time(), aout, in[selecta*32+i]);
        olda[i] = in[selecta*32+i];  }       // Transmit it through the output line

   if (oldb[i] != in[selectb*32+i]) {        // The register has changed
        this->sendOutput(msg.time(), bout, in[selectb*32+i]);
        oldb[i] = in[selectb*32+i];  }       // Transmit it through the output line

  return *this ;
}
```

Figure 9. RegGlob model definition: transition functions [50].

The output function decides if the register has changed, querying *olda* and *oldb* which stores the previous status of the A and B lines. When the register changes, its value is sent through the chosen output (A or B). This model shows a more interesting use of the internal transition function. In this case, we are considering the internal state to decide how the model must react. The internal transition function sees if the *reset* line is activated. In that case, it clears the contents of every register. Then, if the *cen* line was activated, the value of the chosen register is updated with the new input.

3.3. Other basic components

The architectural description is completed with other several DEVS models. We include generic aspects, making a brief description of their behavior. We do not include the definition of the model's transition functions, built as in the previous examples. Details of these models can be found in [49].

WIMCheck = < X, S, Y, δ_{int}, δ_{ext}, λ, D >.

X = CWP $\in$ {0,..., 2^5–1 } $\cup$ WIM $\in$ {0,..., 2^{32}–1};

S = dlLastRES, RES $\in$ {0,..., 2^5–1 } $\cup$ delay_time_WC $\in$ $\boldsymbol{R}_0^+$;

Y = RES $\in$ {0, 1} ;

This circuit checks if the next window to be used will be overwritten. The component consists of a Window Invalid Mask register. It returns the value of the CWP-eth bit of the WIM register.

MEMORY = < X, S, Y, δ_{int}, δ_{ext}, λ, D >

X = DATA $\in$ {0,..., 2^{32}–1 } $\cup$ ADDRESS $\in$ {0,..., 2^{31}–1} $\cup$ ADDRESS_STROBE $\in$ {0,1} $\cup$ BSEL $\in$ {0,..., 2^4–1} $\cup$ RD_WR $\in$ {0,1} $\cup$ RESET $\in$ {0, 1};

S = memory: array(memsize: default 32 Kb), delay_time_M $\in$ $\boldsymbol{R}_0^+$;

Y = DATA $\in$ {0,..., 2^{32}–1 } $\cup$ DTACK $\in$ {0, 1} $\cup$ ERR $\in$ {0, 1}.

The memory is provided with three basic operations: read, write and reset. When a reset is issued, the memory initial image is loaded. The processor writes an address in the bus, and signals the memory using the AS signal when the address is ready. Then, a read/write signal is issued. The memory reacts according with this signal, using an output after a time related with the memory latency.

ADDER = < X, S, Y, δ_{int}, δ_{ext}, λ, D >

X = OPA, OPB $\in$ {0,..., 2^{32}–1 }; **S** = delay_time_A $\in$ $\boldsymbol{R}_0^+$;

Y = RES $\in$ {0,..., 2^{32}–1 } $\cup$ CARRY $\in$ {0, 1}.

The adder receives two inputs. Depending on the result, the Carry bit can be turned on.

ALIGNL/ALIGNS = < X, S, Y, δ_{int}, δ_{ext}, λ, D >

X = OP $\in$ {0,..., 2^{32}–1 } $\cup$ SIZE $\in$ {0,..,3} $\cup$ KIND $\in$ {0,..,3} $\cup$ SIGN $\in$ {0,1} ;

S = delay_time_AL $\in$ $\boldsymbol{R}_0^+$; **Y** = RES $\in$ {0,..., 2^{32}–1 }.

These models are used to align data read/written during the Load/Store operations.

ALU = < X, S, Y, δ_{int}, δ_{ext}, λ, D >

X = OPA, OPB ∈ {0,..., 2^{32}–1} ∪ FCOD ∈ {0,..., 2^4–1} ∪ CIN ∈ {0,1};

Y = RES ∈ {0,...,2^{32}–1 } ∪ CARRY ∈ {0,1} ∪ ZERO ∈ {0,1} ∪ NEGAT ∈ {0,1} ∪ OVFLW ∈ {0,1}.

S = delay_time_ALU ∈ R_0^+;

This model represents the behavior of the integer Arithmetic-Logic Unit. It is capable of executing the following operations: add, sub, addx, subx (add/sub with carry), and, or, xor, andn, orn, xnor (negated and, or, xor).

BOOLEAN GATE = < X, S, Y, δ_{int}, δ_{ext}, λ, D >

X = OP1, OP2 ∈ {0,1}; **S** = delay_time_BG ∈ $\boldsymbol{R}_0^+$; **Y** = RES ∈ {0,1}.

This group of models were included to provide the behavior of the most used Boolean gates: AND, OR, NOT and XOR. They receive binary inputs, producing a result according to the desired operation.

BUS = < X, S, Y, δ_{int}, δ_{ext}, λ, D >

X = **Y** = DATA, ADDRESS ∈ {0,..., 2^{32}–1} ∪ BSEL ∈ {0,..., 2^4–1} ∪ IRQ ∈ {0,..., 2^{15}–1} ∪ CLOCK, AS, RD/WR, DTACK, ERR, RESET, BUSY ∈ {0,1};

S = delay_time_Bus ∈ $\boldsymbol{R}_0^+$.

The bus interprets each of the input signals, providing outputs related with them. If a device which received a 1 in the BGRANTin port needs to write data in the Memory, it writes a 0 in the BGRANTout port (no smaller priority device is able to use the bus). Then, the device starts a bus cycle turning on the *BUSY* signal. The device writes the address to be accessed in the *ADDRESS* lines, and the data to be written in *DATA*. After, the Byte Select Mask *BSEL* to define which byte in the word is used. Finally, it turns on the RD/WRout and AS lines to tell a Write operation was issued. When the memory receives the AS signal, it executes a memory cycle that finishes when the DTACKout line is turned on. The device that issued the write operation receives this signal in its DTACKin line. When the cycle has finished, if BGRANTin is still in 1, the device is able to transfer new data. Otherwise, it turns off the *BUSY* line, allowing a new bus operation by other device.

CCLOGIC = < X, S, Y, δ_{int}, δ_{ext}, λ, D >

X = CARRY, ZERO, NEGAT, OVFLW ∈ {0,1} ∪ COND ∈ {0,..., 2^4–1};

S = delay_time_CC ∈ $\boldsymbol{R}_0^+$; **Y** = RES ∈ {0,1}**.**

This model is used in conditional jumps to decide if a branch must be executed.

CLOCK = < X, S, Y, δ_{int}, δ_{ext}, λ, D >

X = ∅; **S** = period ∈ $\boldsymbol{R}_0^+$; **Y** = RES ∈ {0,1}.

It represents the CPU clock, which period can be configured.

CWPLOGIC = < X, S, Y, δ_{int}, δ_{ext}, λ, D >

X = CWP ∈ {0,..., 2^4–1} ∪ SEL ∈ {0,..., 2^4–1}; **S** = delay_time_CWP ∈ $\boldsymbol{R}_0^+$;

Y = GSEL ∈ {0,..., 2^3–1} ∪ RSEL ∈ {0,..., 2^9–1} ∪ R/G ∈ {0,1}.

This model is used to decide if access to the global registers or the register window is required. It returns the kind of register (Register window/Global) and its number.

INC4 = < X, S, Y, δ_{int}, δ_{ext}, λ, D >

X = OP ∈ {0,..., 2^{32}–1}; **S** = delay_time_INC ∈ $\boldsymbol{R_0}^+$; **Y =** RES ∈ {0,...,2^{32}–1 }.

This model updates the nPC.

IRQLOGIC = < X, S, Y, δ_{int}, δ_{ext}, λ, D >

X = IRQ1,..., IRQ15, PIL0, ..., PIL3 ∈ {0,1} ;

S = int_latency ∈ R_0^+; **Y** = TF {0,1} ∪ TT ∈ {0,..., 2^8–1 }.

This model manages the actions that take place when an interrupt is received. The PIL (Processor Interrupt Level) lines mask the interrupts. If one or more IRQs whose numbers are greater than the PIL are received, the interrupt must be serviced. Then, we see which one has the higher priority, and the TF (Trap Found) bit is turned on. The TT (Trap Type) register is loaded according to the highest level interrupt.

LATCH = < X, S, Y, δ_{int}, δ_{ext}, λ, D >

X = IN ∈ {0,..., 2^{32}–1} ∪ EIN, CLEAR ∈ {0,1};

S = delay_time_LA ∈ $\boldsymbol{R_0}^+$;

Y = OUT ∈ {0,..., 2^{32}–1}.

This model represents a processor register, implemented as a d-latch. The EIN line enable inputs, and the CLEAR line resets the register to zero.

MUL/DIV = < X, S, Y, δ_{int}, δ_{ext}, λ, D >

X = OPA, OPB, YIN ∈ {0,..., 2^{32}–1} ∪ FCOD ∈ {0,..., 2^2–1};

S = delay_time_MUL ∈ $\boldsymbol{R_0}^+$;

Y = RES, YOUT ∈ {0,...,2^{32}–1 } ∪ ZERO, NEGAT, OVFLW ∈ {0,1}.

This model is in charge of making multiplication and divisions, turning on the condition bits.

MUX/ MUX4 = < X, S, Y, δ_{int}, δ_{ext}, λ, D >

X = OPA, OPB, OPC, OPD ∈ {0,..., 2^{32}–1} ∪ SELA, SELB, SELC, SELD ∈ {0,1};

S = delay_time_MUX ∈ $\boldsymbol{R_0}^+$;

Y = OUT ∈ {0,...,2^{32}–1 }.

These models represent 2 or 4 input multiplexers. To choose them, we receive the 4-bit *select* signal whose bit turned on marks which input will be sent through the output.

REGBLOCK = < X, S, Y, δ_{int}, δ_{ext}, λ, D >

X = ASEL, BSEL, CSEL ∈ {0,..., 2^9–1} ∪ CEN, RESET ∈ {0,1} ∪ CIN ∈ {0,...,2^{32}–1 };

S = delay_time_RBL ∈ R_0^+;

Y = AOUT, BOUT ∈ {0,...,2^{32}–1 } ∪ ZERO, NEGAT, OVFLW ∈ {0,1}.

This model is in charge of managing the Register Window.

SHIFTER = < X, S, Y, δ_{int}, δ_{ext}, λ, D >

X = OPA, OPB ∈ {0,..., 2^{32}–1} ∪ FCOD ∈ {0,1};

S = delay_time_SH ∈ R_0^+;

Y = RES ∈ {0,...,2^{32}–1 }.

This model is in charge of implementing a shifter.

SIGNEXT13/SIGNEXT22 =< X, S, Y, δ_{int}, δ_{ext}, λ, D >

X = OP ∈ {0,..., 2^{13}–1} ∪ {0,..., 2^{22}–1};

S = delay_time_SE ∈ $\boldsymbol{R}_0^+$;

Y = RES ∈ {0,...,2^{32}–1 }.

These models extend the sign of an operand of 13 or 22 bits to 32 bits.

TRAPLOGIC = < X, S, Y, δ_{int}, δ_{ext}, λ, D >

X = TRAPS ∈ {0,..., 2^{17}–1} ∪ TRAP_NUMBER ∈ {0,..., 2^6–1};

S = delay_time_TL ∈ $\boldsymbol{R}_0^+$;

Y = TRAP_FOUND ∈ {0, 1} ∪ TRAP_INSTRUCTION ∈ {0,..., 2^{31}–1} ∪ TRAP_TYPE ∈ {0,..., 2^8–1};

This component defines which trap must be serviced, based on a priority system. One of the input lines defines a non-masked trap. Other 7 bits are used to receive the number of a trap that can be masked. The model returns a bit telling if the trap must be serviced, and 8 bits telling the *trap type*. The following table shows the kind and priorities for each trap available:

Line	Description	Priority	Trap Type
INST_ACC_EXCEP	Instruction access exception	5	0x01
ILLEG_INST	Illegal instruction	7	0x02
PRIV_INST	Privileged instruction	6	0x03
WIN_OVER	Window overflow	9	0x05
WIN_UNDER	Window underflow	9	0x06
ADDR_NOT_ALIGN	Address not aligned	10	0x07
DATA_ACC_EXCEP	Data access exception	13	0x09
INST_ACC_ERR	Instruction access error	3	0x21
DATA_ACC_ERR	Data access error	12	0x29
DIV_ZERO	Division by zero	15	0x2A
DATA_ST_ERR	Data store error	2	0x2B

Table 3. Available traps

According with this table, the model analyzes which is the higher priority trap to be serviced. After a delay, it sends the corresponding index through the output ports.

3.4. Control Unit

The Control Unit is in charge of driving the execution flow of the processor. As explained earlier, this model uses several input/output lines. According with the input received, it issues different outputs, acti-

vating the different circuits that were defined previously. Here we show part of its behavior. The specification of the input/output sets is not included, because of its size (details can be found in [49]).

```
Model &UC::externalFunction( const ExternalMessage &msg ) {

  if( msg.port() == CLCK ) {

    if( ! waitfmc ) {
        cycle = (cycle + 1) % numcycles;
        this->holdIn( active, 0 );
    }
        else this->passivate();
  } else if( msg.port() == DTACK ) {
        if( msg.value() == 1 ) waitfmc = 0;
        this->passivate();
        } else if( msg.port() == CCLOGIC ) {
                cclogic = bit( msg.value() );
                this->passivate();
                } else
                 {      string portName;
                        int portNum;

                        nameNum( msg.port().name(), portName, portNum );
                        if( portName == "ir" ) ir[portNum] = bit( msg.value() );
                        else psr[portNum] = bit( msg.value() );
                        this->passivate();
                 }
  return *this;
}

Model &UC::internalFunction( const InternalMessage & )  {
  if( as.val ) waitfmc = 1;
  this->passivate();
  return *this;
}

Model &UC::outputFunction( const InternalMessage &msg )  {
   ...

  // See if the c_en line must be activated
  c_en.val = cycle == c_W && !isBranch( ir ) && !isStore( ir ) && !isWr( ir );

  // Read the Instruction Register and decode the instruction
  ...

  if( isArit( ir ) ) {
    copyBits( ir+19, alu_fcod, 4 );
    enable_mul.val =  isMulDiv( ir );
    enable_alu.val =  isAlu( ir );
    enable_shft.val =  isShft( ir );
  } else {
    enable_mul.val = 0;
    enable_alu.val = 1;
    enable_shft.val = 0;
    if( isWr( ir ) )
      toBits( ALU_XOR, alu_fcod, 4 );
    else
      toBits( ALU_ADD, alu_fcod, 4 );
  }

  ...
  // See branches
  if( isJmp( ir ) )
    incdec_fcod.val = INCDEC_INC;
  else
    incdec_fcod.val = ir[19];

    ...
   // Transmit the outputs

  for( int i = 0; i < numsports; i++ )
    if( needSend( *sportbits[i] ) )
      this->sendOutput( msg.time(), *sOUT[i], sportbits[i]->val );
  for( int i = 0; i < nummports; i++ )
    for( int j = 0; j < mportsizes[ i ]; j++ )
      if( needSend( mportbits[i][j] ) )
        this->sendOutput( msg.time(), *mOUT[i][j], mportbits[i][j].val );
  return *this;
}
```

Figure 10. Control Unit: transition functions.

As we can see, this model is activated by the occurrence of a clock tick. In this case, we check if the Control Unit is waiting a result coming from the memory (*waitfmc*). In that case, we have nothing to do and the model passivates. Otherwise, we register that a clock tick has finished. Other external inputs correspond to the signal DTACK coming from the memory or the CCLOGIC (that is, an input arriving from a register). We also recognize inputs for the Instruction Register (to store a new instruction to execute) or to the PSR (to update the condition codes). The internal transition function records that when the Address Strobe is up, we are waiting the end of a memory transfer. The main tasks of the control unit are executed by the output function. As we can see in the description, the present input values are queried. Depending on the number of clock tick in the instruction cycle, different output lines are activated.

4. THE DIGITAL LOGIC LEVEL

The abstraction level of several models was further detailed, letting the students to analyze the digital logic level of the circuits. In the previous stage, the behavior of these circuits was defined by using atomic models. In this case, some of these models were built using atomic models representing the basic Boolean gates (AND, OR, NOT, XOR). These models (described in the previous section) were used as components that were integrated using digital logic. A coupled model representing the complete circuit replaced the old atomic ones. These modifications, also done in course assignments, show the extensibility and modifiability of Alfa-1. Two of the models implemented this way will be explained following.

4.1. CMP model

The *CMP* is a part of the Address Unit that detects addresses falling out of the program boundaries. The model receives two inputs (through the lines *OPA* and *OPB* that are connected to the BASE and LIMIT registers). As a result it returns the signal *EQ* if both values are equal, or *LW* if A is lower than B.

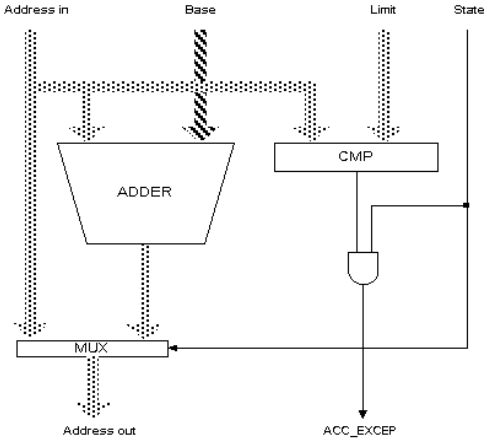

Figure 11. Sketch of the Address Unit.

The model is composed of several one-bit comparators, and coupling n of them generates n-bit comparators. The following figure shows the basic components of this building block:

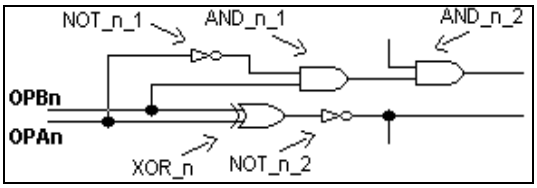

Figure 12. One-bit comparator [50].

This model is formally described by:

$$CM = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, select >$$

X = {OPAn, OPBn / OPAn, OPBn ∈ {0,1} };

Y = {EQ, LW / EQ, LW ∈ {0,1} };

D = {NOT_n_1, NOT_n_2, XOR_n, AND_n_1, AND_n_2 }; where each is an atomic defining the corresponding building block, presented previously in section 3.3.;

$\mathbf{I}_{\mathbf{NOT\_n\_1}}$ = { AND_n_1 }; $\mathbf{I}_{\mathbf{XOR\_n}}$ = {NOT_n_2 };

$\mathbf{I}_{\mathbf{NOT\_n\_2}}$ = { Self }; $\mathbf{I}_{\mathbf{AND\_n\_2}}$ = { Self };

$\mathbf{I}_{\mathbf{AND\_n\_1}}$ = { AND_n_2 }; $\mathbf{I}_{\mathbf{self}}$ = { Self, NOT_n_1, AND_n_1, XOR_n}; and

$\mathbf{Z}_{\mathbf{ij}}$ is built using I, as described earlier, and

Select = D.

The definition of this coupled model using CD++ is presented in the following figure:

```
[top]
components : NOT_n_1@NOT NOT_n_2@NOT XOR_n@XOR AND_n_1@AND AND_n_2@AND
in : OPAn OPBn
out : LW EQ

Link : OPAn@top in@NOT_n_1
Link : OPBn@top ina@XOR_n
Link : OPAn@top inb@XOR_n
Link : OPBn@top inb@AND_n_1
Link : out@NOT_n_1 ina@AND_n_1
Link : out@XOR_n in@NOT_n_2
Link : out@AND_n_2 EQ@top
Link : out@NOT_n_2 LW@top
```

Figure 13. CMP coupled model [50].

First, we define the components of the coupled model (corresponding to the D set). Then, the input/output ports are included (which are related with the X/Y sets defined earlier). Finally, the links show the model influencees (which define the translation function). The *select* function is implicitly defined by the order of definition for the model components.

4.2. Chip Selector

The Chip Selector (*CS*) circuit is devoted to determine if an address is between two others. The model receives a 32-bit address, an Address Strobe (AS), and it returns a Boolean value telling if the address is between the boundaries.

The MASK models provide two 32-bit sets (MAX Mask, MIN Mask) containing the boundaries of the address to be compared. These models, defined originally as latches, were redefined using Boolean gates. The

input address for the chip selector is checked using two comparators, instances of the model defined in the previous section.

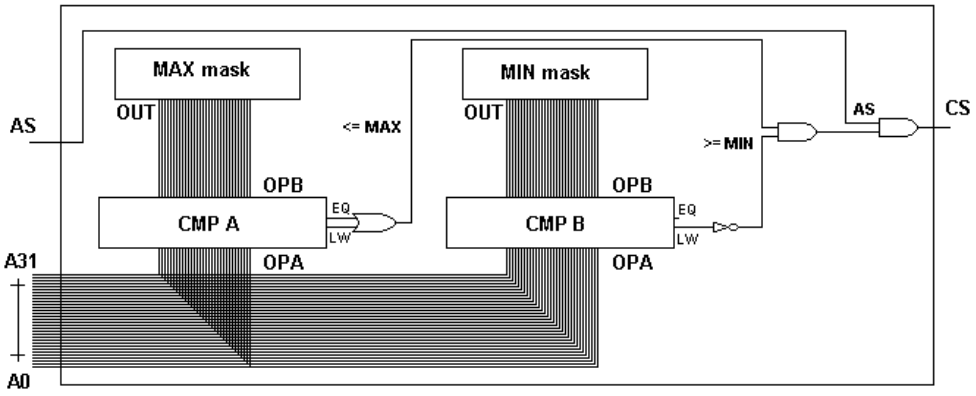

Figure 14. Sketch of the Chip Selector [50].

The result obtained is transmitted through the ports LW and EQ for each of the comparators. Both outputs are ORed for the first register (as we are interested to see if CMP A ≤ MAX). After, the LW output of the second register is inverted (as we are interested to see if CMP B ≥ MIN). If the circuit is enabled, the result obtained is transmitted.

```
[top]
components: MASMAX@MAS MASMIN@MAS CMPA@CMP CMPB@CMP and1@AND and2@AND or@OR not@NOT
in : A31 A30 A29 A28 A27 A26 A25 A24 A23 A22 A21 A20 ... A4 A3 A2 A1 A0 AS
out : CS

Link: A31@top OPA31@CMPA A31@top OPA31@CMPB   Link: A30@top OPA30@CMPA A30@top OPA30@CMPB
...
Link: A1@top OPA1@CMPA A1@top OPA1@CMPB        Link: A0@top OPA0@CMPA A0@top OPA0@CMPB

Link: out31@MASMAX OPB31@CMPA out31@MASMIN OPB31@CMPB
Link: out30@MASMAX OPB30@CMPA out30@MASMIN OPB30@CMPB
...
Link: out0@MASMAX OPB0@CMPA out0@MASMIN OPB0@CMPB
Link: AS ina@and2
Link: eq@CMPA ina@or  lw@CMPA  inb@or
Link: lw@CMPB in@not
Link: out@or ina@and1  out@not inb@and1
Link: out@and1 inb@and2
Link: out@and2 CS@top
```

Figure 15. CS coupled model [50].

5. SIMULATION RESULTS

The present section shows the results obtained when some of the models previously presented are simulated. In the first case, we show the results of a value of 20 incremented by the *INC/DEC* model. The figure shows the model inputs with their timestamps and the output values obtained.

The first step consists of giving an initial value to the circuit (in zero by default). The first event (OP0 = 1 at 00:00:00:00) generates an output only when the model phase changes. As the preparation time for the circuit is 5 time units, this occurs at 00:00:05:000. The second input does not generate changes in the model and no output is issued. In simulated time 10, a new input is inserted through the port OP2. As this value changed, an output is generated at simulated time 15. The following 2 inputs are not registered because the

circuit keeps its present state. The last one increments the value in the register by inserting the value through the FCOD port. The incremented value can be seen 5 time units after.

INPUT	OUTPUT
00:00:00:00 OP0 1 00:00:05:00 OP1 0 00:00:10:00 OP2 1 00:00:15:00 OP3 0 00:00:20:00 OP4 0 00:00:25:00 FCOD 1	00:00:05:000 res0 1 00:00:05:000 res1 0 00:00:05:000 res2 0 00:00:05:000 res3 0 00:00:05:000 res4 0 00:00:15:000 res0 1 00:00:15:000 res1 0 00:00:15:000 res2 1 00:00:15:000 res3 0 00:00:15:000 res4 0 00:00:30:000 res0 1 00:00:30:000 res1 0 00:00:30:000 res2 1 00:00:30:000 res3 0 00:00:30:000 res4 1

Figure 16. Inputs and Outputs for the INC/DEC model.

The following example shows the execution of the *RegGlob* model under different inputs. At the instant 0, the C enable line is activated, allowing write operations in the register. In this case, the register 4 is selected (*csel2=1*, *csel1=0* and *csel0=0*), and the number 0xFFFFFFFF is used as input (*cin0 = ... = cin31=1*). After, in 00:00:01:00, the register 2 is selected (*csel2=0 and csel1=1*), and the number 0x55555555 is input (*cin0=cin2=cin4...=cin30=1*, and *cin1= cin3 = cin5 =... =cin30=1*). The first value is stored in the register 4, and the second in the register 2.

INPUT	OUTPUT
00:00:00:00 cen 1 00:00:00:00 csel2 1 00:00:00:00 cin0 1 ... 00:00:00:00 cin31 1	
00:00:01:00 csel2 0 00:00:01:00 csel1 1 00:00:01:00 cin1 0 00:00:01:00 cin3 0 00:00:01:00 cin5 0 ... 00:00:01:00 cin29 0 00:00:01:00 cin31 0	
00:00:02:00 cen 0 00:00:02:00 asel2 1 00:00:02:00 bsel1 1	00:00:02:010 aout0 1 ... 00:00:02:010 aout31 1 00:00:02:010 bout0 1 00:00:02:010 bout2 1 00:00:02:010 bout4 1 ... 00:00:02:010 bout30 1
00:00:04:00 reset 1	
00:00:05:00 asel2 1 00:00:05:00 asel1 0	00:00:05:010 aout0 0 00:00:05:010 aout2 0 ... 00:00:05:010 aout28 0 00:00:05:010 aout30 0

Figure 17. Inputs/Outputs of RegGlob [50].

At 00:00:02:00, C Enable is deactivated. Therefore, the following operations are devoted to read registers. We see that the value in the register 4 is sent through the A output (*asel=2*) and the register 2 is sent through B (*bsel=1*). As a result, the values previously loaded are transmitted (that is, 0xFFFFFFFF in A, and 0x55555555 in B). After, Reset is activated. Now, we try to read the register 4 at 00:00:05:00, and we obtain the value 0x00000000.

The next test corresponds to the *TrapLogic* model. Here we can see the result obtained after turning on all the trap bits. Due to this, we expect obtaining the index of the highest priority trap that is pending. The re-

sult obtained after the delay time corresponds to the highest one of section 3.4: *Data Store Error* (whose code is TT = 0x2B = 00101101, the result we obtained). Also, the Trap Found flag is turned on.

INPUTS	OUTPUTS
00:00:00:001 / inst_acc_excep / 1.000	00:00:00:052 tf 1
00:00:00:002 / illeg_inst / 1.000	00:00:00:052 tt7 1
00:00:00:003 / priv_inst / 1.000	00:00:00:052 tt6 1
00:00:00:004 / win_over / 1.000	00:00:00:052 tt4 1
00:00:00:005 / win_under / 1.000	00:00:00:052 tt2 1
00:00:00:006 / addr_not_align / 1.000	
00:00:00:007 / data_acc_excep / 1.000	
00:00:00:008 / inst_acc_err / 1.000	
00:00:00:009 / data_acc_err / 1.000	
00:00:00:010 / div_zero / 1.000	
00:00:00:011 / data_st_err / 1.000	
00:00:00:012 / trap_inst / 1.000	

Figure 18. Execution results for the TrapLogic model [50].

Finally, we show two execution examples that are part of a complete program. All the examples were executed in a Pentium processor (133 MHz), using the Linux version of CD++. The average performance for this model was one instruction per second. The source code was translated to binary using the GNU MASM assembler Linker. The executable is used as the initial memory image for the simulator. The first part of the following figure shows part of a program written in assembly language. The second part presents the binary code generated, together with the addresses for each instruction or data (one word each).

```
        set 0x12345678, %r1           ! Load the register 1 with 0x12345678
        st   %r1, [dest]              ! Store it in the "dest" variable
        sth  %r1, [dest+4]            ! Store the high half-word
        sth  %r1, [dest+10]
        stb  %r1, [dest+12]           ! Store the last byte
        stb  %r1, [dest+17]
        stb  %r1, [dest+22]
        stb  %r1, [dest+27]
        unimp
dest:   .ascii "                                "


Initial Image

Addr.             Memory Image               Interpretation
...
040     11000010 00100000 00100000 01001000  Store the register 1 in the address 72
044     11000010 00110000 00100000 01001100  Store the high part of reg. 1 in address 76
048     11000010 00110000 00100000 01010010  Store the high part of reg. 1 in address 82
052     11000010 00101000 00100000 01010100  Store the high byte of reg. 1 in address 84
056     11000010 00101000 00100000 01011001  Store the high byte of reg. 1 in address 89
060     11000010 00101000 00100000 01011110  Store the high byte of reg. 1 in address 94
064     11000010 00101000 00100000 01100011  Store the high byte of reg. 1 in address 99
068     00000000 00000000 00000000 00000000  unimp
072     00100000 00100000 00100000 00100000  "dest" variable (20: space character)
076     00100000 00100000 00100000 00100000
...


Final image

Addr.             Memory Image                Values
...
072     00010010 00110100 01010110 01111000  12 34 56 78
076     01010110 01111000 00100000 00100000  56 78 20 20 (20 = space)
080     00100000 00100000 01010110 01111000  20 20 56 78
084     01111000 00100000 00100000 00100000  78 20 20 20
088     00100000 01111000 00100000 00100000  20 78 20 20
092     00100000 00100000 01111000 00100000  20 20 78 20
096     00100000 00100000 00100000 01111000  20 20 20 78
```

Figure 19. Storing a value in memory.

As we can see, this piece of code copies parts of the number 0x12345678 to certain memory addresses. We show the translation of the binary codes based on the specification of the instruction set of the SPARC

processor. Finally, we show the memory image after the program execution. As we can see, the values stored in memory follow the instructions defined by the executable code.

The following example shows the execution of part of another program. As we can see, the goal is to place a 1 in a given address, and then shift this value to the left, storing the result in the following address. The cycle is repeated 12 times.

```
        set 1, %r1                              ! Load the register 1 with the value 1
cycle:  sll %r1, %r2, %r3                       ! Shift the value the number of times in r2
        stb %r3, [%r2+dest]                     ! Store the result in the variable dest + r2

        subcc %r2, 12, %r0                      ! Repeat the cycle 12 times
        bne cycle
        inc 1, %r2 !Delay slot

        unimp

dest:   .ascii "                          "
        .ascii "                          "
        .ascii "                          "
        .ascii "                          "


Initial Image
Addr.             Memory Image                Interpretation
...
032     10000010 00010000 00100000 00000001  set <1>, 1
036     10000111 00101000 01000000 00000010  Take the register 1, shift and store in R3
040     11000110 00101000 10100000 00111100  Store in address 60 1 byte
044     10000000 10100000 10100000 00001100  substract 12 to R0
048     00010010 10111111 11111111 11111101  Relative jump to address –2 words (40)
052     10000100 00000000 10100000 00000001  increment 1
056     00000000 00000000 00000000 00000000  unimp
060     00100000 00100000 00100000 00100000  Destination variable
064     00100000 00100000 00100000 00100000
068     00100000 00100000 00100000 00100000
...
Final image
...
060     00000001 00000010 00000100 00001000  A value 0x01 shifted 12 times
064     00010000 00100000 01000000 10000000
068     00000000 00000000 00000000 00000000
...
```

Figure 20. Shifting and storing results in memory.

Once the basic behavior of the simulated computer was verified, a thorough integration test was attacked. As explained earlier, each circuit was defined together with a set of input/output values that were encapsulated in an experimental framework. Once each of the models was tested, each operation in the instruction set was checked. The procedure was developed using the verification facilities of DEVS, defining 17100 test cases. The mechanism consisted in creating an experimental framework, which executed an instruction in the instruction set. The execution result was stored in memory, and a memory dump was executed, obtaining the memory state after the execution. This value is checked against the value obtained when the same program is executed in the real architecture, which is included in the testing experimental framework. This procedure allowed us to find some errors derived of the coupled model. For instance, we could see that the division instruction was not working properly. The generated test included the following sentences:

```
        set 274543375,  %r24     ! stores a value in register 24
        set 13908050 , %r22      ! a second value is stored in the register 22
        udiv %r24, %r22 , %r10   ! both values are divided and stored in r10
        st %r10 , [dest]         ! The result is stored in memory
        unimp

.align 4
value:     .ascii "VALUE:"
dest:       .word  FFFFFFFF  !  Result of Test 100
```

Figure 21. Testing routine for the UDIV instruction.

When this example was executed, the testing coupled model found an error:

```
Field   Type          Expected              Found                DIF
=====   ====          ========              ==========           ===
1       int32         19                    1                    ***
```

In this case, the destination should have stored the value 19 (274543375 divided by 13908050). Instead, we have found the value 1, allowing us to see that one of the instructions had an unexpected behavior. In this way we could find errors in some of the instructions that could be fixed. We also found errors in some addition instructions, and in conditional jumps with prediction.

Finally, we show part of the execution of the simulator for the example presented in Figure 20. We show a log file including the messages interchanged between modules. As in other DEVS frameworks, there are four kinds of messages: ***** (used to signal a state change due to an internal event), **X** (used when an external event arrives), **Y** (the model's output) and **done** (indicating that a model has finished with its task). The **I** messages initialize the corresponding models. For each message we show its type, timestamp, value, origin/destination, and the port used for the transmission.

```
Message I / 00:00:00:000 / Root(00) to top(01)      // Initialize the higher level
Message I / 00:00:00:000 / top(01) to mem(02)       // components: memory, bus, CS, etc.
Message I / 00:00:00:000 / top(01) to bus(03)
Message I / 00:00:00:000 / top(01) to csmem(04)
Message I / 00:00:00:000 / top(01) to cpu(05)
Message I / 00:00:00:000 / top(01) to c1(64)
Message I / 00:00:00:000 / top(01) to dpc(65)
Message D / 00:00:00:000 / mem(02) / ... to top(01)  // The models reply the next
Message D / 00:00:00:000 / bus(03) / ... to top(01)  // scheduled event
Message D / 00:00:00:000 / csmem(04) / ... to top(01)
Message I / 00:00:00:000 / cpu(05) to ir(06)        // The CPU initializes the components
Message I / 00:00:00:000 / cpu(05) to pc_add(07)
Message I / 00:00:00:000 / cpu(05) to pc_mux(08)
...
Message * / 00:00:00:000 / Root(00) to top(01)
Message * / 00:00:00:000 / top(01) to cpu(05)
Message * / 00:00:00:000 / cpu(05) to npc(10)       // Take the nPC
Message Y / 00:00:00:000 / npc(10) / out2 / 1.000 to cpu(05)
Message Y / 00:00:00:000 / npc(10) / out5 / 1.000 to cpu(05)
Message D / 00:00:00:000 / npc(10) / ... to cpu(05)
Message X / 00:00:00:000 / cpu(05) / in2 / 1.000 to pc_latch(11)   // Send to pc-inc
Message X / 00:00:00:000 / cpu(05) / op2 / 1.000 to pc_inc(13)     // to increment
Message X / 00:00:00:000 / cpu(05) / in5 / 1.000 to pc_latch(11)   // the value
Message X / 00:00:00:000 / cpu(05) / op5 / 1.000 to pc_inc(13)
Message D / 00:00:00:000 / pc_latch(11) / 00:00:10:000 to cpu(05)  // Schedule the
Message D / 00:00:00:000 / pc_inc(13) / 00:00:10:000 to cpu(05)    // activation of the
Message D / 00:00:00:000 / pc_latch(11) / 00:00:10:000 to cpu(05)  // pc-inc model
Message D / 00:00:00:000 / pc_inc(13) / 00:00:10:000 to cpu(05)
Message D / 00:00:00:000 / cpu(05) / 00:00:00:000 to top(01)
Message D / 00:00:00:000 / top(01) / 00:00:00:000 to Root(00)
Message * / 00:00:00:000 / Root(00) to top(01)
Message * / 00:00:00:000 / top(01) to cpu(05)
Message * / 00:00:00:000 / cpu(05) to pc(12)
Message Y / 00:00:00:000 / pc(12) / out5 / 1.000 to cpu(05)       // Initial address
Message D / 00:00:00:000 / pc(12) / ... to cpu(05)                 // 010000 = 32
...
Message * / 00:00:00:000 / Root(00) to top(01)
Message * / 00:00:00:000 / top(01) to cpu(05)
Message * / 00:00:00:000 / cpu(05) to clock(45)                    // Clock tick
Message Y / 00:00:00:000 / clock(45) / clck / 1.000 to cpu(05)
Message D / 00:00:00:000 / clock(45) / 00:01:00:000 to cpu(05)
Message X / 00:00:00:000 / cpu(05) / clck / 1.000 to cu(43)
Message D / 00:00:00:000 / cu(43) / 00:00:00:000 to cpu(05)
Message D / 00:00:00:000 / cpu(05) / 00:00:00:000 to top(01)
Message D / 00:00:00:000 / top(01) / 00:00:00:000 to Root(00)
Message * / 00:00:00:000 / Root(00) to top(01)
Message * / 00:00:00:000 / top(01) to cpu(05)       // Arrival to the CU
Message * / 00:00:00:000 / cpu(05) to cu(43)        // And activation of the components
Message Y / 00:00:00:000 / cu(43) / a_mux_reg / 1.000 to cpu(05)
Message Y / 00:00:00:000 / cu(43) / b_mux_reg / 1.000 to cpu(05)
Message Y / 00:00:00:000 / cu(43) / enable_alu / 1.000 to cpu(05)
Message Y / 00:00:00:000 / cu(43) / addr_mux / 1.000 to cpu(05)
Message Y / 00:00:00:000 / cu(43) / ir_latch_en / 1.000 to cpu(05)
Message Y / 00:00:00:000 / cu(43) / as / 1.000 to cpu(05)
Message Y / 00:00:00:000 / cu(43) / rd_wr / 1.000 to cpu(05)
```

```
Message Y / 00:00:00:000 / cu(43) / busy / 1.000 to cpu(05)
Message Y / 00:00:00:000 / cu(43) / c_mux2 / 1.000 to cpu(05)
Message Y / 00:00:00:000 / cu(43) / pc_mux0 / 1.000 to cpu(05)
Message D / 00:00:00:000 / cu(43) / ... to cpu(05)
...
Message * / 00:00:10:000 / Root(00) to top(01)
Message * / 00:00:10:000 / top(01) to cpu(05)
Message * / 00:00:10:000 / cpu(05) to pc_latch(11)
Message D / 00:00:10:000 / pc_latch(11) / ... to cpu(05)
Message D / 00:00:10:000 / cpu(05) / 00:00:00:000 to top(01)
Message D / 00:00:10:000 / top(01) / 00:00:00:000 to Root(00)
Message * / 00:00:10:000 / Root(00) to top(01)
Message * / 00:00:10:000 / top(01) to cpu(05)
Message * / 00:00:10:000 / cpu(05) to pc_inc(13)            // Update the nPC
Message Y / 00:00:10:000 / pc_inc(13) / res3 / 1.000 to cpu(05)
Message Y / 00:00:10:000 / pc_inc(13) / res5 / 1.000 to cpu(05)
Message D / 00:00:10:000 / pc_inc(13) / ... to cpu(05)
...
Message * / 00:00:20:001 / Root(00) to top(01)
Message * / 00:00:20:001 / top(01) to mem(02)       // Memory returns the first instr.
Message Y / 00:00:20:001 / mem(02) / dtack / 1.000 to top(01)
Message Y / 00:00:20:001 / mem(02) / out_data0 / 1.000 to top(01)
Message Y / 00:00:20:001 / mem(02) / out_data13 / 1.000 to top(01)
Message Y / 00:00:20:001 / mem(02) / out_data20 / 1.000 to top(01)
Message Y / 00:00:20:001 / mem(02) / out_data25 / 1.000 to top(01)
Message Y / 00:00:20:001 / mem(02) / out_data31 / 1.000 to top(01)
Message D / 00:00:20:001 / mem(02) / ... to top(01)
Message X / 00:00:20:001 / top(01) / in_dtack / 1.000 to bus(03)
Message X / 00:00:20:001 / top(01) / in_data0 / 1.000 to cpu(05)
Message X / 00:00:20:001 / top(01) / in_data13 / 1.000 to cpu(05)
Message X / 00:00:20:001 / top(01) / in_data20 / 1.000 to cpu(05)
Message X / 00:00:20:001 / top(01) / in_data25 / 1.000 to cpu(05)
Message X / 00:00:20:001 / top(01) / in_data31 / 1.000 to cpu(05)
Message D / 00:00:20:001 / bus(03) / 00:00:00:001 to top(01)
...
```

Figure 22. Log file of a simple routine.

The execution cycle starts by initializing the higher level models (memory, CPU, etc.). The message arrived to the CPU model is sent to its lower level components: Instruction Register, PC Adder, PC multiplexer, Control Unit, etc.

When the initialization cycle finishes, the imminent model is executed. In this case, the *nPC* model is activated, transmitting the address of the next instruction. As we can see, the 2nd and 5th bits are returned with a 1 value. That means that the nPC value is 100100 = 36 (as we see in figure 20, the program starts in the address 32). The value is sent to the *pc-inc* model, in charge of adding 4 to this register. The update is finished in 10:000, as the activation time of this model was scheduled using the circuit delay. At that moment, a 4 value is added to the nPC, and we obtain the 3rd and 5th bits in 1 (*res3* and *res5*), that is, 101000 = 40, the next PC. After, the PC is activated and the value 010000 (that is, 32) is obtained. This is the initial address of the program. The following event is the arrival of a clock tick, sent to the processor. The CPU schedules the next tick (in 1:00:000 time units) and transmits the signal to the Control Unit, which activates several components: *a-mux, ALU, Addr-mux, IR*, etc.

We finally see, in the simulated time 20:000, that the memory has returned the first instruction (compare the results with the bit configuration in the address 32). The instruction is sent to the CPU to be stored in the Instruction Register and to follow with the execution. The rest of the instruction cycle is completed in the same way.

We are able to follow the execution flow of any program by analyzing this log file. To simplify the analysis of results, we built a set of scripts using Tk, that lets the students to choose which components should be

considered. In this way, behavior of each of the subcomponents can be followed more easily, and the students can analyze the behavior of the desired subsystem in detail.

6. CONCLUSION

We have presented the use of DEVS to simulate a simple computer. The models were based on the architecture of the SPARC processor, which includes features not existing in simpler CPUs. The tools can be used in Computer Organization courses to analyze and understand the basic behavior of the different levels of a computer system. The interaction between levels can be studied, and experimental evaluation of the system can be done.

The use of DEVS allowed us to have reusable models (in this case, Boolean gates, comparators, multiplexers, latches, etc.). DEVS also allowed us to provide reusable code for different configurations. We provided different machines, one running the digital logic level and the other the instruction set, with different performance in each case, depending on the educational needs. The concept of internal transition functions can be used to improve the definition of the timing properties of each component, letting to define complex synchronization mechanisms. Nevertheless, in this case, most timing delays were represented as simple input/output relations.

We have met all the goals proposed. Alfa-1 is public domain, and has been developed using CD++ (which is also public domain, and it was built using GNU C++). Therefore, the toolkit is available for its use in most existing Computer Organization courses. We described several levels of the architecture (from the Digital Logic level up to the Instruction Set). The assembly language level was also attacked using public domain assemblers that generated executable code that could run in Alfa-1. We have easily extended the components (for instance, including a cache memory that was not included in the first versions). We also have modified existing components (implementing, for instance, Digital Logic versions of some of the circuits). Thorough testing could be done using an approach based on the construction of experimental frameworks associated with testing functions. An experimental framework was also built for the final integrated model.

The most important achievements were related with our educational goals. The whole project was designed with detail as an assignment in a 3rd year Discrete Event Simulation course. The models were formally specified, and the specifications were used by students in a Computer Organization course to build the final version of the architecture. These students had only taken previous prerequisite courses in programming. With only this knowledge the students were able to build all the components here presented. Final integration was planned by a group of undergraduate Teaching Assistants (which also developed the Control Unit and a coupled model representing the whole architecture showed in the Figure 1). Individual and integration testing was also done by 2nd year students. Several of the modifications showed here were developed as course assignments. These facts show the feasibility of the approach from a pedagogical point of view. Upper level courses reported higher success rates and detailed knowledge of the subjects after using Alfa-1.

The tools can be obtained at "http://www.sce.carleton.ca/wainer/usenix". Different experiences can be attacked using this toolkit. In the assembly language level, the students can use existing assemblers to build executables that run in the simulator. A complete analysis of the execution flow at the instruction level can be achieved by tracing the execution in the log file. The students can study the flow of a program and each instruction with detail, starting from the memory image of an executable. The instruction cycle and signal flow in the datapath can be easily inspected. Going deeper, we can see the behavior of those circuits implemented in the digital logic level.

By extending or changing the existing instructions, and implementing the changes in the Control Unit, the students can experience the design of instruction sets. This allows them to have practice in instruction encoding, and to relate instruction definition with the underlying architecture. The students also can include new components (as it was showed with the cache memory example), change existing ones, or implement them using digital logic.

The hierarchical nature of DEVS provides means to go deeper in the hierarchy. For instance, the logical gates could be implemented defining the transistor level (which has not been implemented in this version). We planned to build an Assembler and Linker, but the code generated by those provided by GNU for SPARC plattforms executed straightforward. Nevertheless, the implementation of an assembler and linker are interesting assignments that can be faced to complete the layered view applied in these courses. Also, a debugger for the Alfa-1 architecture could be built, making easier to study the assembly language level.

At present, Alfa-1 is being extended by defining components of the input/output subsystem. Several input/output devices, interfaces and DMA controllers will be simulated. Different transference techniques (polling, interrupts, DMA) will be considered. Likewise, the implementation of different cache management algorithms is being finished. Other tasks faced at present include the definition of a graphical interface to enhance the use of the toolkit. The set of scripts mentioned in section 5 will be used to gather the results of the simulations, and will be used as inputs to be displayed in graphical way. In this way, the study and analysis of the different subsystems will be improved.

7. ACKNOWLEDGEMENTS

I want to thank to the anonymous referees the detailed comments they made to this article. I also thank Prof. Trevor Pearce at SCE, Carleton University, for his help with the final version. Sergio Zlotnik collaborated in the early stage of this project, presented earlier in [51]. The research was partially funded by the Usenix foundation and UBACYT Project JW10. It was developed while Gabriel Wainer was an Assistant Professor at the Computer Sciences Dept. of the Universidad de Buenos Aires in Argentina.

REFERENCES

[1] HENNESY, J.; PATTERSON, D. "Computer Architecture: a quantitative approach". *Morgan Kaufmann, San Francisco*. Second Edition. 1997.
[2] PATTERSON, D. "Computer Organization and Design: the Hardware/Software Interface". *2nd. Edition. University of California, Berkeley*. 1995.
[3] STALLINGS, W. "Computer Organization and Architecture". *Macmillan, New York. 4th. Edition*. 1996.

[4] HEURING, V.; JORDAN, H. "Computer Systems Design and Architecture". *Addison-Wesley*. 1997.
[5] TANENBAUM, A. "Structured Computer Organization", *4th edition, Prentice Hall, New Jersey*, 1999.
[6] BURGER, D.; AUSTIN, T. "The SimpleScalar Tool Set. Version 2.0". *Computer Architecture News*. Vol. 25, (3) pp. 13-25. June 1997.
[7] COE, P.; WILLIAMS, L.; IBBETT, R. "An interactive environment for the teaching of computer architecture". *Proceeding of the Annual Joint Conference Integrating Technology into Computer Science Education*. pp. 33-35. Barcelona, Spain. 1996.
[8] ROSEMBLUM, M.; BUGNION, E.; DEVINE, S., HERROD, S. "Using the SimOS machine simulator to study complex computer systems". *ACM Transactions on Modeling and Computer Simulation*. January 1997.
[9] PEARCE, T. "Notes on p86 Assembly Language and Assembling". *Internal report, Dept. of Systems and Computer Engineering. Carleton University*. http://www.sce.carleton.ca/courses/94201/. 2000.
[10] SHEALY, A.; MALLOY, B.; SYKES, D. "An extensible simulator for the Intel 80x86 processor family". *Proceedings of the 30thAnnual Simulation Symposium*. pp. 157-166. 1997.
[11] MORSIANI, M.; DAVOLI, R. "The MPS Computer System Simulator". *Technical Report UBLCS-99-8, Universitá de Bologna, Italy.* April 1999.
[12] HENNESY, J.; PATTERSON, D. "Appendix A: Assemblers, Linkers and the SPIM simulator. Computer Architecture: a quantitative approach". *Morgan Kaufmann, San Francisco*. Second Edition. 1997.
[13] BABAOGLU, O.; BUSSAN, M.; DRUMMOND, R.; SCHNEIDER, F. "Documentation for the CHIP computer system". *Technical Report TR83-584. Cornell University. Computer Sciences Dept*. December 1983.
[14] BEVILACQUA, R.; GOMEZ, L.; GOMEZ, S. "The PROVIR Virtual Processor". (in Spanish). *M. Sc. Thesis. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires*. 2000.
[15] EL HAJJ, A.; KABALAN, K.; MNEIMNEH, M.; KARABLIEH F. "Microprocessor Simulation and Program Assembling Using Spreadsheets", *Simulation* Vol. 75, No. 2, pp. 82-90, August 2000.
[16] EDMONSON, J.; REILLY, M. "Performance simulation of an ALPHA microprocessor". *IEEE Computer*, May 1998.
[17] I. IKODINOVIC, D. MAGDIC, A. MILENKOVIC, V. MILUTINOVIC. "Limes: A Multiprocessor Simulation Environment for PC Platforms". *Proceedings of Third International Conference on Parallel Processing and Applied Mathematics (PPAM)*, Kazimierz Dolny, Poland, September 14-17, 1999.
[18] BREWER, E.A., DELLAROCAS, C.N., COLBROOK, A. AND WEIHL, W.E. "PROTEUS: A High-Performance Parallel-Architecture Simulator". *Technical Report TR-516, MIT / LCS, Laboratory for Computer Science, Cambridge, MA*, September 1991.
[19] BURNS, M.; GEORGE, A.; WALLACE, B. "Modeling and Simulative Performance Analysis of SMP and Clustered Computer Architectures". *Simulation.* February 2000.
[20] BEDICHEK, R. "Talisman: fast and accurate multicomputer simulation". *Proceedings of SIGMETRICS'95*, Ottawa, Ontario, Canada. May 1995.
[21] SHANMUGAN, K.; FROST, V.; LA RUE, W. "A Block-Oriented Network Simulator (BONeS)". *Simulation.* February 1992.
[22] NGUYEN, A.-T.; MICHAEL, M.; SHARMA, A.; TORRELLAS, J. "The Augmint multiprocessor simulation toolkit for Intel x86 architectures Computer Design". *Proceedings of IEEE International Conference on VLSI in Computers and Processors*, 1996. pp. 486 –490.
[23] KONAS, P., YEW, P. "Improved parallel architectural simulations on shared-memory multiprocessors"; *Proceedings of the 1994 workshop on Parallel and distributed simulation*, 1994, pp. 156 - 159
[24] S. DWARKADAS, J. R. JUMP AND J. B. SINCLAIR. "Execution-driven simulation of multiprocessors address and timing analysis: *ACM Trans. Model. Comput. Simul.* 4, 4 (Oct. 1994), pp. 314 – 338.
[25] BRORSSON, M. "SM-prof: a tool to visualize and find cache coherence performance bottlenecks in multiprocessor programs". *Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pp. 178 – 187, 1995.
[26] HEIN, A.; DAL CIN, M. "Performance and dependability evaluation of scalable massively parallel computer systems with conjoint simulation". *ACM Trans. Model. Comput. Simul.* 8, 4 (Oct. 1998), pp. 333 – 373.
[27] APDUHAN, B.O.; SUEYOSHI, T.; NAMIUCHI, Y.; TEZUKA, T.; ARITA, I. "Experiments of a reconfigurable multiprocessor simulation on a distributed environment". *Proceedings of International Phoenix Conference on Computers and Communications*. Phoenix, AZ. 1992.
[28] TAN, G.S.H.; TEO, Y.M. "Experiences in simulating a declarative multiprocessor". *Proceedings of the 28th Annual Simulation Symposium*, pp. 95 –104. 1995.
[29] ZAGAR, M.; BASCH, D. "Microprocessor Architecture Design with ATLAS." *IEEE Design and Test of Computers*. July 1997.
[30] MATTSSON, S., ELMQVIST, H., BROENINK, J.F. "Modelica: An International effort to design the next generation modeling language" *Journal A, Benelux Quarterly Journal on Automatic Control*, Vol. 38:3, pp. 16-19, September 1997.
[31] BAGRODIA, R. "Designing Efficient Simulations Using Maisie". *Proceedings of the 1991 Winter Simulation Conference*, Dec. 8-11, 1991, Phoenix, AZ, pp. 243-247.
[32] DABNEY, J. HARMAN, T. "Mastering Simulink 4". *Prentice-Hall*. 2001
[33] GAUTHIER, M. "ACSL Reference Manual". *http://www.acslsim.com/*. 1998.
[34] CACI PRODUCTS COMPANY. "MODSIM II, The Language for Object-Oriented Programming", *CACI, La Jolla, California*, 1991.
[35] KIVIAT, P. et al. "SIMSCRIPT: A Simulation Programming Language", CACI, 1973.
[36] FISHWICK, P. "Simulation Model Design and Execution". *Prentice Hall*, 1995.

[37] GHOSH, S. "Hardware Description Languages: Concepts and Principles". An *IEEE Press Original Monograph*, ISBN 0-7803-4744-7, 2000.
[38] THOMAS, D., MOORBY, P. "The Verilog Hardware Description Language". *Kluwer Academic Publishers*, Boston, 1991.
[39] MITSCHELE-THIEL, A. "Systems Engineering with SDL". *JW Wiley*. 2000.
[40] GIAMBIASI, N., ESCUDE, B., GHOSH, S. "GDEVS: A Generalized Discrete Event Specification for Accurate Modeling of Dynamic Systems," *Transactions of the Society for Computer Simulation (SCS) International*, Vol. 17, No. 3, September 2000, pp. 120-134, San Diego, CA.
[41] WAINER, G. "ALFA-0: a simulated computer as an educational tool for Computer Organization". G. Wainer. In *Proceedings of IASTED Applied Modeling and Simulation 1998.* Hawaii, USA.
[42] TROCCOLI, A.; WAINER, G. "CRAPS: an emulator for the SPARC processor" (in Spanish). In *Proceedings of INFOCOM Argentina '98*. Buenos Aires, Argentina.
[43] ISACOVICH, F, MISLEJ, E., WINTERNITZ, F., WAINER, G. "An emulator of the Atari processor" (in Spanish). *Internal Report, Computer Organization course, Computer Sciences Dept., Universidad de Buenos Aires, Argentina* (First Prize at the Students Contest of the 28th Conference on Informatics and Operations Research, Buenos Aires, Argentina). 1999.
[44] ZEIGLER, B.; PRAEHOFER, H.; KIM, T. "Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems". *Academic Press*. 2000.
[45] ZEIGLER, B. "Object-oriented simulation with hierarchical modular models". *Academic Press*, 1990.
[46] WAINER, G.; GIAMBIASI, N. "Application of the Cell-DEVS paradigm for cell spaces modeling and simulation.". To appear in *Simulation.* 2001.
[47] ZEIGLER, B. P. "DEVS Theory of Quantization". *DARPA Contract N6133997K-0007*: ECE Dept., UA, Tucson, AZ. 1998.
[48] RODRIGUEZ, D.; WAINER, G. "New Extensions to the CD++ tool". In *Proceedings of SCS Summer Multiconference on Computer Simulation*. 1999.
[49] DAICZ, S.; TROCCOLI, A.; ZLOTNIK, S.; WAINER, G. "Architectural definition of the ALFA-1 simulated processor" (in Spanish). *Internal report. Departamento de Computación. Universidad de Buenos Aires*. "http://www.dc.uba.ar/people/proyinv/usenix". 1998.
[50] DE SIMONI, L.; ENRIQUE, S.; GLINSKY, E.; PETRONIO, D.; WASSERMANN, D.; WAINER, G. et al. “Definition of components for the ALFA-1 simulated processor”. (in Spanish). *Internal report. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires*. 1998.
[51] DAICZ, S.; TROCCOLI, A.; ZLOTNIK, S.; WAINER, G. "Using the DEVS paradigm to implement a simulated processor". In *Proceedings of 33rd IEEE/SCS Annual Simulation Symposium*. Washington, D.C. U.S.A.