Using the Alfa-1 Simulated Processor for Educational Purposes

GABRIEL A. WAINER
Carleton University
and
SERGIO DAICZ, LUIS F. DE SIMONI, and DEMIAN WASSERMANN
Universidad de Buenos Aires

Alfa-1 is a simulated computer designed for computer organization courses. Alfa-1 and its accompanying toolkit allow students to acquire practical insights into developing hardware by extending existing components. The DEVS formalism is used to model individual components and to integrate them into a hierarchy that describes the detailed behavior of different levels of a computer's architecture. We introduce Alfa-1 and the toolkit, show how to extend existing components, and describe how to use Alfa-1 for educational purposes. We also explain how to assemble, link, and execute applications and how to test new extensions using the testing tools.

Categories and Subject Descriptors: C.0 **[Computer Systems Organization - General]**: Modeling of Computer Architecture; I.6.5 **[Simulation and Modeling]**: Model Development; K.3.1 **[Computers and Education]**: Computer Uses in Education

General Terms: Design

Additional Key Words and Phrases: DEVS formalism, modeling computer architectures, systems specification methodology

1. INTRODUCTION

Computer organization courses are usually based on the description and analysis of the behavior of digital computers. The complexity of these is so high that they are usually described in layers defining different levels of abstraction. These layered descriptions allow greater insight when analyzing a given subsystem. In general, the following levels are described:

a) *digital circuits*: in general, this level is described using Boolean logic;
b) *microprogramming*: microprogrammed or hardwired control units are described using block diagrams, FPGA logic, etc.;
c) *instruction sets* are described using diagrams and algorithmic descriptions;
d) *assembly language* can be expressed using algorithmic descriptions and finite state automata (representing the assembler syntax).

This research was supported by USENIX and NSERC. Authors' addresses: G. A. Wainer, Department of Systems and Computer Engineering, Carleton University, 1125 Colonel By Drive, Ottawa, ON K1S 5B6, Canada; email: gwainer@sce.carleton.ca. URL: http://www.sce.carleton.ca/faculty/wainer.html; S. Daicz, L. F. De Simoni, D. Wassermann, Computer Science Department, Universidad de Buenos Aires, P.B. Pabellón I. Ciudad Universitaria (1428) Buenos Aires, Argentina.

This organization emphasizes understanding the details of the subsystems, but the complex interlayer interactions are not addressed. The introduction of higher levels of abstraction (such as programming languages or operating systems) makes the pedagogical tasks even more difficult. Another consequence of this organizational structure is the difficulty of providing practical experience in every layer. In most cases, training is only provided at the level of the instruction set, resulting in courses that focus on the higher level of the hierarchy, and thus lose the details of the interlevel interactions. In general, these kinds of courses are based on the classic bibliography of the area (for instance, Hennessy and Patterson [1994]; Patterson [1995]; Stallings [1999]; Heuring and Jordan [1997]; and Tanenbaum [1999]), which present detailed behavioral descriptions of the underlying subsystems. These descriptions are essential in understanding the system concepts. Nevertheless, covering the details of each of the subsystems is not feasible.

Consequently, students do not gain an understanding of system behavior as a whole *nor* detailed knowledge of the subsystems. They may even finish with incomplete and erroneous views of how a computer works, which may affect their progress in more advanced courses in an area (e.g., operating systems, embedded systems, computer architectures, etc.). We decided to attack these problems using simulation tools. Our goal is to provide the students with means of designing hardware architectures. Full understanding of the behavior of the different layers and their interaction can be achieved, thanks to the ability to acquire practical experience with the intralayer and interlayer behavior. In the first stage of this project, we analyzed several tools that were built in order to implement computer architectures. We recognized three basic categories of architecture simulator tools:

A. general-purpose tools
B. specific-purpose tools
C. hardware description languages

The results of this study were presented in Wainer et al. [2002]. In that work, we classified our educational needs as follows:

I. *Levels of abstraction*: we need tools that allow the student to gain experience with the different layers described in a computer system. In particular, we need
   1. the ability to describe the multiple abstraction levels studied in computer organization courses; and
   2. the capacity to define different components using a unique approach.

II. *Pedagogical value*: we need tools that can be used easily by students taking computer organization courses. In particular, we require
   1. the possible use by students in the early stages of their careers of new tools or programming languages that can be programmed without extra effort in areas that are out of the scope of computer organization courses;
   2. a fast learning curve, due to the lack of time in one-term courses; and
   3. resources available in the public domain.

III. *Modifiable models* representing the architecture components, including
   1. the ability to extend the architecture to include new components;

2. the ability to modify existing architectures, providing students with the opportunity to experience different configurations; and
3. ease in testing changes in the architecture.

IV. *Study of advanced architectural facilities*, including
1. the ability to reproduce existing architectures; and
2. the capacity to define state-of-the-art architectures.

We analyzed several existing tools according to these needs and classified their advantages in each of these categories. We included different examples of *general- purpose* tools, which can be applied to build any kind of processor by defining an instruction set, the computer organization, and its components. For instance, SimpleScalar [Burger and Austin 1997] allows different architectures to be defined using basic building blocks that can be used to model advanced architectural aspects. HASE (Hierarchical Architecture design and Simulation Environment) allows users to rapidly develop and explore computer architectures with multiple abstraction levels [Coe et al. 1996]. SimOS is a general-purpose tool [Rosenblum et al. 1997] that permits different architectural details to be defined by providing different CPU models (including descriptions of the architecture and components: caches, multiprocessor memory buses, disk drives, etc.). A large number of general-purpose tools are devoted to model multiprocessor simulations. Some examples can be found in Ikodinovic et al. [1999]; Brewer et al. [1991]; Burns et al. [2000]; Bedichek [1995]; Shanmugan et al. [1992]; and Hein and Dal Cin [1998]. None of them meet our educational requirements; instead, they are best suited for higher-level courses to support lectures on computer architecture.

Another possibility is to describe the details of a given architecture using any of the existing *hardware description languages*. For instance, VHDL [Ghosh 2000] can be used for documentation, verification, and synthesis of large digital designs, using structural, dataflow, and behavioral methods. Verilog [Thomas and Moorby 1991] is easier to learn than VHDL, but lacks constructs to support system-level design (structural models are built from gate primitives and other modules). SDL [Mitschele-Thiel 2000] was developed as a description language for reactive systems, and uses extended finite states presented in a graphical form.

Several of the existing tools were built for the *specific purpose* of emulating existing architectures. Some tools focus on the *Intel* 80x86. For instance, Augmint [Nguyen et al. 1996]; Virgo [Pearce 2000]; and Simx86 [Shealy et al. 1997] focus on the instruction set level and the assembly language level of 8086-based computers. Other environments are based on the *MIPS* architecture. MPS [Morsiani and Davoli 1999] defines RAM, ROM, the processor, disks, tapes, printer, and terminal. Spim [Hennessy and Patterson 1997] implements an assembler-extended instruction set for the MIPS, omitting some of the complex details. *Alpha* processor simulators are presented in Edmonson and Reilly [1998], enabling developers to analyze pipelining levels and instruction-level parallelism. THRsim11 [Anderson et al. 1999] emulates the *Motorola 68HC11* microcontroller in its entirety, including memory, CPU registers, I/O ports, a timer, pulse accumulator registers, AD converter registers, and other standard features.

Table I. Comparing Architecture Simulators

Name/Category		I		II			III			IV		
		1	**2**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**
Alpha Emulator	B	-	-	-	-	-	-	-		+	+	-
Alfa-0	B	-	-	+	+		-	-		-	-	
CASLE	B	-	-	+	+			-		-	-	
CHIP	B	-	-	+	+		-	-		-	-	
CPU-Sim	B	-	-	+	+		-	-		-	-	
ESCAPE/DLX	B	+	-	+	+		-	+	-	-	+	
HASE	A	+		-	-		+	+	+	+	+	-
Limes	B	-	-	-	-		-	-		-	+	-
MPS	B	-	-	+	+		-	-		+	-	
PROTEUS	B	-	-	-	-		-	-		-	+	-
PROVIR	B	-	-	+	+		-	-		-	-	
SDL	C	+	+	-	-	-	+	+	+	+	+	
SimOS	A		-	-	-		+	+	+	+	+	-
SimpleScalar	A	-	-	-	-		+	+	+	+	+	-
SimX86	B	-	-	+	+		-	-		+	-	
SPIM	B	-	-				-	-		+		
Verilog	C	+	+	-	-	-	+	+	+	+	+	
VHDL	C	+	+	-	-	-	+	+	+	+	+	
Virgo/SimX86	B	-	-	+	+		-	-		+	-	
Z80/68HC11/Atari emulators	B	-	-	+	+		-	-		-	-	

Other authors emulated systems based on processors that nowadays are obsolete. El Hajj et al. [2000] presented a method for simulating the *Z80* processor using spreadsheets. Q-Emulator [Zidlicky et al. 2002] is a software emulator of the QL Sinclair PC, which interprets the Motorola *68008* instructions, redirecting input and output to a Mac or PC video, keyboard, mouse, disks, sound hardware, and serial ports. A similar approach was taken in PLM [Isacovich et al. 1999]. This work was devoted to providing an emulator for the *Atari* processor, enabling Atari programs to be run on Intel-based computers. The CHIP (Cornell Hypothetical Instructional Processor [Babaoglu et al. 1983]) emulates a *PDP-11* (including dynamic memory mapping, two modes of processor operation, and eight interrupt priority levels). PROVIR [Bevilacqua et al. 2000] is based on the *IBM 360* architecture (including an assembler, a debugger, and the kernel of an operating system).

In our study we found that the most adequate tools, in an educational sense, are specific-purpose tools that define a fictitious computer architecture and are tailored to educational needs. CASLE [Deitz and Adams 1994] is a Web-based computer using its own instruction set. The interface automatically produces an optimizing compiler, assembler, and architectural simulator using the specified architecture. Students can experiment with the effects of changing the number of registers, instruction latencies, optimizations, etc. Alfa-0 [Wainer et al. 2001] and the Rudimentary Computer [Pastor and Sánchez 1997] are two examples of very simple instruction set computers with educational purposes. These computers enable the representation of some

details of the datapath, microprogramming, and instruction set levels. They include a reduced number of registers and instructions in order to simplify the analysis of the problem. CPU-Sim [Skrien 1994] is a Java application that allows users to design simple processors at the microcode level and to execute programs written in machine language. It permits assembly language programs to be written and debugged, and allows analysis of the state of the machine. Finally, ESCAPE [Campenhout et al. 1998] is based on the DLX architecture [Hennessy and Patterson 1997]. The tool enables users to create their own instructions, to modify the numbers of registers, and the size of immediate operands, and so on.

The results of this survey are summarized in Table I. We show a set of different tools, their types, and classify their features according to the categories presented previously. The '+' sign represents a tool that fulfills our goals and the '-' sign represents a tool that does not meet our goals. Column IV.3 shows the tools that are especially tailored for studying multiprocessor architectures.

When we analyze Table I we can see that none of the tools (which, in fact, represent a sample of the different kinds of existing tools) suit our needs. Many tools are strong in a given category but weak in others. If we analyze category I, we can see that only a few tools are able to represent the multiple layers required. Nonetheless, they are weak from the pedagogical standpoint. Tools that are strong candidates from a pedagogical perspective are weak from some others: they are too simple, or do not permit the underlying organization to be modified. This pattern is repeated when we consider the modifiability of tools. The most interesting candidates are too complex in terms of learning. The best choice is the ESCAPE tool, based on the DLX architecture. Nevertheless, this tool implements a nonexistent processor, and each of the abstraction levels are represented using different formalisms, making it more difficult to accomplish a thorough study of the architecture.

We decided to build a new set of simulation tools that could meet all our goals. The tools can be used by anyone who has a basic knowledge of programming techniques; they are public domain, and have been built using public domain software. The tools were built to be fully understood by people who have taken an introductory course in computer programming. The hierarchical and discrete event nature of the problem made DEVS [Zeigler et al. 2000] a good choice to ease the development task. DEVS provides a system-theoretic framework for describing discrete event systems as a composite of submodels that can be simulated by abstract entities. As we will show, all of the levels usually described in computer organization courses can be developed using DEVS as the modeling framework. In DEVS theory, every existing model can be integrated into a hierarchy, allowing reuse of tested models (which we have done extensively in this project).

We started the project as a specific-purpose architecture based on a modern architecture (the SPARC processor). The model-oriented nature of DEVS resulted in a set of components that can be integrated to define existing architectures (at present, Intel and DSP model architectures are being developed). Hence, we have built a general-purpose toolkit that is specialized as a special-purpose tool.

DEVS enables a modeler to focus on the problem to be solved, as abstract simulators are in charge of execution. Behavioral (atomic) or structural (coupled) models can be

defined within a modeling hierarchy. A DEVS atomic model is described by

$\boldsymbol{M} = < X, S, Y, d_{int}, d_{ext}, l, D >$
X: input events set
S: state set
Y: output events set
d_{int}: S → S, internal transition function
d_{ext}: QxX → S, external transition function, with Q = {(s,e)/ s ∈ S, and e ∈ [0,D(s)]}
l: S → Y, output function
D: S → R_0^+ elapsed time function

Each model uses input/output ports in the interface (defined by the X and Y sets) to communicate with other models. When an input external event is received in an input port, it activates the external transition function, which produces a change of state. The new state has an associated lifetime, after which the internal transition function is activated (producing a new state change). The model can generate data to be transmitted through the output ports. The output function, which is in charge of this task, is activated before the execution of the internal transition function.

DEVS atomic models can be used to build coupled models. It was proven that DEVS coupled models are semantically equivalent to atomic models. Consequently, it is possible to build different semantically equivalent modeling hierarchies. A DEVS coupled model is defined by

$\boldsymbol{CM} = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\} >$
X: input events set
Y: output events set
D: index of components, and i ∈ D
$\mathbf{M_i}$: basic DEVS model
$\mathbf{I_i}$: influencees of model i, and j ∈ I_i
$\mathbf{Z_{ij}}$: $Y_i \rightarrow X_j$, the i to j translation function

Each coupled model is composed of a set of basic models (atomic or coupled) connected through their input/output ports. Each component is identified by an index number. The influencees of a model are those receiving the outputs of the model. The translation function transforms the outputs of a model to the inputs of others. First, we create an index of influencees for each model (I_i). We then define which outputs of model M_i are connected to inputs in model M_j by using the set of influencees. When two submodels have external simultaneous events, the *select* function prescribes which of them should be activated first.

There are different tools that implement the theoretical concepts defined by the DEVS formalism (DEVSJava, DevsSim++, DEVS-C++, JDEVS, DEVS-Scheme, etc.). We used the CD++ tool, which is public domain and fits our needs [Rodriguez and Wainer 1999; Wainer et al. 2001; Wainer and Troccoli 2001]. In this tool, atomic models are programmed in C++, enabling users with basic programming skills to develop

new models. Coupled models are defined using a built-in specification language, which represents the model coupling scheme. The detailed design and implementation results of this project are presented in Wainer et al. [2002].

In the following sections we explain how the tool can be used and modified. We also show how the tools were used in computer organization assignments and how to repeat this experience in a classroom. We organized the article according to the model's architecture and organization. Stallings [1999] states that the term computer *architecture* refers to those attributes of a system visible to a programmer, and that computer *organization* refers to the operational units and their interconnections. We start by describing the organizational level, focusing on the definition and modification of components. We then discuss the system architecture level, showing how a programmer can use the services provided by the computer (including examples of executable files). We subsequently present a set of tools devoted to improving testing, and an ongoing effort to building a GUI for the toolkit. Finally, we present a set of activities that can be carried out using the toolkit.

2. THE ORGANIZATION LEVEL

The Alfa-1 simulated computer is built as a set of interacting submodels representing the behavior of the architectural components. As we mentioned, Stallings [1999] defines computer organization as to the operational units and their interconnections that realize architectural specifications. Organizational attributes include those hardware details transparent to the programmer, such as control signals, interfaces between the computer and peripherals and the memory technology. The Alfa-1 processor organization is based in the specification of the integer unit of the SPARC processor [Sun 2001], and is shown in Figure 1. The processor includes 8 global registers (*RegGlob*, shared by every procedure), and 512 organized in windows of 24 registers (*RegBlock*).

The processor includes several special-purpose registers:

1. *PC*s: there are two program counters. PC contains the address of the next instruction, and nPC stores the address of the next PC. If the instruction is a conditional branch, nPC is assigned to PC, and nPC is updated with the jump address.
2. *CWP* (circular window pointer) is a specialized 5-bit register that marks the active window. Every time a new procedure starts, *CWP* is decremented.
3. *PSR* (processor status register) stores the program status.
4. *Y* is used by the product and division operations. Multiplication uses 32-bit operands, producing 64-bit results. The 32 most significant bits are stored in Y, and the remaining bits are stored in the ALU-RES register. The integer division operation takes 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 this number, and the other, the divisor. The integer result is stored in the ALU-RES register and the remainder is stored in the Y register.

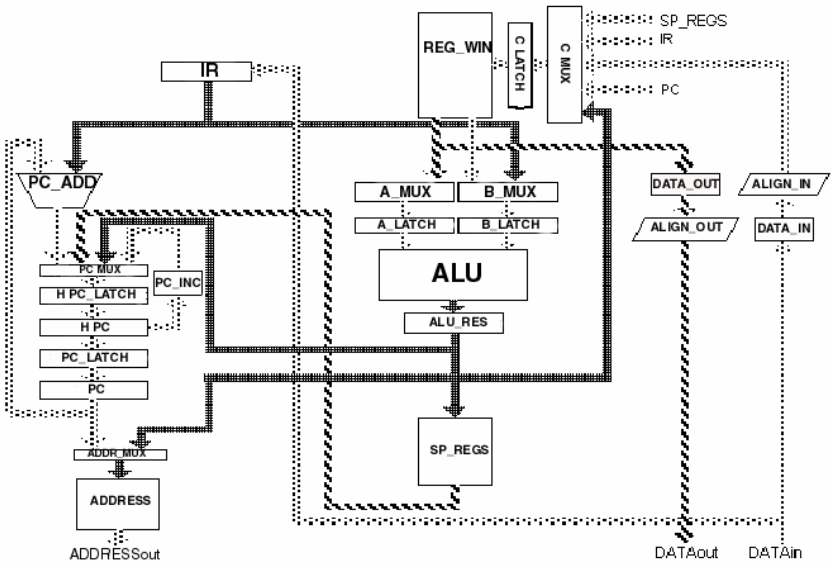

Fig. 1. Organization of the integer unit [Wainer et al. 2002].

5. *BASE* and *SIZE*: we used a flat memory addressing scheme. The BASE register points to the lowest address a program can access. SIZE stores the program size, which represents the maximum addressable memory unit.
6. *WIM* (window invalid mask) is a 32-bit register used to avoid overwriting a register window. When CWP is decremented (because a procedure was called), the WIM bit is verified for the new window. If it is active, an interrupt is raised and the service routine must store the window contents in memory. WIM usually marks the oldest window.
7. *TBR* (trap base register) points to the memory address storing the location of trap routines.

The processor was implemented using the CD++ tool. In a first stage, we specified the behavior of each component, analyzing inputs and outputs of the original circuits. These specifications allowed test cases to be derived (details of this phase can be found in Daicz et al. [1998]). Each subcomponent of the computer was then defined using DEVS as the specification formalism. Using these descriptions, each model was implemented in the CD++ toolkit. Each of the models can be reused in building different architectures. Each model has an associated experimental framework that allows the derived test cases to be executed (details of this phase can be found in De Simoni et al. [1998]). Thirty-five different circuits were defined as DEVS atomic or coupled models and were integrated into the architecture. Several of these models were defined as multicomponents based in digital logic. A set of Boolean gates was included, and these models enabled us to define some of the submodels at the digital logic level. Details about the existing models and their various implementations can be found in Wainer et al. [2002].

```
class AtomicState : public ModelState {
public:
   enum State {
        active,
        passive
   } ;

   State st;
   AtomicState(){};
   virtual ~AtomicState(){};

   AtomicState& operator=(AtomicState& thisState);
   void copyState(BasicState *);
   int  getSize() const;
};
```

Fig. 2. The AtomicState class.

Finally, the main model was built as a coupled model connecting all the submodels defined previously. The CPU executes under the supervision of the control unit. It receives signals from the rest of the processors 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. In addition, connections with the PC, nPC, trap controller, and PSR registers are included. Finally, the data, address and control buses can be accessed. The control unit was defined using microprogramming concepts. Hence, by modifying the control unit, students can gain experience at the microprogramming level.

After running the individual tests for each of the components, we executed integration tests. Integration required combining all of the components and defining their interactions to represent the execution flow of the computer. Execution flow tasks are carried out by the control unit, which, in this case, was built as a DEVS model using several input/output ports representing the CU lines. According to the input received, it issues appropriate outputs by activating the different circuits that were defined previously (details can be found in De Simoni et al. [1998]). Finally, a thorough integration test was executed; the results of this phase are presented in Section 4.

The first step needed is to install the required tools. The instructions to do so can be found at

<http://www.sce.carleton.ca/faculty/wainer/alfa-1.html>.

After installing the tools successfully, different activities can be carried out. Many of them involve modifying or extending the existing components. To do so, atomic models may need to be changed, or new models may be added and integrated with existing coupled models. The following section is devoted to showing how to code new models or modify existing ones, following the guidelines presented in Wainer and Troccoli [2001]. We then show how to introduce new atomic and coupled models into the simulated computer, which may be carried out as a course assignment.

2.1. Defining New Models

In CD++, a new atomic model is created as a new class derived from the *Atomic* base class. The state of the model is composed by the variables that can be changed during a simulation cycle, which are defined in the *AtomicState* class in Figure 2.

When creating a new atomic model, a new class derived from atomic has to be created. *Atomic* is an abstract class that declares a model's API and defines some service functions the user can use to write the model. The *Atomic* class (Figure 3) provides a set of services and requires a set of functions to be redefined:

1. *holdIn(state, Time)* tells the simulator that the model will remain in the state *state* for a period *time*. It corresponds to the D(s) DEVS atomic function.
2. *passivate()* sets the next internal transition time to infinity. The model will only be activated again if an external event is received.
3. *sendOutput(Time, port, BasicMsgValue*)* sends an output message through the port. The time should be set to the current time.
4. *nextChange()* returns the remaining time for the next internal transition.
5. *lastChange()* returns the time since the last state change.
6. *state()* returns the phase of the current model.

The new class should overload the following functions:

1. *virtual Model &externalFunction(const ExternalMessage &)* is invoked when one external event arrives to a port. It corresponds to the δ_{ext} function of the DEVS formalism.

```
class Atomic : public Model {
public:
   virtual ~Atomic();  // Destructor

protected:
   //User defined functions.
   virtual Model &initFunction() = 0;
   virtual Model &externalFunction( const ExternalMessage & );
   virtual Model &internalFunction( const InternalMessage & ) = 0 ;
   virtual Model &outputFunction( const CollectMessage & ) = 0 ;
   virtual string className() const

   //Kernel services
   Time nextChange();
   Time lastChange();

   Model &holdIn( const AtomicState::State &, const Time & ) ;
   Model &sendOutput(const Time &time, const Port & port , Value value)
   Model &passivate();

   Model &state( const AtomicState::State &s )
   { ((AtomicState *)getCurrentState())->st = s; return *this; }

}; // class Atomic
```

Fig. 3. The atomic class.

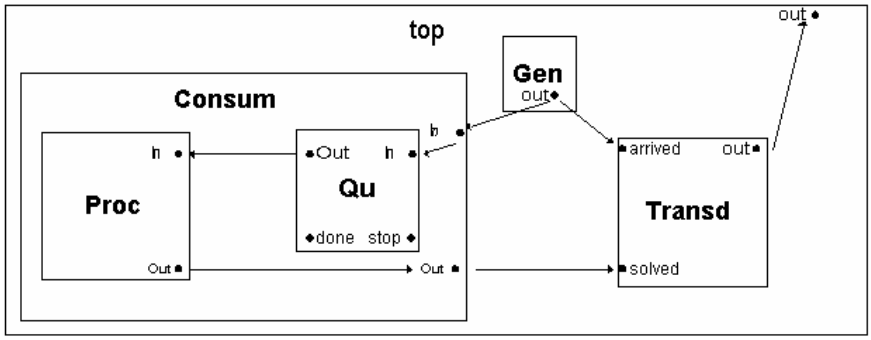

Fig. 4. A consumer/producer scheme [Zeigler et al. 2000].

2. *virtual Model &internalFunction(const InternalMessage &)* corresponds to the δ_{int} function of the DEVS formalism.
3. *virtual Model &initFunction()* is invoked by the simulator at the beginning of the simulation and after the model state has been initialized.
4. *virtual Model &outputFunction(const CollectMessage &)* is in charge of sending all of the output events of the model.

After each atomic model is defined, they can be combined into a multicomponent model. Coupled models are defined using a specification language especially defined for this purpose. The language was built following the formal definitions for DEVS coupled models (Figure 4). Optionally, configuration values for the atomic models can be included.

The **[***top***]** model always defines the coupled model at the top level. As shown in formal specifications presented in Section 1, four properties must be configured: components, output ports, input ports, and links between models.

The following syntax is used:

`Components:` `model_name1[@atomicclass1] [model_name2[@atomicclass2] ...`
This sentence is used to list the components comprising the coupled model. A coupled model may include atomic models and/or other coupled models as components. If we include an atomic component, we must specify an instance and a class name. This allows a coupled model to use more than one instance of an atomic class. For coupled models, only the model name must be given. This model name must be defined as another group in the same file.

`Out``: portname1 portname2 ...` enumerates the output ports of the model. This clause is optional because a model may not have output ports.

`In``: portname1 portname2 ...` enumerates the input ports. This clause is also optional because a coupled model is not required to have input ports.

`Link:` `source_port[@model] destination_port[@model] ...` describes the internal and external coupling scheme. The name of the model is optional. The model used by default is the coupled model currently being defined.

```
[top]
components : Transd@Transducer Gen@Generator Consum
Out : out
Link : out@generator arrived@transducer
Link : out@generator in@Consumer
Link : out@Consumer solved@transducer
Link : out@transducer out

[Consum]
components : Qu@Queue Proc@CPU
in : in                         out : out
Link : in in@qu
Link : out@qu in@Proc
Link : out@Proc done@qu
Link : out@Proc out
```

Fig. 5. Example for the definition of a DEVS coupled model.

Figure 5 shows a sample of a coupled model representing a simple consumer/producer scheme. It consists of three models: a generator that creates data to be consumed, a consumer, and a transducer that measures the consumer speed. The consumer is a coupled model, composed of a processor and a queue to keep waiting jobs.

At the top level of this example, the *Generator* influences the *Transducer* and the *Consumer*. The *Consumer* also influences the *Transducer*, as well as the *Queue*. The *Processor* influences the *Consumer* and the *Queue* influences the *Processor*. Finally, the *Transducer* influences the top model. These interconnections define the influence sets for each of the components. The influencee sets are used to define the translation functions, which transfer data through the input/output ports in the models.

2.2. Modeling at the Organization Level: A Simple Look-Aside Component

The remainder of Section 2 shows simple examples of the educational activities that can be carried out at the organization level (the models developed here were built as assignments in a computer organization course). We show how a user can modify or change existing components, focusing on extending the simulator by adding components external to the processor.

Look-aside components (usually input/output devices) are attached to one of the bus slots. In order to be able to add any new component, we must know the bus protocol used for Alfa-1. If we analyze the descriptions in Daicz et al. [1998], we can see that the bus uses the lines in Table II.

In each data transfer, we can identify an active and a passive component. All the active components are chained by a bus grant line (BGRANT), used to receive a signal from the component with next higher priority and to send a signal to the component with next lower priority (Figure 6). The active component must take over the bus. First, it requests to be the next one to use the bus. If the component requesting the bus receives a 1 over the BGRANT input line, this means that none of the components with higher priority need the bus. Then, it sends a 0 over the BGRANT output line, telling the components with lower priority that it is intending to use the bus. It must then wait until a 0 is received over the BUSY line, meaning that the bus is no longer being used.

Table II. Alfa-1 Bus Lines

Pin Name	*In*	*Out*	*Description*
Data0-31	X	X	Data
A0-31	X	X	Address
Clock	X	X	Clock
AS	X	X	Address Strobe
RD/WR	X	X	Read or Write
DTACK	X	X	Data Acknowledge
Err	X	X	Error
RESET	X	X	Reset
IRQ1-15	X	X	Interrupt Request
Busy	X	X	bus Busy

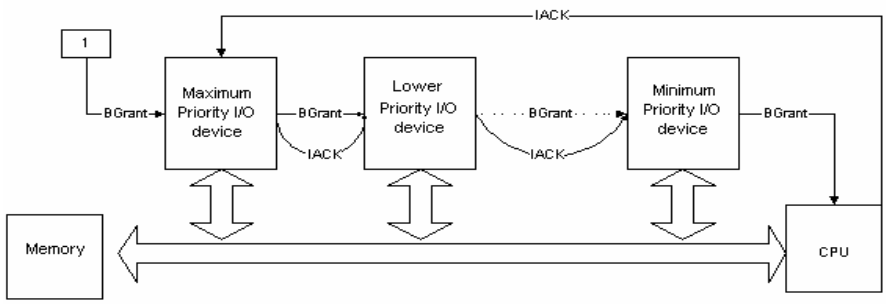

Fig. 6. Active components BGRANT chain [Wainer et al. 2002].

Finally, it will send a 1 over the BUSY line to let the other components know that the bus is being used.

After the bus is taken, the address to be accessed should be written to the address bus (A0-31). The access to memory involves two parameters. First, the active component chooses which 32-bit memory word will be used and sets the address on the address bus. The second parameter allows a 32-bit word to be split into four bytes (D0-7, D8-15, D16-23, D24-31) and it must be specified. The BSEL bits are used to choose the desired bytes: if the active component sets BSEL0, D0-7 will be addressed (BSEL1 corresponds to D8..15, etc.). If the active component requested a write operation (clearing RD/WR), the data to be written should be sent over the selected bytes of the data bus (D0-31). The active component will subsequently send a 1 over the AS line (letting the passive component know that everything is set), and will wait for the DTACK line to be set (which means that the request has been fulfilled). If the operation is a read, the data on the D31-0 lines is collected. The active component must now send a 0 over the AS line to finish the operation. Now the active component can start another operation (it still controls the bus) by setting BGRANT and clearing BUSY to release the bus.

The passive component (for instance the main memory) will use a different protocol. It will first receive a read or write request from the active component, represented by AS being set. It will then receive the address (A0-31) and the bytes selected in the byte mask (BSEL0-3). It then verifies the memory address (which should belong to the ad-

Table III. Bus Lines Used by a Look-Aside Component

Signal	*In*	*Out*
RD/WR	X	
AS	X	
DTACK		X
ERR		X
$D_{31..0}$	X	X
$A_{31..0}$	X	
$Bsel_{3..0}$	X	

dress space of the component). If a write operation is requested, the data to be written should be available over the selected bytes of the data bus (D0-31). It will then output or receive data from Data0-31, according to the action specified by the RD/WR line. When the request has been fulfilled, the passive component will set DTACK. Finally, the AS line must be cleared.

Using this basic information about the bus lines, we show the specification and implementation of a look-aside component. This very simple model can be used as an exercise to show how new models can be incorporated into the existing hierarchy. When a read operation is issued, the look-aside component will return 1s in the data bits (D31-0) selected using the BSEL3-0 mask. If a write operation is issued, it returns an error. Look-aside components are attached to the bus like the memory. Table III shows the signals that should be used.

This atomic model can be defined as follows:

$\boldsymbol{SAMPLELA} = < X, S, Y, d_{int}, d_{ext}, l, D >.$

$\boldsymbol{X} = AS, RD/WR, A \hat{I} \{0,4,\ldots,2^{32}-1\}, BSEL \hat{I} \{0,..,2^{4}-1\};$

$\boldsymbol{S} = responseTime \hat{I} \boldsymbol{R_0}^{+};$

$\boldsymbol{Y}=D \hat{I} \{0,...,2^{32}-1\}, DTACK, ERR;$

In addition, the transition functions are described informally in Figure 7.

As we can see, this component follows the bus specification, and is suitable for attachment to the bus in order to work as a passive component. This behavior is implemented on **δext( )**. It first senses a 1 over the AS line. The component then checks the address space in order to find out whether the request was for its address. In that case, it fills out the selected data bytes with 1's. Finally, the component waits for the active component to clear the AS line. When this happens, the *samplela* component sends a 0 over the DTACK line to end the communication. If we want to simulate a delay in the component's response, the atomic model must wait for some time before starting with the next step. Thus, we call the *hold_in* procedure with the respective delay parameter. The output function **λ( )** sends the signals over the output ports when needed. On the other hand, **δint( )** performs just one operation, *passivate*. This means that the component remains inactive until a new signal arrives on any input port.

```
δext() {
   When (1 is received in the port AS)
        if the value on received over A31..2 is in our address space then
                if (RD/WR=1) then
                        if (BSEL0 = 1) then D7..0 := FFh
                        if (BSEL1 = 1) then D15..8 := FFh
                        if (BSEL2 = 1) then D23..16 := FFh
                        if (BSEL3 = 1) then D31..24 := FFh
                        DTACK:=1
                else
                        ERR:=1
   When (0 received in the port AS) and (1 sent in the port DTACK)
        DTACK:=0;
   hold_in(responseTime)
}

δint() { passivate; }

λ() {
   IF (D31..0 changed) send D31..0 over the D31..0 ports
   IF (DTACK changed) send DTACK over the DTACK port
   IF (ERR is changed) send ERR over the ERR port
}
```

Fig. 7. Behavior of the *samplela* look-aside component.

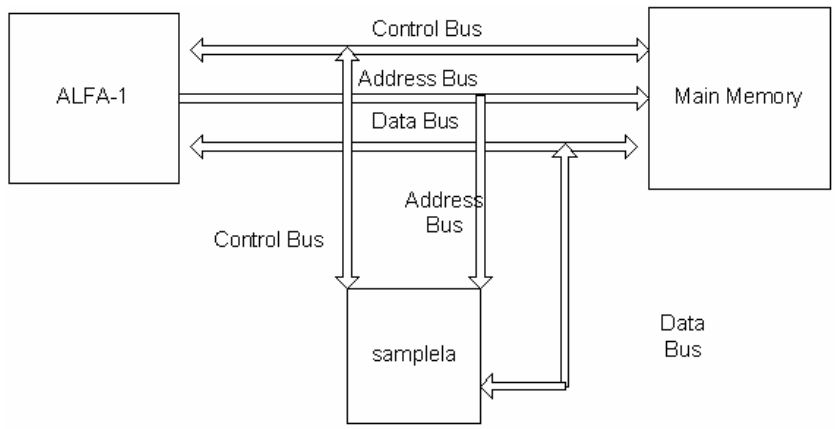

Fig. 8. Look-aside model interconnection.

Figure 9 shows part of the implementation of the transition functions using the CD++ tool. The first lines of the *externalFunction* (**δext( )**) identify the signal that has arrived. Since this function activates every time an input is received, we begin by identifying one signal at a time. We distinguish whether we are sending a response or waiting for a request using the *m_state* variable. There are two arrays named *m_ctlIn* and *m_ctlOut*, containing the present values of the input and output control ports. For instance, the

```
Model & sampleLA::externalFunction( const ExternalMessage & msg) {
   string portName;
   unsigned long portNum;
   unsigned i;

   // Get the port name where a message arrived, and its value
   nameNum(msg.port().name(), portName, portNum );
        for( i=0; i<NUM_CTL_IN_PORT; i++)
        if ( portName == ctlInPortNames[i] )
               m_ctlIn[i]= bit (msg.value());

   if ( portName == string(addrInPortNames[0]) ){
        m_addrIn[portNum]= bit (msg.value());  }
   if (portName == bselInPortNames[0] )
        m_bselIn[portNum]= bit (msg.value());
   if (portName == dataInPortNames[0] )
        m_dataIn[portNum]= bit (msg.value());
   if (m_ctlIn[asin] && (m_state==waiting)) {

  //If someone has requested a read/write operation to the bus
  unsigned long addrRequested = fromBits(m_addrIn,m_addrWidth);
  if ((m_minAddrSpaceLimit <= addrRequested) &&
         (addrRequested < m_maxAddrSpaceLimit)) {
        if (m_ctlIn[rwin] == 0) //if its trying to write to the model
               m_ctlOut[errout].val  = 1; //we send an error
        else  {
               for (unsigned long i = 0 ; i< m_bselWidth; i++)
                  if (m_bselIn[i])
                       for (unsigned long j=0 ; j<m_byteSize ; j++)
                              m_dataOut[i*m_byteSize+j].val = 1;
                       m_ctlOut[dtackout].val = 1;    }
        }
        m_state = sending;
   } else if (m_state == waiting) {
        //If we do not receive an AS line we output 0 on dtack
               m_ctlOut[dtackout].val = 0;
               m_ctlOut[errout].val = 0;
               m_state = waiting;           }
   this->holdIn( active, m_responseTime );
}

Model & sampleLA::internalFunction( const InternalMessage & msg) {
   this->passivate();
}

Model & sampleLA::outputFunction( const InternalMessage & msg) {
   //Outputting data over the ports
   for(unsigned i=0; i<NUM_CTL_OUT_PORT; i++)
        if (needSend(m_ctlOut[i]))
          this->sendOutput(msg.time(),*m_ctlOutPorts[i],m_ctlOut[i].val);
   for(unsigned i=0; i<m_dataWidth; i++){
        if (needSend(m_dataOut[i]))
          this-
   >sendOutput(msg.time(),*m_dataOutPorts[i],m_dataOut[i].val);
   }
}
```

Fig. 9. CD++ samplela look-aside component implementation.

```
components: samplela

Link : OUT_A2@bus A2@samplela
. . .
Link : OUT_A31@bus A31@samplela

Link : OUT_DATA0@samplela IN_DATA0@cpu
. . .
Link : OUT_DATA31@samplela IN_DATA31@cpu

Link : OUT_DATA0@cpu IN_DATA0@samplela
. . .
Link : OUT_DATA31@cpu IN_DATA31@samplela

Link : DTACK@samplela IN_DTACK@bus
Link : OUT_RD_WR@bus RW@samplela

Link : OUT_BSEL0@bus BSEL0@samplela
. .
Link : OUT_BSEL3@bus BSEL3@samplela
```

Fig. 10. The iu.ma definition in Alfa-1.

```
[samplela]

responseTime = 0:0:0:0
minAddr =  65536
maxAddr = 131072
```

Fig. 11. Adding the atomic component samplela to the ALFA1.

rwin constant within the *m_ctlIn* array represents the RD/WR port; hence, *m_ctlIn[rwin]* will contain a 1 when the last input received on that port was 1. We only send data over the output ports if they have changed their values. Consequently, output ports (*m_ctlOut, m_dataOut*) are represented using a data type that distinguishes between the ones that need to be sent and the ones that do not. The *needSend* function is used to query whether a value needs to be sent. The value in the *val* field will be sent over the corresponding output port if needed. Finally, the *internalFunction* (**δint**) and the *outputFunction* (λ) are straightforward implementations of the definitions in Figure 7. Once successfully compiled, the simulator will be able to reference the new component using the *samplela* name. The coupled model file of the Alfa-1 simulator (*iu.ma*) must be edited in order to include the new model in the hierarchy. The lines in Figure 10 should be added in the [top] model:

These definitions correspond to the bus connections of the *samplela* model. More precisely, following the original *iu.ma* connections, the address bus (A2-31), the BSEL signal, and the RD/WR are connected from the *bus* component to the *samplela* component. The DTACK line is connected the other way around (from *samplela* to *bus*). The data bus (D0-31) is connected directly from and to the *CPU* component; this connection is bidirectional due to the characteristics of this bus.

The *samplela* model has a response delay that can be configured using the *responseTime* atomic model parameter. This argument, as well as the address space boundaries *minAddr* and *maxAddr* (lower and higher bounds, respectively) will be passed to the model by adding the lines in Figure 11 at the end of the *iu.ma* file.

2.3. A Sample Look-Through Component

A look-through component (usually a cache memory) is inserted between the microprocessor and the bus, and it should be transparent to the other components attached to the bus. We will now include the specification of a very simple look-through component, which responds to read operations by returning 1's in the data bits selected by BSEL. If a write operation is intended, an error is raised.

Although it may seem similar to the look-aside component, this is not the case. This component is attached between the CPU and the bus, and it must be transparent to any other components when its address space its not being referenced.

The signals handled by a look-through component can be divided into those connected to the CPU and the ones connected to the bus. The lines connected to the CPU must react as a standard bus, whereas those connected to the bus must react as a CPU. In Table IV we show the input/output lines of this model, divided into the two types.

This atomic model can be defined as follows:

$\boldsymbol{SAMPLELT} = < X, S, Y, d_{int}, d_{ext}, l, D >.$

$\boldsymbol{X} = AS(CPU),\ Busy_{in}(CPU), A_{in}\ \hat{I}\{0,...,2^{32}\text{-}1\}, BSEL_{in}\ \hat{I}\ \{0,...,2^{3}\text{-}1\}, RD/\underline{WR}_{in},$
$D_{in}(CPU)\ \hat{I}\ \{0,...,2^{32}\text{-}1\}, Err_{in}, DTACK_{in}, BGrant_{in}, Busy_{in}(bus), BGrant_{in},$
$D_{in}(bus)\ \hat{I}\ \{0,..,2^{32}\text{-}1\};$

$\boldsymbol{S} = responseTime\ \hat{I}\ \boldsymbol{R}_0^{+};$

$\boldsymbol{Y} = Err_{out}, Dtack_{out}, Busy_{out}(CPU), Busy_{out}(bus), Bgrant_{out}, D_{out}(CPU)\ \hat{I}\ \{0,...,2^{32}\text{-}1\}, RD/\underline{WR}_{out}, AS(BUS), A_{out}\ \hat{I}\{0,...,2^{32}\text{-}1\}, BSEL_{out}\ \hat{I}\{0,...,2^{3}\text{-}1\}, D_{out}(BUS)\ \hat{I}\{0,...,2^{32}\text{-}1\};$

In addition, the transition functions are described informally in Figure 12. We can see that these transition functions are similar to the ones in the *samplela* component. The **δext()** functions are the same when the address of the request is in the address space of the *samplelt* component. When the address is referencing another component, the *samplelt* component produces three state changes. First, it takes over the bus. Second, it forwards all of the input signals (data, address, RD/WR, AS, DTACK, ERR) from the CPU to the bus. Third, when the communication has ended, it releases the bus and stops forwarding the signals.

The implementation of these functions using CD++ can be found in the distribution files of the Alfa-1 computer. We have used similar implementation techniques to those used in *samplela*. A variable named *m_forwardingState* is used to distinguish among the three forwarding states previously identified. In this case, the code that copies the input

Table IV. Input/Output Lines of a Look-Through Model

	CPU			***Bus***		
Signal	In	Out	In/Out	In	Out	In/Out
Err		X		X		
Dtack		X		X		
AS	X				X	
RD/$\overline{WR}$	X				X	
Busy			X			X
Bgrant		X		X		
$A_{31..2}$	X				X	
$BSEL_{3..0}$	X				X	
$D_{31..0}$			X			X

Values to the output ports (which takes place if the component is forwarding signals between the bus and the CPU) has been moved into the external function. This way it will be executed only once, when it is needed.

The look-through component must be attached between the CPU and the bus. To do so, we must first detach the CPU component from the bus (shown in Figure 13), and we must connect every line at both sides of the *samplelt* component (as described in Figure 14). Assuming that the new component has been compiled into the simulator and that its name is *samplelt*, we can change the coupling scheme by editing the coupled model file of the Alfa-1 (*iu.ma*). In this case, the lines in Figure 13 should be removed from the *top* model. The lines removed from the *top* model should be replaced by the lines in Figure 14.

After defining this simple look-through component, we can model a complete cache memory. A cache is a small amount of fast memory that reduces the overhead in accessing main memory by storing the most recently used cells. Whenever the CPU needs to perform operations with data stored in the cached cells, the cache memory will be accessed transparently. The decision about which main memory cells should be cached and which should not is made dynamically. In order to enable cached memory cells to change as a program runs, the cache memory includes some information about its contents in a directory. This index stores the main memory addresses of the cached cells. It also includes information to define which cells should be removed from the cache when we need to allocate new cells and there is no space left. There are several cache strategies to translate address bus requests into cache data. The two most widely used are direct and associative mapping (which uses associative memory to do the. translation). The most common replacement strategies are the least-recently-used, least-frequently-used, first-in-first-out, and random [Tanenbaum 1999]. The two most common strategies to update the main memory are to update every change to the cached cells (write-through cache) or not (write-back cache). In the latter case, the cache memory needs to flush cached cells back to the main memory before they can be replaced.

We have developed a model representing a cache memory with write-back update. It is implemented as an extension to the look-through component defined earlier in this section.

```
δext() {
   When (1 is received in the port AS(CPU))
        if the value on received over Ain31..2 is in our address space
   then
           if (RD/WRin=1) then
                 if (BSELin0 = 1) then Dout(CPU)7..0 := FFh
                 if (BSELin1 = 1) then Dout(CPU)15..8 := FFh
                 if (BSELin2 = 1) then Dout(CPU)23..16 := FFh
                 if (BSELin3 = 1) then Dout(CPU)31..24 := FFh
                 DTACKout:=1
           else
                 ERRout:=1
           When (0 is received in AS(CPU)) and (1 is sent in port DTACK-
   out)
               DTACKout:=0;
        else
                wait for BGRANT(BUS)=0
                wait for BUSYin(BUS)=0
                BUSYout(BUS):=1
                start forwarding
                wait for DTACKin(BUS)=1
                DTACKout(CPU):=1
                wait for AS(CPU)=0
                AS(BUS):=0
                BUSYout(BUS):=0
                stop forwarding
                DTACKout(CPU):=0
   hold_in(responseTime)
}

δint() { passivate; }

λ() {
   IF we are forwarding
        Dout(CPU)31..0:=Din(bus)31..0
        Dout(bus)31..0:=Din(CPU)31..0
        Aout31..2:=Ain31..2
        Bselout3..0:=Bselin3..0
        Errout:=Errin
        Dtackout:=Dtackin
        Asout:=Asin
        RD/WRout:=RD/WRin
   IF (Dout(CPU)31..0 changed) send Dout(CPU)31..0
   IF (Dout(bus)31..0 changed) send Dout(bus)31..0
   IF (DTACKout changed) send DTACKout
   IF (ERRout changed) send ERRout
}
```

Fig. 12. Behavior of the *samplelt* look-through component.

All the steps required to implement this model are included in the tool package, enabling users to gain experience with cache memory models. The model is designed to be changed easily without reprogramming the whole model. The associative mapping scheme with a FIFO replacement strategy is thoroughly tested. At present, LFU, LRU, and random replacement strategies are provided as in-progress developments.

```
Link : OUT_DATA0@cpu IN_DATA0@mem ...
Link : OUT_DATA31@cpu IN_DATA31@mem
Link : AS@cpu IN_AS@bus
Link : RD_WR@cpu IN_RD_WR@bus
Link : A0@cpu IN_A0@bus ...
Link : A31@cpu IN_A31@bus
Link : BSEL0@cpu IN_BSEL0@bus ...
Link : BSEL3@cpu IN_BSEL3@bus
```

Fig. 13. Removing connections to define a new coupling scheme.

```
Link : OUT_DATA0@cpu IN_DATA_CPU0@samplelt ...
Link : OUT_DATA31@cpu IN_DATA_CPU31@samplelt
Link : OUT_DATA0@samplelt IN_DATA_CPU0@cpu ...
Link : OUT_DATA31@samplelt IN_DATA_CPU31@cpu
Link : AS@cpu IN_AS@samplelt
Link : RD_WR@cpu IN_RW@samplelt
Link : A0@cpu IN_A0@samplelt ...
Link : A31@cpu IN_A31@samplelt
Link : BSEL0@cpu IN_BSEL0@samplelt ...
Link : BSEL3@cpu IN_BSEL3@samplelt
Link : OUT_DATA_BUS0@samplelt IN_DATA0@mem ...
Link : OUT_DATA_BUS31@samplelt IN_DATA31@mem
Link : AS@samplelt IN_AS@bus
Link : OUT_RW@samplelt IN_RD_WR@bus
Link : OUT_A0@samplelt IN_A0@bus ...
Link : OUT_A31@samplelt IN_A31@bus
Link : BSEL0@samplelt IN_BSEL0@bus ...
Link : BSEL3@samplelt IN_BSEL3@bus
```

Fig. 14. New links to connect a new component.

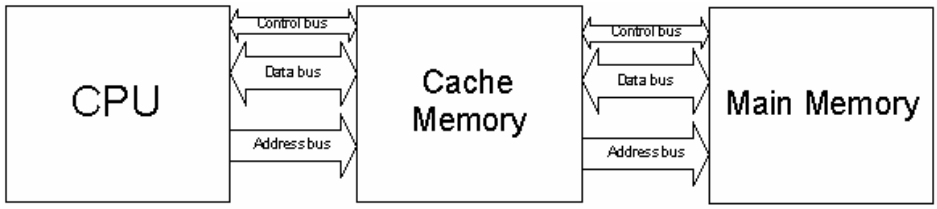

Fig. 15. Cache memory atomic abstraction level.

The first step to consider when defining a new component is how to design the model and the abstraction levels. At the highest level of abstraction, we see the cache as a single component interacting with the CPU and the bus. In order to achieve this behavior, the model follows the standard behavior of the look-through cache previously described. A second level of abstraction considers the internal organization of the cache.

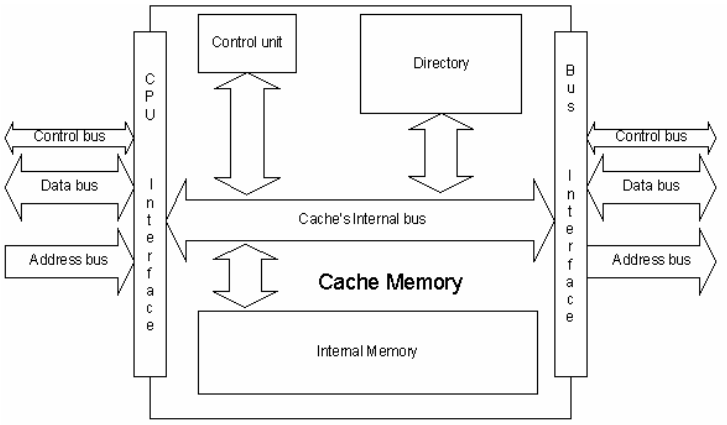

Fig. 16. Cache memory structure.

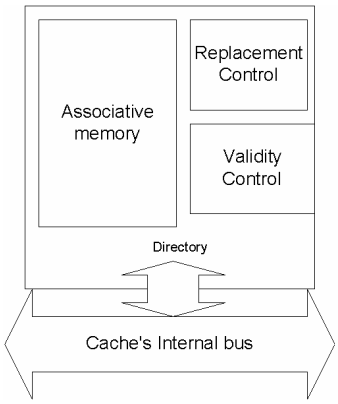

Fig. 17. Directory structure.

We divide the component into five subcomponents connected to two interfaces: one to handle bus communication and the other to handle CPU communication. An internal memory stores cached data, and a directory stores the main memory cells being cached (Figure 15). A control unit coordinates the operation of the other components (Figure 16). A third level of abstraction considers dividing the directory into three components: an associative memory to store the memory address of the cached memory; a replacement control component; and a validity component (see Figure 17). The validity module recognizes which cells have been updated in main memory.

3. THE ARCHITECTURE LEVEL

As explained in Section 1, Stallings [1999] defines computer architecture as those attributes that have a direct impact on the logical execution of a program. Examples of architectural attributes include the instruction set, the number of bits used to represent various data types (e.g., numbers, characters), I/O mechanisms, and techniques for addressing memory. For example, whether a computer will have a multiply instruction is an architectural design issue.

The Alfa-1 architecture is based in the description of the SPARC processor [Sun 2001]. This processor is provided with 520 integer registers divided into windows of 24 registers each, including input, output, and local registers for each procedure. When a procedure starts, 16 registers are reserved (8 local and 8 for 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 in the register ring. *CWP* is decremented each time a new procedure is started. The 32-bit *WIM* register (window invalid mask, one bit per window) avoids the superposition of a window in use by another procedure. When *CWP* is decremented, the hardware checks whether *WIM* is on the new window, and if so, an interrupt is raised. The interrupt service routine saves the content of the window, which will be overwritten. Usually, *WIM* has only a single bit set to 1, marking the oldest window. When that window is reached, the *WIM* rotates one unit. The processor status register includes the result of the last executed instruction. Its contents are interpreted as shown in Table V.

The memory is organized using byte addressing and the Little-Endian standard to store integers. The processor issues a memory access operation by writing an address (and data, if needed) to the bus.

As explained in the Organization section, there is a trap base register used to implement hardware and software interrupts. The register points to a memory address that contains the location of the trap routines. This location is organized as a table whose first 20 bits (trap base address) store the base address of the trap table (Table VI). When an interrupt request is received, the number of the trap to be serviced is stored in the bits 11..4. Therefore, the TBA points to the table position containing the address of the service routine. The last 4 address bits are 0 to guarantee at least 16 bytes to store each routine.

The computer emulates a reduced version of the *instruction set* level of the SPARC architecture. This processor uses instructions with a fixed size of 32 bits, with 8, 16, or 32 bit operands. There are two 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*. Several *jump* instructions are available, including *relative* jumps, *absolute* jumps, *traps*, *calls*, and *return* from traps. Other instructions include changing the movement of the register window, NOPs, and read/write operations on the PSR. There are two execution modes: user and kernel. Certain instructions can only be executed in kernel mode. In addition, the base and size registers are used only when the program is running in user mode.

After installing the tool, the computer is ready to run executable code for the SPARC processor. The first step in developing a program for execution is to write the source code using the SPARC Assembly Language [Sun 2001]. The source code must then be

Table V. Contents of the Processor Status Register

Bits	*Content*	*Description*
31-24	Reserved	
23	N – Negative	1 when the last operation is negative
22	Z – Zero	1 when the last operation is zero
21	V – Overflow	1 when the last operation is overflow
20	C – Carry	1 when 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	Points to the current register window.

Table VI. Contents of the Trap Base Address Table

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)	

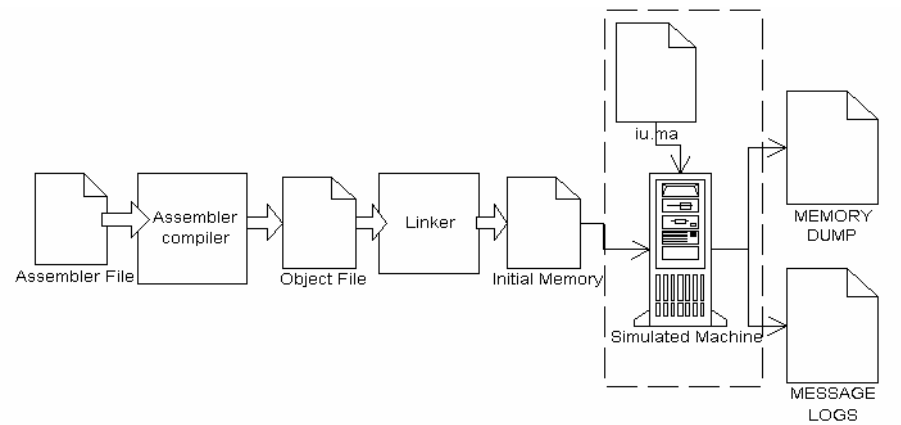

Fig. 18. Process of simulation from the assembler file to the memory dump.

assembled and linked. The resulting executable file must be loaded as a memory image in the simulated computer's memory. Then the simulation tool should be activated. Once the program has ended, the memory contents can be dumped and the memory values can be checked. While the simulation is executing, a detailed log file records component activity, allowing the analysis of changes in the state variables contained in the memory, processor, or bus components. Figure 18 sketches these procedures.

The first lines in Figure 19 show parts of a program written in the SPARC Assembly Language. The second part (Initial Image) gives binary code generated when assembled, along with the absolute addresses for each instruction or data (one word each).

```
! Partial and special copy of a string. r1 : offset over source
! r2 : character to copy. r3 : offset over dest
cycle:  lduh [%r1+source], %r2               ! Load unsigned halfword
        jmpl routine, %r6                    ! Jump and link
        sth %r2, [%r3+dest]                  ! Delay slot
        inc 2, %r1                           ! next character
        inc 4, %r3                           ! next position over dest
        subcc %r1, 12, %r0
        bne cycle                            ! Repeat the Cycle 6 times
        nop
        unimp
routine:jmp %r6+8
        st %r1, [%r3+dest2]                  ! Delay Slot
source: .ascii "xxxx String of text xxxx"    ! Source string
dest:   .ascii "                        "
dest2:  .ascii "                        "
        .ascii "                        "
        .ascii "                        "
        .ascii "                        "
Initial Image
Addr.  Memory Image                          Interpretation
...
032     11000100 00010000 01100000 01001100  load character
036     10001101 11000000 00100000 01000100
040     11000100 00110000 11100000 01100100  save the character
044     01000100 00000000 01100000 00000010  inc 2 to r1
048     10000110 00000000 11100000 00000100  inc 4 to r3
052     10000000 10100000 01100000 00001100  substract 12 to R0
056     00010010 10111111 11111111 11111010  relative jump
060     00000001 00000000 00000000 00000000  nop
064     00000000 00000000 00000000 00000000  unimp
...
076     01111000 01111000 01111000 01111000  x       x       x       x
080     00100000 01010011 01110100 01110010          S       t       r
084     01101001 01101110 01100111 00100000  i       n       g
088     01101111 01100110 00100000 01110100  o       f               T
092     01100101 01111000 01110100 00100000  e       x       t
096     01111000 01111000 01111000 01111000  x       x       x       x
100     00100000 00100000 00100000 00100000
...
Final image
...
076     01111000 01111000 01111000 01111000  x       x       x       x
080     00100000 01010011 01110100 01110010          S       t       r
084     01101001 01101110 01100111 00100000  i       n       g
088     01101111 01100110 00100000 01110100  o       f               T
092     01100101 01111000 01110100 00100000  e       x       t
096     01111000 01111000 01111000 01111000  x       x       x       x
100     01111000 01111000 00100000 00100000  x       x
104     01111000 01111000 00100000 00100000  x       x
108     00100000 01010011 00100000 00100000          S
112     01110100 01110010 00100000 00100000  t       r
116     01101001 01101110 00100000 00100000  i       n
120     01100111 00100000 00100000 00100000  g
...
```

Fig. 19. Execution of a simple routine.

This simple program makes a special copy of a string. We take two-character tokens and copy them to the destination string, separating each two-character token with two blank characters. The token copy is repeated six times. We show the translation of the binary code based on the specification of the instruction set of the SPARC processor. Finally, we show the memory (Final image) after program execution, where the values stored in memory have followed the instructions defined in the executable code.

Along with the memory maps, the execution logs generated by the CD++ tool allow the control flow in the processor datapath to be seen. Figure 20 shows parts of the log file generated when the program is executed. It includes the messages interchanged between executing models, and each message shows its type, timestamp, value, origin/destination, and the port used for the transmission. There are five types of messages: **I** (initializes the corresponding models); ***** (signals a state change due to an internal event); **X** (used when an external event arrives); **Y** (the model's output); and **D** (indicates that a model is done with a task).

The execution cycle starts by initializing the higher-level models (memory, CPU, etc.). When the message arrives at the CPU model, it is relayed to its lower-level components: instruction register, PC adder, PC multiplexer, control unit, etc. When the initialization cycle has finished, 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, meaning that the nPC value is 100100 = 36 (as we see in Figure 20, the program starts in address 32). The value is sent to the *pc-inc* model, which adds 4 to this register. The update is finished in 10:000 time units, 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.

The PC is activated and the value 010000 (32) is obtained afterwards. 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's arrival to the control unit, which activates several components: *a-mux*, *ALU*, *Addr-mux*, *IR*, and so on.

We finally see, at simulated time 20:000, that the memory has returned the first instruction (compare the results with the bit configuration stored in 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 a similar way. Following the log file, or connecting output models to the output lines in the processor, we are able to follow the execution flow for any program.

```
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: mem-
   ory,bus,CS,etc.
Message I/00:00:00:000/top(01) to bus(03)
...
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
Message  D/00:00:00:000/pc_inc(13)/00:00:10:000  to  CPU(05)  //  activation
   of
Message D/00:00:00:000/pc_latch(11)/00:00:10:000 to CPU(05)//pc-inc model
...
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/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 */00:00:00:000/top(01) to CPU(05) // Arrival to the CU and
Message */00:00:00:000/cpu(05) to cu(43)  // 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 */00:00:10:000/cpu(05) to pc_latch(11)
Message D/00:00:10:000/pc_latch(11)/... 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/top(01) to mem(02)//Memory returns the first inst.
Message Y/00:00:20:001/mem(02)/dtack/     1.000 to top(01)
Message Y/00:00:20:001/mem(02)/out_data2/     1.000 to top(01)
Message Y/00:00:20:001/mem(02)/out_data3/     1.000 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_data2/     1.000 to CPU(05)
Message X/00:00:20:001/top(01)/in_data3/     1.000 to CPU(05)
Message X/00:00:20:001/top(01)/in_data6/     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_data14/    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_data31/     1.000 to CPU(05)
Message D/00:00:20:001/bus(03)/00:00:00:001 to top(01)...
```

Fig. 20. Log file of a simple routine.

4. TESTING ALFA-1

Every time the simulated computer is modified, it is necessary to test the resulting changes. In order to improve the testing scheme, we include a testbench that can be applied to future modifications. The testbench consists of a set of test files and a tool that simplifies the detection of errors.

We used a functional testing approach [Beizer 1990], using black-box tests for each of the components. This enables us to test the execution of each of the models without knowing their internal representation. Black-box testing implies that the selection of test data, as well as the interpretation of test results, are based on the functional properties of the software. The primary objective is to assess whether the simulated computer does what it is supposed to do. We have built an experimental framework, consisting of a data generator connected to the model to be tested, and an acceptor that contains information about the desired output value for each one of the generated inputs. These tests were carried out by different groups of students that were provided with only the basic experimental framework, the model specification, and object code for the models. They were not provided with any information about the internal behavior of the models. The results of this testing phase were returned to the original development teams, who fixed the models that had problems. This is the suggested approach for any activity involving modifications to the existing models.

Once individual testing is finished, integration tests must be carried out. The developers of the control unit model carried out the first tests by running sample assembly programs developed as initial simulation test cases. A complete set of tests was then built to provide future developers with a way to test external behavior. In this case, the simulated computer model operates on a finite number of inputs that can be interpreted as a binary bit stream. A complete functional test would consist of subjecting the program to all possible input streams, defining the desired behavior to be obtained in each case. For each input, the model would accept the stream and produce a correct outcome; accept the stream and produce an incorrect outcome; or reject the stream. Because the rejection message is itself an outcome, the problem is reduced to verifying that the correct outcome is produced for every input. This approach was employed for many of the individual components, which have a bounded number of inputs and outputs. Nonetheless, even a short executable program of 10 bytes in length has 2^{80} possible input streams and corresponding outcomes. Hence, complete functional testing in this sense is impractical. For this reason, we have divided the possible inputs in classes and selected representatives of each class to test its behavior. If a representative for a given class fails a test, then an error over this class of inputs has been found.

We chose to use the instruction set as the source of information to divide the inputs. We built a set of programs executing only one instruction. For example, a program using the ***add*** instruction takes two source registers and stores the result in a third. There are different subclasses for each class; for example, we can execute this instruction using the registers *%r1*, *%r2*, and move the result to the *%r3* register. Other kinds of subclasses involve the use of the carry flag. Once the individual classes are tested, combination examples are executed to check interaction influences (Figure 21).

We defined the following test classes according to the SPARC instruction set (we show some examples of each class test).

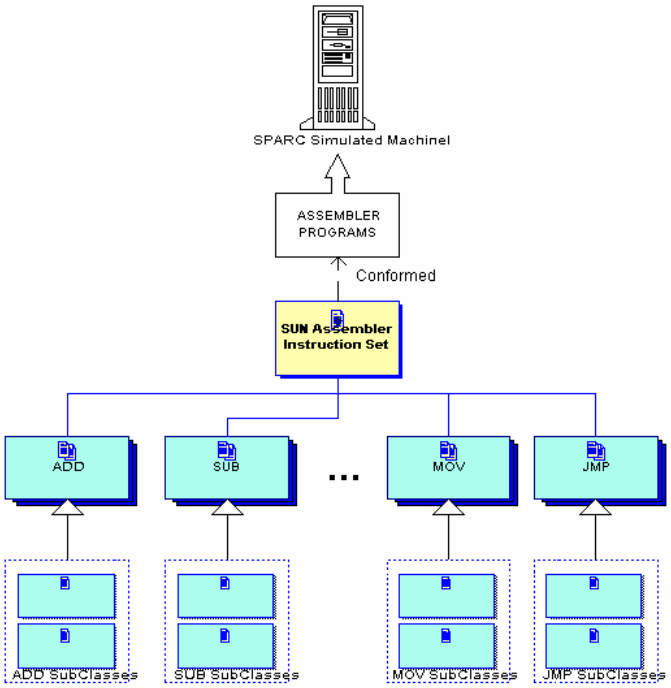

Fig. 21. Classes and subclasses in a simulated computer test.

Store

ST (STORE):

STREG, [ADRRESS]	∀ REG

MOV (MOVE)

MOV REG_a, REG_b	∀ REG_a, REG_b (REGISTER TO REGISTER)
MOV CONST, REG	∀ REG (MOVE A CONSTANT TO A REGISTER)

SET

SET LABEL, REG	∀ REG

Increment, decrement, add, subtract and jumps

INC, DEC, SUB, ADD, SUBCC, ADDCC, Bxx

INC CONST, REG	∀ REG, CONST={0,FFFFFFFF,..}
DEC CONST, REG	∀ REG, CONST={0,FFFFFFFF,..}
ADD REG, CONST, REG	∀ REG, CONST={0,FFFFFFFF,..}
ADD REG, REG, REG	∀ REG, CONST={0,FFFFFFFF,..}
ADDCC REG, CONST, REG	∀ REG, CONST={0,FFFFFFFF,..}
ADDCC REG, REG, REG	∀ REG, CONST={0,FFFFFFFF,..}
SUB REG, CONST, REG	∀ REG, CONST={0,FFFFFFFF,..}
SUB REG, REG, REG	∀ REG, CONST={0,FFFFFFFF,..}
SUBCC REG, CONST, REG	∀ REG, CONST={0,FFFFFFFF,..}
SUBCC REG, REG, REG	∀ REG, CONST={0,FFFFFFFF,..}

Bxx ∀ Bxx, CONST1 {<,=,>} CONST2

Example for each Bxx:

```
                MOV     CONST1, REG1
                MOV     CONST2, REG2
                CMP     REG1, REG2
                Bxx     LABEL_YES
LABEL_NO:       MOV 0, REG3
                BA LABEL_END
LABEL_YES: MOV FFFFFFFF, REG3
LABEL_END: ST REG3, [RES]
```

Load and Store

LD, ST	
LD REG, [LABEL]	∀ REG
LD, ST	∀ REG1, ∀ REG2, CONST en [0..n]

Example for LD and ST:

```
                LD      REG1, [LABEL]
                MOV     CONST, REG2
BEGIN:          ST      REG1, [LABEL], REG2
                LD      REG1, [LABEL], REG2
                DEC     REG2
                CMP     REG2, %G0
                BNE     BEGIN
```

EXCLUSIVE OR and AND Operations

XOR, AND, OR, ANDN, XORcc, ANDcc, ORcc, ANDNcc

XOR, AND, OR, ANDN, XORcc, ANDcc, ORcc, ANDNcc	∀ Bxx, ∀ REG1, ∀ REG2 CONST1 {=, <>, <, >, = not } CONST2

Example for each OP:

```
                MOV     REG1, CONST1
                MOV     REG2, CONST2
                OP      REG1, REG2
                Bxx     LABEL_YES
LABEL_NO:       MOV REG3, 0
                BA LABEL_END
LABEL_YES: MOV REG3, FFFFFFFF
LABEL_END: ST REG3, [RES]
```

Shifts

SLL, SRL, SRA ∀ REG1, ∀ REG2, CONST2 en [0..31]

Example:

```
MOV CONST1, REG1
OP          REG1, CONST2, REG2
ST          REG1, [ORIGIN]
ST          REG2, [DESTINATION]
```

MUL, MULScc, UMUL, UMULScc, DIV, DIVScc, UDIV, UDIVScc " CONST1/2, REG1/2/3

Example:

```
MOV CONST1, REG1
MOV CONST2, REG2
OP          REG1, REG2, REG3
ST          REG1, [A]
ST          REG2, [B]
ST          REG3, [RESULT]
```

```
[sth]
! The sum between registers r1 and r2  and we keep the result in memory

set 2, %r1             !Set the register r1 with 10
set 3,   %r2           !Set the register r2 with 5
add %r1, %r2, %r3      ! Add register r1 and r2 and store the result in r3
st %r3, [dest]         ! Move the result to the memory

unimp
.align 4
dest: .word FFFFFFFF
```

Fig. 22. Add two registers and store the result in memory.

In each of the classes we used a black-box testing approach, based on the construction of an oracle [Howden 1981]. An oracle is a program, process, or body of data that specifies the expected outcome of a set of tests as applied to an object. We used an oracle that specifies whether the expected outcome for a specified input has occurred. The oracle is based on the execution of existing programs generated using assembly language. We know the results for each of these programs because we can reproduce them in a real processor.

For instance, if we have the program defined in the source file in Figure 22, and we assemble and execute it (in a SPARC processor or in the simulated computer), the result obtained should be in register %r3.

The programs belonging to each of the classes are written in assembly language source code, assembled, and then run in a SPARC processor; the results are stored. The assembled programs are then executed in Alfa-1 and the execution results compared with SPARC results. We generated about 100,000 assembly language source programs, with the corresponding expected results. Each of them includes individual operations and combinations of operations that test a combination of all of the proposed classes.

For instance, to test the instruction *BEQ* (jump if equal), we defined examples that represented the *BEQ* class of tests. We used 279 different cases. The assembler code sets two registers with random values and then compares them. In this way, we can predict that the result is a jump to a label, the instructions located in the memory position associated to that label will set a variable which can be checked by analyzing the final memory dump.

In the example shown in Figure 23, the registers are not equal, hence, we expect a result of 0 at the *[dest]* variable in the final memory dump. In (1), the result of comparing both numbers does not result in a jump, as the numbers are not equal. The following operation (2) loads register 10 with 0, and then executes a jump (3) to **LABELEND** where the result is stored in memory (4).

We created a test file for each of the classes. The expected result file includes results using the following format: `VALUE: data type, integer number`. In this example, we include `VALUE: int32, 0`, meaning that we expect an int32 number with value 0 at the memory dump. After testing this file, the result is `File:beq1.TEST OK`, meaning that no error was found for this instruction. After running all the tests, a table was generated with the contents shown in Table VII.

```
set 82,  %r26                  !Set the register 26 with 82
set 543854, %r1                !Set the register 1 with 543854
cmp %r26, %r1                  !Compare the registers
beq LABELYES    (1)            !If they are equal save the result in mem-
   ory
NOP
LABELNO: mov 0, %r10    (2)    !if the numbers are not equal save the
                               !result in memory
         ba LABELEND    (3)
         NOP
LABELYES:      mov 4294967295, %r10   !Equal set FFFF FFFFh
LABELEND:      st %r10, [dest] (4)    !Save the result in memory

unimp
   .align 4
   value:   .ascii "VALUE:"    !The tester will look for the label VALUE
   at
                               !the memory dump
   dest:    .word  FFFFFFFF    !Result of the test
```

Fig. 23. Assembler File *beq1.s*.

Table VII. Table Output Format for the Test Application

Test Number	*Test Class*	*Test Subclass (if any)*	*Number of tests*	*Number of successful tests*	*Bug file*
1	ADD	Add with carry	100.000	100%	
		Add two negatives	100.000	100%	
		Add two positives	100.000	100%	
		Add neg and pos	100.000	100%	
		Add to zero	100.000	100%	
2	BEQ	Numbers are equal	200.000	100%	
		Numbers are not equal	200.000	100%	
3	DIV	Negative numbers	100.000	99%	Divr1-r2.s
…	…	…	…	…	…
N	INC	Force carry	100.000	100%	
		Normal inc	100.000	100%	

This procedure allowed us to find some errors in the coupled model representing the simulated computer. For instance, we found that the division instruction and two condi tional jumps were not working properly. By tracing the execution flow of the programs, we found the source of the errors, which were fixed in less than 8 man/hours.

5. EXECUTION VISUALIZATION

We have begun to develop a GUI to allow students to interact with the simulated computer. The GUI will let users check the system state at any moment. In this section we briefly present an approach to doing so. The GUI is also a source for experimental work, as the definition of a GUI requires detailed knowledge of system architecture. These activities can be used as part of programming courses, or in any existing course on human-computer interaction. It can also be a source for exercises in compiler design

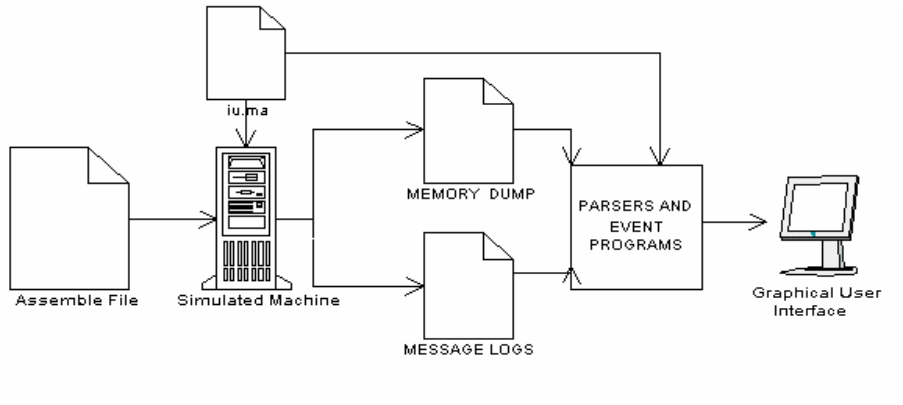

Fig. 24. Dynamic process used to generate a GUI.

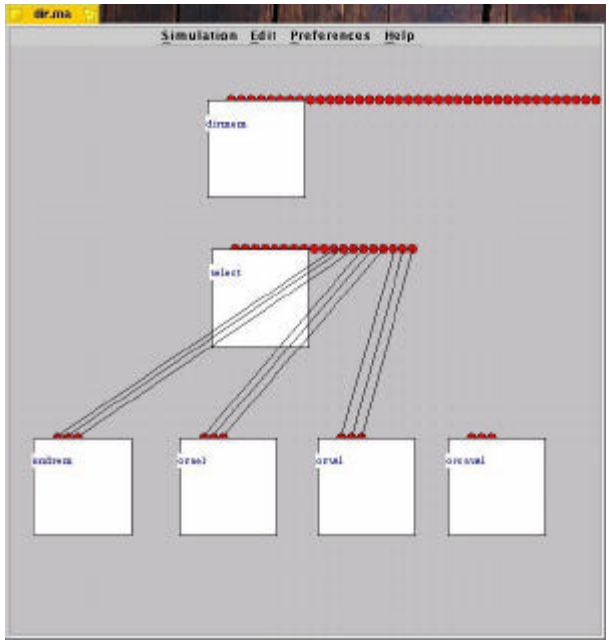

Fig. 25. Snapshot of the GUI.

courses, since parsing log files and analyzing the different circuits and lines activated at each moment are required.

Different types of graphical interfaces can be associated with each layer; for instance, a debugger-like interface for the instruction set level, a diagram representing

the datapath at the microprogramming level, or block diagrams representing boolean gates at the digital logic level.

The log files, which contain the simulation information on every level of abstraction, are the information source for any graphical interface. Any interface developed must be adaptive enough to support changes in the simulated architecture. The basic design of Alfa-1 includes a coupled model of the integer unit (***iu.ma***), defining all the components of the computer. Whenever we need to add a new component, we must include it in the ***iu.ma*** file.

We believe that the GUI must be adapted to new simulated machines and must be able to display information associated with every component in the machine. As shown in Section 3, the simulator generates a log with messages that can be used to show how a program executes. Each message has a source, a destination, and a value representing changes over the component output lines. This information can be used to analyze the present state of any given component. The GUI makes two passes over the log files. In the first pass it automatically creates the ports and links defined for the model. In the second pass the GUI uses the references to the components in order to decide which component is activated, according to the information in the log files.

A second level of interfacing is associated with the computer architecture. We provide a debugger-like interface, allowing interaction in a user-friendly way. Starting with the debugger, the user is able to study lower abstraction levels using different visualization tools. We will provide wizard-like interfaces to help users extend the existing components.

6. EDUCATIONAL ACTIVITIES

In this section we present a set of educational activities that can be carried out using the simulated computer. In order to achieve our goals, we have organized our course according to the different levels provided by Alfa-1. The tool should be used as pedagogical support, as a complement to any of the existing bibliographical references. According to the chosen materials, teaching them could be done top-down (assembly language to digital logic) or bottom-up. Nonetheless, we prefer to use a middle approach, where we first describe the computer organization level and then move to different levels of abstraction.

Once the theoretical concepts on computer organization are introduced, Alfa-1 can be used as a support tool, as it enables analyzing the information flow on the datapath. The following are some possible activities that may be carried out:

1. Given an executable file consisting of only one instruction as memory image, students are required to identify the different components activated by the simulator at the organization level (different executable files can be provided to different students; for instance, the image files in the test directories could be used).
2. Given a log file from which some lines have been deleted, students are required to fill in the missing lines. The log file shows the processor execution flow, and the missing lines correspond to the execution of an instruction at the organization level.

3. Given a complete executable file, students are required to generate an execution log and analyze the execution flow in the datapath by studying the circuits and lines that are activated.
4. Starting with an algorithmic description of the tasks carried out by a complete executable file, create a log file with missing lines. Request that students fill out the missing parts according to the tasks performed by the executable program used to generate the log file.

At this level, we can also propose a number of development tasks associated with modifications in the underlying structure of the computer organization. Some of these activities could include the following:

1. replacing or improving existing components in order to enhance functionality;
2. defining more detailed models of the main memory, including different interleaving strategies;
3. constructing models of input/output devices and connecting them to the system bus;
4. developing a floating point unit and integrating it to the processor;
5. defining virtual memory strategies, and developing a memory management unit (MMU) able to handle virtual memory requests. The MMU must be able to generate fault interrupts;
6. implementing different replacement algorithms in the cache memory;
7. changing the datapath structure, or modifying the number of registers or circuits used. Students can execute a performance analysis of the resulting architecture;
8. improving the performance of some of the existing circuits (for instance, providing single-level atomic models that replace complex coupled ones);
9. defining models of a cache directory using direct mapping techniques. New atomic and coupled models should be created by following the specifications of the current model. New replacements algorithms can be added by modifying the existing FIFO strategy. Once the different strategies are implemented, students can perform benchmarks to measure the performance of the new solutions (for instance, analyzing the hit ratio over the cached cells when running different programs).

After these activities are carried out, students will know the details of the organization layer of Alfa-1. The next steps depend on the general objectives of the course. If study of the digital logic level is a goal, then boolean logic concepts can be applied in different activities, including:

1. analyzing the circuit activation: some of the circuits built using digital logic concepts can be studied in the log files, enabling behavior analysis under different inputs;
2. understanding boolean logic concepts: by providing incomplete log files representing some of the circuits, the students can fill out the missing lines, improving their skills in digital logic analysis.

At this level, we can also propose different development tasks, including:

1. defining new atomic models representing different boolean gates than the ones in the toolkit. Using the new gates, students can build coupled models to redefine existing circuits using different combinatorial rules;
2. extending the existing models to a lower level of abstraction. The component is replaced by a structural model using boolean gates (for instance, use the ALU model and redefine it using the ideas in the *CMP* model in the distribution files);
3. analyzing the interlevel interaction: students should analyze the execution of an instruction and consider their interactions at both levels. In previous assignments, we considered only the organization level. Now we can study how the execution flow in the datapath is implemented at the digital logic level. This can be done following the detailed execution of one instruction that, in the end, activates some of the submodels defined using digital logic.

Besides the digital logic level (which may not be included), we can consider some activities at the microprogramming level. After facing a complete study of microprogramming techniques, the CU model can be used as an example of microcoded architecture. The exhaustive analysis of the input/output lines in the control unit and its internal behavior can be used to understand the micro-operations required to execute every instruction. Details of the microprogrammed architecture can be found in Daicz et al. [1998]. Activities at this level require the introduction of some details at the instruction set level, in order to make clear the use of the microinstructions carrying out processor execution. A very reduced set of instructions of different categories can be presented at this stage (for example, one arithmetic, one control flow and one data transfer) in order to show their implementation.

Activities at this level can improve understanding of interlevel interactions. We first describe the behavior of an existing instruction and then show its implementation in microcode. The microinstructions activate different circuits, and produce data transfers at the organization level. Some of these activities are carried out at the digital logic level. After finishing these activities through detailed practice, the students will know each of these layers and their global interaction in detail.

Some development activities that might be considered at this level include:

1. modifying the control unit structure to introduce pipelining techniques. This involves reorganizing the activation of subcomponents defined as executing simultaneous events;
2. executing the models in parallel, using the parallel version of CD++ in order to reproduce instruction-level parallelism;
3. modifying the internal structure of the CU model in order to implement equivalent FPGA circuits.

Every computer organization course must include different activities at the level of the instruction set. The most important concepts here include the definition of the existing instructions, registers, addressing modes, and input/output functions. These activities are

usually related to the definition of the assembly language level (which, in general, provides one-to-one translations from mnemonic representations into machine language). An important aspect to be explained at this level includes the definition of instruction encoding.

Once the students know the basic concepts related to the architecture level, Alfa-1 can support the teaching of concepts associated with this level of abstraction. Some possible activities include the following:

1. defining applications using the SPARC assembly language in order to expose the details of the processor features;
2. executing trace analysis: using an initial image of a given program and the executable code for a given program, analyze changes after executing each program instruction. This involves changes in memory, registers, cache, and input/output devices;
3. defining interrupt service routines;
4. encoding instruction: study which circuits are activated at each step, according to the encoding of each instruction.

At this level, we propose development tasks that involve modifying the architecture:

1. implementing a MMU at the architecture level, providing virtual addressing schemes (paging, segmentation, demand paging, etc.);
2. defining new instructions to select different replacement algorithms in the cache and the MMU;
3. defining new instructions enabling the activation of individual circuits in the architecture;
4. developing input/output routines using polled loops;
5. developing input/output routines interacting with the look-aside components, analyzing their execution results; and
6. modifying the instruction set, adding new registers, modifying or adding instructions or addressing modes.

Once students finish with activities at this level, they will have a detailed knowledge of each of the levels in a given architecture, and will be ready for more advanced studies. In some cases, the educators will need to address advanced activities. Alfa-1 can be used successfully to reach these goals; and here we show some possible advanced projects using this tool:

1. developing an assembler and a linker that are able to generate instructions encoded for the SPARC processor;
2. reusing existing components to define different architectures (i.e., Motorola, Intel, MIPS, etc.);
3. implementing different input/output processors and models of the devices they control. Different synchronization mechanisms can be included (polling, interrupts, or DMA);
4. defining a SPARC multiprocessor architecture (tightly or loosely coupled), defining multiple instances of the Alfa-1 top model. This requires defining an in-

terconnection mechanism and an information transfer technique (shared memory, message passing);

5. including advanced architectural features in the control unit (multiple stage pipelining, predictive jumps, floating-point vector processors, etc.);
6. defining a microprogramming language whose interpreter is defined as a part of the CU model. The CU should be modified to read microinstructions that can be defined in external files, and to enable students to define new instructions through microprogramming. Add the corresponding instructions to the instruction set and test the correct execution of the instruction.

7. CONCLUSION

We have presented the design of a simulated computer that can be used in computer organization courses to analyze and understand the basic behavior of the different levels of a computer system. The use of DEVS allows us to have reusable models. DEVS also provides a uniform point of view in all layers of description, as every model developed is defined using a single technique. We modeled different levels of abstraction: digital logic, microcode, and the instruction set levels. The models can be used to study interaction between levels and to evaluate systems experimentally. Students can analyze each of the layers in detail by studying the behavior of the components belonging to the layers. Since the Alfa-1 simulator uses all the existing layers, system behavior as a whole can be studied also. Starting with the instruction level, we can analyze the activation flow in the datapath and the reaction of individual components to see how the instruction is carried out.

We have achieved our goals in each of the categories explained previously:

I. *Levels of abstraction*: we were able to describe all the levels of abstraction in the bibliography of the area, ranging from digital logic to the assembly language level. We used a unique approach (the DEVS formalism and a DEVS modeling tool) to achieve these goals.
II. *Pedagogical value*: every component in the simulated computer was developed by undergraduate students in assignments of a computer organization course. The students took a previous course in computer programming. The learning curve included spending some time explaining the basic aspects of DEVS and using the existing toolkit to develop extensions to the original architecture. Initially, a group of 3rd year students of a discrete event simulation course created the formal specifications of the models. These specifications were then used by students in a 2nd year computer organization course to build and test the models using the CD++ tool. Final integration was planned by a group of undergraduate teaching assistants (who also developed the control unit and the coupled model representing the system architecture shown in Figure 1). Individual and integration testing was also done by 2nd year students. All of the modifications shown here were developed as course assignments. The toolkit is public domain, and more details about the implementation, the basic tools used to develop the models, and further improvements can be found at

<http://www.sce.carleton.ca/faculty/wainer/alfa-1.html>.

III. *Modifiability*: we started with a simple version of the simulated computer. This version was extended, the digital logic level was included, and high performance facilities (such as a cache memory) were added without any special effort. The implementations of the formally specified DEVS models were straightforward.
IV. *Advanced architectural facilities*: the tool is based on the description of a SPARC processor, one of the most powerful architectures today. The hierarchical definition of the tool enables the users to experiment with extensions and architectural enhancements based on the redefinition of the existing models.

The use of this tool allowed the students to obtain a complete understanding of computer organization. After using Alfa-1, students in upper-level courses reported higher success rates and detailed knowledge of the subjects.

At present, the set of tools is being completed by including an input/output subsystem. The main input/output devices, the input/output interfaces, DMA controllers, and channels will be simulated. Different transference techniques (polling, interrupts, DMA) will be considered to implement input/output electronics. Other ongoing tasks include finishing the definition of the graphical interface and the implementation of different cache management algorithms.

ACKNOWLEDGMENT

We want to thank to A. Troccoli and S. Zlotnik, who collaborated in the initial efforts of this project. Likewise, several students have actively participated in the development of the models: S. Enrique, E. Glinsky, F. Petronio, D. Altman, P. Barletta, J. Barrionuevo, A. Bender, D. Brignardello, A. Calvo, A. Corvetto, P.Cremona, C. Diuk, D. Grinberg, M. Nuñez Cortés, D. Rubinstein and P. Sor. Finally, we want to thank T. Pearce (SCE Department, Carleton University) who helped us in preparing the final version of this manuscript.

REFERENCES

ANDERSON, K., BUTTRON, J., CLARKE, P., AND ENWALD, M. 1999. WOOKIE: A 68HC11 Emulator *Dr. Dobbs J.* 24, 3, 50-55.

BABAOGLU, O., BUSSAN, M., DRUMMOND, R., AND SCHNEIDER, F. 1983. Documentation for the CHIP computer system. Tech. Rep. TR83-584, Computer Sciences Dept., Cornell Univ., Ithaca, NY.

BEDICHEK, R. 1995. Talisman: Fast and accurate multicomputer simulation. In *Proceedings of the SIGMETRICS'95 Conference* (Ottawa, Ont., Canada).

BEIZER, B. 1990. *Software Testing Techniques*, 2nd ed. Van Nostrand Reinhold, New York, NY.

BEVILACQUA, R., GOMEZ, L., AND GOMEZ, S. 2000. The PROVIR virtual processor. M. Sc. Thesis, Dep. de Computación, Facultad de Ciencias Exactas y Naturales, Univ. de Buenos Aires (in Spanish).

BREWER, E.A., DELLAROCAS, C.N., COLBROOK, A. AND WEIHL, W.E. 1991. PROTEUS: A high-performance parallel-architecture simulator. Tech. Rep. TR-516, MIT / LCS, Laboratory for Computer Science, MIT, Cambridge, MA.

BURGER, D., AND AUSTIN, T. 1997. The SimpleScalar tool set. Version 2.0. *Comput. Architecture News 25*, 3, 13-25.

BURNS, M., GEORGE, A., AND WALLACE, B. 2000. Modeling and simulative performance analysis of SMP and clustered computer architectures. *Simulation 74*, 2, 84-96.

CAMPENHOUT, J., VERPLAETSE, P., AND NEEFS, H. 1998. ESCAPE: Environment for the simulation of computer architectures for the purpose of education. In *Proceedings of the Workshop on Computer Architecture Education* (Las Vegas, NV).

COE, P., WILLIAMS, L., AND IBBETT, R. 1996. An interactive environment for the teaching of computer architecture. In *Proceedings of the Annual Joint Conference Integrating Technology into Computer Science Education* (Barcelona).

DAICZ, S., TROCCOLI, A., ZLOTNIK, S., AND WAINER, G. 1998. Architectural definition of the ALFA-1 simulated processor. Internal report. Computer Science Dept., Univ. de Buenos Aires. http://www.sce.carleton.ca/faculty/wainer/alfa-1.html.

.DEITZ, H., AND ADAMS, G. 1994. CASLE (Compiler/Architecture Simulation for Learning and Experimenting). Online report. http://purcell.ecn.purdue.edu/~casle/Index.html.

DE SIMONI, L., ENRIQUE, S., GLINSKY, E., PETRONIO, F., WASSERMANN, D., AND WAINER, G. 1998. Definition of components for the ALFA-1 simulated processor. Internal report. Computer Science Dept., Univ. de Buenos Aires (in Spanish). http://www.sce.carleton.ca/faculty/wainer/alfa-1.html

EDMONSON, J., AND REILLY, M. 1998. Performance simulation of an ALPHA microprocessor. *IEEE Computer 31*, 5, 50-58.

EL HAJJ, A., KABALAN, K., MNEIMNEH, M., AND KARABLIEH, F. 2000. Microprocessor simulation and program assembling using spreadsheets. *Simulation 75*, 2, 82-90.

GHOSH, S. 2000. *Hardware Description Languages: Concepts and Principles*. IEEE Press.

HEIN, A., AND DAL CIN, M. 1998. Performance and dependability evaluation of scalable massively parallel computer systems with conjoint simulation. *ACM Trans Model. Comput. Simul. 8*, 4, 333-373.

HENNESSY, J. L. AND PATTERSON, D. A. 1996. *Computer Architecture: A Quantitative Approach.* 2nd ed. Morgan Kaufmann, San Francisco, CA.

HEURING, V., AND JORDAN, H. 1997. *Computer Systems Design and Architecture*. Addison-Wesley, Reading, MA.

HOWDEN, W. E. 1981. A survey of dynamic analysis methods. In *Software Testing and Validation Techniques 2*nd ed., E. Miller and W. E. Howden, eds., IEEE Computer Society Press, New York, NY.

IKODINOVIC, I., MAGDIC, D., MILENKOVIC, A., AND MILUTINOVIC, V. 1999. Limes: A multiprocessor simulation environment for PC platforms. In *Proceedings of the Third International Conference on Parallel Processing and Applied Mathematics* (*PPAM*, Kazimierz Dolny, Poland).

ISACOVICH, F., MISLEJ, E., WINTERNITZ, F., AND WAINER, G. 1999. An emulator of the Atari processor Internal Report, Computer Sciences Dept., Univ. de Buenos Aires (in Spanish).

MITSCHELE-THIEL, A. 2000. *Systems Engineering with SDL*. Wiley, New York, NY.

MORSIANI, M., AND DAVOLI, R. 1999. The MPS computer system simulator. Tech. Rep. UBLCS-99-*8*, Univ. de Bologna.

NGUYEN, A.-T., MICHAEL, M., SHARMA, A., AND TORRELLAS, J. 1996. The Augmint multiprocessor simulation toolkit for Intel x86 architecture computer design. In *Proceedings of the IEEE International Conference on VLSI in Computers and Processors* (San Antonio, TX), 486 –490.

PASTOR, E. AND SANCHEZ, F. 1997 The rudimentary computer: A pedagogic processor. In *Proceedings of III Jornadas de Ensenyanza Universitaria sobre Informatica* (*JENUI'97*, Madrid). (In Spanish).

PATTERSON, D. 1995. *Computer Organization and Design: The Hardware/Software Interface*, 2nd ed. University of California, Berkeley.

PEARCE, T. 2000. Notes on p86 Assembly language and assembling. 2000. Internal report, Dept. of Systems and Computer Engineering. Carleton Univ. http://www.sce.carleton.ca/courses/94201/.

RODRIGUEZ, D. AND WAINER, G. 1999. New extensions to the CD++ tool. In *Proceedings of the SCS Summer Computer Simulation Conference* (Chicago, IL), 1-6.

ROSENBLUM, M., BUGNION, E., DEVINE, S., AND HERROD, S. 1997. Using the SimOS machine simulator to study complex computer systems. *ACM Trans. Model. Comput. Simul. 7*, 1, 78-103.

SHANMUGAN, K., FROST, V., LA RUE, W. 1992. A block-oriented network simulator (BONeS). *Simulation 58*, 2.

SHEALY, A., MALLOY, B., AND SYKES, D. 1997. An extensible simulator for the Intel 80x86 processor family. In *Proceedings of the 30th Annual Simulation Symposium* (Atlanta, GA. April), 157-166.

SKRIEN, D. 1994. CPU Sim: A computer simulator for use in an introductory computer organization class. *J. Comput. Higher Education 6*, 1, 3-13.

STALLINGS, W. 1999. *Computer Organization and Architectre*, 4th ed. Macmillan, New York, NY.

SUN MICROSYSTEMS. 2001. *SPARC Assembly Language Reference Manual*. http://docs.sun.com/.

TANENBAUM, A. S. 1999. *Structured Computer Organization*, 4th ed. Prentice Hall, Englewood Cliffs, NJ.

THOMAS, D., AND MOORBY, P. 1991. *The Verilog Hardware Description Language*. Kluwer Academic, Boston, MA.

WAINER, G. 2001. Experiences with DEVS modelling and simulation. *IASTED J. Model. Simul*. (March).

WAINER, G., CHRISTEN, G., AND DOBNIEWSKI, A. 2001. Defining DEVS models with the CD++ toolkit. In *Proceedings of the European Simulation Symposium* (Marseilles).

WAINER, G., DAICZ, S., AND TROCCOLI, A. 2002. Experiences in modeling and simulation of computer architectures in DEVS. *Trans. Soc. Comput. Simul.* (To appear).

WAINER, G., AND TROCCOLI, A. 2001. *CD++ User Manual*. http://www.sce.carleton.ca/faculty/wainer/celldevs.

ZEIGLER, B., PRAEHOFER, H., AND KIM, T. 2000. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press.

ZIDLICKY, R. 2002. QL FAQ and resources pointer. Online report. http://www.geocities.com/SiliconValley/bay/2602/ql.html.

Received November 2001; accepted February 2002