A Teaching Environment to Model and Simulate Computer Processors

Sebastiano PIZZUTILO and Filippo TANGORRA
Dipartimento di Informatica
Università degli Studi di Bari
via Orabona 4, 70126 Bari
ITALY

Abstract: - The paper describes a system to design computer processors and to simulate their behaviour during the execution of assembly user programs. The system, called APE (Architecture Prototyping Environment), is based on a dynamic object oriented definition and use of processor components. After the user choice of the architecture components, the system builds a processor simulator allowing users to study the processor behaviour. The iterative process of modelling the computer architecture, prototyping the corresponding simulator and simulating its behaviour, makes the system particularly useful in the activity of teaching computer architectures.

Key-Words: -Computer simulation, Object oriented prototyping, Teaching environment, Rapid prototyping.

1 Introduction

Computer architecture design and learning both need to proceed in an evolutionary way, beginning with an initial processor design and introducing progressively in the design process or in the learning process the complex organisation of a processor architecture. In fact, the goal is reached by means of a refinement process which starts from a working model, then investigates and evaluates the system features, modifying the initial model or adding further details. The effectiveness of the process lies in the capability of the software tool to investigate and evaluate the computer architecture design and its components.

Generally, simulation is an efficient tool for both a clear comprehension of the functioning of complex systems and the exploration of new solutions to avoid building real systems. In the educational field, computer architecture simulators are adopted in laboratory activities in order to clarify theoretical concepts and to provide access to the internal processor state.

Processor simulators remain the best tools in architecture study, due to their ability to meet different requirements, such as i) focusing only on the main characteristics of complex commercial architectures; ii) reducing costs of developing a simulator, as regards hardware implementation costs in the prototyping activity; iii) reducing the difficulty of following technological progress in updating laboratories with real machines and so on.

Among the best known processor-specific simulators in the didactic field is SPIM [1], which implements Hennessy and Patterson's MIPS R2000/R3000 RISC processors [2]. SPIM is used extensively in university courses.

The choice of the processor for teaching computer architecture depends on the specific learning needs [3]. Generally, current simulators represent specific architecture types, such as RISC or CISC, on a specific hierarchical level of the computer organisation, such as a simplified ISA processor [4] or micro-programming [5, 6] or gate [7] levels .

The drawback of these experiences is that the architectural characteristics cannot be changed. This is critical because, when the computer architecture is fixed, the assembly language that allows the simulator to use the defined architecture, is also fixed [8]. Therefore, if the architecture is modified, the computer simulator must also be re-designed in order to match the new characteristics with the assembly language.

We have developed a flexible tool which allows the user to define, test and dynamically modify computer architectures using the rapid prototyping paradigm.

In this paper, we present the APE (Architecture Prototyping Environment) system, which allows the rapid prototyping of virtual computer architectures with the associated simulator.

In the software development field, the rapid prototyping paradigm essentially consists of building a software system on the basis of user requirements and then of the user evaluation of the prototype. An appropriate interface allows the user to verify whether the prototype fits the design specifications, otherwise new user requirements are formulated to build a new prototype. In this way, the software system is obtained by refining initial requirements until they correspond to user needs [9]. Similarly, APE is aimed at designing and simulating a computer architecture starting from the definition of architectural requirements. From these requirements, it is possible to obtain a software prototype that simulates the defined architecture and

allows the testing of the architecture with its assembly language, in order to verify the coherence between the prototype and the user requirements.

The computer architecture design is performed through an object-oriented approach which involves constructing a system by using objects that encapsulate properties and functions of basic hardware components, such as building blocks.

2 Object-Oriented Approach for the Architecture Component Description

In our approach the computer architecture design is viewed as a collection of hardware components (objects). Register types (program counter, stack pointer, general, index…), ALU, cache memory, storage locations are primitives with which we represent a computer architecture design. These primitives must be designed opportunely in order to support the rapid prototyping paradigm in simulating the computer architecture. The design has to be done by characterising the hardware component with a corresponding reuse of software components [10]. So, we have implemented the software components as objects [11].

Every object has attributes, whose values can also be other objects, and methods which represent the procedures that manage objects. All the objects that share the same attributes and methods are grouped in a class. The structuring of the so defined classes permits the definition of the properties of hardware components (objects) selected in the design process.

The figure 1 reports the main classes diagram of a CISC architecture in UML (Unified Modeling Language) [12], which is de facto a standard in the object-oriented software development.

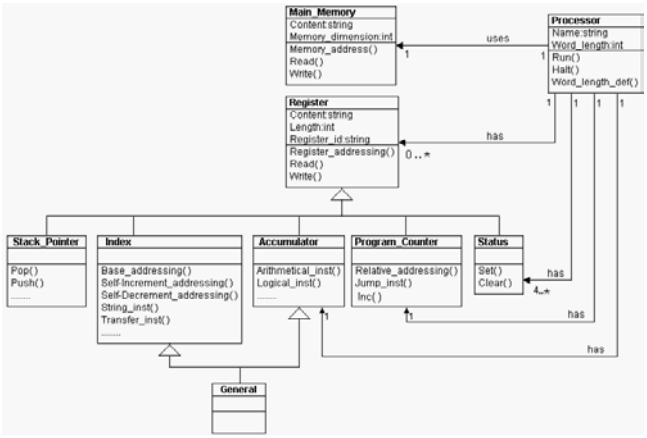

Fig. 1. The main classes of a CISC architecture at the instruction set level.

In our approach, the methods are *machine language instructions* or *addressing methods* associated with a specific architectural component or object. The *addressing methods* can be auto-consistent (for example in the case of immediate or direct addressing) or related to other objects (for example, in case of register or indexed addressing which require, to be executed, the reference of the content of other registers). In the first case, the addressing method is available at the creation of the object instance; in the second one, the method is available only in the presence of the previously defined objects.

The minimal architectural requirement is described in figure 1 by the mandatory participation of *Accumulator, Program_Counter* and *Status* classes in their respective relationships with the *Processor* class.

This approach is very flexible. In fact, it is possible to model a multiprocessor architecture by simply varying the multiplicity indicators "1" and "1" of the relationship "*uses*" between the *Memory* class and the *Processor* class in "1" and "*" or "*" and "*". In this way we can model the basic multiprocessor architecture with a shared memory and a multiprocessor with local memories respectively.

From an implementation point of view, this means that, in the case of multiprocessors with a shared memory (one-to-many relationship), it is possible to associate multiple instances (each with its own register set and its own language) of a *Processor* class to a single *Memory* class instance. In this case, the problem is the representation of multiple processors capable of processing multiple programs simultaneously. On the other hand, the simulator must provide (classes with) algorithms to manage the concurrency in the memory access and to schedule jobs which can be executed efficiently.

In the case of multiprocessors with local memory (many-to-many relationship), the main problem is the displaying of all the components of the independent multiple processors, while providing the same operating system functionalities as the previous case.

In either case, the object oriented approach seems to be able to adequately describe all the architectures (from SISD to MIMD) and to provide their rapid prototype development.

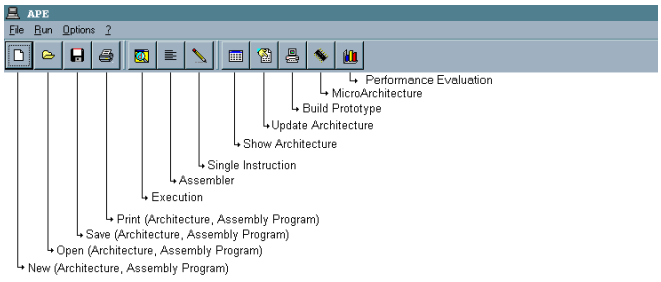

Fig. 2. The main panel of the system for a simulation session.

The set of object instances defined by the user corresponds to the conventional computer architecture;

and the set of activated methods is its assembly language. The user interacts with the system at an high level, by choosing objects which form the desired architecture. This task generates a simulator that is composed of objects selected by the user, while the corresponding methods define the assembler language to use the simulator

3 The APE environment for developing processor simulators

The design aim for APE was to provide teachers/students of the computer architecture courses with a tool that allows the rapid prototyping of processor simulators, which can be used in the laboratory activities. For this purpose, the system supports the processor simulator development providing two steps: the architecture *definition* step and the architecture *test* step. In the same way of the software rapid prototyping approach, APE agrees to repeat the previous steps until a satisfactory processor model has been produced.

This iterative process leads to a definition of the computer architecture closer to user's requirements; therefore it is possible to generate the code corresponding to the defined architecture. This code, when compiled and executed, will provide a simulator of the defined architecture, completely independent from the proposed tool.

3.1 The system overview

The system aims to provide students with a tool to define and simulate computer architectures, which can be used by running programs written in the corresponding language. In particular, the system does not represent specific processors at a specific computer organisation level. So, the user can model the target architecture and control the code execution by displaying the processor's state.

Figure 3 reports the main functions of the system with the meaning of the buttons.

The APE system supports the definition of a processor through an opportune selection of library objects and, at the same time, allows the computer architecture to be simulated using methods of the chosen objects.

The overall structure of APE provides two main subsystems (figure 4), which support the two different user-interaction phases: the *Architecture Definition Subsystem* (APE-ADS) and the *Architecture Test Subsystem* (APE-ATS).

The **APE-ADS subsystem** performs computer architecture definition by providing facilities for entering information on the design requirements. In the left side box of figure 3, the main components of this subsystem are reported.

The ***User interface*** supports three user interaction tasks: the *Architecture design*, the *Architecture updating* and the already defined *Architecture loading*.

The **A*rchitecture design*** task shows the user a set of panels associated with different menu options. Each panel structure corresponds to each object definition and, according to classes designed (fig. 1), it shows the public object methods representing the basic functionalities of the hardware components.

The operations available in the *Architecture design* task include the definition of RISC or CISC architectures by the choice of hardware components (the constituent objects) of the processor type [11].

Once the processor type has been decided, the user models the architecture by selecting, from the associated objects presented in the menu and loaded from the *Data Base of hardware components*, those that will constitute the components of the prototype at the conventional machine level (memory, registers, stack,…) with the desired instruction set and address methods.

For CISC architectures, it is also possible to define the processor components directly at the microprogramming level. In this case, the user chooses the hardware elements according to the set of

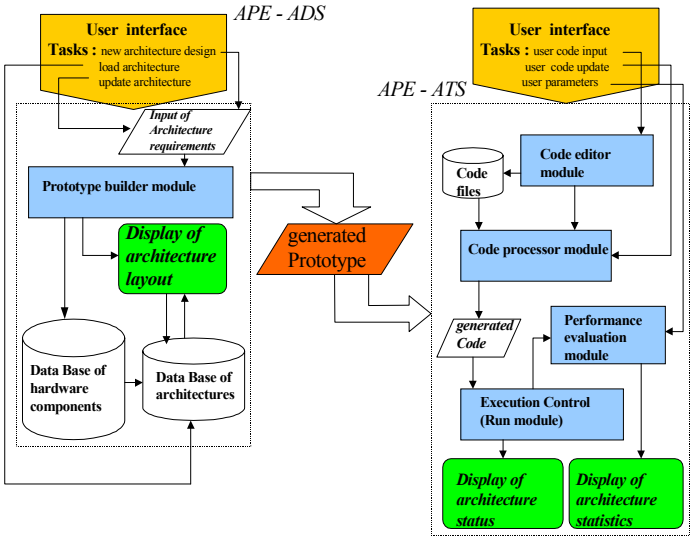

Fig. 3 The APE architecture

predefined classes, indicating the layout of the objects (the register file, the latches, the MIR,…) he/she wishes to include in the computer architecture. At the same time, the user, looking at the components represented in the processor layout, can define the specific parameters of the selected object through a menu-driven dialogue.

The user, filling-in the objects in the panel of this task, will specify the ***Architecture requirements*** that allow the system to simulate the processor defined.

The ***Prototype builder*** module creates the prototype that, starting from the architecture requirements, represents the processor simulator. The Prototype builder uses the Data Base of the hardware components that contains the constituent basic object of RISC and CISC architectures.

The objects populating the Data Base of hardware components are organised in three sets of C++ simulation primitives, representing three built-in module libraries for RISC, CISC and the microprogramming architectures respectively.

The Prototype builder, on the basis of the parameters specified by the user's choice, searches the built-in hardware components in the database and loads the selected components and their methods at the conventional and eventually at the microprogramming level. Therefore, at the conventional level, this module creates a file of hardware objects with their associated addressing methods and the instruction set for their manipulation.

The system creates the file of the hardware components and, for every machine instruction of the defined architecture, the corresponding microprogram. The set of microprograms simulates the processor control storage.

The simulation model can be tested by running the object code generated by a standard compiler, which processes the file of C++ components. The object code is then linked with an executable code, to allow the prototype execution within the APE-ATS subsystem. Alternatively, the code can be linked with an executable code for using the prototype as an off-line computer simulator, for example, by students in laboratory activities, when the defined architecture meets the user's requirements. In this case, the prototype does not allow further modifications of architectural components.

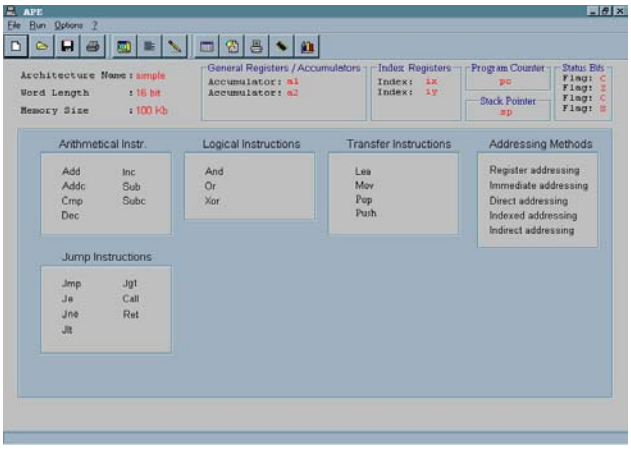

Fig. 4 The architecture layout

The effect of the design process is displayed by the Prototype builder in an ***Architecture layout*** that shows the processor components and actually fires instructions and address methods of the prototype generated. Figure 4 shows an example of a simplified architecture layout of a prototype simulator, ready for the testing phase.

The files containing the built-in classes selected, instantiated by user parameters in the design process and the associated executable simulator are stored in the ***Database of architectures*** for successive updating and testing activities.

In the ***Architecture load*** task, the user has to specify the name of the file in the Database of architectures which contains the prototype to be tested in the simulation process. The file of microprograms is also loaded when the user specifies the simulated architecture he/she wants to use. In this system version, the user can view the MAL symbolic language (the Micro Assembly Language [13]) in order to understand its structure.

Finally, the ***Architecture update*** task allows the user to change or complete an already defined architecture by removing and adding hardware components. The user activates the functions of the Prototype builder with a type of interaction analogous to the architecture design mode and gives new parameters for class instantiations.

The **APE-ATS subsystem** allows the user to accomplish the prototype testing phase of the simulation process. The prototype simulates the behaviour of the designed architecture that can be examined by running assembly (or symbolic microcode) programs, observing the processor's internal status and the result of its computations. In the box on the right side of fig. 3 the main components of this subsystem are illustrated.

The ***User interface*** in this subsystem provides the main interaction tasks necessary to *write-process-run* the code for testing the processor's software prototype.

The task ***User code input*** allows the user to write code (single instructions or programs) in assembly language, using the ***Code editor*** module.

The ***Code Editor*** module stores the code files ready to be processed on disk. The ***Code files*** are loaded and verified by the ***Code processor*** module, which controls the assembly programs syntactically and semantically.

The syntactic analysis discovers the typing errors. During the semantic analysis the module verifies if the program can be executed with the defined architecture and discovers whether some of the program instructions use hardware components, operation codes or address methods not previously selected by the user.

Figure 5 shows the errors noted by the Code processor module, which discovers the references to undefined registers and/or instructions.

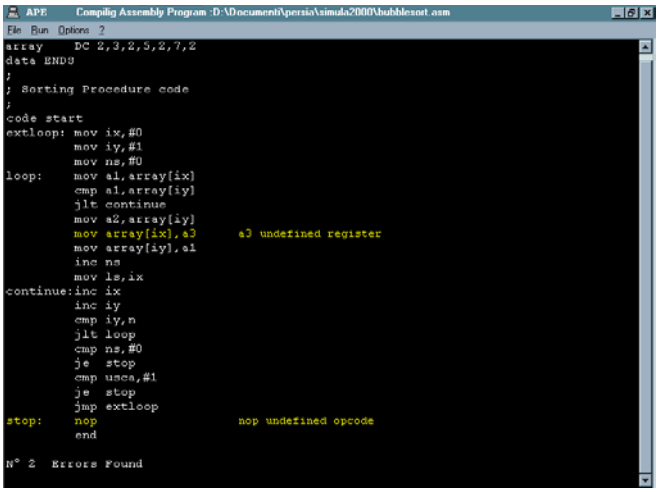

Fig. 5 Code processing task

Once the source code correctness has been verified, the ***Execution control (Run)*** module simulates the effects of the code and of the corresponding microcode execution in a fast mode or in a step by step mode. Therefore, this module shows the processor's status to the user (figure 6) after the execution of a whole program or after each instruction, such as a debugging process.

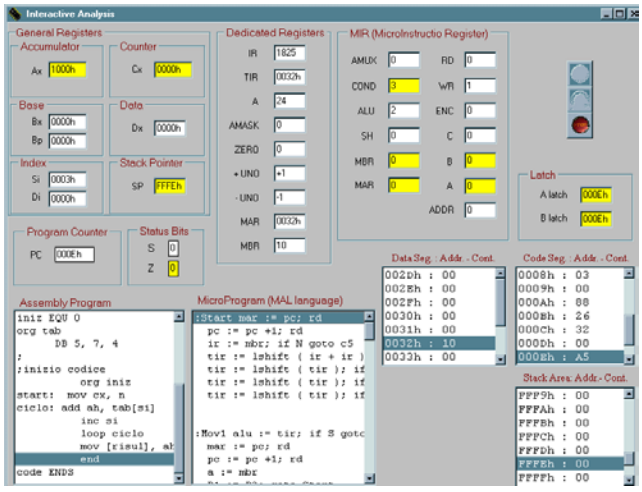

Fig. 6. Status simulation of a CISC architecture.

In order to allow the step-by-step execution of programs at the microprogramming level, the user interaction can be switched to microcode examination. In this case, the Run module shows the user the instructions and the addressing methods allowed by the architecture tested (figure 7 reports the instructions and methods of the simplified architecture layout of figure 4).

Afterwards, the Run module permits the following user activities: a) firing a single instruction showed in the panel, by specifying operands; b) loading the

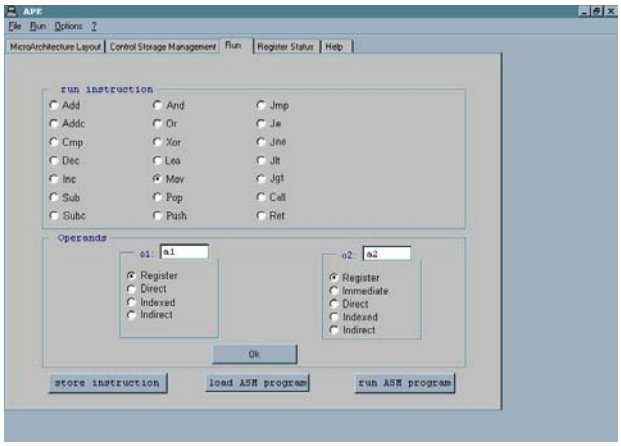

Fig 7. Microprogram panel run for a *simple* prototype.

assembly programs and eventually modifying them using the Code editor (pressing the load ASM program button); c) running the assembly program to be studied step by step at every micro instruction (pressing the run ASM program button)

Figure 8 shows the internal status of the processor ready for the step by step execution of the microprogram corresponding to the instruction *mov a1, a2* of figure 7.

In this way it will be possible to examine at the contents of microarchitecture components in order to understand the semantics of the single microinstruction execution for each assembly instruction of the running program.

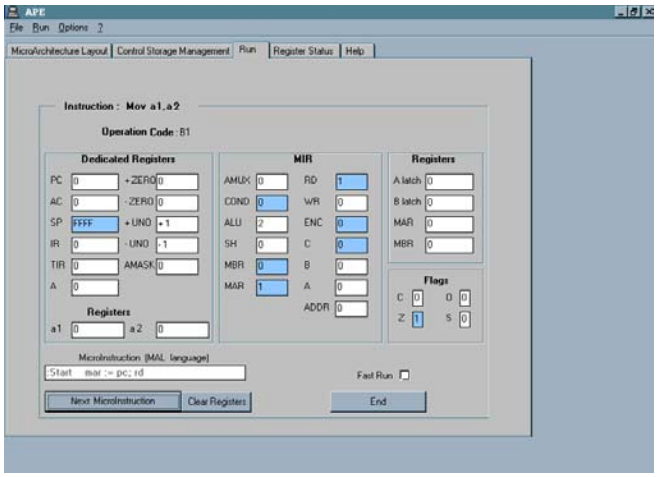

Fig 8. Example of the initial microarchitecture status before running a microprogram for MOV instruction execution

With the ***User parameter*** task, the user can select the measures of a processor's performance from a list of items in order to obtain common statistics on architectural characteristics. The ***Performance evaluation*** module computes statistics on register usage patterns, interruption event counts, timing information and so on.

By running benchmarks, the user can obtain statistics on the computer architecture performance

both at the instruction set level (total execution time, instruction cycle time, etc.) and at the microprogramming level (the usage of a processor's units).

The evaluation of this type of performance is important mainly from a didactic point of view. In this way the student can investigate the effects on processor performance of changing the computer architecture design. The statistical results can be shown in diagrammatic form and can be saved in files.

4. Conclusions

We have presented the implementation of a computer architecture prototyping environment. The system has been successfully used during a computer architecture course, but its facilities can also be extended profitably to the computer design field. In fact, both fields need an evolutionary procedure in the prototyping activity, like as the APE system.

Furthermore, the development of simulation prototypes allows the architecture design to be implemented at a lower cost than the hardware building. In the educational field this means that the alternation of the design and test phases allows the user to verify the learning progress by experimenting small architecture modifications and observing the results. The student can be progressively introduced to the complexity of modern computer architectures, starting from the simulation of a simplified instruction set of a simplified architecture of hypothetical machines. In the design field the same alternation in designing a new architecture helps to identify problems and to test the design.

The main characteristics of our approach consists in the separation of the design and the implementation phases. This helps the user to work on the design without considering implementation details. At the same time, at the end of the architecture definition process, the user obtains a simulator that constitutes a software prototype of the designed architecture.

In the educational environment, the APE has been used to design CISC and RISC architectures, such as Intel 80x86, SPARC (by Sun Microsystems) and SPUR (by D. Patterson and C. Sequin).

The first APE prototype shows some limitations in the computer design field: in fact, the objects representing hardware components are instances of previously defined classes. This constraint restricts the choice of new components and facilities. Therefore, we are studying the possibility of resolving this disadvantage by building *a class/object editor* onto the APE**,** in order to allow the user to easily define new classes corresponding to the new components with different functions, and adding an APE *class/editor inspector* to find inconsistencies in the modified classes model.

Finally, we will extend the APE system to describing and prototyping multiprocessor and non-Von Neumann architecture design.

References:

[1] Larus, J.R.., SPIM 520: A MIPS R2000/R3000 Simulator, www.cs.wisc.edu/~larus/spim.html.

[2] Hennessy, J.L. and D.A. Patterson, *Computer Architecture: A Quantitative approach*, Morgan Kaufmann, San Mateo, Calif., 1990.

[3] Clements, A., Selecting a processor for teaching computer architecture, *Microprocessors and Microsystems*, *23*, 281-290, 1999.

[4] Campbell, R. A., Introducing computer concepts by simulating a simple computer, *ACM SIGCSE Bulletin*, *28*(3), 9-11, 1996.

[5] Cutler, M. and Eckert, R., A Microprogrammed Computer Simulator, *IEEE Transactions on Education*, *30*(3), 135-141, 1987.

[6] Simeonov, S. and Schneider, M., MSIM: An Improved Microcode Simulator. *ACM SIGCSE Bulletin*, *27* (2), 13-18, 1996.

[7] Meyer, R. M., CANALOGS: a logic gate simulator for all seasons, 27th SIGCSE Techn. Symp. on Comp. Science Education, *ACM SIGCSE Bulletin*, *28* (1), 58- 62, 1996.

[8] De Blasi, M. and Tangorra, F., Prolog simulation of computer architecture in laboratory activities, *IEEE Transactions on Education*, *35*(4), 331-337, 1992.

[9] Luqi, Software Evolution Through Rapid Prototyping, *IEEE Computer*, *22* (5), 9-10, 1989.

[10] Kumar, S., Aylor, J.H., Johnson, B.W. and Wulf, Wm. A., Object-Oriented Techniques in Hardware Design, *IEEE Computer*, *27* (6), 64-70, 1994.

[11] Abbattista, F., dell'Aquila, C., Pizzutilo, S. and Tangorra, F., An Object oriented simulator of computer microarchitectures, *Proc. of IASTED Intern. Conf. Modelling and Simulation*, 50-54, Pittsburgh, 2000.

[12] Booch G., Jacobson I., Rumbaugh J et. al., The Unified Modeling Language for Object-Oriented Development Version 1.0, UML Notation Guide, UML Summary, UML Semantics, Rational Software Corporation, January 1997 and the UML 1.1 update of Sept. 1997 - try: official UML-site

[13] Donaldson, J. A Microprogram Simulator and Compiler for an Enhanced Version of Tannenbaum's Mic1 Machine, 26th SIGCSE Techn. Symp. on Comp. Science Education, *ACM SIGCSE Bulletin*, *27* (1), 238- 242, 1995.