A Multi-Level Computer Architecture Simulator

FABIO ABBATTISTA SEBASTIANO PIZZUTILO FILIPPO TANGORRA
Dipartimento di Informatica
Università di Bari
via Orabona 4, 70126 Bari
ITALY
{ }

Abstract: - We describe a simulator, which constitutes a useful supporting tool in all activities involving computer architecture definition, such as designing or teaching. The system, developed for using an object oriented approach, integrates in a single tool different architecture models, such as CISC and RISC architecture types, and different computer levels, such as micro-programming and instruction set levels.

The system allows the user to define the computer architecture at the instruction set level and then to switch automatically to the lower level of the corresponding micro-architecture. It is composed of two parts: a graphical interface, which displays the components of the architecture and allows the user to act directly on component contents; and a manager module, which executes the instructions and micro-instructions. The simulation process consists of a loop, in which two strictly correlated phases are executed: the *definition* phase, in which the user defines the architecture through a choice of hardware components, and a *test* phase, in which the user tests the designed architecture in order to verify the result of the design activity of the previous phase.

The system is written in C++ language and it is used in a didactic environment for the laboratory activity of computer architecture courses. *IMACS/IEEE CSCC'99 Proceedings,* Pages:3701-3707

Key-Words: Architecture Simulation, Computer Architecture Design, Education, Object-Orientation, Prototyping.

1 Introduction

The simulation is recognized as a valid and flexible tool, which leads to both a clear comprehension of the functioning of complex systems and the possibility of exploring new solutions by submitting the model to various experimental conditions. In particular, due to its great versatility, the software simulation allows the user to study and test the system performance, without building the actual system. Moreover, computer simulators are advantageous for the following reasons: a) the low cost of developing a simulator with respect to hardware implementation costs; b) the ability to constantly keep the laboratory up-to-date with technological innovations; c) the possibility of having different architectures implemented on the same workstation for comparative studies; d) the ability to graphically represent events, which cannot be immediately observed in the hardware (register contents, memory).

Most of the current simulators represent specific architectures and cannot be used if architectural characteristics are changed [1-2]. In fact, when the computer architecture is fixed, the assembly language, that permits the use of the defined architecture by the simulator, is also fixed. Therefore, if the architecture is modified, the computer simulator must also be redesigned in order to match the new characteristics with the assembly language. Such a lack of flexibility could be overcome by allowing the user to experiment and verify the architecture performance immediately after the design modifications. This flexibility is common in the simulation field and allows the analyst to use the simulation model in various experimental conditions, but it has not been applied in the computer simulation field. Although languages for hardware description have been devised, the lack of compilers for these languages limits their application and restricts the test of defined architectures with the assembly language. The rapid prototyping paradigm overcomes the previous limits of the computer architecture simulation.

In the software development field, the rapid prototyping paradigm constructs the definitive software system after the user evaluation of a prototype. The prototype is built on the base of user requirements. A graphical interface allows the user to verify whether the prototype fits the desired design specifications, otherwise new user requirements are formulated for a new prototype. In this way, the software system is obtained in an evolutionary fashion, by refining initial requirements until they correspond to user needs. Analogously, we aim at designing a computer architecture, which gives the user the possibility to define architectural requirements. From these requirements it would be possible to obtain a software prototype that simulates the defined architecture and allows to test the architecture with the assembly language, in order to verify the coherence between the prototype and the requirement specifications.

In our approach the computer architecture design is viewed as a collecting of hardware components. 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 characterizing the hardware component with a corresponding reusable software component. So, we have implemented the software components as objects.

Our system supports the computer architecture design through object-oriented prototyping. Object-based prototyping consists in the rapid construction of a system that simulates computer architectures by using objects that encapsulate properties and functions of basic hardware components, such as building blocks.

In the object-oriented technology, the architecture description aspect is separate from the implementative aspect, so the user can concentrate on architecture components, rather than on the simulator code in its working session.

The object-oriented technology is widely used in VSLI design [3] and has been proposed also for hardware description and simulation [4, 5], as it concerns two classes of advantages. The first class is related to the flexibility of the modeling process. The properties of the object-oriented approach (abstraction, encapsulation, inheritance and hierarchy) are closely linked to the physical components, and the designer work is not absorbed in implementation aspects but it is concentrated at a higher level of abstraction. In our project, this is an essential advantage because the architecture designer should be only involved in the choosing of components of the computer architecture.

The second class of advantages is related to the development process for the rapid prototyping. The features of the object-oriented development process (modularity, extensibility, reusability and reduced code) allow to implement a tool in which the user tests the architecture as soon as he finishes choosing components.

We have experimented the system describing various computer architecture types, such as CISC and RISC architecture, at the instruction set level [6, 7] and also we have verified the approach with a different type of description granularity at the micro-programming level.

The paper is organized in three sections. Section 2 analyzes the hierarchy of classes and objects used in the tool; there are shown examples of classes. In section 3, a description of the functional architecture of our system and details of the simulation process are reported. Finally, in section 4 there are reported conclusions.

2 Architecture Components and Objects

The main characteristic of the system is that hardware components of computer architecture are considered as inter-related objects by means of addressing methods, and are used by their method interface (instructions).

We have defined the basic architectural object classes of both main Von Neumann computer types: the CISC and the RISC machines.

The hardware components of the computer architecture are described in terms of objects. Every object has attributes (instance variables), whose values can also be other objects, and methods which represent the procedures that can 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. They are accessible to the designer through the object interface.

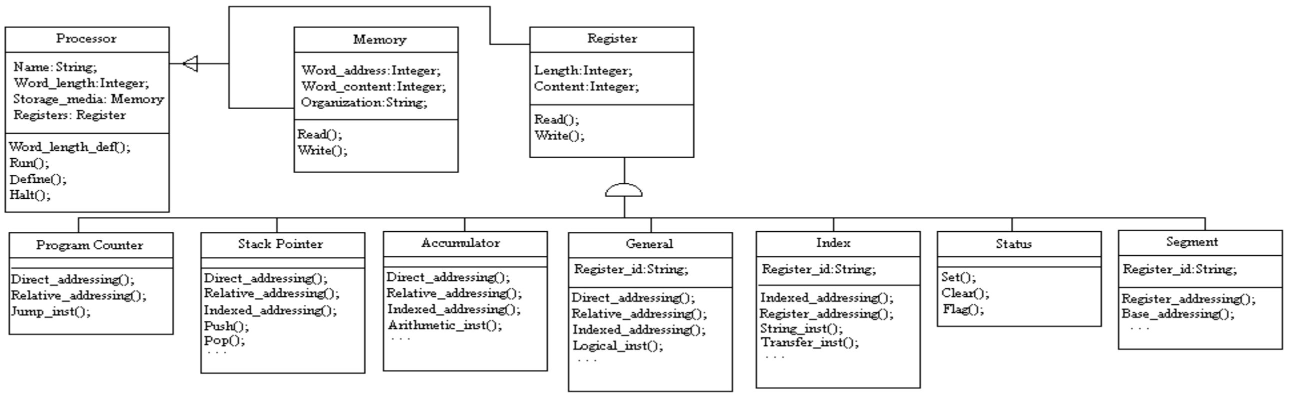
The object structure is composed as follows:

a) the class name identifies the category to which the object belongs and defines the component type of architecture. For example: the memory, the register, and so on.
b) The class properties (attributes) describe object characteristics, such as for example content of the memory word, its length, registers content and so on. Attributes can be defined in terms of other objects. This characteristic is peculiar of the object-oriented approach and is useful to define complex nested objects like hardware components.
c) The object operations define the interface through which the user can consider an object and can modify the object's attributes. In order to do that, the interface invokes methods which are structured in two parts: the *signature* (or *interface*) and the *body*. The first part contains the name of the method and the input/output parameters for its correct execution; the second hidden part contains the implementation (the code) of methods.

In our system, there are private methods not accessible to the user, created merely for implementation purposes, and public methods which correspond to the interface of the object shown to the user (designer, student, assembly program).

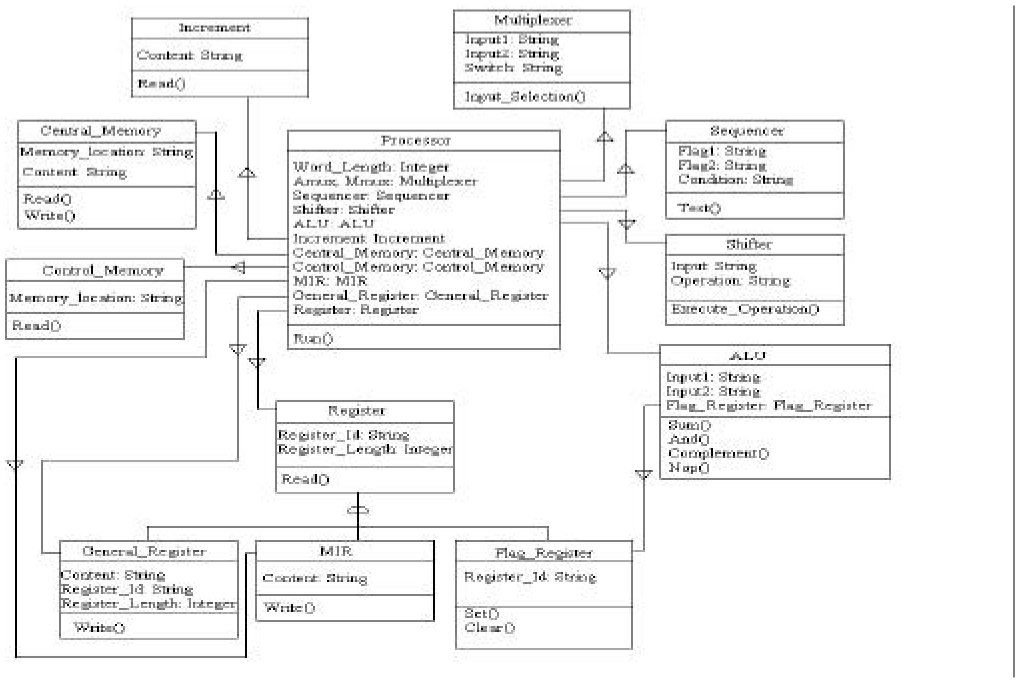
The methods are *machine language instructions* or *addressing methods* associated with a specific architectural component or object.

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 the reference of the content of other registers to be executed). 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.



Processor
Name: String; Word_length: Integer; Storage_media: Memory Registers: Register
Word_length_def(); Run(); Define(); Halt();
Memory
Word_address: Integer; Word_content: Integer; Organization: String;
Read(); Write();
Register
Length: Integer; Content: Integer;
Read(); Write();
Program Counter
Direct_addressing(); Relative_addressing(); Jump_inst();
Stack Pointer
Direct_addressing(); Relative_addressing(); Indexed_addressing(); Push(); Pop(); . . .
Accumulator
Direct_addressing(); Relative_addressing(); Indexed_addressing(); Arithmetic_inst(); . . .
General
Register_id: String;
Direct_addressing(); Relative_addressing(); Indexed_addressing(); Logical_inst(); . . .
Index
Register_id: String;
Indexed_addressing(); Register_addressing(); String_inst(); Transfer_inst(); . . .
Status
Set(); Clear(); Flag();
Segment
Register_id: String;
Register_addressing(); Base_addressing(); . . .

a)



Increment
Content: String
Read()
Multiplexer
Input1: String Input2: String Switch: String
Input_Selection()
Central_Memory
Memory_location: String Content: String
Read() Write()
Processor
Word_Length: Integer Amux, Mmux: Multiplexer Sequencer: Sequencer Shifter: Shifter ALU: ALU Increment: Increment Central_Memory: Central_Memory Control_Memory: Control_Memory MIR: MIR General_Register: General_Register Register: Register
Run()
Sequencer
Flag1: String Flag2: String Condition: String
Test()
Control_Memory
Memory_location: String
Read()
Shifter
Input: String Operation: String
Execute_Operation()
ALU
Input1: String Input2: String Flag_Register: Flag_Register
Sum() And() Complement() Nop()
Register
Register_Id: String Register_Length: Integer
Read()
General_Register
Content: String Register_Id: String Register_Length: Integer
Write()
MIR
Content: String
Write()
Flag_Register
Register_Id: String
Set() Clear()

b)

Figure 1. The main classes of a CISC architecture: a) Instruction set level, b) Microprogramming level.

Figures 1 and 2 report examples of fundamental classes of CISC and RISC architectures. In the diagrams, a class is represented as a rectangle divided into three regions. The top region contains the name of the class, the middle one contains attributes which characterize the properties of the object class: these two regions constitute a hidden part of the class whose data are read or updated only by its methods. Finally, the bottom region contains methods of the class.

An example of object structure is the class “Register_file”, which has been defined to implement the "overlapping register windows" technique (used in RISC architectures to reduce the memory I/O traffic of procedure calls) [8].

Name of the class:

<u>Register_file</u>; the class name.

Attributes:

<u>Length</u>: *integer*; this attribute represents the number of allowed windows.

<u>Windows_counter</u>: *integer*; this attribute represents the counter of the current register window (CWP) and its value is an integer.

<u>Windows</u>: *array of integer*; this attribute represents the link of the current class with the set of register windows, which compose the current class.

Methods:

<u>Length_def</u>(); this method defines (initializes) the value of the “Length” attribute.

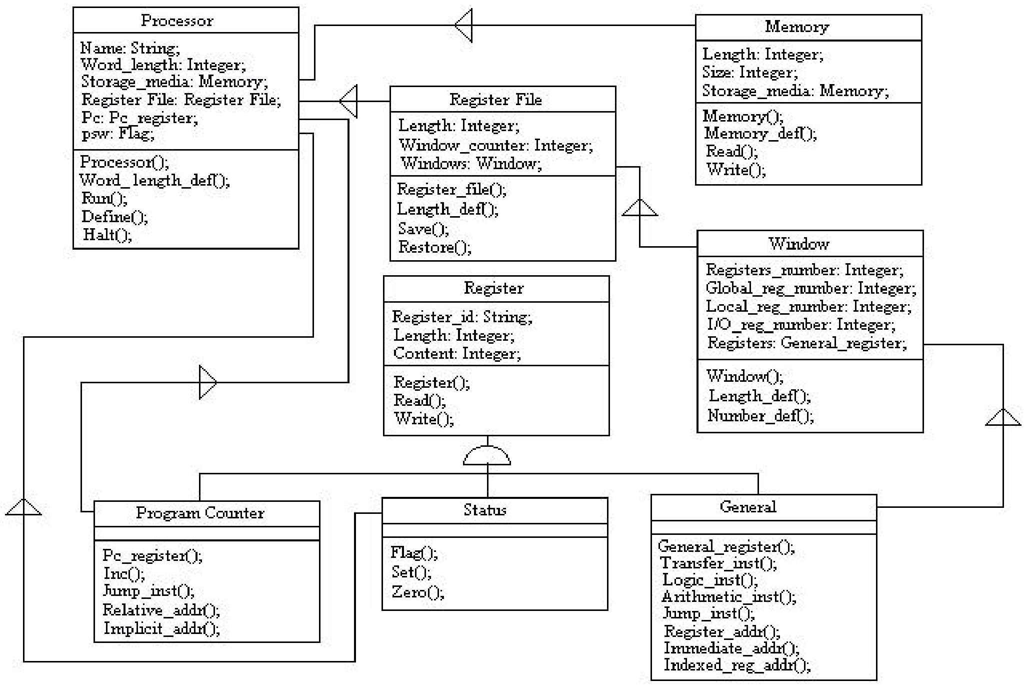


Figure 2. The main classes of a RISC architecture.

Save(); this method creates an object Window, i.e. an instance of the Window class, by overlapping the parameters windows and increasing the CWP.

Restore(); this method removes the last instanced object Window, decreasing the CWP.

In figures, the arcs are relations among classes. The arcs labeled with an arrow indicate that a class (the origin of the arc) constitutes the domain of an attribute of the end class; arcs labeled with a semicircle indicate that the end class is-a super-class of classes from which the arcs start.

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 a high level, by choosing objects which form the desired architecture. This user 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 System Architecture

The overall structure of the system provides two main sub-systems (Fig. 3), which support two different user-interaction phases: the definition of an architecture and the testing of the defined architecture.

The Architecture Definition sub-system is composed by following three modules:

-The User-Interface, which consists of a graphical interface (Windows-like) to help the user in the choosing and naming of basic components of the architecture .

-The Basic Hardware Components Data Base Interface, which, on the basis of the user choice, searches built-in hardware components in the data base and loads the selected components and their methods.

-The Prototype Builder, which generates the simulator. It is invoked when the defined architecture meets the user requirements.

The Architecture Test sub-system is composed by following four modules:

-The User Interface, which consists of a graphical interface that shows the status of the architecture during the execution of the test.

-The Code Editor, which allows the user to write test programs in assembly language.

-The Code Evaluator, which analyzes assembly programs to detect syntax errors or op-codes not supported by the defined architecture.

-The Run module, which gives to the user the possibility to run a program in a fast or in step-by-step mode, and to analyze the status of the architecture after the execution of a whole assembly program or after each instruction.

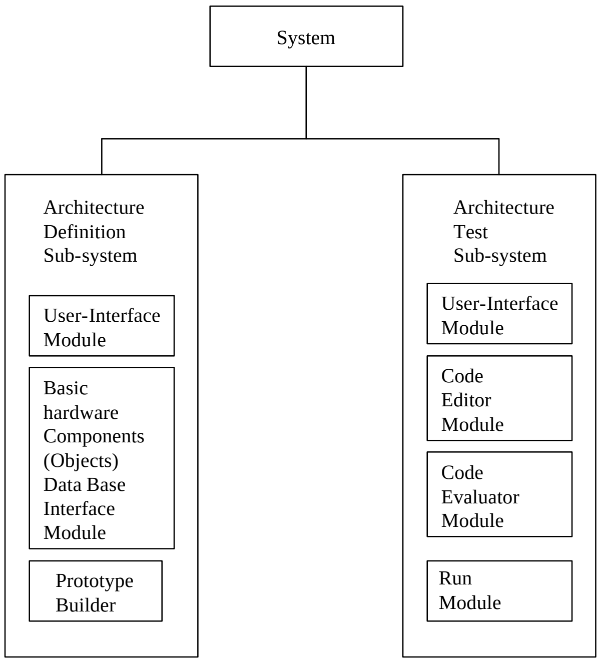

System
Architecture Definition Sub-system
User-Interface Module
Basic hardware Components (Objects) Data Base Interface Module
Prototype Builder
Architecture Test Sub-system
User-Interface Module
Code Editor Module
Code Evaluator Module
Run Module

Figure 3. The system architecture

3.1 System Interaction

The simulation process is represented by a loop in which two strictly correlated phases are executed:

-the *definition* phase, in which the user defines a prototype of the architecture through the choice of hardware components (instances of classes encapsulating methods);

-the *test* phase, in which the user, interacting with the simulator, verifies the prototype defined in the previous phase.

The two phases are iterated until the user is satisfied of the produced architecture.

The great flexibility of the system allows the user to interact at different levels and to switch from a level to another. In fact, the architecture could be defined both at the micro-programming level and at the instruction set level. In the latter case, the tool automatically constructs the corresponding micro-architecture.

***Mode A*) Micro-programming level.**

Definition phase. The user chooses components of this level (ALU, MIR, MPC, control store, etc.) and he/she defines the micro-architecture with the help of a menu-driven graphical interface.

Test phase. The system is able to process single micro-instructions as well as micro-programs edited by the user in the control store. This phase allows the user to verify the processor status after the execution of each micro-instruction. The test phase could lead to a re-definition of some components of the architecture.

***Mode B)* Instruction Set Level.**

Definition phase. The user chooses components of this level (central memory, registers, etc.) and defines the architecture being helped, in this case too, by a menu-driven graphical interface. The user can switch to the corresponding micro-architecture level to test it.

Test phase. The system is able to process single instructions as well as assembly programs, like as in the previous mode. In addition, the user could process the micro-program corresponding to the assembly program edited in the main memory of the defined architecture.

In the phase of *architecture definition*, the system supports the user, providing a list of components, according to the defined classes for the architecture type (see figures 1-2). When the user inserts a new component (object), the system automatically identifies assembler instructions and addressing methods for that hardware component. From an object-oriented point of view, the *signature* of the methods associated to objects composes the assembler language. At the same time, the *body* of the methods will concur to the immediate implementation of the simulator in order to process instructions.

Moreover, the tool doesn’t constraint the user to activate all methods of the defined architecture. This flexibility allows the evolution of complex *vs* simple architecture, as well as complex *vs* simple instruction set. This is particularly useful in a teaching environment to achieve progressive learning of the computer architecture.

The second step of the simulation process is the testing of the defined architecture. In this phase, the user simulates the model by running a test program.

On the basis of the chosen architecture level, the program has to be written in assembly language (instruction set level) or in micro-code (micro-programming level). In the simulated run time, the prototype displays results of the execution in terms of the architecture component’s status (see figure 4 for an example).

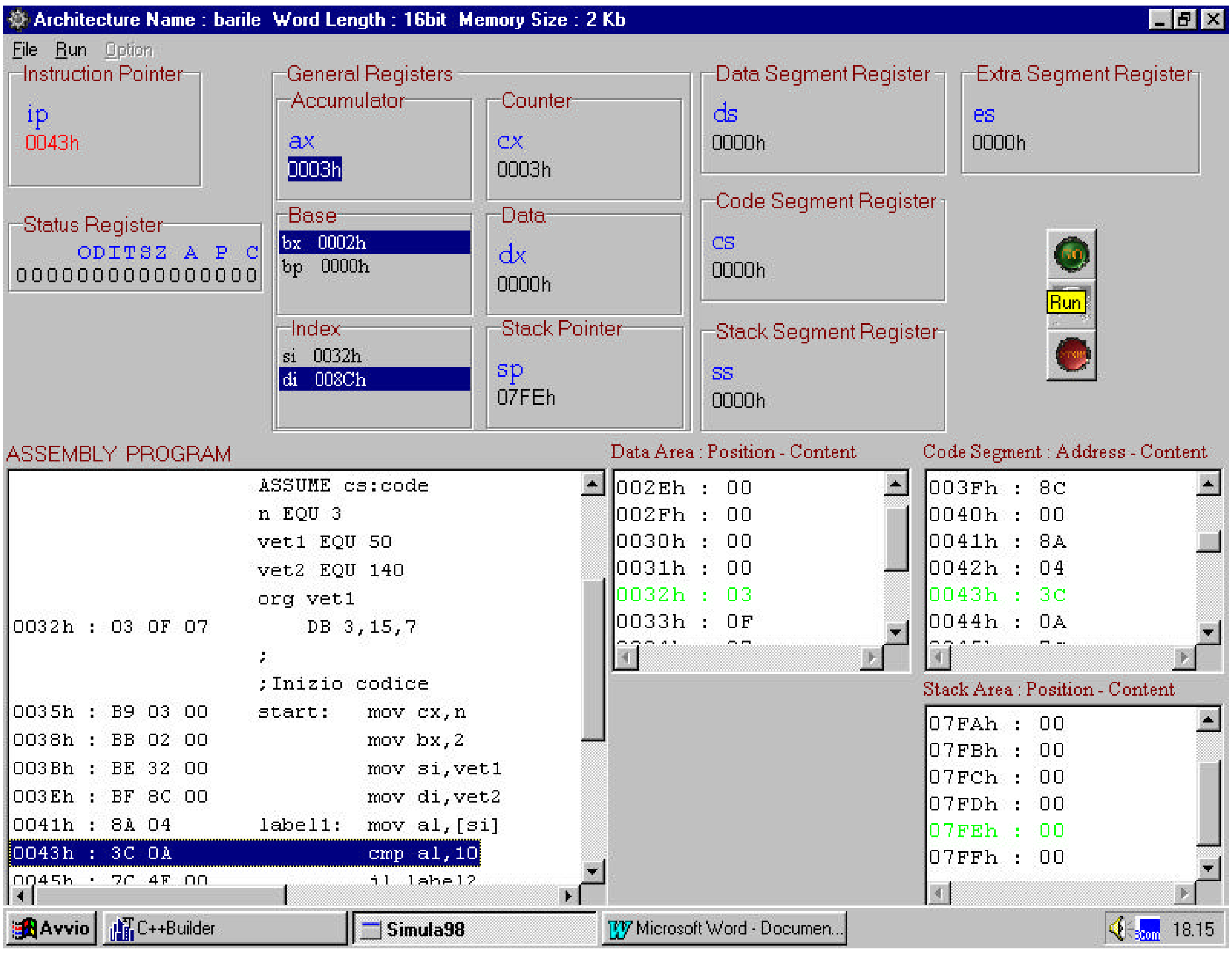
The test can be performed in two different ways:

a) through a *single instruction*: the user initializes the components (register, memory, etc.) and writes the instruction to be executed.

b) through a *program*: the user loads a previously saved program into the memory, or edits a new one. The execution of the program can be performed step-by-step to control the status of the architecture after each instruction.

In both cases, the simulator controls the program syntactically and semantically. The syntactic analysis is aimed at discovering the editing errors. In the semantic analysis, the simulator verifies that the program can be executed with the defined architecture and discovers if some of the program instructions refer to components that the user did not select.

During the execution, the user can easily switch between the two architectural levels, in order to analyze differences in two different run-time environments. In this way it is possible to have complete control over the execution of the test program. In fact, in the step-by-step program execution, users could easily analyze the status of the defined architecture at the end of the execution of


Architecture Name : barile Word Length : 16bit Memory Size : 2 Kb

File Run Option

Instruction Pointer: ip 0043h

General Registers
- Accumulator: ax 0003h
- Counter: cx 0003h
- Base: bx 0002h, bp 0000h
- Data: dx 0000h
- Index: si 0032h, di 008Ch
- Stack Pointer: sp 07FEh

Data Segment Register: ds 0000h

Extra Segment Register: es 0000h

Code Segment Register: cs 0000h

Stack Segment Register: ss 0000h

Status Register
```
   ODITSZ A P C
0000000000000000
```

ASSEMBLY PROGRAM
```
                        ASSUME cs:code
                        n EQU 3
                        vet1 EQU 50
                        vet2 EQU 140
                        org vet1
0032h : 03 0F 07            DB 3,15,7
                        ;
                        ;Inizio codice
0035h : B9 03 00        start:     mov cx,n
0038h : BB 02 00                   mov bx,2
003Bh : BE 32 00                   mov si,vet1
003Eh : BF 8C 00                   mov di,vet2
0041h : 8A 04           label1:    mov al,[si]
0043h : 3C 0A                      cmp al,10
0045h : 7C 4F 00                   jl label2
```

Data Area : Position - Content
```
002Eh : 00
002Fh : 00
0030h : 00
0031h : 00
0032h : 03
0033h : 0F
```

Code Segment : Address - Content
```
003Fh : 8C
0040h : 00
0041h : 8A
0042h : 04
0043h : 3C
0044h : 0A
```

Stack Area : Position - Content
```
07FAh : 00
07FBh : 00
07FCh : 00
07FDh : 00
07FEh : 00
07FFh : 00
```

Figure 4. Status Simulation of a CISC architecture

each instruction. By switching to the micro-architecture level, users can easily perform a step-by-step execution of the micro-code associated to the current assembly instruction. In such a way, users have the possibility to verify the effectiveness of their programs and the adequacy of the defined architecture.

At the end of the execution, the displayed results can be saved on a file, as well as the tested assembly program. Alternatively, if users are not satisfied by the results of the execution, they can simply re-define some of the components of the architecture (preferably at the instruction set level), while the tool will provide the corresponding updating at the micro-programming level. When this iterative process leads to a definition of the computer architecture closer to user's requirements, it is possible to generate the code (written in C++) 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, but it will not provide the possibility to further updating of the components of the architecture. In other words, in a typical work session of the proposed tool, users perform a rapid prototyping life cycle. In fact an architecture produced in the definition phase is equivalent to a prototype to be verified and evaluated and eventually updated (fig. 5). Instead, the code corresponding to the architecture built with our tool, represents the final system, ready to be operational.

4 Conclusions

The main goal of our project was to implement a computer simulator to aid the learning process in a course of computer architecture for undergraduate students. We encouraged students to design and construct their own architectures, having some assembly programs as requisites. We think that the advantages of such a tool could also be extended to the computer design field. These different fields have in common the need to proceed in an evolutionary way, like prototyping activity in software development [9, 10].

In the didactic field, this means that alternating the design and testing phase allows students to be introduced progressively to the complexity of computer architectures, starting with simplified machines and achieving complex architectures. In the computer design field, the

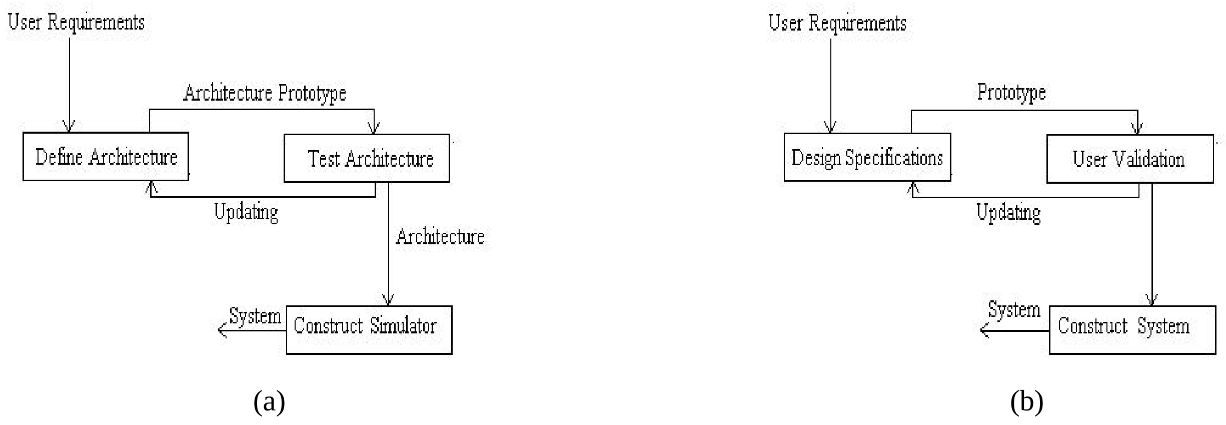


Figure 5. The computer architecture modeling process (a) vs. the rapid prototyping life cycle (b).

same alternation in designing a new architecture helps to identify problems and to test the design performance. It can lead to a redefinition of the architectural requirements in a way similar to the rapid prototyping activity that achieves the final desired architecture with a successive refinement of design requirements.

The main advantage of such an approach consists in the separation of the design and the implementation phases. This helps the user to establish the design without considering the implementation details. At the same time, at the end of the architecture definition process, the user has, at his disposal, a simulator that constitutes a software prototype of the designed architecture.

Another advantage of the tool is the great flexibility of the object-oriented approach. In fact, users are not forced to activate all services (instructions and addressing methods) of the defined architecture. If the focus is, for example, on a small set of characteristics, they can select the subset of functionalities needed for their purposes and deactivate others.

In the educational environment, the proposed simulator has been successfully used to design some real CISC and RISC architectures such as Intel 80x86, SPARC (by Sun Microsystems) and SPUR (by D. Patterson and C. Sequin, University of Berkeley). In the experimentation phase, the tools proved to be easy to use and powerful enough to be effectively used with standard computer architectures.

This approach is limited in the computer design field; in fact in our tool, objects representing hardware components of the architecture are instances of predefined classes, and this constraint restricts the choice of new components with services different from the usual ones. Therefore, the user has to pay great attention to the definition of classes. We are studying the possibility of resolving this disadvantage by adding to the simulator a class/object editor, in order to allow the user himself to easily define new classes corresponding to new components with different functionalities, and a class/editor inspector, to find inconsistencies in the modified classes model.

References:

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

[2] Tangorra F, The Role of the Computer Architecture Simulator in the Laboratory, *ACM SIGCSE Bulletin*, 22(2), 1990, pp. 5-10.

[3] Verschueren jr. A. C., An Object Oriented Design and Simulation System for VLSI, *Microprocessing and Microprogramming*, 30, 1990, pp. 241-246.

[4] Skrien D. and J. Hosack, A multilevel simulator at the register transfer level for use in an introductory machine organization class, *ACM SIGCSE Bulletin*, 23(1), 1991, pp. 5-10.

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

[6] Abbattista F., S. Pizzutilo and F. Tangorra, Object oriented design of architectures at the instruction set level, *In Proc. of 15th IASTED Intern. Conf. Applied Informatics*, ed. M.H. Hamza, 1997, pp. 58-62.

[7] Abbattista F., S. Pizzutilo and F. Tangorra, Object oriented approach to design RISC architectures, *Proc. of 16th IASTED Intern. Conf. Applied Informatics*, ed. M.H. Hamza, 1998, pp. 204-207.

[8] Bardakar D., RISC versus CISC: A Tale of two Chips, *Computer Architecture News*, 25(1), 1997, pp. 1-12.

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

[10] M.M. Tanik and R.T. Yeh, Rapid Prototyping in Software Development, *IEEE Computer*, 22(5), 1989, pp. 9-10.