Int. J. Engng Ed. Vol. 25, No. 2, pp. 228–238, 2009
Printed in Great Britain.

0949-149X/91 $3.00+0.00
© 2009 TEMPUS Publications.

Easy CPU: Simulation-based Learning of Computer Architecture at the Introductory Level*

CECILE YEHEZKEL[1], MATZI ELIAHU[2] AND MIKY RONEN[2]
[1] Davison Institute of Science Education, Weizmann Institute of Science, Rehovot 76100, Israel. E-mail: cecile.yehezkel@weizmann.ac.il
[2] Instructional Systems Technologies Dept., Holon Institute of Technology, Holon Institute of Technology, Holon 58102, Israel. E-mail: matzie@hit.ac.il ronen@hit.ac.il

The interdisciplinary nature of the computer architecture domain and the complexity of both hardware and software make it difficult for instructors to teach students the underlying mechanism of program execution at the introductory level. We present an environment that helps introductory students to understand how the instructions activate the hardware, and how to master basic programming skills in machine language. This environment includes a simulation of a low-level computer machine and a comprehensive set simulation-based activities aimed at scaffolding the learning process. The environment, EasyCPU, displays a schematic model of the computer components and the dynamic processes as well as the flow of information involved in executing the program at the machine level. The environment can control external hardware, in addition to the on-screen I/O simulation. This enables students to develop small but real hardware projects, and thus to experience the interdisciplinary nature of working with hardware and software. The extensive use of EasyCPU (by 7000 students) provided an opportunity to assess its contribution and to a better understanding of the interactions between the computer units and the details of program execution and the data flow within the computer, as well as to the development of programming skills.

Keywords: Simulation-based learning, assembly language, computer architecture.

INTRODUCTION

WHEN TEACHING COMPUTER ARCHITECTURE, instructors encounter difficulties in conveying its main concepts and in illustrating the computer's low-level machine mechanisms underlying program execution. This paper describes a simulation and the associated simulation-based activities designed to improve the learning of computer architecture topics taught in Computer Science and Engineering, Software, and Electrical Engineering (CS, SE, SE and EE) courses. We describe in detail the simulation and the associated simulation-based activities (further referred to as the EasyCPU environment). This educational simulator is aimed at facilitating the teaching of Computer architecture at the introductory level.

Teaching/learning Computer Architecture

The fundamentals of computer architecture are an integral part of the curricula of three engineering domains: Computer Engineering, Software Engineering and Electrical Engineering. According to the computer science periodic curriculum report, issued by a joint task force headed by the ACM and IEEE-CS, Computer Architecture and Organization is defined as one of the core bodies of knowledge in the Computer Science and Software Engineering curriculum, and in the Computer Engineering curriculum [1, 2, 3]. Hughes claims that 'exposure to the physical machine and the software that controls it can assist information technology professionals gain a better understanding of who does what and why it does it . . . perhaps most importantly it can remove the "magic" that is often used to explain seemingly inexplicable problems encountered in using computers' [4, p. 85]. The interdisciplinary nature of the domain, involving both hardware and software, makes it difficult for curriculum designers to cover topics that are at the boundaries of the CS, CE and EE curricula. Cassel et al. (2001) [5] have found that many faculty members who teach this subject teach it outside their areas of specialization and thus are not entirely comfortable with this task. To improve the learning of computer architecture, instructors have searched for better pedagogical methods [6]. This was also our motivation in developing a simulation environment and its associated learning activities.

Tools for teaching computer architecture

In order to understand how a low-level machine functions, the student must learn the basics of its

* Accepted 5 December 2008.

language, the machine language, which is a set of binary-coded instructions. That is why assembly language is used for learning; it is the first practical programming language above machine language, a set of mnemonics. It is selected for professional programming only in extreme cases, for instance, with high real-time constraints. Professional assembly language programming tools, for example, compiler, simulators and debuggers such as Turbo Assembler tools (further referred to as TASM) or Macro Assembler tools are used in the development of programs (textual GUI). Most of them did not evolve, in terms of user-friendly interfaces, such as GUIs of more practical languages. They usually are too sophisticated and complex for introductory-level students. This situation has motivated individual instructors to develop simulators. The field is predestined for simulation [7]. A description of some of these tools can be found in the special issue on specialized computer architecture simulators that see the present and may hold the future [8] as well as a representative brief review of eight simulators in a distributed expertise for teaching computer organization and architecture [5]. Although these tools share many features, since they were designed by individual instructors, they tend to be targeted to specific populations, thus illustrating a conceptual model appropriate to the requirements of specific curricula.

When developing a simulator for educational purposes, we must select an appropriate conceptual model to illustrate the low-level machine [8]. The conceptual model conveys principles of computer architectures such as CISC and RISC. Knuth [9], in his keynote on bottom-up education, claims that 'he has put considerable effort into the design of a RISC machine called MMIX, as an aid to computer science educators. MMIX is intended to be simple and clean yet realistic'. To provide both motivational and kinesthetic learning experiences, instructors have experienced the use of software simulation to activate a robot [10, 11], and Bruce-Lockart et al. [10] found that the TM simulator's ability to handle both physical and abstract models has made it successful for helping students understand the abstract machine defined by the programming language. Some conceptual models are hypothetical, such as the LMC simulator [12, 13], which conveys the general principles of computer architecture but not of an actual computer, whereas others, tend to simulate an actual computer with high fidelity, and still others are simplified models of an actual computer [14, 15]. The environment, EasyCPU, displays a schematic model of the computer components and the dynamic processes as well as the flow of information involved in executing the program at the machine level [8, 16]. The environment and learning activities were the subject of a multi-component research aimed at assessing the contribution of the environment to improving learning [16–8].

EXPERIENCE IN DESIGN AND USE OF SIMULATORS

Challenges in design and assessment of instructional simulators

The development and use of simulators in education are evolving, together with increased computing power and multimedia technology. The technology evolution enables implementation of more sophisticated features in the simulators, such as history recording [19], 3D imaging and 'liveness' (dynamic immediate visual feedback) [20]. Simulators are sometimes referred to as visualizations, to emphasize the aim of illustrating conceptual models and underlying processes that cannot be seen. For areas such as quantum computation where the desired system is not implemented yet, the simulation becomes crucial [21]. In engineering education, professional tools for simulation (MATLAB/SIMULINK, LabVIEW, etc.) are frequently employed by instructors to perform laboratory exercises and to introduce undergraduates to professional tools. Sometimes educators opt to develop their own simulator to fulfill their needs, dictated by the needs of the curriculum and the student population.

The design of new educational simulators should be learner-centred and accompanied by formative evaluation. Moreover, it requires developing methodologies to evaluate their effectiveness [22, 23]. Effectively evaluating simulator utilization in education is essential for further improvements. Campbell and Bourne (2002) describe a quantitative survey of the effectiveness of simulators in the laboratory [25]. They used stand-alone simulations as substitutes for practising physical laboratory exercises. The effectiveness of these simulations is then assessed by comparing the performance in a written exam by students who used simulation and those who used traditional laboratories. Their findings showed that students who used the simulation scored higher. Chaturvedi et al. (2006) share the view that simulation and visualization have great potential for enhancing student learning and the quality of engineering education. They believe that the desired objective is for students to achieve a deeper understanding of basic principles and define the features essential for effectiveness as: interactivity (between the student and the environment), interconnectivity (between subject materials), and hierarchy (gradual learning with succeeding modules) [24].

In CS and CE, most evaluations have focused on algorithm visualization (AV). Unfortunately, the results are not always positive; Hundhausen, Douglas and Stasko (2002) conducted a meta-study on AV evaluation research and concluded that 'how' students use AV technology has a greater impact on its effectiveness than 'what' AV technology is used [26]. They suggest that ethnographic field techniques and observational studies can help us better understand both how and why AV technology might be effective in a realistic situation. Kehoe, Stasko, and Taylor (2001) integrated quantitative and qualitative approaches. The latter was used to observe students' behaviour in a realistic situation, whereas the former assessed the influence of AV on students' understanding [27]. In the research described in [18], quantitative methods were used to assess the effectiveness of the environment, whereas qualitative methods were used to improve our understanding of 'how' and 'why' visualization contributes to learning.

Simulation-based teaching and learning

The simulator cannot generate learning by itself. Simulation-based activities are used to generate a fruitful interaction between the learner and the simulator. Swaak and de Jong [28] emphasize the importance of both model progression and gradually increasing the complexity of assignments to guarantee the effectiveness of simulation-based learning. A well-designed educational environment based on a simulator should support each component of this approach. Simulation-based learning must be supported to help the learner acquire skills and meta-skills and to deepen his understanding of the underlying processes illustrated by the simulator. According to Feisel, and Rosa [29], the early criticisms on simulations focused on the rigidity of simulations, the lack of realism in models, or simulated results that did not adequately represent real-world systems and behaviour, therefore causing the designer to tend to emphasize the realistic aspects of simulation-based learning activities. Recently, Ma and Nickerson [23] have made a comparative review of the literature related to hands-on, simulated and remote laboratories in education. They have observed that the boundaries among the three types of environments are blurred in the sense that most laboratories are mediated by computers. They suggest that with the proper mix of technologies we can find solutions that meet the economic constraints of laboratories by using simulations and remote labs to reinforce conceptual understanding, while at the same time providing enough open-ended interaction to teach design.

One of the principal skills CS and SE students need to acquire in programming is debugging. Ko and Myers (2004), describe debugging as an exploratory activity aimed at investigating a program's behaviour, involving several distinct and interleaving activities: Hypothesizing, observing, restructuring data into different representations, exploring, diagnosing and repairing [30]. Interleaving activities are essential for effective simulation-based learning in any domain. They are essential for acquiring skills and meta-skills and for deepening students' understanding of the topics learned.

Selecting and illustrating an appropriate conceptual model for the simulator are essential for ensuring a fruitful learning process. The simulator must be encapsulated in a course covering the comprehensive theoretical material on the conceptual model, illustrated by the simulator and activities that enable the student to practise and assimilate the new concepts.

THE COURSE

The EasyCPU environment was originally designed as a learning tool for a high-school course on introductory computer organization and assembly language programming. This course is an elective unit in the software engineering and in computer science programs [31, 32]. The textbook [33] covers the theoretical material that is taught in classrooms; this is supplemented by laboratory sessions. The syllabus is given in Table 1 [31].

Although the course is an elective, targeted at a relatively small population of students, since 1998, EasyCPU has been used by more than 7,000 students. It was also used to teach second-year undergraduates at a technological college. About 70 educational institutions in the country have used the program, most of which are high schools, colleges, the Open University and a commercial society that features professional continuous education courses.

Description of the environment

The EasyCPU environment is based on a simplified model of an 8-bit version of the Intel 80X86 microprocessor family. It models the main concepts of the von Neumann architecture, which is still the basis of computer design. The model consists of the CPU, memory segments, input/output components and the bus connections among them. The model of the CPU includes the general registers, instruction and stack pointer registers, flags and a clock. The memory is partitioned into three segments—data, stack and code—each with 256 addressed bytes. Data can be entered directly into the CPU registers and memory cells. The I/O consists of eight simulated LEDs for the output and eight simulated buttons for the input. The data and address buses are represented by eight lines in two different colours to distinguish between them. Three more lines simulate the control lines.

Table 1. Syllabus for the computer organization and assembly language course

Topic	**Class (hours)**	**Lab (hours)**
Number systems	5	2
Computer organization	4	1
Organization and execution of programs	8	–
Basic concepts of assembly language	7	25
Assembling, linking, and loading	8	–
The stack and subprograms	6	8
Interrupts	5	5
From high-level languages to assembly language	2	4
(Total)	**45**	**45**

The Intel 8086 microprocessor is a CISC architecture whose assembly language has more than 3,000 different instructions. This is too many for an introductory student to learn, and yet we want them to achieve basic programming skills in addition to an understanding of how the instructions activate the hardware. Therefore, EasyCPU simulates a subset of these instructions, which was selected to represent the various instruction categories, mnemonics, addressing modes and data types. A special effort was made to keep the EasyCPU assembly language compatible with the Intel X86 instruction set, to enable advanced students make a smooth transition to using a professional environment.

EasyCPU was designed to be operated in two modes, Basic and Advanced, to enable a gradual increase in the complexity of the tasks assigned [28, 34]. The basic-mode enables the novice student to learn the syntax and semantics of individual assembly language instructions. The advanced-mode provides the students with a visual display and development tools for creating their own programs and simulates the program's execution. EasyCPU offers a comprehensive set of scaffolding activities that enable the student to learn the basics of computer architecture and assembly language. The activities cover the main topics of the course syllabus (Table 2).

The activities address different instructional goals and are based on the activation of the Basic and Advanced modes that we will now describe.

Basic mode

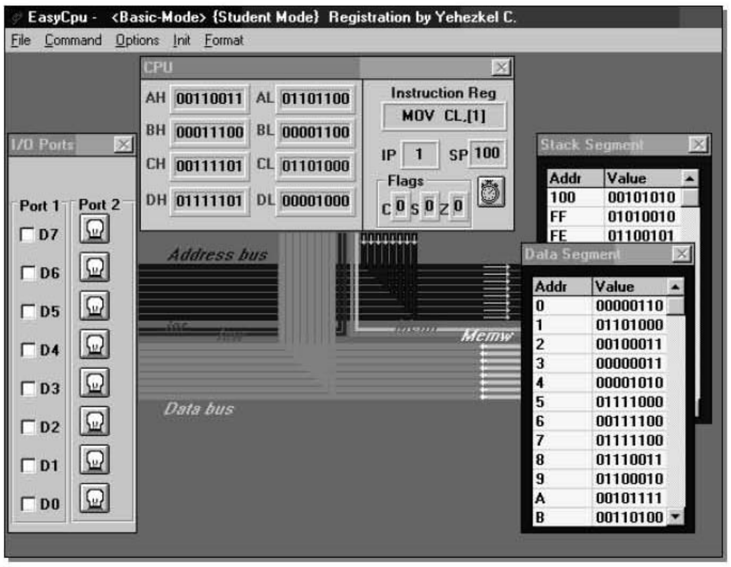
The Basic mode screen in Figure 1 shows the visualization of the execution of instruction MOV CL,[1]. To execute this instruction, the CPU reads the contents of the memory addressed by 1 and places this value into register CL.

The control, address, and data busses connecting the different units are animated to illustrate the read/write cycle type (memory or I/O): arrows slide on the address bus from the CPU to the memory, the control line MemR lights up, then arrows slide on the data bus from the data segment to the CPU.

In Basic mode activities, students are asked to pay attention to transmission of the data from one unit to another during the execution of a single instruction; this way they can learn to identify aspects of instruction execution involving a bus

Table 2: The activity-set

#	Contents
1	Instruction syntax and memory access: registers operations within the CPU, memory read/write access.
2	How to run a program. Editing, compiling and executing.
3	Addressing modes: registers, immediate, direct and indirect.
4	Testing a program: syntax and run-time error detection and correction.
5	I/O access, bus transfers and bus cycle identification
6	The endless loop
7	Arithmetic instructions and their effect on the zero, carry and sign flags.
8	Jump instructions: control loop, testing and boundary conditions.
9	Logic Instructions: NOT, AND, OR, shift and rotate instructions.
10	Constants and variables: identifying an arithmetic progression.
11	Programming the control unit of an elevator.
12	Activating real hardware: using the timer in the computer to play music.


EasyCpu - <Basic-Mode> {Student Mode} Registration by Yehezkel C.
File Command Options Init Format
CPU
I/O Ports
Port 1
Port 2
D7
D6
D5
D4
D3
D2
D1
D0
Address bus
Data bus
Memw
Stack Segment
Data Segment

AH 00110011 AL 01101100
BH 00011100 BL 00001100
CH 00111101 CL 01101000
DH 01111101 DL 00001000

Instruction Reg: MOV CL,[1]

IP 1 SP 100

Flags: C 0 S 0 Z 0

Addr	Value
100	00101010
FF	01010010
FE	01100101

Addr	Value
0	00000110
1	01101000
2	00100011
3	00000011
4	00001010
5	01111000
6	00111100
7	01111100
8	01110011
9	01100010
A	00101111
B	00110100

Fig. 1. Basic mode during the execution of MOV CL [1].

read cycle from those involving a bus write cycle (see an example in Appendix I). In addition, students can modify the processor's clock, in order to control the rate of data transfer between components of the computer (CPU, memory, I/O). They can choose the format of the data display to be binary, hex, decimal, or signed decimal.

The instructional goals addressed by the Basic Mode activities are as follows:

a) To learn the structure and classification of the instruction set and to identify their mnemonics;
b) To learn the syntax of the assembly language and to understand addressing modes;
c) To understand the mechanism of instruction execution and of memory and I/O read/write cycles.

In Basic mode activities, students can define various types of instructions: data transfers, logical operations, arithmetic operations, I/O, stack and control. Instructions are defined by selections in dialogue boxes specialized for each instruction. The dialogue box functions as a wizard to enable the student to choose instruction operands and addressing modes according to the alternatives allowed by the syntax. Once the instruction is defined, its syntax is displayed and its execution is animated. The basic mode is used in activities 1 and 3 where students learn new types of instructions and addressing modes. Part of the third activity is presented in Appendix I. Thereafter, the use of the basic advanced mode is combined, for instance, in activity 5 to introduce Input/Ouput instructions and bus transfers, or in activity 7 to show how arithmetic instructions affect flags or, like in activity 8, to show the action of logic instructions and to illustrate the differences between *rotate* and shift instructions.

Advanced mode

The Advanced mode is designed for students who have attained a basic knowledge of assembly language instructions, enabling them to develop programs. In effect, the Advanced mode functions as an integrated development environment for developing and simulating assembly language programs (Figure 2).

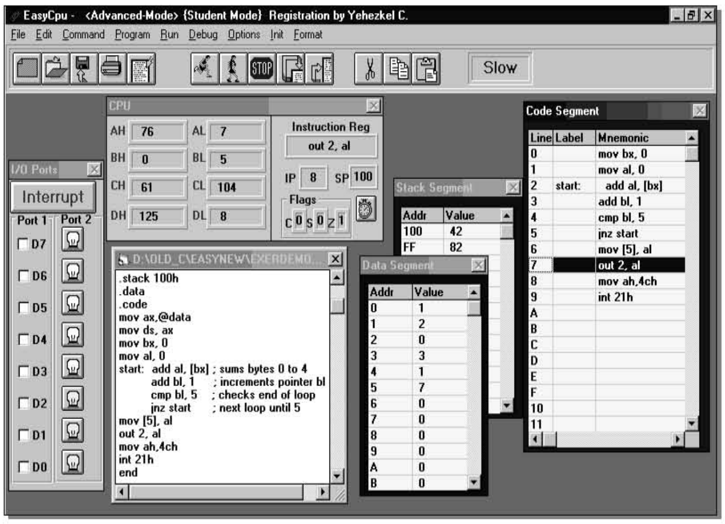
The environment visualizes the processes taking place within the computer by simultaneously displaying the source code, the data and stack segments, and an on-screen simulation of I/O ports. After writing code in the program editor, assembly and linking are simulated, followed by a simulation of the execution of the program. Students can step through a program, observing the state of the computer after each step.

EasyCPU generates messages to help students correct the syntax. To detect run-time errors in the program, they may use the debugging tools to change the data in the CPU and the memory, and they can correct and rerun the program. Figure 2 above shows a screenshot of a sample program running in the Advanced mode. The program adds the content of the first five memory bytes, saves the results in the memory byte at address five, and outputs the result to output port 2.

Instructional goals addressed by the Advanced Mode activities are as follows:

a) To acquire basic skills in the use of the move, arithmetic, logic and control instructions;
b) To understand the structure of a program;
c) To understand the process of executing a program;
d) To become familiar with the stack data structure and the actions executed on it;
e) To learn to build structured programs with subroutines;
f) To introduce the interrupts and understand their implementation.

In the activity-set (Table 2), the aim of the early use of the advanced mode in activity 2 is to illustrate the execution process of a small program


EasyCpu - <Advanced-Mode> {Student Mode} Registration by Yehezkel C.
File Edit Command Program Run Debug Options Init Format
STOP
Slow
CPU
I/O Ports
Interrupt
Port 1
Port 2
D7
D6
D5
D4
D3
D2
D1
D0
D:\OLD_C\EASYNEW\EXERDEMO...
Stack Segment
Data Segment
Code Segment

AH	76	AL	7
BH	0	BL	5
CH	61	CL	104
DH	125	DL	8

Instruction Reg: out 2, al

IP 8 SP 100

Flags: C 0 S 0 Z 1

```
.stack 100h
.data
.code
mov ax,@data
mov ds, ax
mov bx, 0
mov al, 0
start:  add al, [bx] ; sums bytes 0 to 4
        add bl, 1     ; increments pointer bl
        cmp bl, 5     ; checks end of loop
        jnz start     ; next loop until 5
mov [5], al
out 2, al
mov ah,4ch
int 21h
end
```

Stack Segment

Addr	Value
100	42
FF	82

Data Segment

Addr	Value
0	1
1	2
2	0
3	3
4	1
5	7
6	0
7	0
8	0
9	0
A	0
B	0

Code Segment

Line	Label	Mnemonic
0		mov bx, 0
1		mov al, 0
2	start:	add al, [bx]
3		add bl, 1
4		cmp bl, 5
5		jnz start
6		mov [5], al
7		out 2, al
8		mov ah,4ch
9		int 21h
A		
B		
C		
D		
E		
F		
10		
11		

Fig. 2: Advanced mode showing a program.

and to practice basic editing, compilation and basic debugging tools. This activity introduces the definition of syntax errors and run-time errors and provides guidelines for minor debugging to correct a short program. Activity 4 consolidates testing skills. The students are progressively asked to make significant modifications of programs and, finally, to write their own programs. To write meaningful programs, the students have to understand the flags' roles (activity 7) and the action of conditional jumps, and to master control loops (activity 8). A part of the eighth activity is presented in Appendix II. In activity 9 students learn logic and shift instructions and are asked to calculate successive exponents of 2, and to implement variations of running-lights on the LED-array as a function of the user's switch-array inputs. Activity 10 introduces the use of constants and variables—students are asked to print the ABCs on a text-screen. In activity 11, students are asked to program the control unit of an elevator.

The EasyCPU environment can control external hardware in addition to the on-screen I/O simulation. This enables students to develop small but actual hardware projects, and thus to experience the interdisciplinary nature of working with hardware and software. For example, they can write a program that plays musical tones by using the *output* instructions, the computer timer, and the speaker of the PC platform, as required in activity 12 to generate musical tones. A unique feature allows the development of independent software modules that can be activated in EasyCPU through *input/output* assembly instructions. Such a module was developed for students to practice interfacing basic peripherals on an educational lab-kit. The GUI used in simulating the hardware lab-kit connected to the PC allows students to interactively test the activation of the peripherals by "clicking" the devices on the GUI and then to directly program peripherals (timer, parallel ports) and toggle switches, LEDs, and Seven-segments. Software communication between the lab-kit simulator and the EasyCPU environment enables the student to program the lab-kit hardware and debug the program with both the simulated and the hardware lab-kit. These features provide simplified tools supplied in a professionally Integrated Development Environment.

Empirical evaluation and experience

We quantitatively assessed the environment's contribution to the development students' understanding and assimilation of the course materials as well as their acquirement of programming skills. The study was divided into two parts. The first part was a survey aimed at evaluating the contribution of EasyCPU to understanding the course materials, and the second part was a survey targeted to assess the contribution of EasyCPU to the acquisition of programming skills.

The study population

The study was performed with 271 high-school students in eleven classes from eight different high schools. All classes learned the same introductory course of Computer Organization as part of the Computer Science and Software Engineering curricula. Uniform assessment tools were provided by formal assessment of student performance in the national matriculation examination. The data were collected and statistically analyzed.

The population of the study was divided into two groups:

1) the experimental group, composed of students from nine classes which studied the course using EasyCPU,
2) the control group, composed of 58 students from two other classes which used TASM tools in the same course.

To identify an experiment and a control group of equivalent ability, we used a pretest administered to the students at the end of the first trimester; it covered the topics learned without support of any computer tools. During the first trimester, the course was based only on theoretical learning. The pretest consisted of 20 closed-questions mainly aimed at evaluating student understanding of the theoretical topics learned (the list of the topics is presented in Table 3).

The following statistical analysis was performed on the results of the pretest and the posttest:

1) The test was used to test frequency differences between groups;
2) Student t-test was employed to test differences between the means of the two groups;
3) One-way ANOVA test, followed by the post-hoc Duncan's multiple range test, was used to analyze differences among several groups.

Table 3. Topics covered in pretest and posttest

Topics covered in the pretest	Topics covered in the posttest
Data representation and binary arithmetic	Data representation/binary arithmetic-operations
Inside the CPU & memory organization.	Memory space
Computer units and interconnections	Stack mechanisms
Instruction components- symbolic writing	Instructions structure and addressing modes
Basic translation process:	Detailed translation process:
Instruction level	Program level
Instruction execution processes	Program execution processes
Operations performed by instructions	Control Instructions

Duncan's multiple range test revealed that the mean scores of four classes were similar (i.e. not significantly different among themselves; N = 78). The experimental group consisted of three classes ($E_1 = 71.5$ (SD = 13.5), $E_2 = 69.2$ (SD = 8.1), ($E_3 = 72.2$ (SD = 15.5); N = 55) and one control class ($C_1 = 73.9$ (SD = 11.2); N = 23)). All students were in the eleventh grade and the classes were taught by the teachers who had taught computer science fundamentals the previous year. All teachers had experience teaching the course on Computer Organization and Assembly Language.

The two posttests, namely, the written examination and laboratory examination, were based on the format of the matriculation examination.

Formal assessment

According to the instructions of the Israeli Ministry of Education, the formal assessment was performed with uniform assessment tools used for the matriculation examination. Assessment of student performances was based on a combination of a traditional (i.e. written examination) and a laboratory examination. All examinations were supervised by external examiners appointed by the Ministry of Education.

The written examination included a set of 10 closed-questions. Its weight was 30% of the final grade. The first part was targeted mainly at the materials learned in the first trimester, such as the instruction fetch process and machine codes, with no use of computer. For practical reasons, the examinations were not performed simultaneously. Therefore, questions were selected by examiners from a standard bank of questions classified by topics.

The laboratory examination was performed in a computer laboratory where students were asked to write a short program. Its weight was 70% of the final grade. The grading was based on a set of criteria listed in a standard grading form published by the Ministry of Education (Table 4).

Evaluating understanding of low-level machine

The contribution of the EasyCPU environment to the assimilation and understanding of basic concepts of computer architecture was assessed by a pretest—posttest analysis. The pretest, as described above, was used to identify the control and the experimental group. The posttest consisted of a set of 14 closed-questions administered by the end of the second trimester as a preparation for the matriculation examination, to evaluate the contribution of computer tools (EasyCPU and TASM) to students' understanding. In the posttest, some questions, aimed at evaluating the understanding of topics that were covered in the pretest, were inserted intentionally. The topics covered are presented in Table 4.

The ANOVA test for the variable laboratory examination resulted in $F_{11,232} = 11.41$, $p = 0.0001$. The mean scores of only two experimental classes among the three were significantly higher than the mean score of the control class ($E_1 = 68.7$ (SD = 17.5) and $E_3 = 58.7$ (SD = (18.5)), two experimental classes versus a control class $C_1 = 52.6$ (SD = 16.5)) and the third experiment class's performance was similar to that of the control class ($E_2 = 50.5$ (SD = 14.6)).

Two classes out of three (72% of the students in the experimental group) significantly outperformed the control group in the posttest.

Evaluating the acquirement of programming skills

Evaluation of the effectiveness of the EasyCPU in developing programming skills was based on the laboratory part of the matriculation examination as described above. The same statistical analysis procedure as in the previous survey was performed on the results. The ANOVA test for the variable laboratory examination revealed a significant $F_{10,236} = 2.11$, $p = 0.0242$. Duncan's multiple range test revealed that the mean scores of all three experiment classes were significantly higher than the mean score of the control class ($E_1 = 94.6$ (SD = 7.6), $E_2 = 91.9$ (SD = 9.3), $E_3 = 90.8$ (SD = 13.7), experimental classes versus the control class $C_1 = 82.2$ (SD = 23.4)).

Students who used the EasyCPU environment acquired better programming skills than those who learned with the TASM environment. This study provided us with a quantitative evaluation of the environment's effectiveness, whereas the research deepened our understanding of both 'how' and 'why' the environment contributes to learning.

In the multi-component research, described in [16, 17, 18], qualitative methods were used to improve the understanding of both 'how' and 'why' the environment contributes to learning topics in computer architecture. Here we summarize the findings of this extensive research. Two phases of the research were aimed at examining the use of the environment during the performance of two specific types of activities:

1) program development of a basic embedded system,
2) testing the program.

The findings showed that:

1) concretization of the embedded system pre-

Table 4. Uniform criteria for grading laboratory examination

Examination criteria	Points
Expertise in operating the environment	15
Expertise in the GUI	
Expertise in information retrieval	
Expertise in working modes	
Programming and debugging skills	25
Addressing modes, flags, loop control, variables, stacks, program structure	
Understanding the theoretical aspects of the problems	10
Solution correctness	15
Program structure	5
Total points	70

sented in the simulator helped students in conceptualizing the system and in developing programming skills,
2) the EasyCPU environment facilitated the students' investigations of the detailed behaviour of the program.

In the final phase of the research, described in [17, 18], the contribution of the EasyCPU environment to student understanding of a conceptual model of a computer was evaluated. This was done by investigating the mental models that students construct in the two phases of the learning set-up: before practising and after practising in the environment. The findings support the view that the environment was critical in enabling the construction of a viable mental model, a process that did not occur from textbook learning alone.

Summary of the results

Students who learned with the EasyCPU environment performed a comprehensive set of activities with gradually increased complexity. In the Basic mode, before acquiring basic programming skills, students were asked to follow up the simulated execution of a single instruction and to observe the interaction between the different units. This learning phase facilitates a better understanding of the low-level machine mechanism. In the Advanced mode, students practised basic programming skills; they were asked to test intentionally faulty programs and then to write small programs of their own. The environment provides the student with debugging tools accessible to the novice. The research findings led us to conclude that:

1) the environment renders the displayed information and the program's execution comprehensible;
2) the students' improved control of the tools provided by the environment and the feedback they gain from the visual display encourages them to refine the testing procedure and deepens their comprehension of the program's execution.

The qualitative study conducted on small groups of students provided in-depth insight of students' operations in the environment when developing and testing a program. The qualitative results support and explain the quantitative findings of the survey on the laboratory examination.

Anecdotes of the experience

In order to standardize the examinations' modes of the elective units of the curriculum, the Israeli Ministry of Education decided to replace the traditional matriculation exam in the CS and SE programs with individual mini-projects. The EasyCPU environment, which was originally intended to support novice student learning with a closed set of activities, offers limited tools (a reduced instruction set and I\O controls) and was not adapted for the development of mini-projects. Amazingly, students found several ideas for the development of small embedded mini-projects. They used the simulator to develop something original, implementing a basic embedded system-like panel control of a microwave, a fitness treadmill, alarm and air conditioning systems. For several years we had some ideas in mind, specifically to further develop a more robust environment and to provide the student with more realistic Input/Output modules for developing small embedded systems. We would like to update the design of its GUI, improve the syntax and debugging tools, enhance visual tracing of a program's execution and insert new features such as rewind tools. The environment is still used to teach the unit in spite of the availability of competing up-to-date environments, even though there has been no development and support for several years. In fact, it has exceeded our expectations in terms of lifetime (more than 10 years), as well as in its diversity and widespread use (students' original implementations).

CONCLUSIONS

The most critical decision in designing the simulator was the choice of conceptual model. We opted for a simplified model of a real computer; our goal was to facilitate the entrance of novices into the domain of computer architecture, enabling them to grasp the fundamentals of program execution and to acquire basic assembly language programming skills. Although simplified, the model is compatible with an actual computer in order to ease the novice's transition from EasyCPU to a professional environment. The findings showed that use of the environment can improve students' understanding of basic concepts of computer architecture and programming skills. The long-term, extensive use of the environment has demonstrated that it has successfully satisfied the educational needs of novice students in computer architecture. This achievement can be attributed to the simplicity of the design of the simulator (in spite of its severe limitations) and the activity-set, which were tailored to the course's objectives and content. This experience provided insight and perspectives on simulation-based learning. The key features for fruitful simulation-based learning are the adequacy between course content, learning materials, activities, and the conceptual model illustrated in the simulator and its suitability to targeted populations. These key features can be achieved by learner-centred design accompanied by formative evaluation.

Acknowledgments—We would like to thank the teachers and students who participated in this research.

REFERENCES

1. ACM/IEEE Joint Task Force on Computing Curricula, Computing Curricula 2001 Computer Science Final December, (2000). http://acm.org/education/curric_vols/cc2001.pdf
2. ACM/IEEE Joint Task Force on Computer Engineering Curricula, December, (2004). http://www.eng.auburn.edu/ece/CCCE
3. ACM/IEEE Joint Task Force on Computing Curricula, Software Engineering 2004 Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering, A Volume of the Computing Curricula Series, August, (2004). http://sites.computer.org/ccse/SE2004Volume.pdf
4. L. Hughes, *CSE Special Issue on Teaching Hardware-software*, **17**(2), 2007, pp. 85–86.
5. L. Cassel, D. Kumar, K. Bolding, J. Davies, M. Holliday, J. Impagliazzo, M. Pearson, G. Wolffe, and W. Yurcik, Distributed expertise for teaching computer organization and architecture (ITiCSE 2000 Working Group Report), *ACM SIGCSE Bulletin*, **33**(2), 2001, pp. 111–126.
6. IEEE Micro, *Special Issue on Computer Architecture Education*, **20**(30), 2000.
7. H. T. Salzer and I. Levin, Spreadsheet-based Logic Controller for Teaching Fundamentals of Requirements Engineering. *Int. J. Eng. Educ*. **20**(6), 2004, pp. 939–948.
8. JERIC, Special issue on *specialized* computer architecture simulators that see the present and may hold the future, *J.Educ. Resources in Computing* **1**(4), 2001.
9. D. Knuth, Bottom-up education, *keynote in Proceedings of 8th annual conference on Innovation and technology in computer science education ITiCSE'03*, Thessaloniki, Greece, ACM Press, (2003).
10. M. Bruce-Lockhart, T. S. Norvell and Y. Cotronis, Program and Algorithm Visualization in Engineering and Physics. *Electronic Notes Theoretical Computer Science*, **178**, 2007, pp. 111–119.
11. S. L. Gordon and J. Wolfer, A Python-Based Assembler for a Custom, Robot-Centric, Instruction Set, *Proceedings of the International Conference on Engineering and Computer Education,* 2007, pp. 24–28.
12. I. Pedrosa, A.J. Mendes and M. Zenha Rela, edu.LMC and Other LMC Simulation Approaches: Contributions to Computer Architecture Education Using the LMC Paradigm. *Education for the 21st Century 2006*, (2006) pp. 393–397.
13. G. Wolffe, W. Yurcik, H. Osborne, and M. A. Holliday, Teaching computer organization/architecture with limited resources using simulators, *33th SIGCSE Technical Symposium on Computer Science Education*, Covington, KY, (2002) pp. 176–181.
14. WWW Computer Architecture Page—Simulators, http://www.cs.wisc.edu/~arch/www/tools.html [online], last modified: 29 Feb 2008, (Accessed 31 March 2008).
15. W. Yurcik and Osborne, H., A crowd of little man computers: visual computer simulator teaching tools. *Proceedings of the 33nd Conference on Winter Simulation*, (2001), pp. 1632–1639.
16. C. Yehezkel, C., M. Ben-Ari, and T. Dreyfus, The contribution of visualization to learning computer architecture, *CSE on Special Issue on Teaching Hardware-software*, **2**(17), 2007, pp. 117–127.
17. C. Yehezkel, A Visualization Environment for Computer Architecture, Ph.D. Dissertation, Weizmann Institute of Science, (2004).
18. C. Yehezkel, M. Ben-Ari and T. Dreyfus, Computer architecture and mental models. *36th SIGCSE Technical Symposium on Computer Science Education*. St Louis, MO, (2005) pp. 101–105.
19. L. Davidovitch, A. Parush and A. Shtub, Simulation-based Learning in Engineering Education: Performance and Transfer in Learning Project, *J. Eng. Amer. Soc. Eng. Educ.* **Oct**., 2006, pp. 289–300. http://findarticles.com/p/articles/mi_qa3886/is_200610/ai_n16810355/pg_1
20. C. D. Hundhausen and J. L. Brown, What you see is what you code: a radically dynamic algorithm visualization development model for novice learners, *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, (2005), pp. 163–170.
21. Barbosa, Lula, B. and Lima, A. F., Symbolic and numeric quantum circuit simulation, *proceedings of the First International Conference on Quantum, Nano, and Micro Technologies (ICQNM'07)*, (2007) pp. 6–10.
22. G. Donzellini and D. Ponta, A Simulation Environment for e-Learning in Digital Design, *IEEE Transactions on Industrial Electronics*, **54**(6), 2007, pp. 3078–3085.
23. J. Ma and J.V. Nickerson, Hands-on, simulated, and remote laboratories: A comparative literature review, *ACM Computer Survey*, **38**(3), 2006, pp. 1–24.
24. S. K. Chaturvedi and O. Akan, *Simulation and Visualization Enhanced Engineering Education*, International Mechanical Engineering Education Conference, Beijing, China (2006). www.asme.org/Education/College/2006_Proceedings.cfm
25. J. O. Campbell, R. J. Bourne, P. J. Mosterman and J. A. Brodersen, The Effectiveness of learning simulators in electronic laboratories, *J. Eng. Educ.* **91**(1), 2002 pp. 81–87.
26. C. D. Hundhausen, S. A. Douglas and J. T. Stasko, A meta-study of algorithm visualization effectiveness, *J. Visual Languages and Computing,* **13**(3), 2002, pp. 259–290.
27. C. Kehoe, J. Stasko and A. Taylor, Rethinking the evaluation of algorithm animation as learning aids: An observational study, *Int. J. Hum. Computer Studies*, **54**(2), 2001, pp. 265–284.
28. J. Swaak and T. de Jong, Discovery simulations and the assessment of intuitive knowledge, *J. Comp. Assisted Learning*, **17**(3), 2001, 284–294.
29. L. D. Feisel, and A. J. Rosa, The role of the laboratory in undergraduate engineering education, *J. Eng. Educ.* **94**(1), 2005, pp. 121–130.
30. A. Ko and B. Myers, Designing the Whyline: a debugging interface for asking questions about program failures, in: *Proceedings of the ACM SIGCHI 2004*, ACM Press, New York, 2004, pp. 151–158.

31. J. Gal-Ezer and D. Harel, Curriculum and course syllabi for high school CS program. *Computer Science Education*, **9**(2), 1999, pp. 114–147. http://www.openu.ac.il/Personal_sites/download/galezer/curr_and_syll.pdf
32. B. Haberman and A. Cohen, A high-school program in software engineering, *Int. J. Eng. Educ.* Special Issue: Trends in Pre-college Engineering and Technology Education, **23**(1), 2007, pp. 15–23.
33. H. Zilberman, D. Kraus, D. Lupo, and I. Zeratsky, *Computer Organization and Assembly Language*. Tel Aviv, Open University, (1999).
34. T. de Jong, W. R. van Joolingen, J. Swaak, K. Veermans, R. Limbach, S. King, and D. Guerghian, Self-directed Learning in Simulation-based discovery environments, *J. Comp. Assisted Learning*, **14**, 1998, pp. 235–246.

Cecile Yehezkel received her Ph.D. degree in Science Teaching from the Weizmann Institute of Science. She is one of the heads of an educational programme at the Weizmann Institute of Science, Davidson Institute, and she is teaching at the School of Engineering at Bar-Ilan University. Her research interests focus on human-computer interaction, computer architecture, modelling and simulation and engineering education.

Matzi Eliahu received his Ph.D. from the Technion—Israel Institute of Technology in Haifa and his M.Sc. from Ben-Gurion University in Beer-Sheva. He is currently a lecturer at HIT—Holon Institute of Technology and also at Ariel University Centre of Samaria. His research interests are technology instruction issues at HIT and robotics navigation at Ariel.

Miky Ronen is the chair of the Instructional Systems Technologies Department at the Holon Academic Institute of Technology and a fellow teaching professor at the Department of Education at Haifa University. Her research focuses on the instructional design of interactive learning environments and on the incorporation of technology in the teaching and learning process.

APPENDIX I: A TASK IN BASIC MODE

Instructional goals:

- Identify addressing modes;
- Identify memory access cycles.

Immediate addressing, where the operand is the data itself:

- Select the Copy Data command from the Command menu and build the command MOV CL,1. This is a 'move register, data' command, which inserts a data value into a register. Press OK for confirmation and follow the execution of the command.
- Describe verbally the action performed by the command.

Direct addressing, where the operand is the address of the data:

- Select the Copy Data command from the Command menu and build the command MOV CL,[1].
- This is a 'move register, memory' command, which copies data from memory to a register. Use Direct addressing for the source operand. Press OK for confirmation and follow the execution of the command.
- Describe verbally the action performed by the command. Follow the data transfer along the lines of the addresses, data, and control. As shown in Figure 1, the control line MemR is activated (lights up) and the data is output from the data segment in memory.
- Why do you think this action is called 'reading from memory'?

Indirect addressing, where the operand indicates the name of the register containing the data address; in indirect addressing, the 16-bit register BX functions as a pointer. It is a combination of the BL and BH registers:

- Select Copy Data from the Command menu and build the command MOV CL,[BX].
- This is a 'move register, memory' command. Use Indirect addressing for the target operand. Press OK for confirmation and follow the execution of the command. Follow the data transfer along the address, data, and control lines.
- Describe verbally the way the operation is executed.
- Compare this instruction with the execution of the following instructions:
  MOV CL,1
  MOV CL,[1]
  MOV CL,BX

APPENDIX II: A TASK IN ADVANCED MODE

Instructional goals:

- Learning to characterize syntax and run-time errors, and to debug them;
- Implementing addressing methods;
- Enhancing skills in writing loops;
- Practising how to run and debug a program.

In this activity you will be asked to correct the errors of a student who was asked to write a program that adds the first 5 bytes of the data memory, writes the sum into the memory byte addressed 5, and outputs the sum to an LED port. Here is the main part of the program:

```
mov al, 0
start:  mov bx, 0
        add [5], [bx]
        add bl, 1
        cmp bl, 5
        jnz start
out 2, [5]
```

- Detect and correct the two syntax errors. Recompile until there are no more syntax errors.
- Detect and correct the run-time errors. Run the program and compare the results to the expected ones. Reload the program run it step-by-step in order to detect run-time errors. Check and correct the program.