A Synthesis Course in Hardware Architecture, Compilers, and Software Engineering

Noam Nisan
School of Computer Science
and Engineering
The Hebrew University of Jerusalem
Jerusalem 91904 Israel
972-2-658-5730
noam.nisan@gmail.com

Shimon Schocken
Efi Arazi School of
Computer Science
IDC Herzliya
P.O.B. 167 Herzliya, Israel 46150
972-9-952-7231
shimon@idc.ac.il

ABSTRACT
We describe a synthesis course that provides a hands-on treatment of many hardware and software topics learned in computer science (CS) programs. Using a modular series of twelve projects, we walk the students through the gradual construction of a simple hardware platform and a modern software hierarchy, yielding a basic yet powerful computer system. In the process of building the computer, the students gain a first-hand understanding of how hardware and software systems are designed and how they work together, as one enterprise. The course web site contains all the materials necessary to run this course in open source, and students and instructors are welcome to use and extend them freely. The course projects are modular and self-contained, and any subset of them can be implemented in any order and in any programming language. Therefore, they comprise a flexible library of exercises that can be used in many applied CS courses. This paper gives a description of the approach and the course, juxtaposed against general educational principles underlying meaningful learning.

Categories and Subject Descriptors
1. K.3.2 [Computers and Education]: Computer and Information Science Education – *Computer Science Education.*

General Terms
Design.

Keywords
Design, abstraction, compilers, architecture, software, meaningful learning.

2. INTRODUCTION
Seven years ago, the *IEEE-CS/ACM Computing Science Curricula Report* [7] voiced a concern about the increasing specialization in CS education, and about the students' diminishing ability to focus on major ideas and themes that cut across traditional course lines. Since then, the CS field continued to grow in scope and complexity, and CS courses are becoming more detailed and specialized. As a result of this trend, students often lose the forest for the trees, failing to appreciate the underlying abstractions, interfaces, and contracts that hold the CS discipline together.

We believe that there is a great pedagogical virtue in exposing students to the big picture, without losing rigor. We propose to do so using a theme-oriented approach, in particular courses that draw from multiple CS disciplines in some applied and hands-on context. There have been several proposals describing end-to-end courses that integrate various systems and architecture topics, e.g. [8]. One such course, presently taught in several universities, is described in this paper. The theme of this course is building a complete computer system from scratch – hardware and software – in one semester. The hardware is built using a simple version of HDL and a hardware simulator, and the software can be built in any programming language. As we describe below, the course gives a unified treatment of numerous ideas, concepts, algorithms and techniques studied in other CS courses.

The course and the teaching approach employ many pedagogical principles advocated by the literature on *meaningful learning* [1]. Section 2 gives an overview of these principles, and Section 3 describes the course. Section 4 discusses how meaningful learning principles are employed in the course and the approach.

3. MEANINGFUL LEARNING
Different learning strategies have a profound impact on how learners recall and use the learned concepts. For example, knowledge gained via rote learning is usually triggered only by the context in which it was taught. In contrast, meaningful learning strategies build complex knowledge structures in the learner's mind [2]. As a result, they facilitate recall and extraction of learned concepts through a rich set of triggers and associations. Thus, meaningful learning can yield non-specific transfer [4], in which taught concepts and skills are used "on demand", in new and unexpected contexts. Achieving meaningful learning is a challenging educational objective, especially when it comes to teaching "soft" concepts [5] like abstraction, modularization, and reduction. Such concepts are typically based on fundamental ideas, applicable in multiple domains, and this generality is precisely what makes it difficult to impart them in the framework of a single course. Yet, once a meaningful understanding of soft concepts has been acquired, the very same generality facilitates a significant, powerful and non-specific transfer [15].

Different strategies have been proposed for promoting meaningful learning. Ausubel [1, 2] recommends using *advance organizers* – thematic structures that learners can use to organize and interpret new incoming information [10]. For example, class discussions of "big ideas" and "big picture" perspectives help establish connections between new concepts and already-acquired cognitive structures and underlying themes. In a similar way, Cuoco, Goldenberg and Mark [6] call for organizing teaching around *habits of minds*, e.g. abstraction and modularization, as powerful means for facilitating non-specific transfer. A similar strategy is recommended by Bruner [4], who calls for emphasizing *fundamental ideas* that cut across several areas, and by Schwill [15], who recommends focusing on *CS fundamental ideas*. Another strategy to enhance deep-seated learning is to make it *active*: "What I hear I forget; what I see I remember; what I do I understand" (attributed to Confucius). Indeed, numerous sources, e.g. [13], indicate that when engaged in *learning-by-doing*, students gain better and lasting retention of the learned material. *Contextualization* is also an important factor affecting meaningful learning [12, 16]. In this framework, learning processes are viewed as social activities that take place within the context of a culture, a community, and a history. *Project-based learning* [3] – another highly effective teaching method – is rooted in the frameworks of both learning-by-doing and contextualization, since projects are often drawn from real world contexts that are adjacent to the course topics. Meaningful learning can also benefit from *collaboration* [9].

We reviewed several strategies that, according to the literature, promote meaningful learning. Later in the paper we describe how these strategies come to play in the teaching approach and course that we have developed. We now turn to describe the course itself.

4. THE COURSE

The course that we describe in this paper is presently taught in several universities under a variety of titles ranging from the formal "Digital Systems Construction" to the playful "From Nand to Tetris". The only pre-requisite to this course is introduction to programming, although most students take it after having taken several core courses like algorithms and digital architectures.

The explicit goal of the course is to construct a general purpose computer system in one semester. The implicit goal is to provide a practical exposition of some of the most important abstractions in applied computer science, and to create a powerful synthesis of knowledge gained in key courses like digital architectures, compilers, operating systems, and software engineering. The course achieves this integration through a series of twelve construction projects. Each project is designed to (i) understand an important hardware or software abstraction, (ii) implement and unit-test the abstraction, turning it into an executable module, and (iii) integrate the module with the overall computer system built throughout the course. The computer system is built gradually, and bottom up. The twelve projects and the course are supported by a textbook [11] and a web site that gives all the course lectures, software, and project materials in open source [14]. The complete course plan and project flow is shown in figure 1.

The first four of the twelve projects focus on constructing the chip-set and architecture of a simple Von-Neumann computer. The remaining eight projects evolve around the design and implementation of a typical modern software hierarchy. In particular, we motivate and build an assembler, a virtual machine, a simple Java-like language and its associated compiler, a mini operating system, and a sample high-level application. The overall course consists of three parts, as follows:

Hardware (4 projects): We begin the course with an overview of key topics in gate logic and low-level hardware design. Following this introduction, we describe and build a typical 16-bit chip-set (ALU, registers, CPU, and RAM) using a simple dialect of HDL (Hardware Description Language) that can be self-learned in a few hours. The students implement our chip specifications as HDL programs that can be executed on a visual hardware simulator provided by us. For example, Figure 2 describes a simulation of a typical logic gate (Xor), running on our hardware simulator. The chip API (names of the chip and its I/O pins) and test script are supplied by us; The HDL program is written by the student; The contract is as follows: when the hardware simulator runs the student's HDL program on our test script, it must generate an output file which is identical to our compare-file.

After building a basic chip set, we define an instruction set and guide the students through the process of piecing together the chips into a hardware architecture capable of executing machine language programs written in the given instruction set. Altogether, the students build 30 combinational and sequential chips, culminating in an overall computer-on-a-chip architecture. The simulated computer uses bit-mapped memory segments that allow displaying pixels on the screen and reading scan-codes from a standard keyboard, enabling basic user interaction.

Low-level software (4 projects): After completing the hardware design, we introduce an assembly language specification and guide the students through the process of writing an assembler for it. We then discuss the benefits of a Virtual Machine (VM)

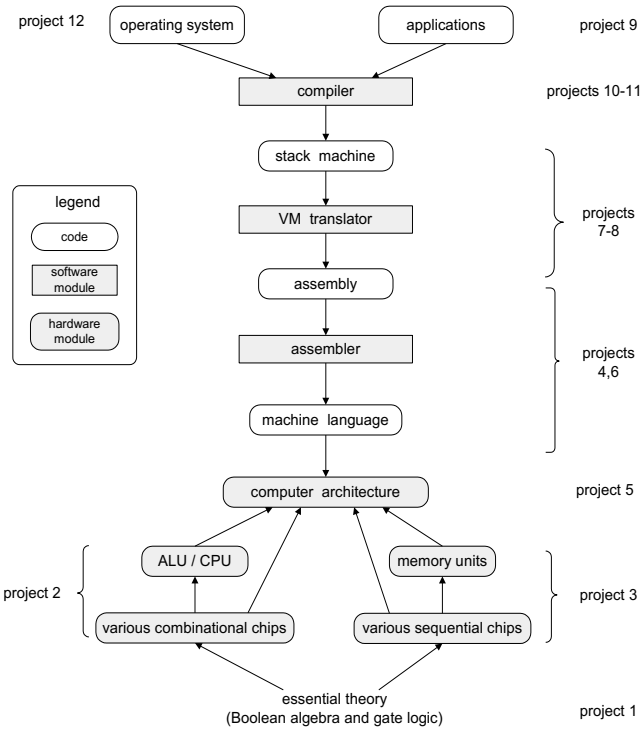

Figure 1. Overall course plan. Each project corresponds to either 1 or 2 course-weeks, and project numbers indicate the normal order in which they are done. E.g. before developing the Jack compiler (projects 10-11), we write an application in the Jack language (project 9), to get acquainted with it.

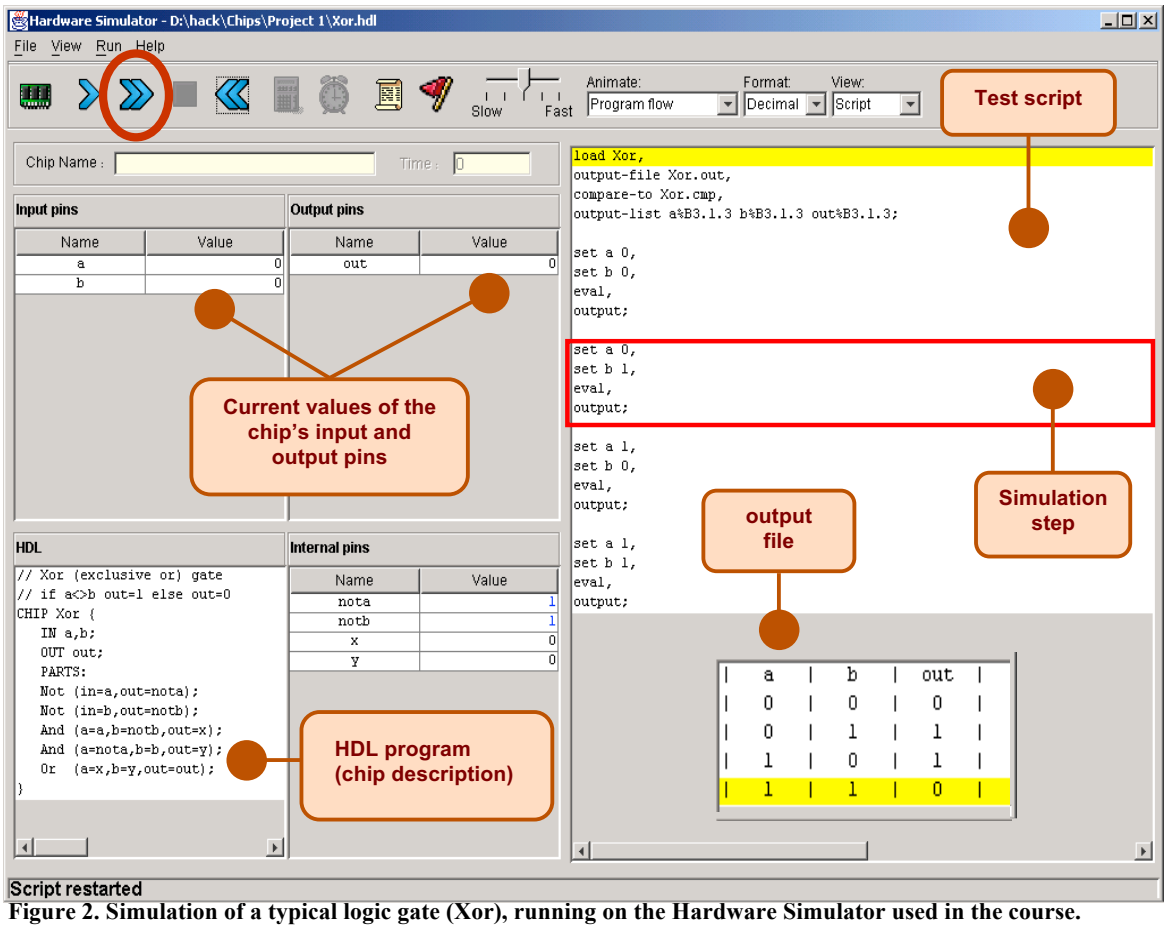

Figure 2. Simulation of a typical logic gate (Xor), running on the Hardware Simulator used in the course.

approach in modern computer systems, and present a stack-based VM abstraction and an associated VM language. Next, we guide the students through the process of writing a VM implementation – a program that translates programs written in this VM language into programs written in our previously implemented assembly language. The resulting code runs on the same computer that the students built in the previous projects. We note in passing that the VM built in our course is modelled after the Java Virtual Machine (JVM) paradigm.

High-level software (4 projects): We begin by introducing the syntax of a simple, Java-like, object-based programming language, called Jack. Following a discussion of key compilation techniques like parsing and code generation, we provide an API and a design plan for constructing a Jack compiler – a program that translates a collection of Jack class files into the previously defined VM code. This completes the basic design of our computer system. Next, we motivate the need for an operating system (OS), and guide the students through the construction of a mini OS, written in Jack. The OS endows the computer with various services like math functions, string processing, input/output functionality, and basic graphics and I/O operations. We celebrate the end of this journey by writing some applications for our computer – typically simple games like Tetris or Space Invaders that lend themselves to object-based and event-driven implementations. For example, Figure 3 presents a Space Invaders game, implemented by one of our students in Jack. The program was then translated by the compiler developed by the students into a VM file written in the VM language (a typical pop/push language) presented in the course. This latter program can either be translated further into a machine language program that can run on the computer chip using the Hardware Simulator (see Figure 2), or it can be executed directly on a VM Emulator, as seen in Figure 3. Each alternative has its own educational benefits.

How can so much ground be covered in a single course? We do it by adhering to several pedagogical and engineering principles. First, we require that the constructed computer system will be fast enough, but no faster. And by "fast enough" we mean that it has to deliver a satisfying user experience. For example, if the computer's graphics is sufficiently smooth to support the animation required by simple computer games, then there is no need to optimize relevant hardware or software modules any further. Thus, the scope of each built module is defined pragmatically: as long as the module passes a set of operational tests supplied by us, there is no need to optimize it further.

In addition, there is no design uncertainty. Too often, CS students are told something like "go build a program that does this and that". The students are then left to their own devices, having to figure out three very different things: how to *design* the program, how to *implement* it, and how to *test* it systematically. Our course

eliminates two-thirds of this uncertainty: we give detailed design specifications and API's for each hardware and software module,

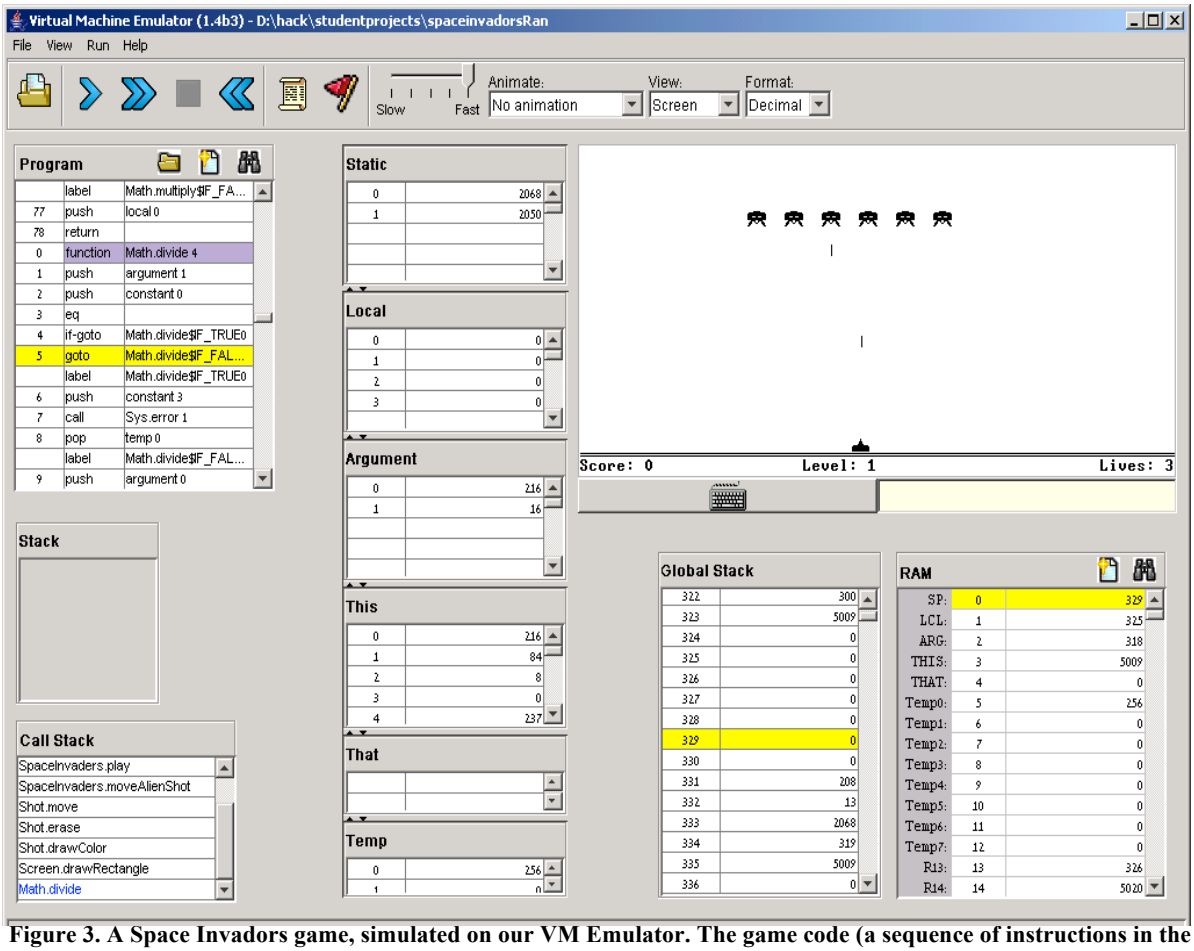

Figure 3. A Space Invadors game, simulated on our VM Emulator. The game code (a sequence of instructions in the VM language) is shown in the top left. The game animation appears in the emulated screen in the top right. All other elements are "inside view" of the Virtual Machine implementation and the RAM that hosts it.

and we provide all the necessary test scripts and programs. In short, we give a step-by-step specification for building and testing the entire computer.

Finally, in any real hardware or software implementation project, most of the work is spent on handling exceptions like faulty inputs and special conditions. In our course, however, we ignore special conditions and assume error-free inputs. For example, when the students develop the adder chip, they can assume that there will be no overflow; and when writing the compiler, they can assume that the source programs contain no syntax errors, and so on. Although learning to handle exceptions is an important educational objective, we believe that it is equally important to assume, at least temporarily, an error-free world. This allows the students to focus on fundamentals ideas and core concepts, rather than spend endless time on handling exceptions, as is sometimes done when writing compilers.

The rationale for these concessions is pragmatic. First, without them, there would be no way to complete this course in one semester. Second, the topics that we leave out of the course (efficiency, design, and error handling) are covered in other CS courses in the program. Third, any one of the limitations inherent in our computer system (and there are many of them, to be sure) provides a clearly stated and well-motivated extension project for advanced students or follow-up courses.

Tools: In order to facilitate the instruction of this course, we built several open-source software tools which can run as-is under either Windows or Unix. In particular, the course web site features a *Hardware Simulator*, a *CPU Emulator*, a *VM Emulator*, and executable *Jack compiler* and *OS*. These programs comprise the complete tool-set necessary for building and testing all the hardware and software systems presented in the course. The students download the software tools to their home computers, and complete the course without needing access to a lab of any sort. Each software tool is accompanied by a detailed tutorial, also available in the course web site.

Experience: So far, about a thousand students have taken the course in four universities (Hebrew University, Haifa University, IDC Herzliya, and Harvard), and many universities used parts of the course materials in various courses. The course is typically positioned as an elective that can be taken at any stage in the program following the introductory programming course. Typically, the course is offered to both undergraduate and graduate students. The course is highly popular, and student feedbacks are exceptionally positive. Outside universities, the course is self-learned by professionals off the web, and their

reactions are quite enthusiastic, as can be seen in the book reviews[1]. We were invited to give many lectures about the course in industry, e.g. a Google tech-talk[2].

5. DISCUSSION

This section discusses some of the educational principles underlying the course, using the meaningful learning framework described in section 2.

Throughout the course, students are exposed to *hard concepts* like RAM design and symbol tables through a set of hands-on construction projects. At the same time, *soft concepts* like abstraction and modularity – which can also be viewed as *habits of minds* – are acquired implicitly through the design and architecture of our projects materials and API's. We break each project into well-defined modules that can be unit-tested separately, and provide all the necessary test programs. This allows us to introduce another important soft concept: multiple points of view. In each project, students are required to assume three different perspectives: user's view, architect's view, and implementer's view. Thus, students are expected to move freely between various levels of abstraction, depending on the context in which the system comes to play. Thus, consistent with the literature's recommendation, soft concepts like abstraction and modularity are not acquired locally, in specific projects, but rather globally, throughout the course.

The computer system that we build throughout the course is constructed bottom-up. In order to promote continuity and enable what the literature calls *advance organizer*s, each project and lecture begins with a description of its immediate neighbours: the previous, lower-level project, viewed as a server that provides abstract building blocks from which the current project can be implemented, and the next, higher-level project, viewed as a client that will later use the abstract services that the current project seeks to implement. This helps the students focus on the "big ideas" that cut through the course and connect new, about-to-be-taught concepts to previous knowledge.

Another key aspect of this course is *active learning*. Students build the computer in a gradual manner, with the level of difficulty increasing from one project to the next. This gradual complexity is also preserved within the individual projects. In each project, students are asked to begin by building a simple version of the needed system, and then extend it into the final implementation. For example, in the assembler project, we begin by building a translator for assembly programs without labels and symbolic variables; next, we build a symbol table and extend the previously-built translator into a full-blown symbolic assembler. Similar gradual constructions are used in building the VM translator, the compiler, and the operating system.

Various aspects of *contextualization* are present: We encourage the students to work on the projects in pairs, fostering collaboration. This places the learning process in a social context. In each lecture the historical background of the taught concepts is discussed, and the likes of Ada Lovelace, Von Neumann, Turing, Shannon are discussed in the very context of their contributions. Finally, and importantly, all the CS concepts and abstractions discussed in this course appear *in-vivo* and *in-context*. That is, concepts like linked lists, hash tables, depth-first search, recursion, and so on, are taught only when they are actually needed in the practical implementation of some module in the built computer. Thus, numerous CS concepts learned in other courses come to play in a live and exciting context – building a general-purpose computer from the ground up.

[1] http://www.amazon.com/review/product/026214087X/ref=dp_top_cm_cr_acr_txt?%5Fencoding=UTF8&showViewpoints=1

[2] http://video.google.com/videoplay?docid=7654043762021156507

6. REFERENCES

[1] Ausubel, D. P. (1963). *The Psychology of Meaningful Verbal Learning.* New York, NY: Grune & Stratton.

[2] Ausubel, D. P. (2000). *The Acquisition and Retention of Knowledge: A Cognitive View*. Dordrecht, the Netherlands: Kluwer Academic Publishers.

[3] Blumenfeld, P. C., Soloway, E. Marx, R. W., Krajcik, J. S., Guzdial, M., and Palinscar, A. (1991). Motivating project-based learning: sustaining the doing, supporting the learning. *Educational Psychologist*, 26 (3&4), 369-398.

[4] Bruner, J. S. (1960). *The Process of Education*. Cambridge, Ma: Harvard University Press.

[5] Corder, C. (1990). *Teaching Hard Teaching Soft: A Structured Approach to Planning and Running Effective Training Courses*. Gower, Aldershot, UK.

[6] Cuoco, A., Goldenberg, E. P. and Mark, J. (1996). Habits of mind: An organizing principle for mathematics curriculum. *J. of Mathematical Behavior*, 15(4), 375-402.

[7] IEEE Computer Society/ACM. (2001). *Computing Curricula 2001: Final Report*. Available on hwww.sigcse.org/cc2001/, retrieved July 6th, 2008.

[8] Johnson, S. and Ruehr, F. (2005). An integrated course in architecture and compilers. *J. of Computing Sciences in Colleges*, 21(1), 191-198.

[9] Jonassen, D. H., Peck, K. L., and Wilson B. G. (1999). *Learning with Technology: A Constructivist Perspective*. Upper Saddle River, NJ: Merrill Publishing.

[10] Mayer, R. (2003). *Learning and Instruction*. New Jersey: Pearson Education, Inc.

[11] Nisan, N. and Schocken, S. (2005). *The Elements of Computing Systems*. Cambridge, MA: MIT Press.

[12] van Oers, B. (1996). Learning mathematics as a meaningful activity. In L. P. Steffe, P. Nesher, P. Cobb, G. A. Goldin, and B. Greer (Eds.) *Theories of Mathematical Learning*. Hillsdale, NJ: Lawrence Erlbaum Associates. 91-114.

[13] Schank, R.C., Berman, T. R., and Macpherson, K. A. (1999). Learning by doing. In C. M. Reigeluth (Ed.) *Instructional-Design Theories and Models: A new paradigm of instructional technology*, vol. II. Mahwah, NJ: Lawrence Erlbaum Associates. 161-182.

[14] Schocken, S., Nisan, N. (2005): TECS Course site, www.idc.ac.il/tecs

[15] Schwill, A. (1994). Fundamental ideas of computer science. *Bulletin of European Association for Theoretical Computer Science*, 53, 274-295.

[16] Vygotsky, L. S. (1962). *Thought and Language*. Trans. by E. Hanfmann and G. Vakar. Cambridge, MA: MIT Press.