A Synthesis Course in Hardware Architecture, Compilers and Software Engineering

Shimon Schocken
IDC Herzliya

Noam Nisan
Hebrew University

Michal Armoni
Weizmann Institute

March, 2009

ABSTRACT

We describe a synthesis course that provides a hands-on and cross-section 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.

INTRODUCTION

In 2001, the *IEEE-CS/ACM Computing Science Curricula Report* 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 [1]. In the decade that followed the publication of this report, the CS field continued to grow in scope and complexity, and CS courses have become more detailed and specialized. As a result of this trend, students continue to lose the forest for the trees, failing to appreciate the underlying abstractions, interfaces, and contracts that hold the applied CS discipline together.

Creating synthesis courses that give an integrated view of the complex and sprawling field of applied CS is a challenging undertaking. Such courses can crumble under their own weight, becoming either too ambitious, or too simplistic. Yet the effort is well spent, as there is a great pedagogical virtue in exposing students to the big picture, without losing rigor. Indeed, we are certainly not the first, and hopefully not the last, to propose the development of synthesis courses in applied CS, e.g. [2] and [3].

During the last ten years we've been developing and teaching one such course, which is the subject of this paper. Unlike other courses in this category, ours is an "end-to-end" affair, in which students build a complete general-purpose computer system – hardware and software – in one semester. The hardware platform is built from the ground up using a simple version of HDL and a supplied hardware simulator, and the software hierarchy is built by realizing a series of given API's that can be implemented in any programming language.

All our lectures, course materials, API's, software tools and projects, as well as all the resources mentioned in this paper, are available on the web, freely and in open source [4]. A book describing the approach is also available [5].

THE COURSE

The course objective is to give an integrated, hands-on view of applied computer science, as it unfolds in the process of building a general-purpose computer system – hardware and software – in one semester. The course is taught in traditional university settings as well as in self-organized web-based settings, 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; all the hardware, compilation, and systems knowledge necessary for building the computer are gained in the course itself.

The explicit goal of the course is to construct a general-purpose computer system from first principles. 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 key topics from 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 complete course plan and project flow is shown in figure 1.

Note that each module in the course plan treats the module below it as a black-box abstraction. For example, the compiler of our high-level language -- called *Jack* -- generates VM code without worrying how the VM language is implemented; the VM translator, in turn, generates assembly code without worrying how the assembly language is implemented, and so on. This modularity enables teaching the course in any desired order, although the "full experience" entails building the complete computer from the ground up, in the order shown in Figure 1.

The first five of the twelve projects focus on constructing the chip-set and architecture of a simple Von-Neumann hardware platform, called *Hack*. 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 compiler for a simple Java-like language, a basic operating system, and a sample high-level application. The overall course consists of three parts, as we now turn to describe.

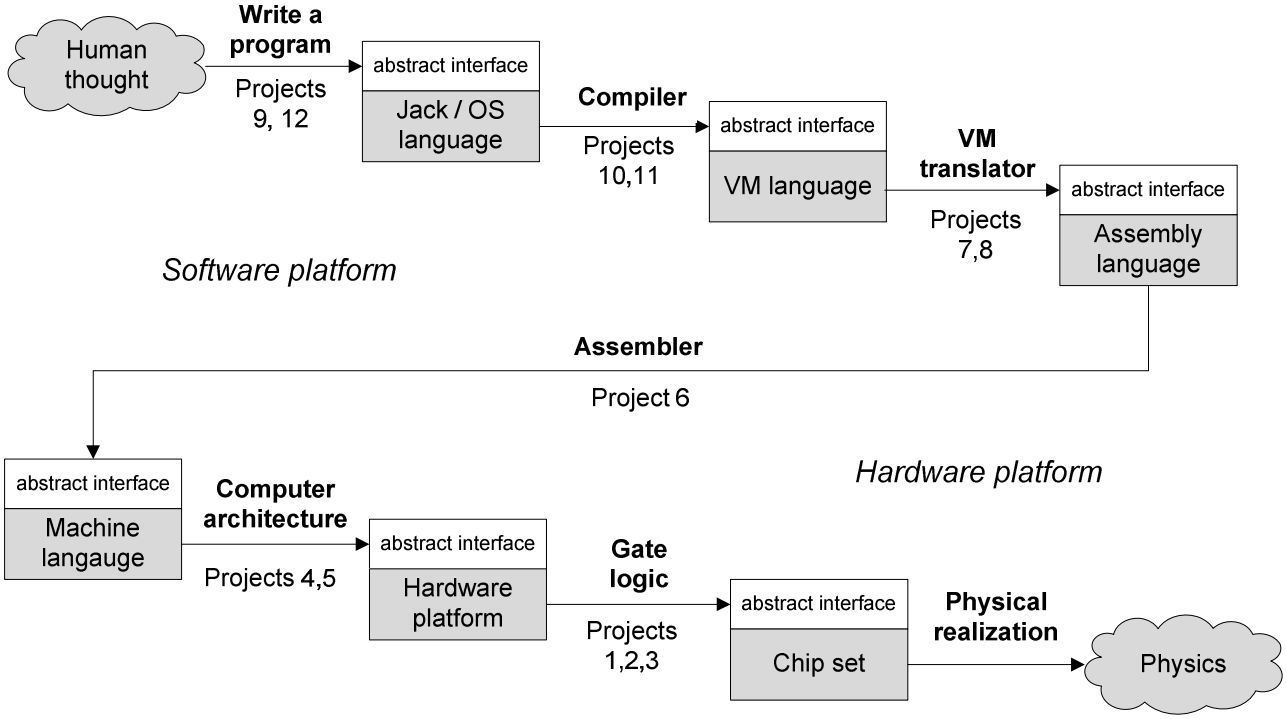

Figure 1. The course outline, comprising 12 projects and instructional modules in which the students build a chip set, a computer architecture, an assembler, a VM translator, a compiler for a simple Java-like language, a basic operating system, and an illustrative application.

Hardware (projects 1-5): 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 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 supplied 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 the supplied test script, it must generate an output file which is identical to the supplied compare-file.

After building and unit-testing a basic chip set, we present an instruction set and guide the students through the process of piecing together the previously-built chips into hardware architecture capable of executing machine language programs written in the given instruction set. Altogether, the students build 33 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 a simulated screen and reading scan-codes from a standard keyboard, enabling basic user interaction.

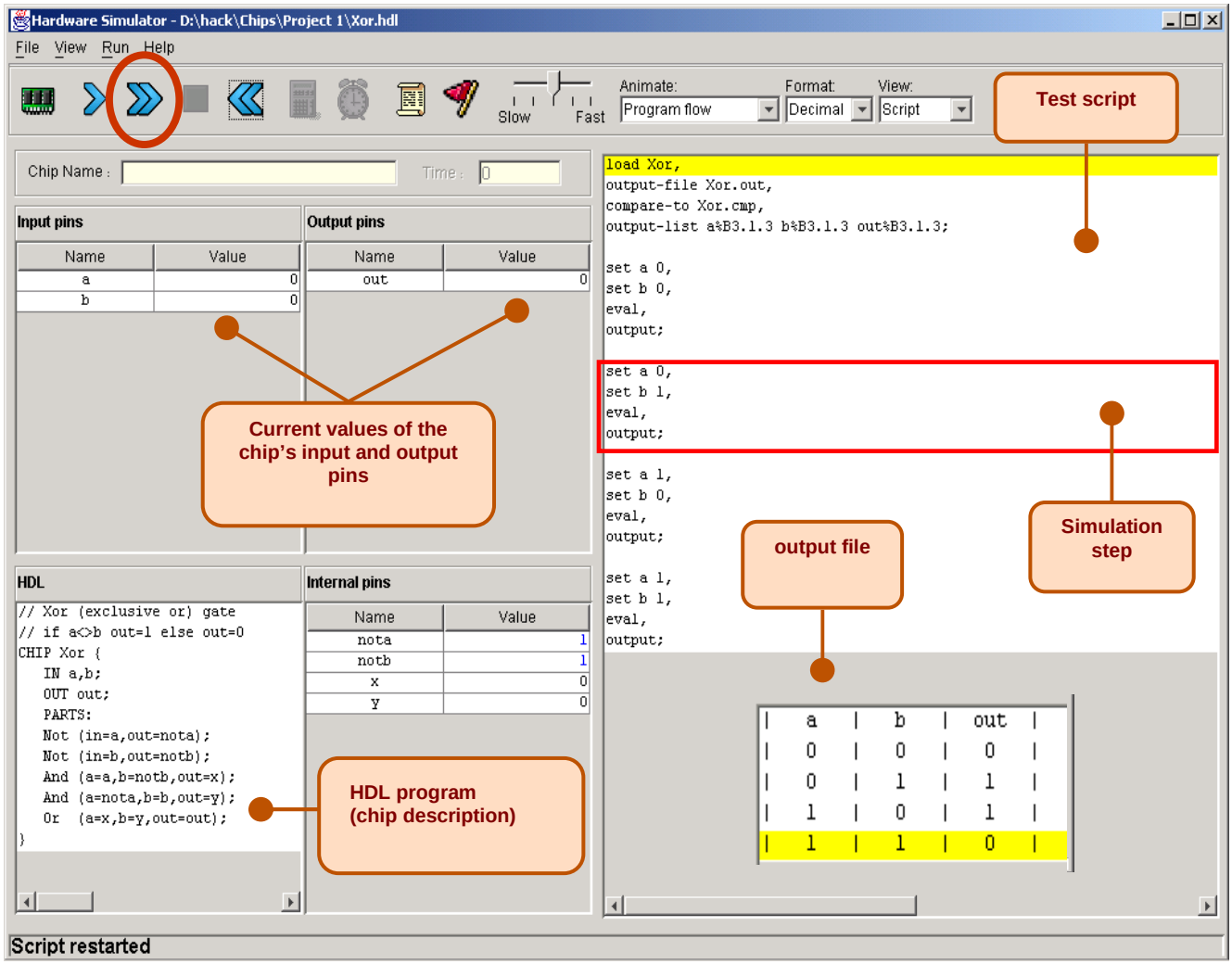

Figure 2. Simulation of a typical logic gate (Xor),
running on the supplied hardware simulator used in the course.

Intermediate software (projects 6-8): 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 and role of a Virtual Machine (VM) approach in modern computer systems, and present a stack-based VM abstraction and an associated VM language resembling a simple version of Java's Bytecode. 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 projects 1-5. We note in passing that the VM built in our course is quite similar to Java's JVM and .NET's CLR paradigms.

High-level software (projects 9-12): 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 (next page) presents a Space Invaders game, implemented in Jack by one of our students. The program -- about 500 hundred lines of Jack code -- was then translated by the compiler developed by the student into a VM file written in our VM language (a typical pop/push language). 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.

Tools: In order to facilitate the instruction of the course and the construction of the projects, 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 any additional lab services. Each software tool is accompanied by a detailed tutorial, also available in the course web site.

METHODOLOGY

The computer system that we build throughout the course is constructed bottom-up. To promote continuity, each project and lecture begins with a description of its immediate neighbours: the previously developed lower-level module, viewed now as a black-box abstraction providing abstract building blocks from which the current module can be implemented, and the next, higher-level module, viewed as a client that will later use the abstract services that the current project seeks to implement. This perspective helps the students focus on the big ideas that cut through the course and connect new, about-to-be-taught concepts to previous knowledge.

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 implement a symbol table API and extend the previously-built translator into a full-blown symbolic assembler. Similar gradual constructions are used to stage and unit-test the development of the VM translator, compiler, and operating system.

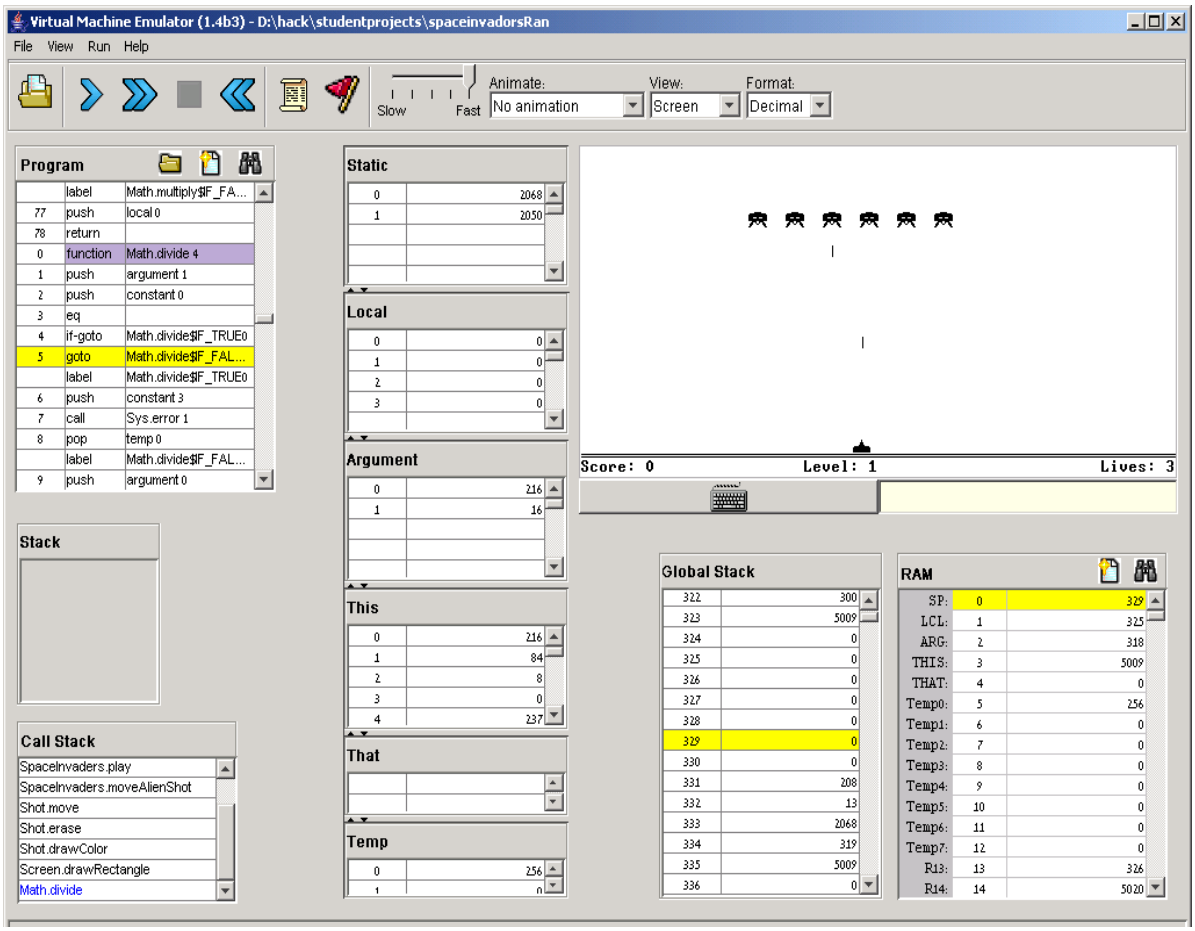

Figure 3. Space Invadors game, simulated on the supplied VM Emulator.
The game code (a sequence of instructions in the VM language) is shown in the top left. The game animation appears in the emulated screen in the top right. All other elements are "inside view" of the Virtual Machine implementation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

In each lecture, we strive to mention the historical background of the taught concepts. The heroes of the course are the likes of Ada Lovelace, John Von Neumann, Allan Turing, and Claude Shannon, and their contributions to applied CS are discussed in the very context of the modules in which they come to play. 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, many key CS concepts learned in other courses come to play in a live and exciting context – building a general-purpose computer from the ground up.

COMPLEXITY

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, 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. For a more complete discussion of our approach on containing complexity, see

FIELD EXPERIENCE

Since we've made the course materials freely available on the web, thousands of students from India, China, Africa, and western countries have built the Hack/Jack computer system. This learning experience took place in traditional academic settings as well as in self-organized study groups, ranging from Harvard University to the University of the People. The vast majority of the students who completed this course knew nothing about hardware or compilation upon starting their journey. And yet many of them were happy campers, expressing publicly the joy and satisfaction they've experienced building the machine.

In addition to regular students, the course is taken by many self-learners, having nothing but intellectual curiosity and pursuit of knowledge to motivate their work (for a typical self-organized course site, see [7]). The course is typically positioned as an elective that can be taken at any stage in the program, following an introductory programming course.

REFERENCES

[1] Computing Curricula 2001. Final Report, IEEE Computer Society/ACM. (2001). www.sigcse.org/cc2001, Page 27.

[2] Bryant, R.E. and Hallaron, D.R., 2010, Computer Systems: A Programmer's Perspective, *Addison Wesley; 2nd edition.*

[3] Patt, Y. and Patel, 1999, S. Introduction to Computer Systems: from Bits and Gates to C and Beyond, *Mcgraw-Hill*.

[4] Nisan, N., Schocken, S. 2005. *TECS Course Web Site*. www.idc.ac.il/tecs

[5] Nisan, N., Schocken, S. 2005. The Elements of Computing Systems. Cambridge, MA: MIT Press.

[6] Schocken, S., 2009, Taming Complexity in Large Scale Systems Projects, *working paper*.

[7] A do-it-yourself CS course, based on the approach, developed and administered by Parag Shah: http://tinyurl.com/3jotumk