

# Preface

*What I hear, I forget; What I see, I remember; What I do, I understand.*  
—Confucius, 551–479 BC

Once upon a time, every computer specialist had a gestalt understanding of how computers worked. The overall interactions among hardware, software, compilers, and the operating system were simple and transparent enough to produce a coherent picture of the computer’s operations. As modern computer technologies have become increasingly more complex, this clarity is all but lost: the most fundamental ideas and techniques in computer science—the very essence of the field—are now hidden under many layers of obscure interfaces and proprietary implementations. An inevitable consequence of this complexity has been specialization, leading to computer science curricula of many courses, each covering a single aspect of the field.

We wrote this book because we felt that many computer science students are missing the forest for the trees. The typical student is marshaled through a series of courses in programming, theory, and engineering, without pausing to appreciate the beauty of the picture at large. And the picture at large is such that hardware and software systems are tightly interrelated through a hidden web of abstractions, interfaces, and contract-based implementations. Failure to see this intricate enterprise in the flesh leaves many students and professionals with an uneasy feeling that, well, they don’t fully understand what’s going on inside computers.

We believe that the best way to understand how computers work is to build one from scratch. With that in mind, we came up with the following concept. Let’s specify a simple but sufficiently powerful computer system, and have the students build its hardware platform and software hierarchy from the ground up, starting with nothing more than elementary logic gates. And while we are at it, let’s do it right. We say this because building a general-purpose computer from first principles is a huge undertaking. Therefore, we identified a unique educational opportunity not only to

build the thing, but also to illustrate, in a hands-on fashion, how to effectively plan and manage large-scale hardware and software development projects. In addition, we sought to demonstrate the ability to construct, through recursive ascent and human reasoning, fantastically complex and useful systems from nothing more than a few primitive building blocks.

---

## Scope

The book exposes students to a significant body of computer science knowledge, gained through a series of hardware and software construction tasks. These tasks demonstrate how theoretical and applied techniques taught in other computer science courses are used in practice. In particular, the following topics are illustrated in a hands-on fashion:

- *Hardware:* Logic gates, Boolean arithmetic, multiplexors, flip-flops, registers, RAM units, counters, Hardware Description Language (HDL), chip simulation and testing.
- *Architecture:* ALU/CPU design and implementation, machine code, assembly language programming, addressing modes, memory-mapped input/output (I/O).
- *Operating systems:* Memory management, math library, basic I/O drivers, screen management, file I/O, high-level language support.
- *Programming languages:* Object-based design and programming, abstract data types, scoping rules, syntax and semantics, references.
- *Compilers:* Lexical analysis, top-down parsing, symbol tables, virtual stack-based machine, code generation, implementation of arrays and objects.
- *Data structures and algorithms:* Stacks, hash tables, lists, recursion, arithmetic algorithms, geometric algorithms, running time considerations.
- *Software engineering:* Modular design, the interface/implementation paradigm, API design and documentation, proactive test planning, programming at the large, quality assurance.

All these topics are presented with a very clear purpose: building a modern computer from the ground up. In fact, this has been our topic selection rule: The book focuses on the minimal set of topics necessary for building a fully functioning computer system. As it turns out, this set includes many fundamental ideas in applied computer science.

## Courses

The book is intended for students of computer science and other engineering disciplines in colleges and universities, at both the undergraduate and graduate levels. A course based on this book is “perpendicular” to the normal computer science curriculum and can be taken at almost any point during the program. Two natural slots are “CS-2”—immediately after learning programming, and “CS-199”—a capstone course coming at the end of the program. The former course can provide a systems-oriented introduction to computer science, and the latter an integrative, project-oriented systems building course. Possible names for such courses may be Constructive Introduction to Computer Science, Elements of Computing Systems, Digital Systems Construction, Computer Construction Workshop, Let’s Build a Computer, and the like. The book can support both one- and two-semester courses, depending on topic selection and pace of work.

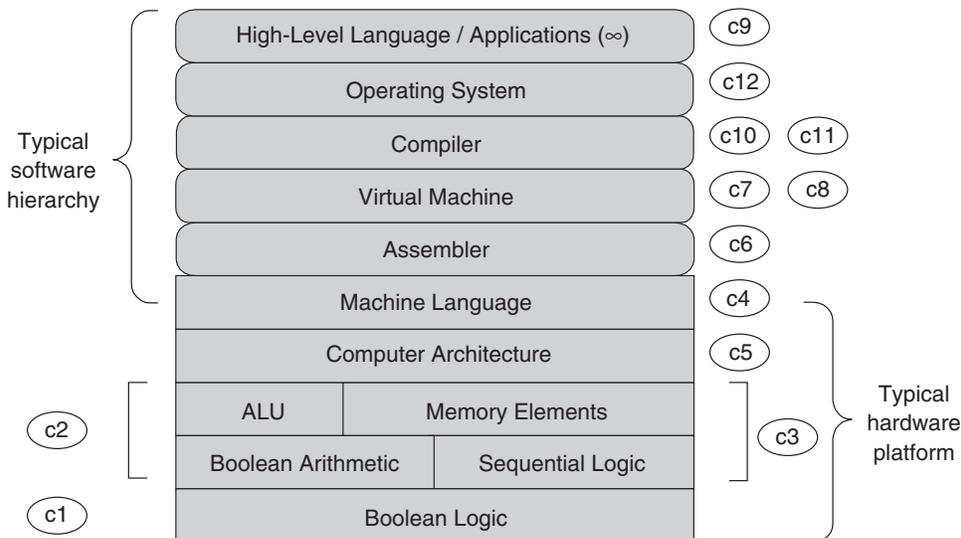
The book is completely self-contained, requiring only programming (in any language) as a prerequisite. Thus, it lends itself not only to computer science majors, but also to computer-savvy students seeking to gain a hands-on view of hardware architectures, operating systems, and modern software engineering in the framework of one course. The book and the accompanying Web site can also be used as a self-study learning unit, suitable to students from any technical or scientific discipline following a programming course.

---

## Structure

The introduction chapter presents our approach and previews the main hardware and software abstractions discussed in the book. This sets the stage for chapters 1–12, each dedicated to a key hardware or software abstraction, a proposed implementation, and an actual project that builds and tests it. The first five chapters focus on constructing the hardware platform of a simple modern computer. The remaining seven chapters describe the design and implementation of a typical multi-tier software hierarchy, culminating in the construction of an object-based programming language and a simple operating system. The complete game plan is depicted in figure P.1.

The book is based on an *abstraction-implementation* paradigm. Each chapter starts with a Background section, describing relevant concepts and a generic hardware or software system. The next section is always Specification, which provides a clear



**Figure P.1** Book and proposed course map, with chapter numbers in circles.

statement of the system’s abstraction—namely, the various services that it is expected to deliver. Having presented the *what*, each chapter proceeds to discuss *how* the abstraction can be implemented, leading to a (proposed) Implementation section. The next section is always Perspective, in which we highlight noteworthy issues left out from the chapter. Each chapter ends with a Project section that provides step-by-step building instructions, testing materials, and software tools for actually building and unit-testing the system described in the chapter.

---

## Projects

The computer system described in the book is *for real*—it can actually be built, and it works! A reader who takes the time and effort to gradually build this computer will gain a level of intimate understanding unmatched by mere reading. Hence, the book is geared toward active readers who are willing to roll up their sleeves and build a computer from the ground up.

Each chapter includes a complete description of a stand-alone hardware or software development project. The four projects that construct the computer platform are built using a simple Hardware Description Language (HDL) and simulated on a hardware simulator supplied with the book. Five of the subsequent software projects

(assembler, virtual machine I and II, and compiler I and II) can be written in any modern programming language. The remaining three projects (low-level programming, high-level programming, and the operating system) are written in the assembly language and high-level language implemented in previous projects.

**Project Tips** There are twelve projects altogether. On average, each project entails a weekly homework load in a typical, rigorous university-level course. The projects are completely self-contained and can be done (or skipped) in any desired order. Of course the “full experience” package requires doing all the projects in their order of appearance, but this is just one option.

When we teach courses based on this book, we normally make two significant concessions. First, except for obvious cases, we pay no attention to optimization, leaving this very important subject to other, more specific courses. Second, when developing the translators suite (assembler, VM implementation, and compiler), we supply error-free test files (source programs), allowing the students to assume that the inputs of these translators are error-free. This eliminates the need to write code for handling errors and exceptions, making the software projects significantly more manageable. Dealing with incorrect input is of course critically important, but once again we assume that students can hone this skill elsewhere, for example, in dedicated programming and software design courses.

---

## Software

The book’s Web site ([www.idc.ac.il/tecs](http://www.idc.ac.il/tecs)) provides the tools and materials necessary to build all the hardware and software systems described in the book. These include a hardware simulator, a CPU emulator, a VM emulator, and executable versions of the assembler, virtual machine, compiler, and operating system described in the book. The Web site also includes all the project materials—about two hundred test programs and test scripts, allowing incremental development and unit-testing of each one of the twelve projects. All the supplied software tools and project materials can be used as is on any computer equipped with either Windows or Linux.

---

## Acknowledgments

All the software that accompanies the book was developed by our students at the Efi Arazi School of Computer Science of the Interdisciplinary Center Herzliya, a new

Israeli university. The chief software architect was Yaron Ukrainitz, and the developers included Iftach Amit, Nir Rozen, Assaf Gad, and Hadar Rosen-Sior. Working with these student-developers has been a great pleasure, and we feel proud and fortunate to have had the opportunity to play a role in their education. We also wish to thank our teaching assistants, Muawyah Akash, David Rabinowitz, Ran Navok, and Yaron Ukrainitz, who helped us run early versions of the course that led to this book. Thanks also to Jonathan Gross and Oren Baranes, who worked on related projects under the excellent supervision of Dr. Danny Seidner, to Uri Zeira and Oren Cohen, for designing an integrated development environment for the Jack language, to Tal Achituv, for useful advice on open source issues, and to Aryeh Schnall, for careful reading and meticulous editing suggestions.

Writing the book without taking any reduction in our regular professional duties was not simple, and so we wish to thank esti romem, administrative director of the EFI Arazi School of Computer Science, for holding the fort in difficult times. Finally, we are indebted to the many students who endured early versions of this book and helped polish it through numerous bug reports. In the process, we hope, they have learned first-hand that insight of James Joyce, that *mistakes are the portals of discovery*.

Noam Nisan

Shimon Schocken