

# PREFACE

This textbook has evolved from EECS 100, the first computing course for computer science, computer engineering, and electrical engineering majors at the University of Michigan, that Kevin Compton and the first author introduced for the first time in the fall term, 1995.

EECS 100 happened because Computer Science and Engineering faculty had been dissatisfied for many years with the lack of student comprehension of some very basic concepts. For example, students had a lot of trouble with pointer variables. Recursion seemed to be “magic,” beyond understanding.

We decided in 1993 that the conventional wisdom of starting with a high-level programming language, which was the way we (and most universities) were doing it, had its shortcomings. We decided that the reason students were not getting it was that they were forced to memorize technical details when they did not understand the basic underpinnings.

The result is the bottom-up approach taken in this book. We treat (in order) MOS transistors (very briefly, long enough for students to grasp their global switch-level behavior), logic gates, latches, logic structures (MUX, Decoder, Adder, gated latches), finally culminating in an implementation of memory. From there, we move on to the Von Neumann model of execution, then a simple computer (the LC-2), machine language programming of the LC-2, assembly language programming of the LC-2, the high level language C, recursion, pointers, arrays, and finally some elementary data structures.

We do not endorse today’s popular information hiding approach when it comes to learning. Information hiding is a useful productivity enhancement technique after one understands what is going on. But until one gets to that point, we insist that information hiding gets in the way of understanding. Thus, we continually build on what has gone before, so that nothing is magic, and everything can be tied to the foundation that has already been laid.

We should point out that we do not disagree with the notion of top-down *design*. On the contrary, we believe strongly that top-down design is correct design. But there is a clear difference between how one approaches a design problem (after one understands the underlying building blocks), and what it takes to get to the point where one does understand the building blocks. In short, we believe in top-down design, but bottom-up learning for understanding.

---

## WHAT IS IN THE BOOK

The book breaks down into two major segments, a) the underlying structure of a computer, as manifested in the LC-2; and b) programming in a high level language, in our case C.

### The LC-2

We start with the underpinnings that are needed to understand the workings of a real computer. Chapter 2 introduces the bit and arithmetic and logical operations on bits. Then we begin to build the structure needed to understand the LC-2. Chapter 3 takes the student from MOS transistor, step by step, to a real memory. Our real memory consists of 4 words of 3 bits each, rather than 64 megabytes. The picture fits on a single page (Figure 3.20), making it easy for a student to grasp. By the time the students get there, they have been exposed to all the elements

that make memory work. Chapter 4 introduces the Von Neumann execution model, as a lead-in to Chapter 5, the LC-2.

The LC-2 is a 16-bit architecture that includes physical I/O via keyboard and monitor; TRAPs to the operating system for handling service calls; conditional branches on N, Z, and P condition codes; a subroutine call/return mechanism; a minimal set of operate instructions (ADD, AND, and NOT); and various addressing modes for loads and stores (direct, indirect, base+offset, and an immediate mode for loading effective addresses).

Chapter 6 is devoted to programming methodology (stepwise refinement) and debugging, and Chapter 7 is an introduction to assembly language programming. We have developed a simulator and an assembler for the LC-2. Actually, we have developed two simulators, one that runs on Windows platforms and one that runs on UNIX. The Windows simulator is available on the website and on the CD-ROM. Students who would rather use the UNIX version can download and install the software from the web at no charge.

Students use the simulator to test and debug programs written in LC-2 machine language and in LC-2 assembly language. The simulator allows on-line debugging (deposit, examine, single-step, set breakpoint, and so on). The simulator can be used for simple LC-2 machine language and assembly language programming assignments, which are essential for students to master the concepts presented throughout the first 10 chapters.

Assembly language is taught, but not to train expert assembly language programmers. Indeed, if the purpose was to train assembly language programmers, the material would be presented in an upper-level course, not in an introductory course for freshmen. Rather, the material is presented in Chapter 7 because it is consistent with the paradigm of the book. In our bottom-up approach, by the time the student reaches Chapter 7, he/she can handle the process of transforming assembly language programs to sequences of 0s and 1s. We go through the process of assembly step-by-step for a very simple LC-2 Assembler. By hand assembling, the student (at a very small additional cost in time) reinforces the important fundamental concept of translation.

It is also the case that assembly language provides a user-friendly notation to describe machine instructions, something that is particularly useful for the second half of the book. Starting in Chapter 11, when we teach the semantics of C statements, it is far easier for the reader to deal with ADD R1, R2, R3 than with 0001001010000011.

Chapter 8 deals with physical input (from a keyboard) and output (to a monitor). Chapter 9 deals with TRAPs to the operating system, and subroutine calls and returns. Students study the operating system routines (written in LC-2 code) for carrying out physical I/O invoked by the TRAP instruction.

The first half of the book concludes with Chapter 10, a treatment of stacks and data conversion at the LC-2 level, and a comprehensive example that makes use of both. The example is the simulation of a calculator, which is implemented by a main program and 11 subroutines.

## The Language C

From there, we move on to C. The C programming language occupies the second half of the book. By the time the student gets to C, he/she has an understanding of the layers below.

The C programming language fits very nicely with our bottom-up approach. Its low-level nature allows students to see clearly the connection between software and the underlying hardware. In this book we focus on basic concepts such as control structures, functions, and arrays. Once basic programming concepts are mastered, it is a short step for students to learn more advanced concepts such as objects and abstraction.

Each time a new construct in C is introduced, the student is shown the LC-2 code that a compiler would produce. We cover the basic constructs of C (variables, operators, control, functions), pointers, recursion, arrays, structures, I/O, complex data structures, and dynamic allocation.

Chapter 11 is a gentle introduction to high-level programming languages. At this point, students have dealt heavily with assembly language and can understand the motivation behind what high-level programming languages provide. Chapter 11 also contains a simple C program, which we use to kick-start the process of learning C.

Chapter 12 deals with values, variables, constants, and operators. Chapter 13 introduces C control structures. We provide many complete program examples to give students a sample of how each of these concepts are used in practice. LC-2 code is used to demonstrate how each C construct affects the machine at the lower levels.

In Chapter 14, students are exposed to techniques for debugging high-level source code. Chapter 15 introduces functions in C. Students are not merely exposed to the syntax of functions. Rather they learn how functions are actually executed using a run-time stack. A number of examples are provided.

Chapter 16 teaches recursion, using the student's newly gained knowledge of functions, activation records, and the run-time stack. Chapter 17 teaches pointers and arrays, relying heavily on the students' understanding of how memory is organized. Chapter 18 introduces the details of I/O functions in C, in particular, streams, variable length argument lists, and how C I/O is affected by the various format specifications. This chapter relies on the student's earlier exposure to physical I/O in Chapter 8. Chapter 19 concludes the coverage of C with structures, dynamic memory allocation, and linked lists.

Along the way, we have tried to emphasize good programming style and coding methodology by means of examples. Novice programmers probably learn at least as much from the programming examples they read as from the rules they are forced to study. Insights that accompany these examples are highlighted by means of lightbulb icons that are included in the margins.



We have found that the concept of pointer variables (Chapter 17) is not at all a problem. By the time students encounter it, they have a good understanding of what memory is all about, since they have analyzed the logic design of a small memory (Chapter 3). They know the difference, for example, between a memory location's address and the data stored there.

Recursion ceases to be magic since, by the time a student gets to that point (Chapter 16), he/she has already encountered all the underpinnings. Students understand how stacks work at the machine level (Chapter 10), and they understand the call/return mechanism from their LC-2 machine language programming experience, and the need for linkages between a called program and the return to the caller (Chapter 9). From this foundation, it is not a large step to explain functions by introducing run-time activation records (Chapter 15), with a lot of the mystery about argument passing, dynamic declarations, and so on, going away. Since a function can call a function, it is one additional small step (certainly no magic involved) for a function to call itself.

## How to Use this Book

We have discovered over the past two years that there are many ways the material in this book can be presented in class effectively. We suggest six presentations below.

1. The Michigan model. First course, no formal prerequisites. Very intensive, this course covers the entire book. We have found that with talented, very highly motivated students, this works best.

2. Normal usage. First course, no prerequisites. This course is also intensive, although less so. It covers most of the book, leaving out Sections 10.3 and 10.4 of Chapter 10, Chapters 16 (recursion), 18 (the details of C I/O), and 19 (data structures).
3. Second course. Several schools have successfully used the book in their second course, after the students have been exposed to programming with an object-oriented programming language in a milder first course. In this second course, the entire book is covered, spending the first two-thirds of the semester on the first 10 chapters, and the last one-third of the semester on the second half of the book. The second half of the book can move more quickly, given that it follows both Chapter 1–10 and the introductory programming course, which the student has already taken. Since students have experience with programming, lengthier programming projects can be assigned. This model allows students who were introduced to programming via an object-oriented language to pick up C, which they will certainly need if they plan to go on to advanced software courses such as operating systems.
4. Two quarters. An excellent use of the book. No prerequisites, the entire book can be covered easily in two quarters, the first quarter for Chapters 1–10, the second quarter for Chapters 11–19.
5. Two semesters. Perhaps the optimal use of the book. A two-semester sequence for freshmen. No formal prerequisites. First semester, Chapters 1–10, with supplemental material from Appendix C, the Microarchitecture of the LC-2. Second semester, Chapters 11–19 with additional substantial programming projects so that the students can solidify the concepts they learn in lectures.
6. A sophomore course in computer hardware. Some universities have found the book useful for a sophomore level breadth-first survey of computer hardware. They wish to introduce students in one semester to number systems, digital logic, computer organization, machine language and assembly language programming, finishing up with the material on stacks, activation records, recursion, and linked lists. The idea is to tie the hardware knowledge the students have acquired in the first part of the course to some of the harder to understand concepts that they struggled with in their freshman programming course. We strongly believe the better paradigm is to study the material in this book before tackling an object-oriented language. Nonetheless, we have seen this approach used successfully, where the sophomore student gets to understand the concepts in this course, after struggling with them during the freshman year.

## Some Observations

**Understanding, not Memorizing** Since the course builds from the bottom up, we have found that less memorization of seemingly arbitrary rules is required than in traditional programming courses. Students understand that the rules make sense since by the time a topic is taught, they have an awareness of how that topic is implemented at the levels below it. This approach is good preparation for later courses in design, where understanding of and insights gained from fundamental underpinnings is essential to making the required design tradeoffs.

**The Student Debugs the Student's Program** We hear complaints from industry all the time about CS graduates not being able to program. Part of the problem is the helpful teaching assistant, who contributes far too much of the intellectual component of the student's program, so the student never has to really master the art. Our approach is to push the student to do the job without the teaching assistant (TA). Part of this comes from the bottom-up approach where memorizing is minimized and the student builds on what he/she already knows. Part of this is

the simulator, which the student uses from day one. The student is taught debugging from the beginning and is required to use the debugging tools of the simulator to get his/her programs to work from the very beginning. The combination of the simulator and the order in which the subject material is taught results in students actually debugging their own programs instead of taking their programs to the TA for help . . . and the common result that the TAs end up writing the programs for the students.

**Preparation for the Future: Cutting Through Protective Layers** In today's real world, professionals who use computers in systems but remain ignorant of what is going on underneath are likely to discover the hard way that the effectiveness of their solutions is impacted adversely by things other than the actual programs they write. This is true for the sophisticated computer programmer as well as the sophisticated engineer.

Serious programmers will write more efficient code if they understand what is going on beyond the statements in their high-level language. Engineers, and not just computer engineers, are having to interact with their computer systems today more and more at the device or pin level. In systems where the computer is being used to sample data from some metering device such as a weather meter or feedback control system, the engineer needs to know more than just how to program in FORTRAN. This is true of mechanical, chemical, and aeronautical engineers today, not just electrical engineers. Consequently, the high-level programming language course, where the compiler protects the student from everything "ugly" underneath, does not serve most engineering students well, and certainly does not prepare them for the future.

**Rippling Effects Through the Curriculum** The material of this text clearly has a rippling effect on what can be taught in subsequent courses. Subsequent programming courses can not only assume the students know the syntax of C but also understand how it relates to the underlying architecture. Consequently, the focus can be on problem solving and more sophisticated data structures. On the hardware side, a similar effect is seen in courses in digital logic design and in computer organization. Students start the logic design course with an appreciation of what the logic circuits they master are good for. In the computer organization course, the starting point is much further along than when students are seeing the term Program Counter for the first time. Feedback from Michigan faculty members in the follow-on courses have noticed substantial improvement in student's comprehension, compared to what they saw before students took EECS 100.

---

## ACKNOWLEDGMENTS

This book has benefited greatly from important contributions of many, many people. At the risk of leaving out some, we would at least like to acknowledge the following.

First, Professor Kevin Compton. Kevin believed in the concept of the book since it was first introduced at a curriculum committee meeting that he chaired at Michigan in 1993. The book grew out of a course (EECS 100) that he and the first author developed together, and co-taught the first three semesters it was offered at Michigan in fall 1995, winter 1996, and fall 1996. Kevin's insights into programming methodology (independent of the syntax of the particular language) provided a sound foundation for the beginning student. The course at Michigan and this book would be a lot less were it not for Kevin's influence.

Several other students and faculty at Michigan were involved in the early years of EECS 100 and the early stages of the book. We are particularly grateful for the help of Professor David Kieras, Brian Hartman, David Armstrong, Matt Postiff, Dan Friendly, Rob Chappell, David Cybulski, Sangwook Kim, Don Winsor, and Ann Ford.

We also benefited enormously from TAs who were committed to helping students learn. The focus was always on how to explain the concept so the student gets it. We acknowledge, in particular, Fadi Aloul, David Armstrong, David Baker, Rob Chappell, David Cybulski, Amolika Gurujee, Brian Hartman, Sangwook Kim, Steve Maciejewski, Paul Racunas, David Telehowski, Francis Tseng, Aaron Wagner, and Paul Watkins.

We were delighted with the response from the publishing world to our manuscript. We ultimately decided on McGraw-Hill in large part because of the editor, Betsy Jones. Once she checked us out, she became a strong believer in what we are trying to accomplish. Throughout the process, her commitment and energy level have been greatly appreciated. We also appreciate what Michelle Flomenhoft has brought to the project. It has been a pleasure to work with her.

Our book has benefited from extensive reviews provided by faculty members at many universities. We gratefully acknowledge reviews provided by Carl D. Crane III, Florida, Nat Davis, Virginia Tech, Renee Elio, University of Alberta, Kelly Flangan, BYU, George Friedman, UIUC, Franco Fummi, Universita di Verona, Dale Grit, Colorado State, Thor Gulsrud, Stavanger College, Brad Hutchings, BYU, Dave Kaeli, Northeastern, Rasool Kenarangui, UT at Arlington, Joel Kraft, Case Western Reserve, Wei-Ming Lin, UT at San Antonio, Roderick Loss, Montgomery College, Ron Meleshko, Grant MacEwan Community College, Andreas Moshovos, Northwestern, Tom Murphy, The Citadel, Murali Narayanan, Kansas State, Carla Purdy, Cincinnati, T. N. Rajashekhara, Camden County College, Nello Scarabottolo, Universita degli Studi di Milano, Robert Schaefer, Daniel Webster College, Tage Stabell-Kuloe, University of Tromsø, Jean-Pierre Steger, Burgdorf School of Engineering, Bill Sverdlik, Eastern Michigan, John Tronto, St. Michael's College, Murali Varansi, University of South Florida, Montanez Wade, Tennessee State, and Carl Wick, US Naval Academy.

In addition to all these people, there were others who contributed in many different and sometimes unique ways. Space dictates that we simply list them and say thank you. Susan Kornfield, Ed DeFranco, Evan Gsell, Rich Belgard, Tom Conte, Dave Nagle, Bruce Shriver, Bill Sayle, Steve Lumetta, Dharma Agarwal, and David Lilia.

Finally, if you will indulge the first author a bit: This book is about developing a strong foundation in the fundamentals with the fervent belief that once that is accomplished, students can go as far as their talent and energy can take them. This objective was instilled in me by the professor who taught me how to be a professor, Professor William K. Linvill. It has been more than 35 years since I was in his classroom, but I still treasure the example he set.

---

## A FINAL WORD

We hope you will enjoy the approach taken in this book. Nonetheless, we are mindful that the current version will always be a work in progress, and both of us welcome your comments on any aspect of it. You can reach us by email at [patt@ece.utexas.edu](mailto:patt@ece.utexas.edu) and [sjp@crhc.uiuc.edu](mailto:sjp@crhc.uiuc.edu). We hope you will.

Yale N. Patt  
Sanjay J. Patel