Computer Science

Volume 2

J. Stanley Warford

Computer Science

Volume 2

J. Stanley Warford
Pepperdine University

D. C. Heath and Company
Lexington, Massachusetts / Toronto

Address editorial correspondence to:

D. C. Heath
125 Spring Street
Lexington, MA 02173

Acquisitions Editor: Carter Shanklin
Developmental Editor: Katherine Pinard
Production Editor: Jennifer Brett
Designer: Judith Miller
Production Coordinator: Lisa Merrill
Photo Researcher: Ann M. Barnard

Cover Image: Digital Art/Los Angeles

Photo Credits: p. 666—The Bettmann Archive; p. 703—Iowa State University, Public Information Photo Department; p. 757—Ian Yeomans/Susan Griggs Agency Ltd.; p. 825—The Computer Museum; p. 908—The Thinking Machine, Inc.; p. 904—Camermann Inc.

Published simultaneously in Canada.

Printed in the United States of America.

International Standard Book Number: 0-669-24977-7

Library of Congress Catalog Number: 90-84807

10 9 8 7 6 5 4 3 2 1

Preface

This book is the outgrowth of an introductory computer science course taught at Pepperdine University, where the book has been class-tested for five years. The course, as well as the book, is primarily for majors in computer science who intend a more in-depth study later, and secondarily for nonmajors who desire a strong background in computers so that they can deal with them effectively in their chosen fields. The book is designed for a two- or three-semester course.

The Physics Model

The approach taken in this book is unique. The idea is best illustrated by looking at the typical curriculum in physics, a much older discipline. Like computer science, physics is a broad area. Over the years a traditional introductory physics course, which is uniform in content from school to school, has evolved. Physics educators realize that the first course should teach some problem-solving techniques and give the student some laboratory experience. In addition, they realize the importance of introducing the student to all the main areas of study, including mechanics, thermodynamics, electricity, magnetism, and modern physics. In-depth mastery of the subfields of physics is postponed for later courses in which the instructor can assume that the student has already been exposed to the main concepts.

The traditional design of the introductory computer science sequence lacks the breadth of the typical physics sequence. Computer science includes the study of hardware, language theory, algorithms, data structures, architecture, and operating systems. Although the recent trend is to incorporate more topics from data structures into the introductory course, most introductory sequences in computer science neglect hardware, language theory, architecture, and operating systems.

The physics curriculum generally recognizes that the concepts of motion, force, and energy as developed in classical Newtonian mechanics are the foundation on which all the other topics can be presented. However, physics books rarely present just mechanics—they develop the concepts in mechanics first and then move on to the other topics.

Similarly, this introductory computer science text begins with algorithm design in a high-order language, but is not confined to it. Although Pascal programming accounts for much of the content, a substantial portion of the text is devoted to computer organization down to the logic gate level. Students should get an overall picture of the discipline of computer science in their first year of study. Indeed, if they do not get a unified picture of the discipline then, when will they get it later?

Summary of Contents

Computers operate at several levels of abstraction; programming in Pascal at a high level of abstraction is only part of the story. This book presents a unified concept of computer systems based on the level structure of Figure P.1.

The book is divided into six parts corresponding to six of the seven levels of Figure P.1:

Level 7	Applications
Level 6	High-order languages
Level 3	Machine
Level 5	Assembly
Level 4	Operating system
Level 1	Logic gate

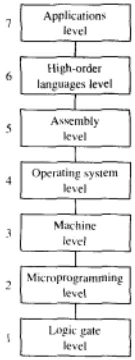


Figure P.1

The level structure of a typical computer system.

Volume 1 of the two-volume edition covers Levels 7 and 6, and Volume 2 covers Levels 3, 5, 4, and 1. Microprogramming, Level 2, is beyond the scope of this book.

The text generally presents the levels top-down, from the highest to the lowest. Level 3, the machine level, is discussed before Level 5, the assembly level, for pedagogical reasons. In this one instance, it is more natural to revert temporarily to a bottom-up approach so that the building blocks of the lower level will be in hand for construction of the higher level.

Level 7 is a single chapter on applications programs. It presents the idea of levels of abstraction and establishes the framework for the remainder of the book. A few concepts of relational databases are presented as an example of a typical computer application. It is assumed that students have experience with text editors or word processors.

Level 6 consists of 10 chapters on algorithm design and data structures in Pascal (Chapters 2–11). The treatment of Pascal is fairly complete and includes separate chapters on recursion and pointers. Some topics, such as passing procedures as parameters, are not included.

Students learn best by studying and imitating complete programs, so the Pascal chapters present the material in a case-study format. Generalizations then follow the examples. As much as possible, Pascal language features are discussed in the context of the specific programs.

Two design methodologies are integrated into the Pascal chapters—stepwise refinement and top-down design. Although these two techniques are closely related (some would say they are identical), they are treated separately. Stepwise refinement is a tool for developing a single main program or a single module. Top-down design is a tool for partitioning a program into modules. (In this book, a module is defined as a main program or a procedure or a function.) The design methodologies and associated terminology are simplified to avoid intimidating the student.

Two other topics integrated throughout the Pascal chapters are assertions and statement execution counts. Assertions are introduced with nested `if` statements as a tool for reasoning about the behavior of executable code. Formal proofs of correctness are left for a later course, but the book lays the groundwork by giving the students the ability to formulate a strong assertion. Statement execution counts are introduced with loops. Given the execution time of a program with a small amount of data, the student is shown how to estimate the execution time of the program with a large amount of data. The numerical exercises included in these chapters give the student a feel for the usefulness of time-complexity results.

Level 3 is the machine level. Its two chapters describe Pep/5, a hypothetical computer designed to illustrate computer concepts. The Pep/5 computer is a classical von Neumann machine. The CPU contains an accumulator, an index register, a base register, a program counter, a stack pointer, and an instruction register. It has four addressing modes—immediate, direct, indexed, and stack relative. The Pep/5 operating system, in simulated read-only memory (ROM), can load and execute programs in hexadecimal format from students' text files. Students run short programs on the Pep/5 simulator and learn that executing a store instruction to ROM does not change the memory value.

Students learn the fundamentals of information representation and computer organization at the bit level. Because a central theme of this book is the relationship of the levels to one another, the Pep/5 chapters show the relationship between the ASCII representation (Level 3) and Pascal variables of type `char` (Level 6). They also show the relationship between two's complement representation (Level 3) and Pascal variables of type `integer` (Level 6).

Level 5 is the assembly level. The text presents the concept of the assembler as a translator between two levels—assembly and machine. It introduces Level 5 symbols and the symbol table.

The unified approach really pays off here. Chapters 14 and 15 present the compiler as a translator from a high-order language to assembly language. In previous chapters students learned a specific Level 6 language, Pascal, and a specific von Neumann machine, Pep/5. These chapters continue the theme of relationships between the levels by showing the correspondence between (a) assignment statements at Level 6 and load/store instructions at Level 5, (b) loops and `if` statements at Level 6 and branching instructions at Level 5, (c) arrays at Level 6 and indexed addressing at Level 5, (d) procedure calls at Level 6 and the run-time stack at Level 5, (e) function and procedure parameters at Level 6 and stack-relative addressing at Level 5, and (f) `case` statements at Level 6 and jump tables at Level 5.

The beauty of the unified approach is that the text can implement many of the examples from the Pascal chapters at this lower level. For example, the run-time stack illustrated in the recursive examples of an earlier chapter corresponds directly to the hardware stack in Pep/5 main memory. Students gain an understanding of the compilation process by translating manually between the two levels.

This approach provides a natural setting for the discussion of central issues in computer science. For example, the book presents structured programming at Level 6 versus the possibility of unstructured programming at Level 5. It discusses the goto controversy and the structured programming/efficiency tradeoff, giving concrete examples from languages at the two levels. As in the Pascal chapters, the style of presentation involves drawing general conclusions from specific examples.

Chapter 16, Language Translation Principles, introduces students to computer science theory. Now that students know intuitively how to translate from a high-level language to assembly language, we pose the fundamental question underlying all of computing, What can be automated? The theory naturally fits in here because students now know what a compiler (an automated translator) must do. They learn about parsing and finite state machines—deterministic and nondeterministic—in the context of recognizing Pascal and Pep/5 assembly language tokens. This chapter includes an automatic translator between two small languages, which illustrates lexical analysis, parsing, and code generation. The lexical analyzer is an implementation of a finite state machine. What could be a more natural setting for the theory?

Level 4 consists of two chapters on operating systems. Chapter 17 is a description of process management. Two sections, one on loaders and another on interrupt handlers, illustrate the concepts with the Pep/5 operating system. Four instructions have unimplemented opcodes that generate software interrupts. The operating system stores the process control block of the user's running process on the system stack while the interrupt service routine interprets the instruction. The classic state transition diagram for running and waiting processes in an operating system is thus reinforced with a specific implementation of a suspended process. The chapter concludes with a description of concurrent processes and deadlocks. Chapter 18 describes storage management, both main memory and disk memory.

Level 1 uses two chapters to present combinational networks and sequential networks. Chapter 19 emphasizes the importance of the mathematical foundation of computer science by starting with the axioms of boolean algebra. It shows the relation between boolean algebra and logic gates, then describes some common SSI and MSI logic devices. Chapter 20 again illustrates the fundamental concept of a finite state machine through the state transition diagrams of sequential circuits. It concludes with the construction of the data section of the Pep/5 computer. The same machine model is thus used from the Pascal level to the logic gate level, providing a complete, unifying picture of the entire system.

Unifying Themes

In physics, fundamental concepts of motion, force, and energy are developed in one area of study and carried over into other areas, thus providing a unifying framework. Unifying themes of this book include abstraction, languages, and finite state machines.

The fundamental space/time tradeoff is another recurring theme. Execution-time analysis begins with the first `while` loop in Chapter 5 and continues throughout the text. The tradeoff occurs in software when the availability of extra memory may permit a faster algorithm, and in hardware when a two-level network may require more gates to implement a binary function than a multilevel one.

The Denning Report

This text reflects the recent recommendations of the Denning Report, "Computing as a Discipline."[1] That report identifies the following nine subareas of computer science on which the introductory sequence should be based:

1. Algorithms and data structures
2. Programming languages
3. Architecture
4. Numeric and symbolic computation
5. Operating systems
6. Software methodology and engineering
7. Databases and information retrieval
8. Artificial intelligence and robotics
9. Human-computer communication

This book emphasizes five areas (1, 2, 3, 5, 6), touches on two others (4, 7), and admits that two areas are beyond its scope (8, 9). Even with the omissions, I believe that this book achieves the goal enunciated in the Denning Report that the "introductory sequence should bring out the underlying unity of the field and should flow from topic to topic in a pedagogically natural way."

Additional Features

Exercises and Programming Assignments In a course based on this book, homework assignments consist of exercises, which are handwritten, and problems, which are programming assignments for computer execution. There are an average of nearly 40 exercises and programming assignments in each chapter. These two types of assignments reflect the difference between analysis and design. Neglecting analysis is like trying to teach children to write (design) before they can read (analyze). At the introductory level, analysis is just as important as design. If students do not get explicit practice in reasoning about control structures, it is more difficult for them to locate logical errors in the loops they write. After all, debugging is analysis, not design.

Software Tools One goal of this text is to give the student useful software tools. Accordingly, I have tried to present the "best" known algorithms. The sequential

1. Peter J. Denning et al., "Computing as a Discipline," *Computer* 22 (February 1989): 63–70. © 1989 by IEEE.

search algorithm installs the search key after the last item in the list, so the loop has only one test. The binary search algorithm is free of the subtle errors described by Pattis.[2] The random number generator is based on that recommended by Park and Miller.[3] The sequential file update algorithm is the balanced-line algorithm as described by Levy.[4] The section on sorting uses the taxonomy of sort algorithms from Merritt.[5] The version of quick sort is one that executes in time *n* log *n* even in the case when the original array is approximately in order.

Use in the Curriculum

With such broad coverage, some instructors may wish to omit some of the material when designing their introductory sequence. To provide maximum flexibility for curriculum design, the model is again the traditional introductory physics textbook, which usually contains many topics and applications that may be omitted depending on the interest of the instructor.

Students are introduced to the concept of an abstract data type as applied to the stack in Chapter 9. Although other data structures topics are normally included in the introductory sequence, it is possible to omit half of Chapter 10 (Dynamic Storage Allocation) and all of Chapter 11 (Data Structures), trading off this depth for the breadth that comes by studying the lower levels. Later material in the book is not dependent on these omitted topics, which can be left to a more advanced data structures course.

In the remainder of the book, Chapters 12–14 must be covered sequentially. Chapters 15 (Compiling to the Assembly Level) and 16 (Language Translation Principles) can be covered in either order. I often skip ahead to Chapter 16 to initiate a large software project, writing an assembler for a subset of Pep/5 assembly language, so students will have sufficient time to complete it during the semester. Chapter 20 (Sequential Networks) is obviously dependent on Chapter 19 (Combinational Networks), but neither depends on Chapter 18 (Storage Management), which may be omitted. Figure P.2, a chapter dependency graph, summarizes the possible chapter omissions.

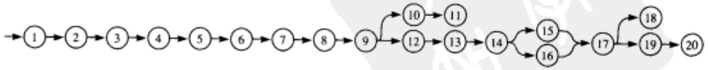


Figure P.2

A chapter dependency graph.

2. Richard E. Pattis, "Textbook Errors in Binary Searching," *ACM SIGCSE Bulletin* 20 (February 1988): 190–94.

3. Stephen K. Park and Keith W. Miller, "Random Number Generators: Good Ones Are Hard to Find," *Communications of the ACM* 31 (October 1988): 1192–1201.

4. Michael R. Levy, "Modularity and the Sequential File Update Problem," *Communications of the ACM* 25 (June 1982): 362–67.

5. Susan M. Merritt, "An Inverted Taxonomy of Sorting Algorithms," *Communications of the ACM* 28 (January 1985): 96–98.

In addition to possible chapter omissions, some sections within chapters may be omitted; examples include sections 6.4 (Scope of Identifiers), 6.5 (Random Numbers), 13.5 (Some Typical Architectures), 15.4 (Data Types at Level 5), 17.3 (Concurrent Processes), and 17.4 (Deadlocks). Selected topics within sections may also be omitted; examples include specific algorithms such as matrix multiplication and recursive merge sort.

Support Materials

Pep/5 Assembler/Simulator Disk Machine-readable source code for the Pep/5 system is available to adopters from the publisher or the author (Bitnet address: warford@pepvax). The package, complete with assembler and trace facilities, is written in Pascal and runs on MS-DOS, MacOS, or UNIX systems. (Please specify which system you use.) The software may be copied freely without express permission.

Instructor's Guide An Instructor's Guide containing solutions to exercises and overhead transparency masters of the figures and program listings is available to adopters. Teaching hints and suggestions on how to structure the course, based on classroom experience teaching computer science from a unified perspective, are also included.

Test Disk The exercises and problems in the book have been combined with additional test items and are available on disk in ASCII format. These can be imported into your word processor to facilitate the printing of exams.

Test Item File A printed Test Item File contains the same questions that are available on the Test Disk.

Program Disk All the programs from the book and the data files necessary to test the programming problems are available on disk.

Acknowledgments

The nroff document-preparation utility on Pepperdine's VAX UNIX system was valuable for preparing early versions of the manuscript. I was able to design most of the figures with MacDraw on a Macintosh. I commend the folks at Apple Computers for such a great desktop graphics system. I believe the quality of the book is significantly better than it would have been without these tools.

Pep/1 had 16 instructions, one accumulator, and one addressing mode. Pep/2 added indexed addressing. John Vannoy wrote both simulators in ALGOL W. Pep/3 had 32 instructions and was written in Pascal as a student software project by Steve Dimse, Russ Hughes, Kazuo Ishikawa, Nancy Brunet, and Yvonne Smith. In an early review Harold Stone suggested many improvements to the Pep/3 architecture that were incorporated into Pep/4 and carried into Pep/5. Pep/4 had special

stack instructions, simulated ROM, and software interrupts. Pep/5 is a more orthogonal design, allowing any instruction to use any addressing mode. John Rooker wrote the Pep/4 system and an early version of Pep/5. Gerry St. Romain implemented a MacOS version and an MS-DOS version.

More than any other book, Tanenbaum's *Structured Computer Organization* has influenced this text.[6] This text extends the level structure of Tanenbaum's book by adding the high-order programming level and the applications level at the top.

The following reviewers of the manuscript in its various stages of development shaped the final product significantly: Wayne P. Bailey, Northeast Missouri State University; Fadi Deek, New Jersey Institute of Technology; William Decker, University of Iowa; Gerald S. Eisman, San Francisco State University; Victoria Evans, University of Nevada at Reno; David Garnick, Bowdoin College; Ephraim P. Glinert, Rensselaer Polytechnic Institute; Dave Hanscom, University of Utah; Michael Hennessy, University of Oregon; Michael Johnson, Oregon State University; Robert Martin, Middlebury College; Richard H. Mercer, Pennsylvania State University—Berks Campus; Randy Molmen, Baldwin-Wallace College; Peter Ng, New Jersey Institute of Technology; Bernard Nudel, University of Michigan; Carolyn Oberlink, Western Michigan University; Wolfgang Pelz, University of Akron; James F. Peters III, Kansas State University; James C. Pleasant, East Tennessee State University; Eleanor Quinlan, Ohio State University; Glenn A. Richard, State University of New York at Stony Brook; David Rosser, Ramsey, N.J.; Scott Smith, State University of New York at Plattsburgh; Harold S. Stone, The Interfactor, Inc.; J. Peter Weston, Daniel Webster College; and Norman E. Wright, Brigham Young University.

Don Thompson, Don Hancock, Carol Adjemian, Chelle Boehning, and Janet Davis taught sections of the course on which this book is based. They contributed suggestions and exercises. Thanks especially to Gerry St. Romain, who provided, based on his experience, literally hundreds of ideas that were incorporated into the text. Those who typed early drafts of the manuscript include Julie Teichrow, Russ Hughes, and Janet Davis. Joe Piasentin provided artistic consultation.

It has been a pleasure to work with the publisher of this book, D. C. Heath. Karin Ellison provided constant enthusiasm and encouragement from the beginning. Carter Shanklin guided the project on to completion. Thanks to Kitty Pinard for suggesting the sidebars and to Scott Smith of SUNY Plattsburgh for writing them. Judy Miller did a magnificent job designing the book, and Jennifer Brett did a superb job producing it, no small feat for a technical book of this length.

I am fortunate to be at an institution that is committed to excellence in undergraduate education. Pepperdine University in the person of Ken Perrin provided the creative environment and the professional support in which the idea behind this project was able to evolve. My wife, Ann, provided endless personal support. To her I owe an apology for the time this project has taken, and my greatest thanks.

Stan Warford

6. Andrew S. Tanenbaum, *Structured Computer Organization* (Englewood Cliffs, N.J.: Prentice-Hall, 1984).

Brief Contents

Contents

Chapter

12 Information Representation

One of the most significant inventions of mankind is the printed word. The words on this page represent information stored on paper, which is conveyed to you as you read. Like the printed page, computers have memories for storing information. The central processing unit (CPU) has the ability to retrieve information from its memory much like you take information from words on a page.

Reading and writing, words and pages

Some computer terminology is based on this analogy. The CPU *reads* information from memory and *writes* information into memory. The information itself is divided into *words*. In some computer systems large sets of words, usually anywhere from a few hundred to a few thousand, are grouped into *pages*.

Information representation at Level 3

In Pascal, at Level 6, information takes the form of values that you store in a variable in main memory or in a file on disk. This chapter shows how the computer stores that information at Level 3. Information representation at the machine level differs significantly from that at the high-order languages level. At Level 3, information representation is less human-oriented. Later chapters will discuss information representation at the intermediate levels, Levels 5 and 4, and show how they relate to Levels 6 and 3.

12.1 Unsigned Binary Representation

The Mark I computer

Early computers were electromechanical. That is, all their calculations were performed with moving switches called *relays*. The Mark I computer, built in 1944 by [illegible] Howard H. Aiken of Harvard University, was such a machine. Aiken had procured financial backing for his project from Thomas J. Watson, president of International Business Machines (IBM). The relays in the Mark I computer could compute much faster than the mechanical gears that were used in adding machines at that time.

The ENIAC computer

Even before the completion of Mark I, John V. Atanasoff, working at Iowa State University, had finished the construction of an electronic computer to solve systems of linear equations. In 1941, John W. Mauchly visited Atanasoff's laboratory and in 1946, in collaboration with J. Presper Eckert at the University of Pennsylvania, built the famous Electronic Numerical Integrator and Calculator

(ENIAC). ENIAC's 19,000 vacuum tubes could perform 5000 additions per second compared to 10 additions per second with the relays of the Mark I. Like the ENIAC, present-day computers are electronic, although their calculations are performed with integrated circuits (ICs) instead of with vacuum tubes. Each IC contains thousands of transistors similar to the transistors in radios.

Binary Storage

Electronic computer memories cannot store numbers and letters directly. They can only store electrical signals. When the CPU reads information from memory, it is detecting a signal whose voltage is about equal to that produced by three flashlight batteries.

Computer memories are designed with a most remarkable property. Each storage location contains either a high-voltage signal or a low-voltage signal, never anything in between. The storage location is like being pregnant. Either you are or you are not. There is no halfway.

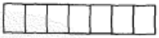
The word *digital* means that the signal stored in memory can only have a fixed number of values. *Binary* means that only two values are possible. Practically all computers on the market today are binary. Hence, each storage location contains either a high voltage or a low voltage. The state of each location is also described as being either on or off, or, alternatively, as containing either a 1 or a 0.

Each individual storage unit is called a *binary digit* or *bit*. A bit can be only 1 or 0, never anything else, such as 2, 3, A, or Z. This is a fundamental concept. Every piece of information stored in the memory of a computer, whether it is the amount you owe on your credit card or your street address, is stored in binary as 1's and 0's.

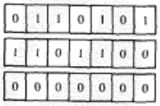
In practice, the bits in a computer memory are grouped together into *cells*. A seven-bit computer, for example, would store its information in groups of seven bits, as Figure 12.1 shows. You can think of a cell as a group of boxes, each box containing a 1 or a 0, nothing else. The first two lines in Figure 12.1(c) are impossible because the values in some boxes differ from 0 or 1. The last is impossible because each box must contain a value. A bit of storage cannot contain nothing.

Different computers have different numbers of bits in each cell, although most computers these days have eight bits per cell. This chapter will show examples with several different cell sizes to illustrate the general principle.

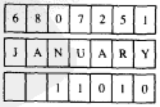
Information such as numbers and letters must be represented in binary form to be stored in memory. The representation scheme used to store information is called a *code*. This section examines a code for storing unsigned integers. The remainder of this chapter describes codes for storing other kinds of data. The next chapter examines codes for storing program commands in memory.



(a) A seven-bit cell.



(b) Some possible values in a seven-bit cell.



(c) Some impossible values in a seven-bit cell.

Figure 12.1

A seven-bit memory cell in main memory.

Integers

Numbers must be represented in binary form to be stored in a computer's memory. The particular code depends on whether the number has a fractional part or is an