third edition

introduction to computing systems

from bits & gates to C/C++ & beyond

```
101001 000110 0011
010111 101010 0011
```

```
int Add(int x,
{
  return x + y
}
```

```
LDR R0, R6,
LDR R1, R6,
ADD R2, R0,
STR R2, R6,
RET
```

Mc
Graw
Hill

yale n. patt
sanjay j. patel

introduction to computing systems

third edition

introduction to computing systems

from bits & gates to C/C++ & beyond

Yale N. Patt
The University of Texas at Austin

Sanjay J. Patel
University of Illinois at Urbana-Champaign

Mc
Graw
Hill

INTRODUCTION TO COMPUTING SYSTEMS: FROM BITS & GATES TO C/C++ & BEYOND, THIRD EDITION

Published by McGraw-Hill Education, 2 Penn Plaza, New York, NY 10121. Copyright ©2020 by McGraw-Hill Education. All rights reserved. Printed in the United States of America. Previous editions ©2004, and 2001. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written consent of McGraw-Hill Education, including, but not limited to, in any network or other electronic storage or transmission, or broadcast for distance learning.

Some ancillaries, including electronic and print components, may not be available to customers outside the United States.

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 LCR 24 23 22 21 20 19

ISBN 978-1-260-15053-7 (bound edition)
MHID 1-260-15053-4 (bound edition)
ISBN 978-1-260-42474-4 (loose-leaf edition)
MHID 1-260-42474-X (loose-leaf edition)

Executive Portfolio Manager: *Suzy Bainbridge*
Product Developer: *Heather Ervolino*
Senior Marketing Manager: *Shannon O'Donnell*
Content Project Manager: *Jeni McAtee*
Buyer: *Sandy Ludovissy*
Designer: *David Hash*
Content Licensing Specialist: *Melissa Homer*
Cover Image: *Front Cover Computing art ©Yale N. Patt; Abstract image in green, orange and yellow ©Design Pics/Tim Antoniuk; Abstract image in orange, green and yellow ©Design Pics/Tim Antoniuk; Computer laptops on a blue abstract background ©loops7/ Getty Images; Illustration of streams of binary code ©Shutterstock/Tavarius*
Compositor: *Aptara®, Inc.*

All credits appearing on page or at the end of the book are considered to be an extension of the copyright page.

Library of Congress Cataloging-in-Publication Data

Names: Patt, Yale N., author. | Patel, Sanjay J., author.
Title: Introduction to computing systems : from bits & gates to c/c++ & beyond / Yale N. Patt, The University of Texas at Austin, Sanjay J. Patel, University of Illinois at Urbana-Champaign.
Description: Third edition. | New York, NY : McGraw-Hill, 2020.
Identifiers: LCCN 2018050089 | ISBN 9781260150537 (acid-free paper) | ISBN 1260150534 (acid-free paper)
Subjects: LCSH: Computer science. | C (Computer program language)
Classification: LCC QA76 .P367 2019 | DDC 004–dc23 LC record available at https://lccn.loc.gov/2018050089

mheducation.com/highered

To the memory of my parents,
Abraham Walter Patt A"H and Sarah Clara Patt A"H,
who taught me to value “learning”
even before they taught me to ride a bicycle.

To my loving family,
Amita, Kavi, and Aman.

Contents

Preface

Finally, the third edition! We must first thank all those who have been pushing us to produce a third edition. Since the publication of the second edition was so long ago, clearly the material must be out of date. Wrong! Fundamentals do not change very often, and our intent continues to be to provide a book that explains the fundamentals clearly in the belief that if the fundamentals are mastered, there is no limit to how high one can soar if one's talent and focus are equal to the task.

We must also apologize that it took so long to produce this revision. Please know that the delay in no way reflects any lack of enthusiasm on our part. We are both as passionate about the foundation of computing today as we were 25 years ago when we overhauled the first course in computing that eventually became the first edition of this book. Indeed, we have both continued to teach the course regularly. And, as expected, each time we teach, we develop new insights as to what to teach, new examples to explain things, and new ways to look at things. The result of all this, hopefully, is that we have captured this in the third edition.

It is a pleasure to finally be writing this preface. We have received an enormous number of comments from students who have studied the material in the book and from instructors who have taught from it. It is gratifying to know that a lot of people agree with our approach, and that this agreement is based on real firsthand experience learning from it (in the case of students) and watching students learn from it (in the case of instructors). The excitement displayed in their correspondence continues to motivate us.

Why the Book Happened

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

EECS 100 happened at Michigan 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.

Our result was the bottom-up approach taken in this book, where we continually build on what the student already knows, only memorizing when absolutely necessary. We did not endorse then and we do not endorse now the 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.

Major Changes in the Third Edition

The LC-3

A hallmark of our book continues to be the LC-3 ISA, which is small enough to be described in a few pages and hopefully mastered in a very short time, yet rich enough to convey the essence of what an ISA provides. It is the LC "3" because it took us three tries to get it right. Four tries, actually, but the two changes in the LC-3 ISA since the second edition (i.e., changes to the LEA instruction and to the TRAP instruction) are so minor that we decided not to call the slightly modified ISA the LC-4.

The LEA instruction no longer sets condition codes. It used to set condition codes on the mistaken belief that since LEA stands for Load Effective Address, it should set condition codes like LD, LDI, and LDR do. We recognize now that this reason was silly. LD, LDI, and LDR load a register from memory, and so the condition codes provide useful information – whether the value loaded is negative, zero, or positive. LEA loads an address into a register, and for that, the condition codes do not really provide any value. Legacy code written before this change should still run correctly.

The TRAP instruction no longer stores the linkage back to the calling program in R7. Instead, the PC and PSR are pushed onto the system stack and popped by the RTI instruction (renamed Return from Trap or Interrupt) as the last instruction in a trap routine. Trap routines now execute in privileged memory (x0000 to x2FFF). This change allows trap routines to be re-entrant. It does not affect old code provided the starting address of the trap service routines, obtained from the Trap Vector Table, is in privileged memory and the terminating instruction of each trap service routine is changed from RET to RTI.

As before, Appendix A specifies the LC-3 completely.

The Addition of C++

We’ve had an ongoing debate about how to extend our approach and textbook to C++. One of the concerns about C++ is that many of its language features are too far abstracted from the underlying layers to make for an easy fit to our approach. Another concern is that C++ is such a vast language that any adequate coverage would require an additional thousand pages. We also didn’t want to drop C completely, as it serves as a de facto development language for systems and hardware-oriented projects.

We adopted an approach where we cover the common core of C and C++ from Chapters 11 through 19. This common core is similar to what was covered in the second edition, with some minor updates. Chapter 20 serves as a transition, which we aspired to make very smooth, to the core concepts of C++. With this approach, we get to explore the evolution between C and C++, which serves as a key learning opportunity on what changes were essential to boost programmer productivity.

In particular, we focus on classes in C++ as an evolution from structures in C. We discuss classes as a compiler construct, how method calls are made, and the notion of constructors. We touch upon inheritance, too, but leave the details for subsequent treatment in follow-on courses.

An important element of C++ is the introduction of container classes in the Standard Template Library, which is a heavily utilized part of the C++ language. This provides an opportunity to dive deep into the vector class, which serves as a continuation of a running example in the second half around the support for variable-sized arrays in high-level languages, or in particular, C’s lack of support for them.

Other Important Updates

Although no chapter in the book has remained untouched, some chapters have been changed more than others. In Chapter 2, we expanded the coverage of the floating point data type and the conversion of fractions between decimal and binary numbers in response to several instructors who wanted them. We moved DeMorgan’s Laws from Chapter 3 to Chapter 2 because the concept is really about AND and OR functions and not about digital logic implementation. In Chapter 3, we completely overhauled the description of state, latches, flip-flops, finite state machines, and our example of a danger sign. We felt the explanations in the second edition were not as clear as they needed to be, and the concepts are too important to not get right. We revised Chapter 4 to better introduce the LC-3, including a different set of instructions, leading to our first complete example of a computer program.

Our organization of Chapters 8, 9, and 10 was completely overhauled in order to present essentially the same material in a more understandable way. Although most of our treatment of data structures waits until we have introduced C in the second half of the book, we felt it was important to introduce stacks, queues, and character strings as soon as the students have moved out of programming in machine language so they can write programs dealing with these data structures and see how these structures are actually organized in memory. We moved our discussion of subroutines up to Chapter 8 because of their importance in constructing richer programs.

We also introduced recursion in Chapter 8, although its main treatment is still left for the second half of the book. Both the expressive power of recursion and its misuse are so common in undergraduate curricula that we felt dealing with it twice, first while they are engrossed in the bowels of assembly language and again after moving up to the richness of C, was worthwhile.

Chapter 9 now covers all aspects of I/O in one place, including polling and interrupt-driven I/O. Although the concept of privilege is present in the second edition, we have put greater emphasis on it in the third edition. Our coverage of system calls (the trap routines invoked by the TRAP instruction) appears in Chapter 9. All of the above reduce Chapter 10 to simply a comprehensive example that pulls together a lot of the first half of the book: the simulation of a calculator. Doing so requires 12 subroutines that are laid out in complete detail. Two concepts that are needed to make this happen are stack arithmetic and ASCII/binary conversion, so they are included in Chapter 10.

We reworked all the examples in Chapters 11 through 19 to use the latest ANSI Standard C or C18. We also added more coding examples to further emphasize points and to provide clarity on complex topics such as pointers, arrays, recursion, and pointers to pointers in C. In Chapter 16, we added additional sections on variable-sized arrays in C, and on multidimensional arrays.

Chapter Organization

The book breaks down into two major segments, (a) the underlying structure of a computer, as manifested in the LC-3; and (b) programming in a high-level language, in our case C and C++.

The LC-3

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-3. Chapter 3 takes the student from an MOS transistor, step by step, to a “real” memory and a finite state machine.

Our real memory consists of four words of three bits each, rather than 16 gigabytes, which is common in most laptops today. Its description fits on a single page (Figure 3.20), making it easy for a student to grasp. By the time students get there, they have been exposed to all the elements needed to construct the memory. The finite state machine is needed to understand how a computer processes instructions, starting in Chapter 4. Chapter 4 introduces the von Neumann execution model and enough LC-3 instructions to allow an LC-3 program to be written. Chapter 5 introduces most of the rest of the LC-3 ISA.

The LC-3 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).

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-3 that runs on Windows, Linux, and Mac0S platforms. It can be downloaded from the web at no charge.

Students use the simulator to test and debug programs written in LC-3 machine language and in LC-3 assembly language. The simulator allows online debugging (deposit, examine, single-step, set breakpoint, and so on). The simulator can be used for simple LC-3 machine language and assembly language programming assignments, which are essential for students to master the concepts presented throughout the first ten 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-3 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 writing programs in Chapters 8, 9, and 10, and for providing many of the explanations in 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 to have to struggle with 0001001010000011.

Chapter 8 introduces three important data structures: the stack, the queue, and the character string, and shows how they are stored in memory. The subroutine call/return mechanism of the LC-3 is presented because of its usefulness both in manipulating these data structures and in writing programs. We also introduce recursion, a powerful construct that we revisit much more thoroughly in the second half of the book (in Chapter 17), after the student has acquired a much stronger capability in high-level language programming. We introduce recursion here to show by means of a few examples the execution-time tradeoffs incurred with recursion as a first step in understanding when its use makes sense and when it doesn't.

Chapter 9 deals with input/output and some basic interaction between the processor and the operating system. We introduce the notions of priority and privilege, which are central to a systems environment. Our treatment of I/O is all physical, using keyboard data and status registers for input and display data and status registers for output. We describe both interrupt-driven I/O and I/O under program control. Both are supported by our LC-3 simulator so the student can write interrupt drivers. Finally, we show the actual LC-3 code of the trap service routines that the student has invoked with the TRAP instruction starting in Chapter 4. To handle interrupt-driven I/O and trap service routines, we complete the description of the LC-3 ISA with details of the operation of the Return from Trap or Interrupt (RTI) and TRAP instructions.

The first half of the book concludes with Chapter 10, a comprehensive example of a simple calculator that pulls together a lot of what the students have learned in Chapters 1 through 9.

Programming in C and C++

By the time the student gets to the second part of the textbook, he/she has an understanding of the layers below. In our coverage of programming in C and C++, we leverage this foundation by showing the resulting LC-3 code generated by a compiler with each new concept in C/C++.

We start with the C language because it provides the common, essential core with C++. 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 in C++.

Each time a new high-level construct is introduced, the student is shown the LC-3 code that a compiler would produce. We cover the basic constructs of C (variables, operators, control, and functions), pointers, arrays, recursion, I/O, complex data structures, and dynamic allocation. With C++, we cover some basic improvements over C, classes, and containers.

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 is used in practice. LC-3 code is used to demonstrate how each C construct affects the machine at the lower levels.

Chapter 14 introduces functions in C. Students are not merely exposed to the syntax of functions. Rather they learn how functions are actually executed, with argument-passing using a run-time stack. A number of examples are provided.

In Chapter 15, students are exposed to techniques for testing their code, along with debugging high-level source code. The ideas of white-box and black-box testing are discussed.

Chapter 16 teaches pointers and arrays, relying heavily on the student's understanding of how memory is organized. We discuss C's notions of fixed size and variable-length arrays, along with multidimensional array allocation.

Chapter 17 teaches recursion, using the student's newly gained knowledge of functions, stack frames, and the run-time stack. 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 discusses structures in C, dynamic memory allocation, and linked lists.

Chapter 20 provides a jump-start on C++ programming by discussing its roots in C and introducing the idea of classes as a natural evolution from structures. We also cover the idea of containers in the standard template library, to enable students to quickly jump into productive programming with C++.

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 16) 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 17), he/she has already encountered all the underpinnings. Students understand how stacks work at the machine level (Chapter 8), and they understand the call/return mechanism from their LC-3 machine language programming experience, and the need for linkages between a called program and the return to the caller (Chapter 8). From this foundation, it is not a large step to explain functions by introducing run-time stack frames (Chapter 14), 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.

The Simulator/Debugger

The importance of the Simulator/Debugger for testing the programs a student writes cannot be overemphasized. We believe strongly that there is no substitute for hands-on practice testing one's knowledge. It is incredibly fulfilling to a student's education to write a program that does not work, testing it to find out why it does not work, fixing the bugs himself/herself, and then seeing the program run correctly. To that end, the Simulator/Debugger has been completely rewritten. It runs on Windows, Linux, and MacOS while presenting the same user interface (GUI) regardless of which platform the student is using. We have improved our incorporation of interrupt-driven I/O into the Simulator's functionality so students can easily write interrupt drivers and invoke them by interrupting a lower priority executing program. ...in their first course in computing!

Alternate Uses of the Book

We wrote the book as a textbook for a freshman introduction to computing. We strongly believe that our motivated bottom-up approach is the best way for students to learn the fundamentals of computing. We have seen lots of evidence showing that in general, students who understand the fundamentals of how the computer works are better able to grasp the stuff that they encounter later, including the high-level programming languages that they must work in, and that they can learn the rules of these programming languages with far less memorizing because everything makes sense. For us, the best use of the book is a one-semester freshman course for particularly motivated students, or a two-semester sequence where the pace is tempered.

Having said that, we recognize that others see the curriculum differently. Thus, we hasten to add that the book can certainly be used effectively in other ways. In fact, each of the following has been tried, and all have been used successfully:

Two Quarters, Freshman Course

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 11–20. The pace is brisk, but the entire book can be covered easily in two academic quarters.

One-Semester, 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. The rationale is that after exposure to high-level language programming in the first course, the second course should treat at an introductory level digital logic, basic computer organization, and assembly language programming. The first two-thirds of the semester is spent on Chapters 1–10, and the last third on Chapters 11–20, teaching C programming, but also showing how some of the magic from the students' first course can be implemented. Coverage of functions, activation records, recursion, pointer variables, and data structures are examples of topics where getting past the magic is particularly useful. The second half of the book can move more quickly since the student has already taken an introductory programming course. This model also allows students who were introduced to programming with an object-oriented language to pick up C, which they will almost certainly need in some of their advanced software courses.

A Sophomore-Level Computer Organization Course

The book has been used to delve deeply into computer implementation in the sophomore year. The semester is spent in Chapters 1 through 10, often culminating in a thorough study of Appendix C, which provides the complete microarchitecture of a microprogrammed LC-3. We note, however, that some very important ideas in computer architecture are not covered in the book, most notably cache memory, pipelining, and virtual memory. Instructors using the book this way are encouraged to provide extra handouts dealing with those topics. We agree that they are very important to the student's computer architecture education, but we feel they are better suited to a later course in computer architecture and design. This book is not intended for that purpose.

Why LC-3, and Not ARM or RISCV?

We have been asked why we invented the LC-3 ISA, rather than going with ARM, which seems to be the ISA of choice for most mobile devices, or RISCV, which has attracted substantial interest over the last few years.

There are many reasons. First, we knew that the ISA we selected would be the student's first ISA, not his/her last ISA. Between the freshman year and graduation, the student is likely to encounter several ISAs, most of which are in commercial products: ARM, RISCV, x86, and POWER, to name a few.

But all the commercial ISAs have details that have no place in an introductory course but still have to be understood for the student to use them effectively. We could, of course, have subset an existing ISA, but that always ends up in questions of what to take out and what to leave in with a result that is not as clean as one would think at first blush. Certainly not as clean as what one can get when starting from scratch. It also creates an issue whenever the student uses an instruction in an exam or on an assignment that is not in the subset. Not very clean from a pedagogical sense.

We wanted an ISA that was clean with no special cases to deal with, with as few opcodes as necessary so the student could spend almost all his/her time on the fundamental concepts in the course and very little time on the nuances of the instruction set. The formats of all instructions in the LC-3 fit on a single page. Appendix A provides all the details (i.e., the complete data sheet) of the entire LC-3 ISA in 25 pages.

We also wanted an instruction set that in addition to containing only a few instructions was very rich in the breadth of what it embraced. So, we came up with the LC-3, an instruction set with only 15 four-bit opcodes, a small enough number that students can absorb the ISA without even trying. For arithmetic, we have only ADD instead of ADD, SUB, MUL, and DIV. For logical operations, we have AND and NOT, foregoing OR, XOR, etc. We have no shift or rotate instructions. In all these cases, the missing opcodes can be implemented with procedures using the few opcodes that the LC-3 provides. We have loads and stores with three different addressing modes, each addressing mode useful for a different purpose. We have conditional branches, subroutine calls, return from trap or interrupt, and system calls (the TRAP instruction).

In fact, this sparse set of opcodes is a feature! It drives home the need for creating more complex functionality out of simple operations, and the need for abstraction, both of which are core concepts in the book.

Most importantly, we have found from discussions with hundreds of students that starting with the LC-3 does not put them at a disadvantage in later courses. On the contrary: For example, at one campus students were introduced to ARM in the follow-on course, while at another campus, students were introduced to x86.

In both cases, students appreciated starting with the LC-3, and their subsequent introduction to ARM or x86 was much easier as a result of their first learning the fundamental concepts with the LC-3.

A Few Observations

Having now taught the course more than 20 times between us, we note the following:

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 are 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 content 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 the day he/she writes his/her first program. The student is taught debugging from his/her first program and is required from the very beginning to use the debugging tools of the simulator to get his/her programs to work. 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 ... with the too-frequent result that the TAs end up writing the programs for the students.

Preparation for the Future: Cutting Through Protective Layers

Professionals who use computers in systems today 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 MATLAB. 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 it 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/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. Faculty members who have taught the follow-on courses have noticed substantial improvement in students' comprehension compared to what they saw before students took our course.

Acknowledgments

It's been 15 years since the second edition came out, and about 20 years since we first put together a course pack of notes that eventually morphed into the first edition. Through those years, a good number of people have jumped in, volunteered help, adopted the book, read through drafts, suggested improvements, and so on. We could easily fill many pages if we listed all their names. We are indebted to each of them, and we deeply appreciate their contributions.

Still, there are some folks we feel compelled to acknowledge here, and we apologize to all those who have helped immeasurably, but are not mentioned here due to space limitations.

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. Kevin co-taught the course at Michigan the first four times it was offered, in 1995–1997. His 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 colleagues and students were our major go-to guys when it came to advice, and insights, on the current version of the book and its future. Wen-mei Hwu, Veynu Narasiman, Steve Lumetta, Matt Frank, Faruk Guvinilir,

Chirag Sakuja, Siavash Zanganeh, Stephen Pruett, Jim Goodman, and Soner Onder have been particularly important to us since the second edition.

McGraw-Hill has been an important partner in this undertaking, starting with Betsy Jones, our first editor. We sent the manuscript of our first edition to several publishers and were delighted with the many positive responses. Nonetheless, one editor stood out. Betsy immediately checked us out, and once she was satisfied, she became a strong believer in what we were trying to accomplish. Throughout the process of the first two editions, her commitment and energy, as well as that of Michelle Flomenhoft, our first development editor, were greatly appreciated. Fifteen years is a long time between editions, and with it have come a whole new collection of folks from McGraw-Hill that we have recently been privileged to work with, especially Dr. Thomas Scaife, Suzy Bainbridge, Heather Ervolino, and Jeni McAtee.

Our book has benefited from extensive comments by faculty members from many universities. Some were in formal reviews of the manuscript, others in e-mails or conversations at technical conferences. We gratefully acknowledge input from Professors Jim Goodman, Wisconsin and Aukland; Soner Onder, Michigan Tech; Vijay Pai, Purdue; Richard Johnson, Western New Mexico; Tore Larsen, Tromso; Greg Byrd, NC State; Walid Najjar, UC Riverside; Sean Joyce, Heidelberg College; James Boettler, South Carolina State; Steven Zeltmann, Arkansas; Mike McGregor, Alberta; David Lilja, Minnesota; Eric Thompson, Colorado, Denver; Brad Hutchings, Brigham Young; 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 Guisrud, Stavanger College; Dave Kaeli, Northeastern; Rasool Kenarangui, UT, Arlington; Joel Kraft, Case Western Reserve; Wei-Ming Lin, UT, San Antonio; Roderick Loss, Montgomery College; Ron Meleshko, Grant MacEwan Community College; Andreas Moshovos, Toronto; 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 Tromsoe; Jean-Pierre Steger, Burgdorf School of Engineering; Bill Sverdlik, Eastern Michigan; John Trono, St. Michael's College; Murali Varansi, University of South Florida; Montanez Wade, Tennessee State; Carl Wick, US Naval Academy; Robert Crisp, Arkansas; Allen Tannenbaum, Georgia Tech; Nickolas Jovanovic, Arkansas–Little Rock; Dean Brock, North Carolina–Asheville; Amar Raheja, Cal State–Pomona; Dayton Clark, Brooklyn College; William Yurcik, Illinois State; Jose Delgado-Frias, Washington State; Peter Drexel, Plymouth State; Mahmoud Manzoul, Jackson State; Dan Connors, Colorado; Massoud Ghyam, USC; John Gray, UMass–Dartmouth; John Hamilton, Auburn; Alan Rosenthal, Toronto; and Ron Taylor, Wright State.

We have continually been blessed with enthusiastic, knowledgeable, and caring TAs who regularly challenge us and provide useful insights into helping us explain things better. Again, the list is too long – more than 100 at this point. Almost all were very good; still, we want to mention a few who were particularly helpful. Stephen Pruett, Siavash Zangeneh, Meiling Tang, Ali Fakhrzadehgan, Sabee Grewal, William Hoenig, Matthew Normyle, Ben Lin, Ameya Chaudhari,

Nikhil Garg, Lauren Guckert, Jack Koenig, Saijel Mokashi, Sruti Nuthalapati, Faruk Guvenilir, Milad Hashemi, Aater Suleman, Chang Joo Lee, Bhargavi Narayanasetty, RJ Harden, Veynu Narasiman, Eiman Ebrahimi, Khubaib, Allison Korczynski, Pratyusha Nidamaluri, Christopher Wiley, Cameron Davison, Lisa de la Fuente, Phillip Duran, Jose Joao, Rustam Miftakhutdinov, Nady Obeid, Linda Bigelow, Jeremy Carillo, Aamir Hasan, Basit Sheik, Erik Johnson, Tsung-Wei Huang, Matthew Potok, Chun-Xun Lin, Jianxiong Gao, Danny Kim, and Iou-Jen Liu.

Several former students, former colleagues, and friends reviewed chapters in the book. We particularly wish to thank Rich Belgard, Alba Cristina de Melo, Chirag Sakhuja, Sabee Grewal, Pradip Bose, and Carlos Villavieja for their help doing this. Their insights have made a much more readable version of the material.

We have been delighted by the enthusiastic response of our colleagues at both The University of Texas at Austin and the University of Illinois, Urbana-Champaign, who have taught from the book and shared their insights with us: Derek Chiou, Jacob Abraham, Ramesh Yerraballi, Nina Telang, and Aater Suleman at Texas, and Yuting Chen, Sayan Mitra, Volodymyr Kindratenko, Yih-Chun Hu, Seth Hutchinson, Steve Lumetta, Juan Jaramillo, Pratik Lahiri, and Wen-mei Hwu at Illinois. Thank you.

Also, there are those who have contributed in many different and often unique ways. Space dictates that we leave out the detail and simply list them and say thank you. Amanda, Bryan, and Carissa Hwu, Mateo Valero, Rich Belgard, Aman Aulakh, Minh Do, Milena Milenkovic, Steve Lumetta, Brian Evans, Jon Valvano, Susan Kornfield, Ed DeFranco, Evan Gsell, Tom Conte, Dave Nagle, Bruce Shriver, Bill Sayle, Dharma Agarwal, David Lilja, Michelle Chapman.

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 50 years since I was in his classroom, but I still treasure the example he set.

A Final Word

We are hopeful that you will enjoy teaching or studying from this third edition of our book. However, as we said in the prefaces to both previous editions, we are mindful that the current version of the book 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 and sjp@illinois.edu. We hope you will.

Yale N. Patt
Sanjay J. Patel
September, 2019

CHAPTER

1 Welcome Aboard

1.1 What We Will Try to Do

Welcome to *From Bits and Gates to C and Beyond*. Our intent is to introduce you over the next xxx pages to the world of computing. As we do so, we have one objective above all others: to show you very clearly that there is no magic to computing. The computer is a deterministic system—every time we hit it over the head in the same way and in the same place (provided, of course, it was in the same starting condition), we get the same response. The computer is not an electronic genius; on the contrary, if anything, it is an electronic idiot, doing exactly what we tell it to do. It has no mind of its own.

What appears to be a very complex organism is really just a very large, systematically interconnected collection of very simple parts. Our job throughout this book is to introduce you to those very simple parts and, step-by-step, build the interconnected structure that you know by the name *computer*. Like a house, we will start at the bottom, construct the foundation first, and then go on to add layer after layer, as we get closer and closer to what most people know as a full-blown computer. Each time we add a layer, we will explain what we are doing, tying the new ideas to the underlying fabric. Our goal is that when we are done, you will be able to write programs in a computer language such as C using the sophisticated features of that language and to understand what is going on underneath, inside the computer.

1.2 How We Will Get There

We will start (in Chapter 2) by first showing that any information processed by the computer is represented by a sequence of 0s and 1s. That is, we will encode all information as sequences of 0s and 1s. For example, one encoding of the letter *a* that is commonly used is the sequence 01100001. One encoding of the decimal number *35* is the sequence 00100011. We will see how to perform operations on such encoded information.

Once we are comfortable with information represented as codes made up of 0s and 1s and operations (addition, for example) being performed on these representations, we will begin the process of showing how a computer works. Starting in Chapter 3, we will note that the computer is a piece of electronic equipment and, as such, consists of electronic parts operated by voltages and interconnected by wires. Every wire in the computer, at every moment in time, is at either a high voltage or a low voltage. For our representation of 0s and 1s, we do not specify exactly how high. We only care whether there is or is not a large enough voltage relative to 0 volts to identify it as a 1. That is, the absence or presence of a reasonable voltage relative to 0 volts is what determines whether it represents the value 0 or the value 1.

In Chapter 3, we will see how the transistors that make up today's microprocessor (the heart of the modern computer) work. We will further see how those transistors are combined into larger structures that perform operations, such as addition, and into structures that allow us to save information for later use. In Chapter 4, we will combine these larger structures into the von Neumann machine, a basic model that describes how a computer works. We will also begin to study a simple computer, the LC-3. We will continue our study of the LC-3 in Chapter 5. *LC-3* stands for Little Computer 3. We actually started with LC-1 but needed two more shots at it before (we think) we got it right! The LC-3 has all the important characteristics of the microprocessors that you may have already heard of, for example, the Intel 8088, which was used in the first IBM PCs back in 1981. Or the Motorola 68000, which was used in the Macintosh, vintage 1984. Or the Pentium IV, one of the high-performance microprocessors of choice for the PC in the year 2003. Or today's laptop and desktop microprocessors, the Intel Core processors – I3, I5, and I7. Or even the ARM microprocessors that are used in most smartphones today. That is, the LC-3 has all the important characteristics of these "real" microprocessors without being so complicated that it gets in the way of your understanding.

Once we understand how the LC-3 works, the next step is to program it, first in its own language (Chapter 5 and Chapter 6), and then in a language called *assembly language* that is a little bit easier for humans to work with (Chapter 7). Chapter 8 introduces representations of information more complex than a simple number – stacks, queues, and character strings, and shows how to implement them. Chapter 9 deals with the problem of getting information into (input) and out of (output) the LC-3. Chapter 9 also deals with services provided to a computer user by the operating system. We conclude the first half of the book (Chapter 10) with an extensive example, the simulation of a calculator, an app on most smartphones today.

In the second half of the book (Chapters 11–20), we turn our attention to high-level programming concepts, which we introduce via the C and C++ programming languages. High-level languages enable programmers to more effectively develop complex software by abstracting away the details of the underlying hardware. C and C++ in particular offer a rich set of programmer-friendly constructs, but they are close enough to the hardware that we can examine how code is transformed to execute on the layers below. Our goal is to enable you to write short, simple programs using the core parts of these programming languages, all the while being able to comprehend the transformations required for your code to execute on the underlying hardware.

We'll start with basic topics in C such as variables and operators (Chapter 12), control structures (Chapter 13), and functions (Chapter 14). We'll see that these are straightforward extensions of concepts introduced in the first half of the textbook. We then move on to programming concepts in Chapters 15–19 that will enable us to create more powerful pieces of code: Testing and Debugging (Chapter 15), Pointers and Arrays in C (Chapter 16), Recursion (Chapter 17), Input and Output in C (Chapter 18), and Data Structures in C (Chapter 19).

Chapters 20 is devoted to C++, which we present as an evolution of the C programming language. Because the C++ language was initially defined as a superset of C, many of the concepts covered in Chapters 11–19 directly map onto the C++ language. We will introduce some of the core notions in C++ that have helped establish C++ as one of the most popular languages for developing real-world software. Chapter 20 is our Introduction to C++.

In almost all cases, we try to tie high-level C and C++ constructs to the underlying LC-3 so that you will understand what you demand of the computer when you use a particular construct in a C or C++ program.

1.3 Two Recurring Themes

Two themes permeate this book that we as professors previously took for granted, assuming that everyone recognized their value and regularly emphasized them to students of engineering and computer science. However, it has become clear to us that from the git-go, we need to make these points explicit. So, we state them here up front. The two themes are (a) the notion of abstraction and (b) the importance of not separating in your mind the notions of hardware and software. Their value to your development as an effective engineer or computer scientist goes well beyond your understanding of how a computer works and how to program it.

The notion of abstraction is central to all that you will learn and expect to use in practicing your craft, whether it be in mathematics, physics, any aspect of engineering, or business. It is hard to think of any body of knowledge where the notion of abstraction is not critical.

The misguided hardware/software separation is directly related to your continuing study of computers and your work with them.

We will discuss each in turn.

1.3.1 The Notion of Abstraction

The use of abstraction is all around us. When we get in a taxi and tell the driver, "Take me to the airport," we are using abstraction. If we had to, we could probably direct the driver each step of the way: "Go down this street ten blocks, and make a left turn." And, when the driver got there, "Now take this street five blocks and make a right turn." And on and on. You know the details, but it is a lot quicker to just tell the driver to take you to the airport.

Even the statement "Go down this street ten blocks …" can be broken down further with instructions on using the accelerator, the steering wheel, watching out for other vehicles, pedestrians, etc.

Abstraction is a technique for establishing a simpler way for a person to interact with a system, removing the details that are unnecessary for the person to interact effectively with that system. Our ability to abstract is very much a productivity enhancer. It allows us to deal with a situation at a higher level, focusing on the essential aspects, while keeping the component ideas in the background. It allows us to be more efficient in our use of time and brain activity. It allows us to not get bogged down in the detail when everything about the detail is working just fine.

There is an underlying assumption to this, however: *when everything about the detail is just fine*. What if everything about the detail is not just fine? Then, to be successful, our ability to abstract must be combined with our ability to *un*-abstract. Some people use the word *deconstruct*—the ability to go from the abstraction back to its component parts.

Two stories come to mind.

The first involves a trip through Arizona the first author made a long time ago in the hottest part of the summer. At the time he was living in Palo Alto, California, where the temperature tends to be mild almost always. He knew enough to take the car to a mechanic before making the trip and tell him to check the cooling system. That was the abstraction: cooling system. What he had not mastered was that the capability of a cooling system for Palo Alto, California, is not the same as the capability of a cooling system for the summer deserts of Arizona. The result: two days in Deer Lodge, Arizona (population 3), waiting for a head gasket to be shipped in.

The second story (perhaps apocryphal) is supposed to have happened during the infancy of electric power generation. General Electric Co. was having trouble with one of its huge electric power generators and did not know what to do. On the front of the generator were lots of dials containing lots of information, and lots of screws that could be rotated clockwise or counterclockwise as the operator wished. Something on the other side of the wall of dials and screws was malfunctioning and no one knew what to do. As the story goes, they called in one of the early giants in the electric power industry. He looked at the dials and listened to the noises for a minute, then took a small screwdriver from his pocket and rotated one screw 35 degrees counterclockwise. The problem immediately went away. He submitted a bill for $1000 (a lot of money in those days) without any elaboration. The controller found the bill for two minutes' work a little unsettling and asked for further clarification. Back came the new bill:

```
Turning a screw 35 degrees counterclockwise:  $  0.75
Knowing which screw to turn and by how much:    999.25
```

In both stories the message is the same. It is more efficient to think of entities as abstractions. One does not want to get bogged down in details unnecessarily. And as long as nothing untoward happens, we are OK. If there had been no trip to Arizona, the abstraction "cooling system" would have been sufficient. If the electric power generator never malfunctioned, there would have been no need for the power engineering guru's deeper understanding.

As we will see, modern computers are composed of transistors. These transistors are combined to form logic "gates"—an abstraction that lets us think in terms of 0s and 1s instead of the varying voltages on the transistors. A logic circuit is a further abstraction of a combination of gates. When one designs a logic circuit out of gates, it is much more efficient to not have to think about the internals of each gate. To do so would slow down the process of designing the logic circuit. One wants to think of the gate as a component. But if there is a problem with getting the logic circuit to work, it is often helpful to look at the internal structure of the gate and see if something about its functioning is causing the problem.

When one designs a sophisticated computer application program, whether it be a new spreadsheet program, word processing system, or computer game, one wants to think of each of the components one is using as an abstraction. If one spent time thinking about the details of each component when it was not necessary, the distraction could easily prevent the total job from ever getting finished. But when there is a problem putting the components together, it is often useful to examine carefully the details of each component in order to uncover the problem.

The ability to abstract is the most important skill. In our view, one should try to keep the level of abstraction as high as possible, consistent with getting everything to work effectively. Our approach in this book is to continually raise the level of abstraction. We describe logic gates in terms of transistors. Once we understand the abstraction of gates, we no longer think in terms of transistors. Then we build larger structures out of gates. Once we understand these larger abstractions, we no longer think in terms of gates.

The Bottom Line Abstractions allow us to be much more efficient in dealing with all kinds of situations. It is also true that one can be effective without understanding what is below the abstraction as long as everything behaves nicely. So, one should not pooh-pooh the notion of abstraction. On the contrary, one should celebrate it since it allows us to be more efficient.

In fact, if we never have to combine a component with anything else into a larger system, and if nothing can go wrong with the component, then it is perfectly fine to understand this component only at the level of its abstraction.

But if we have to combine multiple components into a larger system, we should be careful not to allow their abstractions to be the deepest level of our understanding. If we don't know the components below the level of their abstractions, then we are at the mercy of them working together without our intervention. If they don't work together, and we are unable to go below the level of abstraction, we are stuck. And that is the state we should take care not to find ourselves in.

1.3.2 Hardware vs. Software

Many computer scientists and engineers refer to themselves as hardware people or software people. By hardware, they generally mean the physical computer and all the specifications associated with it. By software, they generally mean the programs, whether operating systems like Android, ChromeOS, Linux, or Windows, or database systems like Access, MongoDB, Oracle, or DB-terrific, or application programs like Facebook, Chrome, Excel, or Word. The implication is that the person knows a whole lot about one of these two things and precious little about the other. Usually, there is the further implication that it is OK to be an expert at one of these (hardware OR software) and clueless about the other. It is as if there were a big wall between the hardware (the computer and how it actually works) and the software (the programs that direct the computer to do their bidding), and that one should be content to remain on one side of that wall or the other.

The power of abstraction allows us to "usually" operate at a level where we do not have to think about the underlying layers all the time. This is a good thing. It enables us to be more productive. But if we are clueless about the underlying layers, then we are not able to take advantage of the nuances of those underlying layers when it is very important to be able to.

That is not to say that you must work at the lower level of abstraction and not take advantage of the productivity enhancements that the abstractions provide. On the contrary, you are encouraged to work at the highest level of abstraction available to you. But in doing so, if you are able to, at the same time, keep in mind the underlying levels, you will find yourself able to do a much better job.

As you approach your study and practice of computing, we urge you to take the approach that hardware and software are names for components of two parts of a computing system that work best when they are designed by people who take into account the capabilities and limitations of both.

Microprocessor designers who understand the needs of the programs that will execute on the microprocessor they are designing can design much more effective microprocessors than those who don't. For example, Intel, AMD, ARM, and other major producers of microprocessors recognized a few years ago that a large fraction of future programs would contain video clips as part of e-mail, video games, and full-length movies. They recognized that it would be important for such programs to execute efficiently. The result: most microprocessors today contain special hardware capability to process those video clips. Intel defined additional instructions, initially called their MMX instruction set, and developed special hardware for it. Motorola, IBM, and Apple did essentially the same thing, resulting in the AltiVec instruction set and special hardware to support it.

A similar story can be told about software designers. The designer of a large computer program who understands the capabilities and limitations of the hardware that will carry out the tasks of that program can design the program so it executes more efficiently than the designer who does not understand the nature of the hardware. One important task that almost all large software systems need to carry out is called sorting, where a number of items have to be arranged in some order. The words in a dictionary are arranged in alphabetical order. Students in a class are often graded based on a numerical order, according to their scores on the final exam. There is a large number of fundamentally different programs one can write to arrange a collection of items in order. Donald Knuth, one of the top computer scientists in the world, devoted 391 pages to the task in *The Art of Computer Programming*, vol. 3. Which sorting program works best is often very dependent on how much the software designer is aware of the underlying characteristics of the hardware.

The Bottom Line We believe that whether your inclinations are in the direction of a computer hardware career or a computer software career, you will be much more capable if you master both. This book is about getting you started on the path to mastering both hardware and software. Although we sometimes ignore making the point explicitly when we are in the trenches of working through a concept, it really is the case that each sheds light on the other.

When you study data types, a software concept, in C (Chapter 12), you will understand how the finite word length of the computer, a hardware concept, affects our notion of data types.

When you study functions in C (Chapter 14), you will be able to tie the *rules* of calling a function with the hardware implementation that makes those rules necessary.

When you study recursion, a powerful algorithmic device (initially in Chapter 8 and more extensively in Chapter 17), you will be able to tie it to the hardware. If you take the time to do that, you will better understand when the additional time to execute a procedure recursively is worth it.

When you study pointer variables in C (in Chapter 16), your knowledge of computer memory will provide a deeper understanding of what pointers provide, and very importantly, when they should be used and when they should be avoided.

When you study data structures in C (in Chapter 19), your knowledge of computer memory will help you better understand what must be done to manipulate the actual structures in memory efficiently.

We realize that most of the terms in the preceding five short paragraphs may not be familiar to you *yet*. That is OK; you can reread this page at the end of the semester. What is important to know right now is that there are important topics in the software that are very deeply interwoven with topics in the hardware. Our contention is that mastering either is easier if you pay attention to both.

Most importantly, most computing problems yield better solutions when the problem solver has the capability of both at his or her disposal.

1.4 A Computer System

We have used the word *computer* more than two dozen times in the preceding pages, and although we did not say so explicitly, we used it to mean a system consisting of the software (i.e., computer programs) that directs and specifies the processing of information and the hardware that performs the actual processing of information in response to what the software asks the hardware to do. When we say "performing the actual processing," we mean doing the actual additions, multiplications, and so forth in the hardware that are necessary to get the job done. A more precise term for this hardware is a *central processing unit* (CPU), or simply a *processor* or *microprocessor*. This textbook is primarily about the processor and the programs that are executed by the processor.

1.4.1 A (Very) Little History for a (Lot) Better Perspective

Before we get into the detail of how the processor and the software associated with it work, we should take a moment and note the enormous and unparalleled leaps of performance that the computing industry has made in the relatively short time computers have been around. After all, it wasn't until the 1940s that the first computers showed their faces. One of the first computers was the ENIAC (the Electronic Numerical Integrator and Calculator), a general purpose electronic computer that could be reprogrammed for different tasks. It was designed and built in 1943–1945 at the University of Pennsylvania by Presper Eckert and his colleagues. It contained more than 17,000 vacuum tubes. It was approximately 8 feet high, more than 100 feet wide, and about 3 feet deep (about 300 square feet of floor space). It weighed 30 tons and required 140 kW to operate. Figure 1.1 shows three operators programming the ENIAC by plugging and unplugging cables and switches.

About 40 years and many computer companies and computers later, in the early 1980s, the Burroughs A series was born. One of the dozen or so 18-inch boards that comprise that machine is shown in Figure 1.2. Each board contained 50 or more integrated circuit packages. Instead of 300 square feet, it took up around 50 to 60 square feet; instead of 30 tons, it weighed about 1 ton, and instead of 140 kW, it required approximately 25 kW to operate.

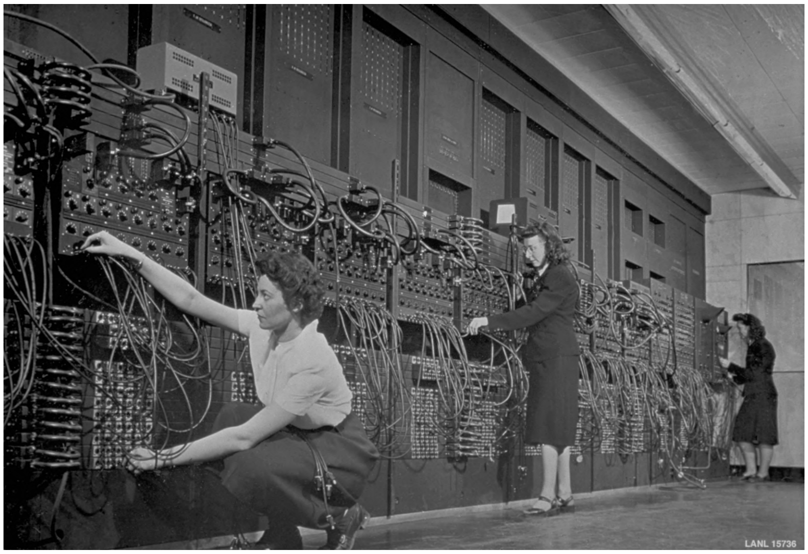

Figure 1.1 The ENIAC, designed and built at University of Pennsylvania, 1943–45. ©Historical/Getty Images

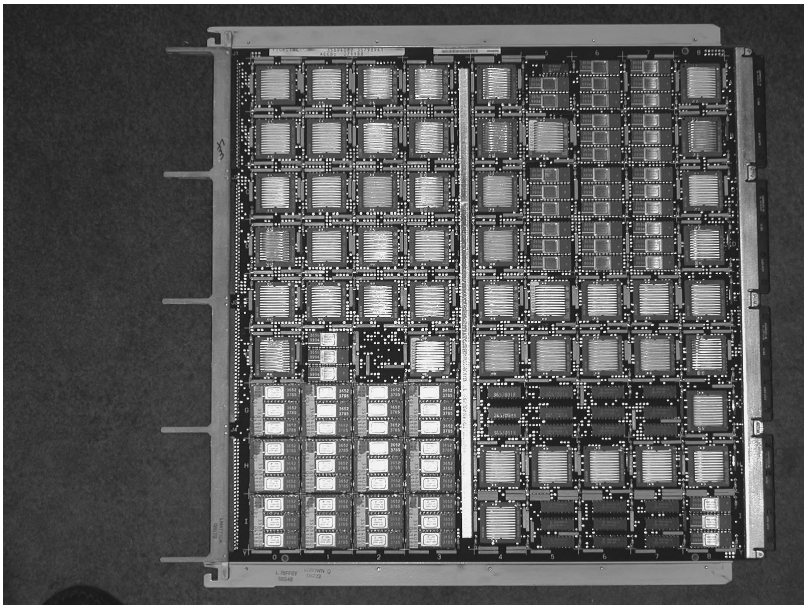

Figure 1.2 **A processor board, vintage 1980s. Courtesy of Emilio Salguerio**

Fast forward another 30 or so years and we find many of today's computers on desktops (Figure 1.3), in laptops (Figure 1.4), and most recently in smartphones (Figure 1.5). Their relative weights and energy requirements have decreased enormously, and the speed at which they process information has also increased enormously. We estimate that the computing power in a smartphone today (i.e., how fast we can compute with a smartphone) is more than four million times the computing power of the ENIAC!

Figure 1.3 **A desktop computer. ©Joby Sessions/Future/REX/Shutterstock**

Figure 1.4 **A laptop. ©Rob Monk/Future/REX/Shutterstock**

Figure 1.5 **A smartphone. ©Oleksiy Maksymenko/imageBROKER/REX/Shutterstock**