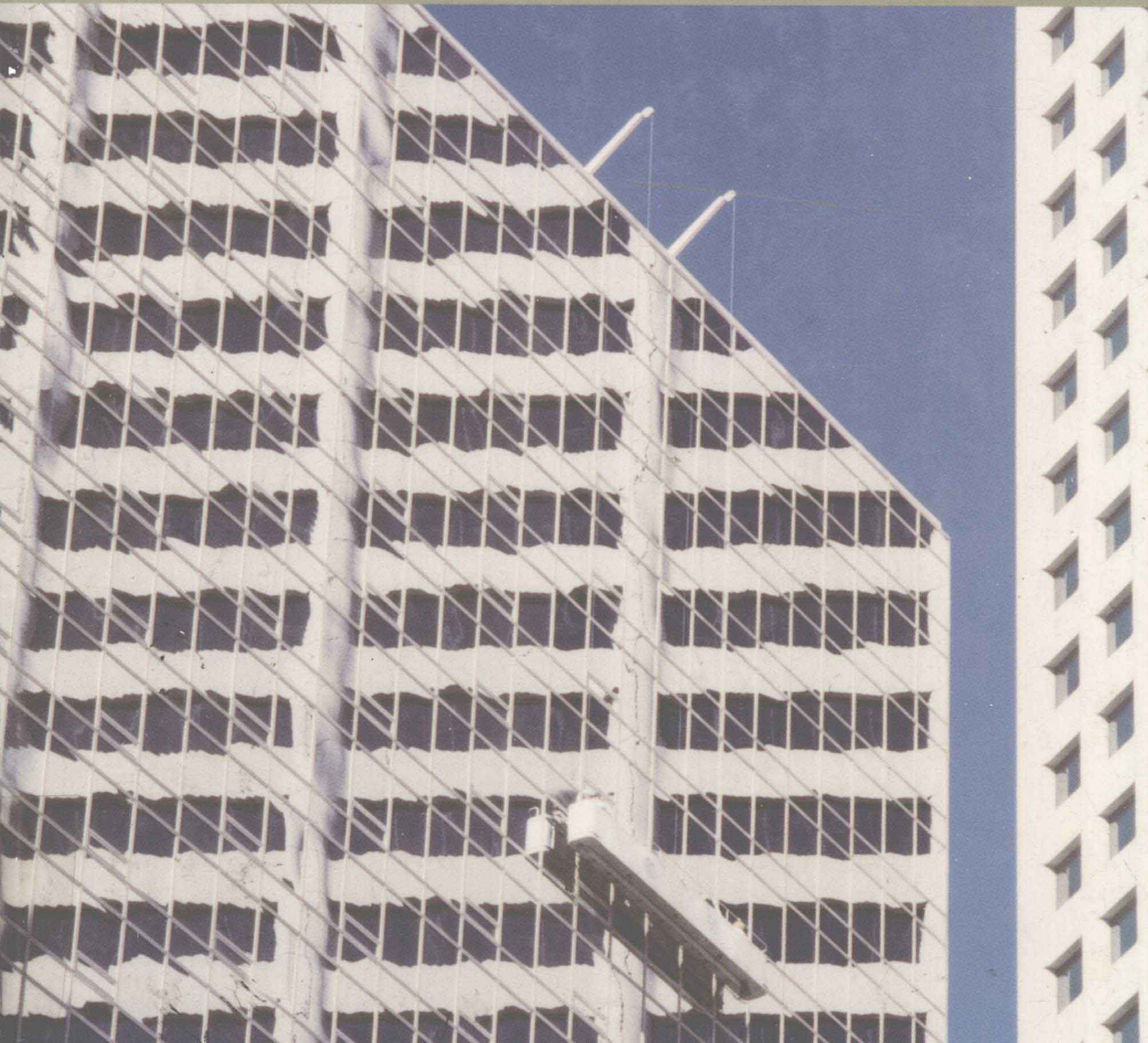
ADVANCED COMPUTER ARCHITECTURE

A Systems Design Approach

RICHARD Y. KAIN



ADVANCED COMPUTER ARCHITECTURE

A Systems Design Approach

RICHARD Y. KAIN

Department of Electrical Engineering
University of Minnesota

PRENTICE HALL, Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging-in-Publication Data

Kain, Richard Y.
Advanced computer architecture : a systems design approach
Richard Y. Kain
p. cm.
Includes bibliological references and index.
ISBN: 0-13-007741-0 (hard cover)
1. Computer architecture I . Title.
QA76.9.A73K363 1996
004.2'2--dc20 95-18657
CIP

Acquisitions editor: **Alan Apt**
Production editor: **Bayani Mendoza deLeon**
Copy editor: **Peter Zurita**
Cover designer: **Bruce Kenselaar**
Buyer: **Donna Sullivan**
Editorial assistant: **Shirley McGuire**

Printed in the United States of America

10 9 8 7 6 5 4 3 2

cover photograph copyright 1992 by Richard Y. Kain

ISBN 0-13-007741-0

Prentice-Hall International (UK) Limited, London
Prentice-Hall of Australia Pty. Limited, Sydney
Prentice-Hall Canada Inc., Toronto
Prentice-Hall Hispanoamericana, S.A., Mexico
Prentice-Hall of India Private Limited, New Delhi
Prentice-Hall of Japan, Inc., Tokyo
Simon & Schuster Asia Pte. Ltd., Singapore
Editora Prentice-Hall do Brasil, Ltda., Rio de Janeiro

PREFACE

The man who sees two or three generations
is like one who sits in the conjurer's booth at a fair,
and sees the tricks two or three times.
They are meant to be seen only once.

— Schopenhauer

Often tricks are repeated over generations, but are still meant to surprise the audience, according to Schopenhauer. It is instructive to view this from the other side—the magician must learn basic tricks before performing and astounding an audience. The history of computer design follows this pattern: Architectures have progressed through several "generations," usually defined in terms of the implementation technology, and many "tricks" have been rediscovered and re-used. Many techniques have been reinvented, because the old ones have been forgotten. I believe that although designers face different trade-offs in different technologies, they work with the same basic bag of tricks to conjure up new architectures.

In this book, I describe many architectural tricks and position them within a useful structure. The structure should clarify the similarities and differences between architectures by exhibiting not only the basic tricks and techniques, but also the relationships between software and hardware levels of system implementation and operation.

As designers ponder basic design trade-offs and seek good solutions to their design challenges, they should study the requirements and design options for the complete system. The system will include many modules arranged in levels, ranging from the processor outward to operating systems, programming languages, and application structures. For a number of years, it has been obvious that the design questions, design options, and approaches to the design questions have many similarities across the levels of the system. By being aware of the needs of the application, programming language structure, and operating system functions when designing the processor architecture, a design that supports a more efficient system might be developed.

Historical Designs. The book includes examples from a number of programming languages, operating systems, and processor designs, some new and some old. By exposing you to diverse examples, I hope to convince you that many design options are feasible, at least in the context of specific design environments. I also show you details needed to complete the designs, or I challenge you to find the details as you work through the problems.

Even though an arbitrary combination of design choices might not produce an effective system, we should consider and remember design techniques developed by our predecessors, because their techniques might be useful in many situations.

Many old design strategies are revived or reinvented as technology evolves and the design environment changes. We should not let the good designs of the past be forgotten, because once forgotten, it is a waste of time and energy to reinvent them later. And then they will have to be debugged all over again!

Why Did I Choose this Approach? Klapp [KLAP86] suggests that one contributor to the current "lag in meaning" is that people do not have enough time to ponder recent ideas. Time for "wondering" is time for finding new connections and relationships. The rapid introduction of new computer systems, programming languages, and myriad applications keeps designers and implementers busy meeting pressing deadlines. This may leave them with insufficient time to wonder about and ponder the overall scene.

My musings about how to approach architectural issues resulted in this book; this is the current version of my ongoing quest for a coherent personation of computer design issues. I am trying to join issues and techniques from software and hardware levels. I believe that better system designs can be achieved when designers start with a clear conception of the design problem and are familiar with the approaches used by their predecessors to solve similar problems at all levels of the overall system.

I have chosen to present a coherent approach to computer system design that encompasses many, if not most, of the design problems and solution options, starting from the structures of contemporary programming languages and operating systems, extending inward to the processor's architecture and its implementation. Many common design issues and certain approaches and solution options could be applied at several levels of system implementation. To emphasize the commonalty and to impose structure on the presentation, I tend to place general principles and design approaches at the beginning of each chapter and then progress in a top–down manner (from programming languages toward the processor and its implementation), using examples from actual programming languages, operating systems, and processors. In this manner, I detail and illustrate the problems and the designs that solve those problems.

There is considerable overlap between the material in some sections of this book and parts of certain computer science courses, particularly courses on the structure of high-level languages and operating systems, with a pinch of database topics thrown in. A reader who has studied these topics before might skim over these sections to see how I present the material (which does not always conform to the conventional terminology used in the specialized field). Although these topics might not be considered to be architectural topics by some purists, they are connected with processor architectures and with features that lead to high performance in real applications environments. Recall that the ultimate purpose of computer science is to understand how to develop algorithms and data structures that use computing machines to effectively solve large, difficult problems. Therefore, techniques that are useful in computer science have to be implemented somewhere within the total system. Thus, one should look with an eye toward how they can be supported within a system's architecture. The ultimate outcome of this thinking might be a software solution, but it might suggest direct processor support for features that otherwise might not be considered important. For example, if one were taking a view that functional necessity (in a mathematical sense) is the only criterion for including a feature within a processor, one would have difficulty implementing a secure system.

This book is a major rewrite that started from my earlier two-volume book [KAIN89]. I changed the emphasis in programming examples from Ada to C++ and from the MC68020 processor to RISC processors. I expanded and clarified examples and explanations and added more examples from contemporary systems. I added more details about sharing and coordination between cooperating processes. I reduced the explanations of many introductory topics and rely now on the reader's background with programming in C++ or a similar language and with the design of RISC processors, at the level of Patterson and Hennessy [PATT93]. Some background material and some specialized topics have been moved into appendices. The fact that some material has moved to an appendix does not reduce its importance; it merely suggests that it could be skipped by a reader desiring to obtain an overview of the majority view of what constitutes computer architecture. I believe that with sufficient basic background, the reader can progress through the chapters, referring to the appendices only for detailed information about certain processors. Along this vein, note that the last appendix contains a listing of significant portions of several processor instruction sets; thus the reader can perform exercises that involve writing realistic programs for these processors, which will give an appreciation for the significant differences among several design approaches.

About Performance Topics. In contemporary practice, to get a paper accepted for publication in a computer architecture journal or conference, the authors must simulate the proposed design and exercise that simulator against "typical" programs. Many performance tables are presented. I have chosen to downplay this aspect of architectural studies, because they do not affect the logical structure of the architect's design options. Furthermore, the ultimate measure of a system's performance requires an incredibly detailed simulation of the complete design, which clearly reaches to the scope of a large project, and therefore is beyond the scope of this text.

Another reason for omitting performance measures is that the results depend upon the particular programs used to exercise the simulator. In many studies, these are quite narrow, typically relying on certain Unix programs that use few, if any, floating-point operations or on numeric applications that heavily exercise floating-point arithmetic. One might question the degree to which these represent "typical" programs. And often one sees a significant variance among the results across the set of programs. These comments should not be taken as criticism of all papers that contain performance statistics; rather I wish to raise your level of skepticism about performance results that do appear in the literature. A careful look at what was simulated and what programs were used should give a hint about the degree to which the results might apply to a different environment.

General and Specific Designs. The book presents optional ways to satisfy many real-world goals that challenge designers of contemporary systems and processors. In most chapters and appendices, one basic problem domain and the fundamental approaches to those problems are developed at the beginning. Uses of these basic approaches to specific situations at various levels are amplified in the chapter. Some concepts, design details, and comparisons are presented and detailed in problems rather than in the text. Therefore, designers are advised to look at the problems in conjunction with the text.

Background. I assume that the reader is familiar with at least one processor design, one operating system, one programming language (preferably C++), and the design of basic pipelined RISC processors. I do not assume any experience with multithread systems or with secure systems, which are covered extensively in Chapters 6 through 9.

Structure of the Book. The book's structure and approach reflect my belief that the best way to understand a set of related problems and design issues is first, to see the general techniques that could be used, and, second, to see how those have been applied, both in straight and varied manners and in various combinations, within real systems. To illustrate the common features, it is necessary to structure the text around common problems or around common elements of the system design. The book can be divided into three parts. The first part has a structure based on the von Neumann breakdown of a uniprocessor system—there are memory, control, and functional elements to each design. The second part addresses issues that arise when multiple threads of control or thought are simultaneously active within the overall system. The third part deals with the additional constraints imposed on the system by protection and security concerns. A brief outline of the topics in each chapter and appendix follows:

Chapter 1. Illusions: An introduction to the basic assumptions or illusions that programmers use to cope with the complexity of computer systems. The illusions reappear in the other chapters, where we point out how design options can support the illusions.

Chapter 2. Instruction Set Design: A quick review of basic ideas in processor structure, which become the basis for many examples in the following chapters. This is supplemented by material in Appendices A through D, where specific processor architectures are detailed.

Chapter 3. Memory Organization: The process of getting from the name of a memory object to an access to a physical location holding a set of bits; issues of name mapping implementations (from symbol tables to hash tables and hierarchical tables); linking; segmentation; paging; cache memories; memory interconnection structures. The material in Appendices E and F provides an essential background for the discussion in this chapter. Appendix G discusses the functions of associative memories and their implementation.

Chapter 4. Single-Stream Control: Basic structures for specifying control flow through a sequentially processed program; Harvard and von Neumann architectures; instruction representations; and brief comments regarding microprogramming. This material is supplemented by Appendix H, which covers the nonsequential control structures in Prolog.

Chapter 5. Object-Oriented Processing: Object types and classes; type-based modularization; and processor and system support for type-based modularization. This material is supplemented by Appendix I, covering the LISP language and the design of a machine to support that language.

Chapter 6. Single I-Stream Parallelism: Importance of static discovery or development of parallelism; ways to express parallelism within programs; finding and adding parallelism to programs; SIMD array processors; MIMD array processors; pipeline and barrel processors; and VLIW processors. Dynamic detection of parallelism; superscalar architectures. Ways to restructure programs to enhance the possibility of parallel execution. Appendix J covers systolic arrays, which can perform special algorithms using static parallelism.

Chapter 7. Parallelism by Message Passing: Models of message passing; functional correctness; programming models; interprocess and intermodule communication; and input/output interactions. This material is supplemented by Appendix K, covering data flow systems and their implementation. Some correctness proof arguments in Appendix L relate directly to these models.

Chapter 8. Shared-Resource Systems: Shared-memory systems; functional correctness models; synchronization and coherence for programs and caches; basic uninterruptible instructions and their usage; implementations that support the single copy and program order illusions; barrier instructions. Appendix L discusses some reasoning strategies useful with shared-resource systems.

Chapter 9. Protection and Security: Protection schemes using page and segment structures. Security requirements and their logical consequences, including the Bell and LaPadula security model and the Biba integrity model; basic design approaches that might assure security properties; reference monitor design approaches; and capability and type enforcement approaches to secure system design.

Appendix A. SPARC Summary: The essential architectural features of the SPARC processors, which has evolved from the early Berkeley RISC designs.

Appendix B. Alpha AXP: Essential features of this DEC microprocessor design, which is based on several interesting design options, including the use of low-level software to implement essential functions.

Appendix C. MC680x0 Processors: Some features of this architecture, with emphasis on the MC68020, chosen because it includes some interesting features that disappeared in later "compatible" processors; these include a flexible coprocessor interface design and instructions for calling modules that execute somewhat independently. These processors were inside all initial models of the Macintosh family of personal computers.

Appendix D. Stack-Oriented Systems: The B 5700 and HP 3000 systems; stack structures to support stacked allocations for activation blocks; special addressing modes that access local objects allocated within the stacked activation blocks. The B 5700 uses the stack in place of registers in a load/store architecture.

Appendix E. Naming Memory Objects: Techniques for specifying objects in programming languages and within processor instructions. Special emphasis on nested naming environments and two-dimensional addressing, because these pose important design problems and implementation options.

Appendix F. Memory Allocation: Basic allocation policies and the supporting management data structures; and options for processor support for these functions.

Appendix G. Associative Memories: Operations provided in general-purpose associative memories (more flexible than the simple equality-search memories used in memory accessing). Several implementation strategies.

Appendix H. Prolog: The structure of this language based on "facts" and "rules" collected within a database that serves as both the memory and the program. Some emphasis on the difficulties of implementing this language with parallelism.

Appendix I. List Processing: Structure of a list processing language, emphasizing the data structures, the run-time definition capabilities, and the overlap between data and program in this language. An overview of the Symbolics processors designed for efficient execution of LISP programs. Some emphasis on their choice of list representations, which led to efficiency.

Appendix J. Systolic Arrays: Structure of systolic arrays, demonstrating the difference between passing a program across data (the conventional view) and passing data through an array (the systolic view). Mathematical descriptions of these systems. Unfortunately, this approach seems to apply to only a few classes of algorithms.

Appendix K. Data Flow Systems: Structure and implementation of these systems, which are based on asynchronous activations of processing modules that communicate by passing tokens across communication paths. These structures provide a basis for some proofs discussed in the next appendix.

Appendix L. Reasoning and Proofs: A few approaches to proving properties of systems and programs. History matrices and induction arguments.

Examples. The text's programming examples use various programming languages, frequently C++, which is only partially explained in the text. In a similar manner, the processor examples emphasize features of several microprocessors—the SPARC, the DEC Alpha AXP, and the Motorola MC680x0, especially the MC68020.[1] These three processor architectures are surveyed in Appendices A through C. Processors that directly support stack and list structures are presented in Appendices D and I.

Many examples are based on features from other programming languages, operating systems, multiprocessor structures, and processor designs. I have not tried to make you an expert on all details of any of these designs; there are always myriad details needed to fashion a practical realization of an effective system.

Trends and Concept List. Each chapter (or appendix) ends with three sections, the first, titled "Trends," contains a brief projection of some apparent trends related to the material covered in the chapter. The second section, entitled "Concept List," contains a listing of the important concepts that were discussed in the chapter. And the third is the problem section.

Problems. I believe that solving problems is an important part of learning from a text, for only in this way can you appreciate the subtle design details or the subtle consequences of choosing certain design options. Some problems are

[1] I have received several comments to the effect that this architecture has been supplanted by subsequent developments, which is indeed true. I continue to use it because two of its features are architecturally interesting, but have been dropped from subsequent processors in the same series.

straightforward. Other problems pose design goals that may not be completely specified. Therefore, part of your problem time will be spent interpreting the problem statement. Some interesting and challenging problems place you in the role of design consultant and critic—you will be given a design proposal to critique and asked to examine its implications for the complete system. The critique problems should be considered thoughtfully; I do not recommend trying to solve them in a single session. You should think about the proposal and then give your mind at least overnight to ruminate on the proposal's effects on a complete system before you complete your critique. Except for the introductory first chapter, every chapter and appendix contains some problems related to the material in the chapter or appendix.

The Book as a Text. The book is written for use as a text. Seniors majoring in computer science and graduate students in electrical engineering should be able to cover the material in two quarters or in an intense single semester course. A single semester schedule may require compressing things, making it a challenge to leave sufficient time for reflection and thought about the points being made. The single semester schedule will require eliminating many of the appendices from the classroom presentation.

The Book as a Design Tool. The book is useful as a design tool. It incorporates the essential features of and the concepts behind many diverse designs. For this reason, it should be useful to both architects and implementers, enabling architects to have more design options at their command and implementers to understand the roles of their module(s) within a complete system.

About the Language (in the Book). Contemporary writers face many problems concerning the "correct" usage of the English language. Regarding computer systems or other technological artifacts, there is the problem of when the systems or languages are, or were, in common usage. This difficulty leads the author to question which tenses should be used when discussing these artifacts. I want to explain my approach to these difficulties immediately, so the choices will not lead the reader into incorrect or deceptive interpretations or conclusions. I have chosen the following linguistic options:

1. All languages, systems, and designs are described in the present tense, even though some of them are obviously obsolete as I write and others may become obsolete soon after the book becomes available. A design or language can become obsolete for many nontechnical reasons. These factors do not imply that all ideas within the design are not useful, but rather, the design environment might have changed as technology or applications evolve.
2. I use C++ syntax for all programs that could be written in C++ and for programs incorporating concepts or structures that could become compatible extensions of C++. However, C++ is not a "rich" language, in the sense that

it does not present all possible general structures and implementation challenges. Thus, some programming examples illustrate structures or concepts that would never be considered compatible with C++. One example is the static nesting of procedure definitions. Examples that are unlikely to be considered compatible with C++ are expressed in a Pascal/Ada syntax, which is not explained in detail. The reader should be able to read these programs and understand their structures by working from the first chapter forward; except in a few easily detected places, the reader is not asked to produce any programs incompatible with C++ or common assembly language structures. I believe that most, if not all, example programs are syntactically correct, but this is not a programming text, so it should not be used as an accurate reference for program syntax.

3. Programming languages do not support boldface symbols, but I do use boldface to emphasize keywords within programs and to distinguish keywords from common English words within the body of the text. For example, I talk about the **for** loop structure using a boldface font for **for** when it denotes the keyword in the text or in programs. I also put all symbols that appear in programs into the same sans-serif font used for programs.
4. Within assembly-level programs, I have tried to use the manufacturer's syntax, explaining it only as it seems necessary.
5. All comments in C++ programs are marked using the C++ style—with double slashes to set off each comment, which runs to the end of the line. This comment structure may not be compatible with the conventions of the programming language used in the remainder of the line. Comments in other programs are sometimes expressed in the Ada style, using a double dash to set off the comment.

Best Wishes. I hope that you will learn interesting and useful things from the book. In particular, I hope that you will appreciate the connections between software and hardware design issues, leading you to a better appreciation of the interlocking nature of these issues and their solutions. I hope that you will learn optional ways to approach difficult, but not impossible, design goals and constraints. Also, I hope that you will learn about a few historically important designs that were innovative in their times; many deserve continuing study to assist the understanding of future designers. Perhaps in your time you will write a book that follows this tradition and that you will contribute to keeping alive my approach of integrating contemporary, historical, and untried (but logically interesting) design approaches!

Richard Y. Kain
Minneapolis, Minnesota

ACKNOWLEDGMENTS

My students and colleagues have provided many helpful suggestions for improving this text, and essential encouragement for this project. The students in my classes have tolerated sketchy drafts of this book.

I extend special thanks to Mark Smotherman and Matthew Frank for voluntary reviews of the manuscript; they gave very helpful suggestions. Three students from my class contributed significant editorial suggestions; I thank Derek Lee, Scott vander Linde, and Greg Younker for this assistance.

The Prentice Hall reviewers, who made many helpful suggestions, include Stephen J. Hartley, Dan C. Marinescu, Paul W. Ross, Robert Seban, and Douglas Reeves. Several of them suggested that the draft had too much material; this led me into a vast late-stage reorganization that moved one-third of the material into the appendices. Of course, they are not responsible for any of the contents of the book—after all, they probably do not agree with all of it anyhow!

I received some interesting suggestions from people watching the comp.arch bulletin board on the Internet, though I did not place priority on all of their suggestions. Special thanks go to George Papp, who saw my appeal and responded with helpful processor manuals.

The original impetus for this book came from Prentice Hall editors who suggested that I make a significant revision of the previous book, compressing it and updating material; Tom McElwee was the impetus in that direction. When he moved to marketing, Bill Zobrist ably took over and reined me in to complete the project. Chris Certain, the local representative, was quite helpful in getting materials in a timely manner. The production editing was started by Bayani DeLeon and finished by Rose Kernan, an able production editor who tolerated last minute changes, including adding her name to these acknowledgments.

People at the AmiPro help line also deserve some thanks for getting me out of several binds during the writing and production stages of this project.

Many people contributed to the previous book from which this was derived by a tortuous process; those not mentioned here influenced this text, indirectly through the old version.

Special thanks go to my wife, Katherine Simon Frank, without whose help and encouragement this would not have been possible. Now we can both enjoy life more as we celebrate the fact that the book is done!

CONTENTS