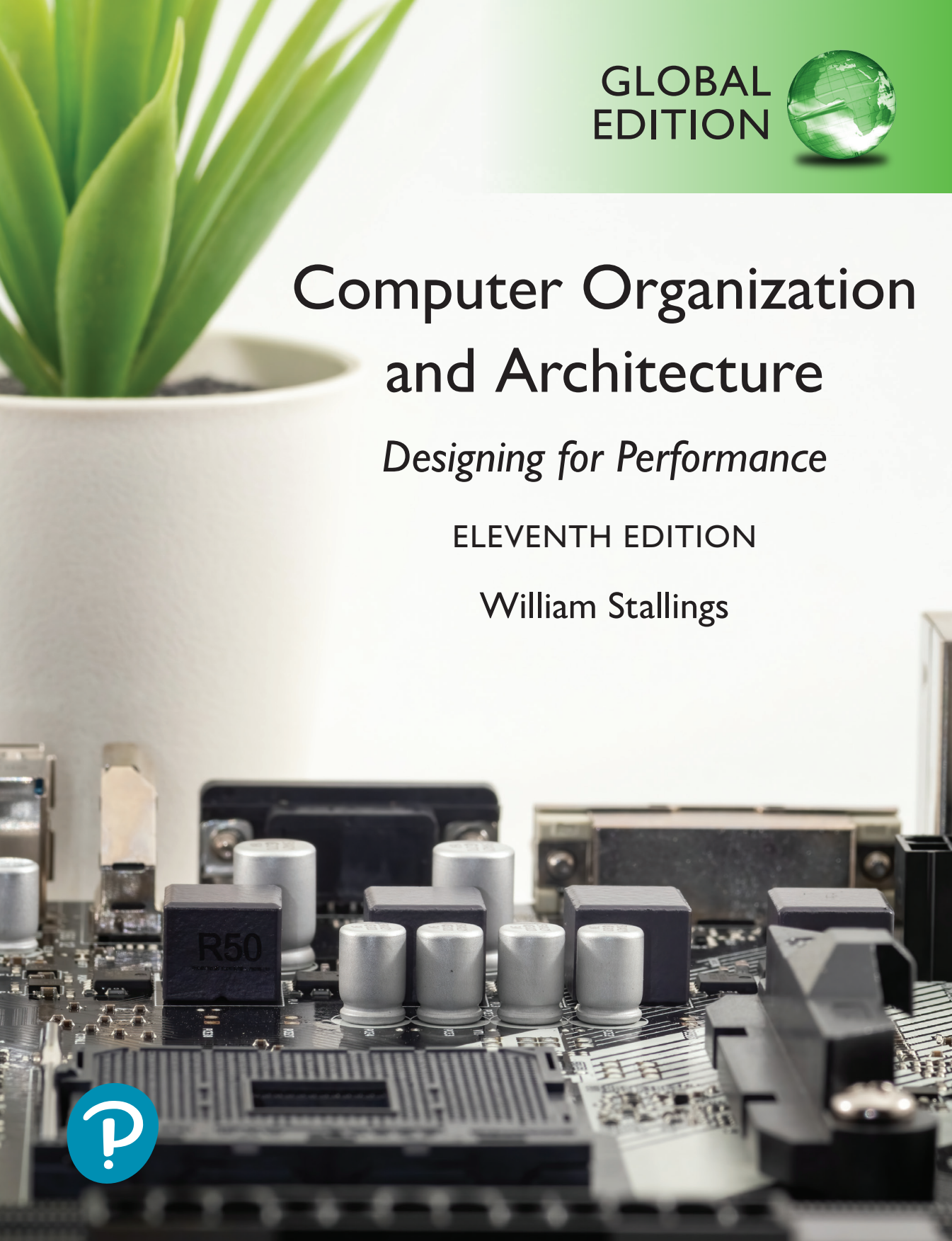
GLOBAL EDITION

Computer Organization and Architecture

Designing for Performance

ELEVENTH EDITION

William Stallings



Digital Resources for Students

Your eBook provides 12-month access to digital resources on the Companion Website. Refer to the preface in the textbook for a detailed list of resources.

Follow the instructions below to register for the Companion Website for William Stallings' *Computer Organization and Architecture*, Eleventh Edition, Global Edition.

1. Go to www.pearsonglobaleditions.com
2. Enter the title of your textbook or browse by author name.
3. Click Companion Website.
4. Click Register and follow the on-screen instructions to create a login name and password.

ISSSTY-FLAIL-EMERY-DUVET-BLUNT-RISES

Use a coin to scratch off the coating and reveal your access code.
Do not use a sharp knife or other sharp object as it may damage the code.

Use the login name and password you created during registration to start using the online resources that accompany your textbook.

IMPORTANT:

This access code can only be used once. This subscription is valid for 12 months upon activation and is not transferable.

For technical support, go to https://support.pearson.com/getsupport

Computer Organization and Architecture

Designing for Performance

Eleventh Edition

This page is intentionally left blank

Computer Organization and Architecture

Designing for Performance

Eleventh Edition

Global Edition

William Stallings

330 Hudson Street, New York, NY 10013

Cover Art: *zairiazmal/Shutterstock*

Pearson Education Limited
KAO Two
KAO Park
Hockham Way
Harlow
CM17 9SR
United Kingdom

and Associated Companies throughout the world

Visit us on the World Wide Web at: www.pearsonglobaleditions.com

Please contact https://support.pearson.com/getsupport/s/contactsupport with any queries on this content.

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

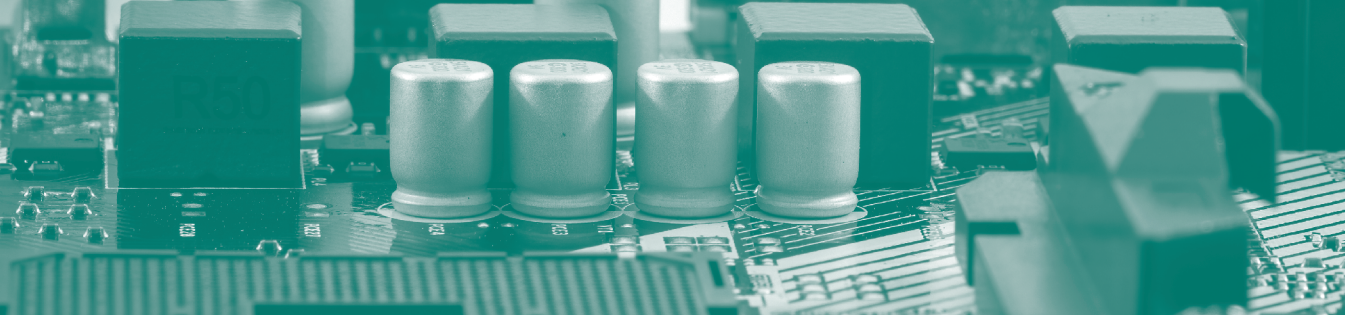
ISBN 10: 1-292-42010-3 (print)
ISBN 13: 978-1-292-42010-3 (print)
ISBN 13: 978-1-292-42008-0 (uPDF eBook)

To Tricia
my loving wife, the kindest
and gentlest person

This page is intentionally left blank

CONTENTS

This page is intentionally left blank



PREFACE

WHAT'S NEW IN THE ELEVENTH EDITION

Since the tenth edition of this book was published, the field has seen continued innovations and improvements. In this new edition, I try to capture these changes while maintaining a broad and comprehensive coverage of the entire field. To begin this process of revision, the tenth edition of this book was extensively reviewed by a number of professors who teach the subject and by professionals working in the field. The result is that, in many places, the narrative has been clarified and tightened, and illustrations have been improved.

Beyond these refinements to improve pedagogy and user-friendliness, there have been substantive changes throughout the book. Roughly the same chapter organization has been retained, but much of the material has been revised and new material has been added. The most noteworthy changes are as follows:

- **Multichip Modules:** A new discussion of MCMs, which are now widely used, has been added to Chapter 1.
- **SPEC benchmarks:** The treatment of SPEC in Chapter 2 has been updated to cover the new SPEC CPU2017 benchmark suite.
- **Memory hierarchy:** A new chapter on memory hierarchy expands on material that was in the cache memory chapter, plus adds new material. The new Chapter 4 includes:
  - —Updated and expanded coverage of the principle of locality
  - —Updated and expanded coverage of the memory hierarchy
  - —A new treatment of performance modeling of data access in a memory hierarchy
- **Cache memory:** The cache memory chapter has been updated and revised. Chapter 5 now includes:
  - —Revised and expanded treatment of logical cache organization, including new figures, to improve clarity
  - —New coverage of content-addressable memory
  - —New coverage of write allocate and no write allocate policies
  - —A new section on cache performance modeling.
- **Embedded DRAM:** Chapter 6 on internal memory now includes a section on the increasingly popular eDRAM.

- **Advanced Format 4k sector hard drives:** Chapter 7 on external memory now includes discussion of the now widely used 4k sector hard drive format.
- **Boolean algebra:** The discussion on Boolean algebra in Chapter 12 has been expanded with new text, figures, and tables, to enhance understanding.
- **Assembly language:** The treatment of assembly language has been expanded to a full chapter, with more detail and more examples.
- **Pipeline organization:** The discussion on pipeline organization has been substantially expanded with new text and figures. The material is in new sections in Chapters 16 (Processor Structure and Function), 17 (RISC), and 18 (Superscalar).
- **Cache coherence:** The discussion of the MESI cache coherence protocol in Chapter 20 has been expanded with new text and figures.

SUPPORT OF ACM/IEEE COMPUTER SCIENCE AND COMPUTER ENGINEERING CURRICULA

The book is intended for both an academic and a professional audience. As a textbook, it is intended as a one- or two-semester undergraduate course for computer science, computer engineering, and electrical engineering majors. This edition supports recommendations of the ACM/IEEE Computer Science Curricula 2013 (CS2013). CS2013 divides all course work into three categories: Core-Tier 1 (all topics should be included in the curriculum); Core-Tier-2 (all or almost all topics should be included); and Elective (desirable to provide breadth and depth). In the Architecture and Organization (AR) area, CS2013 includes five Tier-2 topics and three Elective topics, each of which has a number of subtopics. This text covers all eight topics listed by CS2013. Table P.1 shows the support for the AR Knowledge Area provided in this textbook. This book also supports the ACM/IEEE Computer Engineering Curricula 2016 (CE2016). CE2016 defines a necessary body of knowledge for undergraduate computer engineering, divided into twelve knowledge areas. One of these areas is Computer Architecture and Organization (CE-CAO), consisting of ten core knowledge areas. This text covers all of the CE-CAO knowledge areas listed in CE2016. Table P.2 shows the coverage.

Table P.1 Coverage of CS2013 Architecture and Organization (AR) Knowledge Area

IAS Knowledge Units	Topics	Textbook Coverage
Digital Logic and Digital Systems (Tier 2)	• Overview and history of computer architecture • Combinational vs. sequential logic/Field programmable gate arrays as a fundamental combinational sequential logic building block • Multiple representations/layers of interpretation (hardware is just another layer) • Physical constraints (gate delays, fan-in, fan-out, energy/power)	—Chapter 1 —Chapter 12
Machine Level Representation of Data (Tier 2)	• Bits, bytes, and words • Numeric data representation and number bases • Fixed- and floating-point systems • Signed and twos-complement representations • Representation of non-numeric data (character codes, graphical data)	—Chapter 10 —Chapter 11

IAS Knowledge Units	Topics	Textbook Coverage
Assembly Level Machine Organization (Tier 2)	• Basic organization of the von Neumann machine • Control unit; instruction fetch, decode, and execution • Instruction sets and types (data manipulation, control, I/O) • Assembly/machine language programming • Instruction formats • Addressing modes • Subroutine call and return mechanisms (cross-reference PL/Language Translation and Execution) • I/O and interrupts • Shared memory multiprocessors/multicore organization • Introduction to SIMD vs. MIMD and the Flynn Taxonomy	—Chapter 1 —Chapter 8 —Chapter 13 —Chapter 14 —Chapter 15 —Chapter 19 —Chapter 20 —Chapter 21
Memory System Organization and Architecture (Tier 2)	• Storage systems and their technology • Memory hierarchy: temporal and spatial locality • Main memory organization and operations • Latency, cycle time, bandwidth, and interleaving • Cache memories (address mapping, block size, replacement and store policy) • Multiprocessor cache consistency/Using the memory system for inter-core synchronization/atomic memory operations • Virtual memory (page table, TLB) • Fault handling and reliability	—Chapter 4 —Chapter 5 —Chapter 6 —Chapter 7 —Chapter 9 —Chapter 20
Interfacing and Communication (Tier 2)	• I/O fundamentals: handshaking, buffering, programmed I/O, interrupt-driven I/O • Interrupt structures: vectored and prioritized, interrupt acknowledgment • External storage, physical organization, and drives • Buses: bus protocols, arbitration, direct-memory access (DMA) • RAID architectures	—Chapter 3 —Chapter 7 —Chapter 8
Functional Organization (Elective)	• Implementation of simple datapaths, including instruction pipelining, hazard detection, and resolution • Control unit: hardwired realization vs. microprogrammed realization • Instruction pipelining • Introduction to instruction-level parallelism (ILP)	—Chapter 16 —Chapter 17 —Chapter 18 —Chapter 19
Multiprocessing and Alternative Architectures (Elective)	• Example SIMD and MIMD instruction sets and architectures • Interconnection networks • Shared multiprocessor memory systems and memory consistency • Multiprocessor cache coherence	—Chapter 20 —Chapter 21
Performance Enhancements (Elective)	• Superscalar architecture • Branch prediction, Speculative execution, Out-of-order execution • Prefetching • Vector processors and GPUs • Hardware support for multithreading • Scalability	—Chapter 17 —Chapter 18 —Chapter 20

Table P.2 Coverage of CE2016 Computer Architecture and Organization (AR) Knowledge Area

Knowledge Unit	Textbook Coverage
History and overview	Chapter 1—Basic Concepts and Computer Evolution
Relevant tools, standards and/or engineering constraints	Chapter 3—A Top-Level View of Computer Function and Interconnection
Instruction set architecture	Chapter 13—Instruction Sets: Characteristics and Functions Chapter 14—Instruction Sets: Addressing Modes and Formats Chapter 15—Assembly Language and Related Topics
Measuring performance	Chapter 2—Performance Concepts
Computer arithmetic	Chapter 10—Number Systems Chapter 11—Computer Arithmetic
Processor organization	Chapter 16—Processor Structure and Function Chapter 17—Reduced Instruction Set Computers (RISCs) Chapter 18—Instruction-Level Parallelism and Superscalar Processors Chapter 19—Control Unit Operation and Microprogrammed Control
Memory system organization and architectures	Chapter 4—The Memory Hierarchy: Locality and Performance Chapter 5—Cache Memory Chapter 6—Internal Memory Technology Chapter 7—External Memory
Input/Output interfacing and communication	Chapter 8—Input/Output
Peripheral subsystems	Chapter 3—A Top-Level View of Computer Function and Interconnection Chapter 8—Input/Output
Multi/Many-core architectures	Chapter 21—Multicore Computers
Distributed system architectures	Chapter 20—Parallel Processing

OBJECTIVES

This book is about the structure and function of computers. Its purpose is to present, as clearly and completely as possible, the nature and characteristics of modern-day computer systems.

This task is challenging for several reasons. First, there is a tremendous variety of products that can rightly claim the name of computer, from single-chip microprocessors costing a few dollars to supercomputers costing tens of millions of dollars. Variety is exhibited not only in cost but also in size, performance, and application. Second, the rapid pace of change that has always characterized computer technology continues with no letup. These changes cover all aspects of computer technology, from the underlying integrated circuit technology used to construct computer components to the increasing use of parallel organization concepts in combining those components.

In spite of the variety and pace of change in the computer field, certain fundamental concepts apply consistently throughout. The application of these concepts depends on the current state of the technology and the price/performance objectives of the designer.

The intent of this book is to provide a thorough discussion of the fundamentals of computer organization and architecture and to relate these to contemporary design issues.

The subtitle suggests the theme and the approach taken in this book. It has always been important to design computer systems to achieve high performance, but never has this requirement been stronger or more difficult to satisfy than today. All of the basic performance characteristics of computer systems, including processor speed, memory speed, memory capacity, and interconnection data rates, are increasing rapidly. Moreover, they are increasing at different rates. This makes it difficult to design a balanced system that maximizes the performance and utilization of all elements. Thus, computer design increasingly becomes a game of changing the structure or function in one area to compensate for a performance mismatch in another area. We will see this game played out in numerous design decisions throughout the book.

A computer system, like any system, consists of an interrelated set of components. The system is best characterized in terms of structure—the way in which components are interconnected, and function—the operation of the individual components. Furthermore, a computer's organization is hierarchical. Each major component can be further described by decomposing it into its major subcomponents and describing their structure and function. For clarity and ease of understanding, this hierarchical organization is described in this book from the top down:

- **Computer system:** Major components are processor, memory, I/O.
- **Processor:** Major components are control unit, registers, ALU, and instruction execution unit.
- **Control unit:** Provides control signals for the operation and coordination of all processor components. Traditionally, a microprogramming implementation has been used, in which major components are control memory, microinstruction sequencing logic, and registers. More recently, microprogramming has been less prominent but remains an important implementation technique.

The objective is to present the material in a fashion that keeps new material in a clear context. This should minimize the chance that the reader will get lost and should provide better motivation than a bottom-up approach.

Throughout the discussion, aspects of the system are viewed from the points of view of both architecture (those attributes of a system visible to a machine language programmer) and organization (the operational units and their interconnections that realize the architecture).

EXAMPLE SYSTEMS

This text is intended to acquaint the reader with the design principles and implementation issues of contemporary operating systems. Accordingly, a purely conceptual or theoretical treatment would be inadequate. To illustrate the concepts and to tie them to real-world design choices that must be made, two processor families have been chosen as running examples:

- **Intel x86 architecture:** The x86 architecture is the most widely used for nonembedded computer systems. The x86 is essentially a complex instruction set computer (CISC)

with some RISC features. Recent members of the x86 family make use of superscalar and multicore design principles. The evolution of features in the x86 architecture provides a unique case-study of the evolution of most of the design principles in computer architecture.

- **ARM:** The ARM architecture is arguably the most widely used embedded processor, used in cell phones, iPods, remote sensor equipment, and many other devices. The ARM is essentially a reduced instruction set computer (RISC). Recent members of the ARM family make use of superscalar and multicore design principles.

Many, but by no means all, of the examples in this book are drawn from these two computer families. Numerous other systems, both contemporary and historical, provide examples of important computer architecture design features.

PLAN OF THE TEXT

The book is organized into six parts:

- Introduction
- The computer system
- Arithmetic and logic
- Instruction sets and assembly language
- The central processing unit
- Parallel organization, including multicore

The book includes a number of pedagogic features, including the use of interactive simulations and numerous figures and tables to clarify the discussion. Each chapter includes a list of key words, review questions, and homework problems. The book also includes an extensive glossary, a list of frequently used acronyms, and a bibliography.

INSTRUCTOR SUPPORT MATERIALS

Support materials for instructors are available at the **Instructor Resource Center (IRC)** for this textbook, which can be reached through the publisher's Web site www.pearsonglobaleditions.com. The IRC provides the following materials:

- **Projects manual:** Project resources including documents and portable software, plus suggested project assignments for all of the project categories listed subsequently in this Preface.
- **Solutions manual:** Solutions to end-of-chapter Review Questions and Problems.
- **PowerPoint slides:** A set of slides covering all chapters, suitable for use in lecturing.
- **PDF files:** Copies of all figures and tables from the book.
- **Test bank:** A chapter-by-chapter set of questions.

- **Sample syllabuses:** The text contains more material than can be conveniently covered in one semester. Accordingly, instructors are provided with several sample syllabuses that guide the use of the text within limited time. These samples are based on real-world experience by professors with the first edition.

STUDENT RESOURCES

For this new edition, a tremendous amount of original supporting material for students has been made available online. The **Companion Web Site**, at www.pearsonglobaleditions.com, includes a list of relevant links organized by chapter. To aid the student in understanding the material, a separate set of homework problems with solutions are available at this site. Students can enhance their understanding of the material by working out the solutions to these problems and then checking their answers. The site also includes a number of documents and papers referenced throughout the text. Students can subscribe to the Companion Web Site by using the access codes provided on the inside front cover.

PROJECTS AND OTHER STUDENT EXERCISES

For many instructors, an important component of a computer organization and architecture course is a project or set of projects by which the student gets hands-on experience to reinforce concepts from the text. This book provides an unparalleled degree of support for including a projects component in the course. The instructor's support materials available through the IRC not only includes guidance on how to assign and structure the projects but also includes a set of user's manuals for various project types plus specific assignments, all written especially for this book. Instructors can assign work in the following areas:

- **Interactive simulation assignments:** Described subsequently.
- **Research projects:** A series of research assignments that instruct the student to research a particular topic on the Internet and write a report.
- **Simulation projects:** The IRC provides support for the use of the two simulation packages: SimpleScalar can be used to explore computer organization and architecture design issues. SMPCache provides a powerful educational tool for examining cache design issues for symmetric multiprocessors.
- **Assembly language projects:** A simplified assembly language, CodeBlue, is used and assignments based on the popular Core Wars concept are provided.
- **Reading/report assignments:** A list of papers in the literature, one or more for each chapter, that can be assigned for the student to read and then write a short report.
- **Writing assignments:** A list of writing assignments to facilitate learning the material.
- **Test bank:** Includes T/F, multiple choice, and fill-in-the-blank questions and answers.

This diverse set of projects and other student exercises enables the instructor to use the book as one component in a rich and varied learning experience and to tailor a course plan to meet the specific needs of the instructor and students.

INTERACTIVE SIMULATIONS

An important feature in this edition is the incorporation of interactive simulations. These simulations provide a powerful tool for understanding the complex design features of a modern computer system. A total of 20 interactive simulations are used to illustrate key functions and algorithms in computer organization and architecture design. At the relevant point in the book, an icon indicates that a relevant interactive simulation is available online for student use. Because the animations enable the user to set initial conditions, they can serve as the basis for student assignments. The instructor's supplement includes a set of assignments, one for each of the animations. Each assignment includes several specific problems that can be assigned to students.

ACKNOWLEDGMENTS

This new edition has benefited from review by a number of people, who gave generously of their time and expertise. The following professors provided a review of the entire book: Nikhil Bhargava (Indian Institute of Management, Delhi), James Gil de Lamadrid (Bowie State University, Computer Science Department), Debra Calliss (Computer Science and Engineering, Arizona State University), Mohammed Anwaruddin (Wentworth Institute of Technology, Dept. of Computer Science), Roger Kieckhafer (Michigan Technological University, Electrical & Computer Engineering), Paul Fortier (University of Massachusetts Darthmouth, Electrical and Computer Engineering), Yan Zhang (Department of Computer Science and Engineering, University of South Florida), Patricia Roden (University of North Alabama, Computer Science and Information Systems), Sanjeev Baskiyar (Auburn University, Computer Science and Software Engineering), and (Jayson Rock, University of Wisconsin-Milwaukee, Computer Science). I would especially like to thank Professor Roger Kieckhafer for permission to make use of some of the figures and performance models from his course lecture notes.

Thanks also to the many people who provided detailed technical reviews of one or more chapters: Rekai Gonzalez Alberquilla, Allen Baum, Jalil Boukhobza, Dmitry Bufistov, Humberto Calderón, Jesus Carretero, Ashkan Eghbal, Peter Glaskowsky, Ram Huggahalli, Chris Jesshope, Athanasios Kakarountas, Isil Oz, Mitchell Poplingher, Roger Shepherd, Jigar Savla, Karl Stevens, Siri Uppalapati, Dr. Sriram Vajapeyam, Kugan Vivekanandarajah, Pooria M. Yaghini, and Peter Zeno,

Professor Cindy Norris of Appalachian State University, Professor Bin Mu of the University of New Brunswick, and Professor Kenrick Mock of the University of Alaska kindly supplied homework problems.

Aswin Sreedhar of the University of Massachusetts developed the interactive simulation assignments.

Professor Miguel Angel Vega Rodriguez, Professor Dr. Juan Manuel Sánchez Pérez, and Professor Dr. Juan Antonio Gómez Pulido, all of University of Extremadura, Spain, prepared the SMPCache problems in the instructor's manual and authored the SMPCache User's Guide.

Todd Bezenek of the University of Wisconsin and James Stine of Lehigh University prepared the SimpleScalar problems in the instructor's manual, and Todd also authored the SimpleScalar User's Guide.

Finally, I would like to thank the many people responsible for the publication of the book, all of whom did their usual excellent job. This includes the staff at Pearson, particularly my editor Tracy Johnson, her assistant Meghan Jacoby, and project manager Bob Engelhardt. Thanks also to the marketing and sales staffs at Pearson, without whose efforts this book would not be in front of you.

ACKNOWLEDGMENTS FOR THE GLOBAL EDITION

Pearson would like to acknowledge and thank the following for their work on the Global Edition.

Contributor

Asral Bahari Jambek (Universiti Malaysia Perlis, Malaysia)
Carl Barton (Birkbeck University of London)
Sutep Tongngam (NIDA, Thailand)

Reviewers

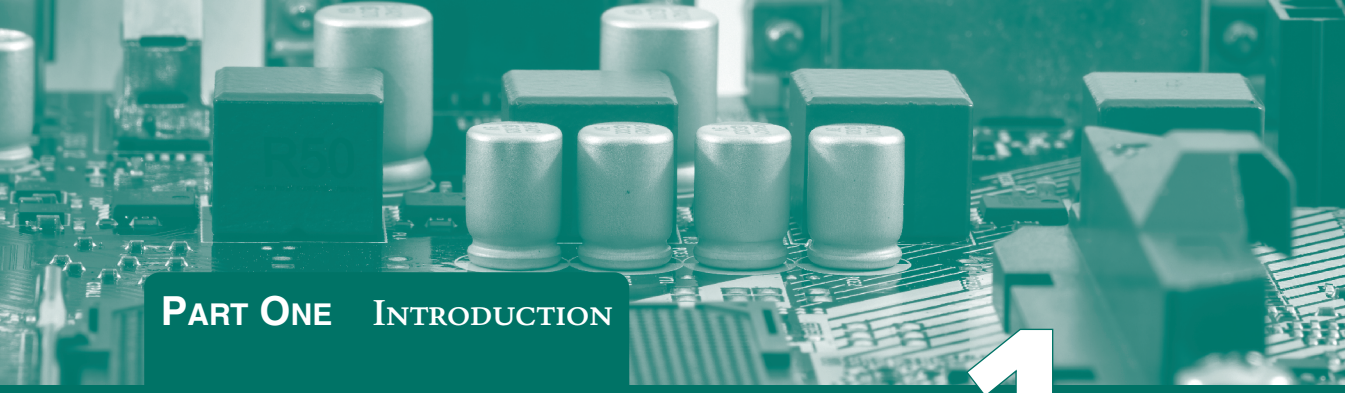
Rana Ejaz Ahmed (American University of Sharjah, UAE)
Ritesh Ajoodha (University of the Witwatersrand, Johannesburg)
Asral Bahari Jambek (Universiti Malaysia Perlis, Malaysia)
Patricia E.N. Lutu (University of Pretoria, South Africa)
Sezer Gören Uğurdağ (Yeditepe Üniversitesi, Turkey)

About the Author

Dr. William Stallings has authored 18 textbooks, and counting revised editions, over 70 books on computer security, computer networking, and computer architecture. In over 30 years in the field, he has been a technical contributor, technical manager, and an executive with several high-technology firms. Currently, he is an independent consultant whose clients have included computer and networking manufacturers and customers, software development firms, and leading-edge government research institutions. He has 13 times received the award for the best computer science textbook of the year from the Text and Academic Authors Association.

He created and maintains the Computer Science Student Resource Site at ComputerScienceStudent.com. This site provides documents and links on a variety of subjects of general interest to computer science students (and professionals). He is a member of the editorial board of *Cryptologia*, a scholarly journal devoted to all aspects of cryptology.

Dr. Stallings holds a PhD from MIT in computer science and a BS from Notre Dame in electrical engineering.



PART ONE INTRODUCTION

CHAPTER 1

BASIC CONCEPTS AND COMPUTER EVOLUTION

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Explain the general functions and structure of a digital computer.
- Present an overview of the evolution of computer technology from early digital computers to the latest microprocessors.
- Present an overview of the evolution of the x86 architecture.
- Define embedded systems and list some of the requirements and constraints that various embedded systems must meet.

1.1 ORGANIZATION AND ARCHITECTURE

In describing computers, a distinction is often made between *computer architecture* and *computer organization*. Although it is difficult to give precise definitions for these terms, a consensus exists about the general areas covered by each. For example, see [VRAN80], [SIEW82], and [BELL78a]; an interesting alternative view is presented in [REDD76].

Computer architecture refers to those attributes of a system visible to a programmer or, put another way, those attributes that have a direct impact on the logical execution of a program. A term that is often used interchangeably with computer architecture is **instruction set architecture (ISA)**. The ISA defines instruction formats, instruction opcodes, registers, instruction and data memory; the effect of executed instructions on the registers and memory; and an algorithm for controlling instruction execution. **Computer organization** refers to the operational units and their interconnections that realize the architectural specifications. Examples of architectural attributes include the instruction set, the number of bits used to represent various data types (e.g., numbers, characters), I/O mechanisms, and techniques for addressing memory. Organizational attributes include those hardware details transparent to the programmer, such as control signals; interfaces between the computer and peripherals; and the memory technology used.

For example, it is an architectural design issue whether a computer will have a multiply instruction. It is an organizational issue whether that instruction will be implemented by a special multiply unit or by a mechanism that makes repeated use of the add unit of the system. The organizational decision may be based on the anticipated frequency of use of the multiply instruction, the relative speed of the two approaches, and the cost and physical size of a special multiply unit.

Historically, and still today, the distinction between architecture and organization has been an important one. Many computer manufacturers offer a family of computer models, all with the same architecture but with differences in organization. Consequently, the different models in the family have different price and

performance characteristics. Furthermore, a particular architecture may span many years and encompass a number of different computer models, its organization changing with changing technology. A prominent example of both these phenomena is the IBM System/370 architecture. This architecture was first introduced in 1970 and included a number of models. The customer with modest requirements could buy a cheaper, slower model and, if demand increased, later upgrade to a more expensive, faster model without having to abandon software that had already been developed. Over the years, IBM has introduced many new models with improved technology to replace older models, offering the customer greater speed, lower cost, or both. These newer models retained the same architecture so that the customer's software investment was protected. Remarkably, the System/370 architecture, with a few enhancements, has survived to this day as the architecture of IBM's mainframe product line.

In a class of computers called microcomputers, the relationship between architecture and organization is very close. Changes in technology not only influence organization but also result in the introduction of more powerful and more complex architectures. Generally, there is less of a requirement for generation-to-generation compatibility for these smaller machines. Thus, there is more interplay between organizational and architectural design decisions. An intriguing example of this is the reduced instruction set computer (RISC), which we examine in Chapter 15.

This book examines both computer organization and computer architecture. The emphasis is perhaps more on the side of organization. However, because a computer organization must be designed to implement a particular architectural specification, a thorough treatment of organization requires a detailed examination of architecture as well.

1.2 STRUCTURE AND FUNCTION

A computer is a complex system; contemporary computers contain millions of elementary electronic components. How, then, can one clearly describe them? The key is to recognize the hierarchical nature of most complex systems, including the computer [SIMO96]. A hierarchical system is a set of interrelated subsystems; each subsystem may, in turn, contain lower level subsystems, until we reach some lowest level of elementary subsystem.

The hierarchical nature of complex systems is essential to both their design and their description. The designer need only deal with a particular level of the system at a time. At each level, the system consists of a set of components and their interrelationships. The behavior at each level depends only on a simplified, abstracted characterization of the system at the next lower level. At each level, the designer is concerned with structure and function:

- **Structure:** The way in which the components are interrelated.
- **Function:** The operation of each individual component as part of the structure.

In terms of description, we have two choices: starting at the bottom and building up to a complete description, or beginning with a top view and decomposing the system into its subparts. Evidence from a number of fields suggests that the top-down approach is the clearest and most effective [WEIN75].

The approach taken in this book follows from this viewpoint. The computer system will be described from the top down. We begin with the major components of a computer, describing their structure and function, and proceed to successively lower layers of the hierarchy. The remainder of this section provides a very brief overview of this plan of attack.

Function

Both the structure and functioning of a computer are, in essence, simple. In general terms, there are only four basic functions that a computer can perform:

- **Data processing:** Data may take a wide variety of forms, and the range of processing requirements is broad. However, we shall see that there are only a few fundamental methods or types of data processing.
- **Data storage:** Even if the computer is processing data on the fly (i.e., data come in and get processed, and the results go out immediately), the computer must temporarily store at least those pieces of data that are being worked on at any given moment. Thus, there is at least a short-term data storage function. Equally important, the computer performs a long-term data storage function. Files of data are stored on the computer for subsequent retrieval and update.
- **Data movement:** The computer's operating environment consists of devices that serve as either sources or destinations of data. When data are received from or delivered to a device that is directly connected to the computer, the process is known as *input–output* (*I/O*), and the device is referred to as a *peripheral*. When data are moved over longer distances, to or from a remote device, the process is known as *data communications*.
- **Control:** Within the computer, a control unit manages the computer's resources and orchestrates the performance of its functional parts in response to instructions.

The preceding discussion may seem absurdly generalized. It is certainly possible, even at a top level of computer structure, to differentiate a variety of functions, but to quote [SIEW82]:

> There is remarkably little shaping of computer structure to fit the function to be performed. At the root of this lies the general-purpose nature of computers, in which all the functional specialization occurs at the time of programming and not at the time of design.

Structure

We now look in a general way at the internal structure of a computer. We begin with a traditional computer with a single processor that employs a microprogrammed control unit, then examine a typical multicore structure.

SIMPLE SINGLE–PROCESSOR COMPUTER Figure 1.1 provides a hierarchical view of the internal structure of a traditional single-processor computer. There are four main structural components:

- **Central processing unit (CPU):** Controls the operation of the computer and performs its data processing functions; often simply referred to as **processor**.
- **Main memory:** Stores data.
- **I/O:** Moves data between the computer and its external environment.
- **System interconnection:** Some mechanism that provides for communication among CPU, main memory, and I/O. A common example of system

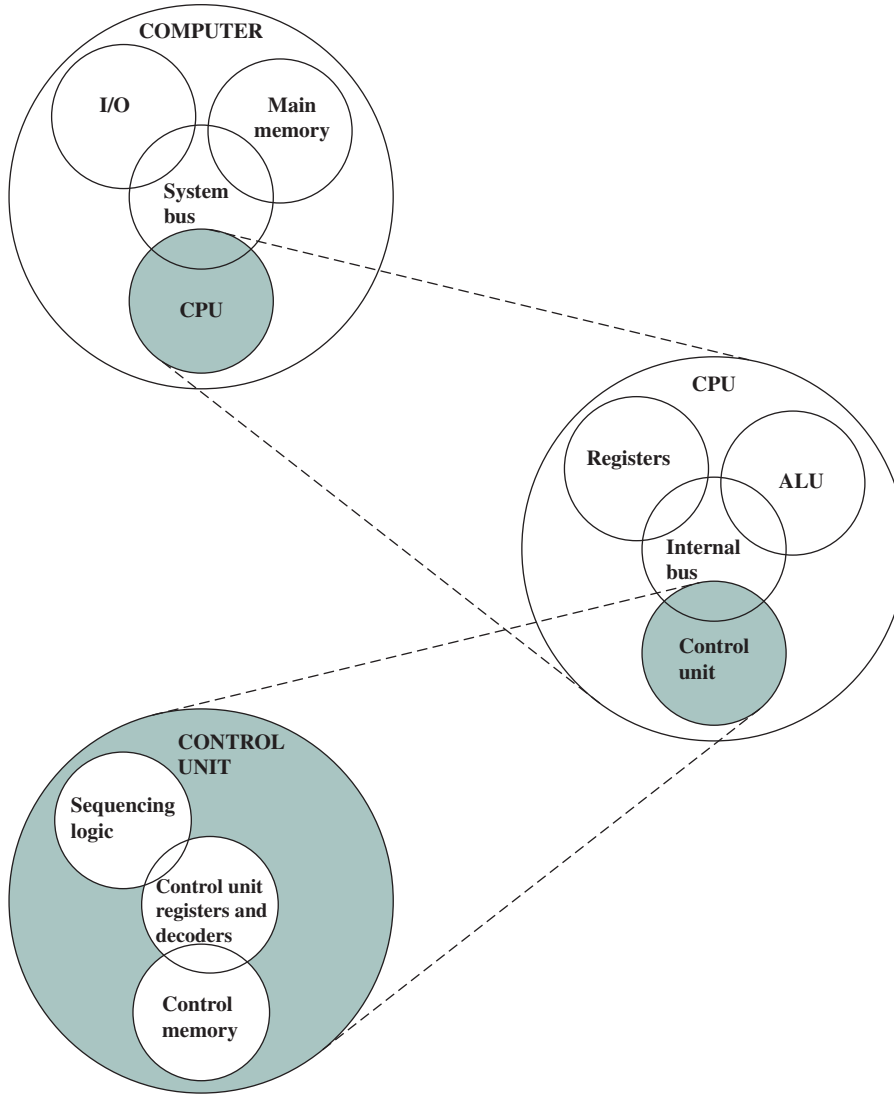


Figure 1.1 The Computer: Top-Level Structure

interconnection is by means of a **system bus**, consisting of a number of conducting wires to which all the other components attach.

There may be one or more of each of the aforementioned components. Traditionally, there has been just a single processor. In recent years, there has been increasing use of multiple processors in a single computer. Some design issues relating to multiple processors crop up and are discussed as the text proceeds; Part Five focuses on such computers.

Each of these components will be examined in some detail in Part Two. However, for our purposes, the most interesting and in some ways the most complex component is the CPU. Its major structural components are as follows:

- **Control unit:** Controls the operation of the CPU and hence the computer.
- **Arithmetic and logic unit (ALU):** Performs the computer's data processing functions.
- **Registers:** Provides storage internal to the CPU.
- **CPU interconnection:** Some mechanism that provides for communication among the control unit, ALU, and registers.

Part Three covers these components, where we will see that complexity is added by the use of parallel and pipelined organizational techniques. Finally, there are several approaches to the implementation of the control unit; one common approach is a *microprogrammed* implementation. In essence, a microprogrammed control unit operates by executing microinstructions that define the functionality of the control unit. With this approach, the structure of the control unit can be depicted, as in Figure 1.1. This structure is examined in Part Four.

MULTICORE COMPUTER STRUCTURE As was mentioned, contemporary computers generally have multiple processors. When these processors all reside on a single chip, the term *multicore computer* is used, and each processing unit (consisting of a control unit, ALU, registers, and perhaps cache) is called a *core*. To clarify the terminology, this text will use the following definitions.

- **Central processing unit (CPU):** That portion of a computer that fetches and executes instructions. It consists of an ALU, a control unit, and registers. In a system with a single processing unit, it is often simply referred to as a *processor*.
- **Core:** An individual processing unit on a processor chip. A core may be equivalent in functionality to a CPU on a single-CPU system. Other specialized processing units, such as one optimized for vector and matrix operations, are also referred to as cores.
- **Processor:** A physical piece of silicon containing one or more cores. The processor is the computer component that interprets and executes instructions. If a processor contains multiple cores, it is referred to as a **multicore processor**.

After about a decade of discussion, there is broad industry consensus on this usage.

Another prominent feature of contemporary computers is the use of multiple layers of memory, called *cache memory*, between the processor and main memory.

Chapter 4 is devoted to the topic of cache memory. For our purposes in this section, we simply note that a cache memory is smaller and faster than main memory and is used to speed up memory access, by placing in the cache data from main memory, that is likely to be used in the near future. A greater performance improvement may be obtained by using multiple levels of cache, with level 1 (L1) closest to the core and additional levels (L2, L3, and so on) progressively farther from the core. In this scheme, level n is smaller and faster than level $n + 1$.

Figure 1.2 is a simplified view of the principal components of a typical multicore computer. Most computers, including embedded computers in smartphones and tablets, plus personal computers, laptops, and workstations, are housed on a motherboard. Before describing this arrangement, we need to define some terms. A **printed circuit board (PCB)** is a rigid, flat board that holds and interconnects chips and other electronic components. The board is made of layers, typically two to ten, that interconnect components via copper pathways that are etched into

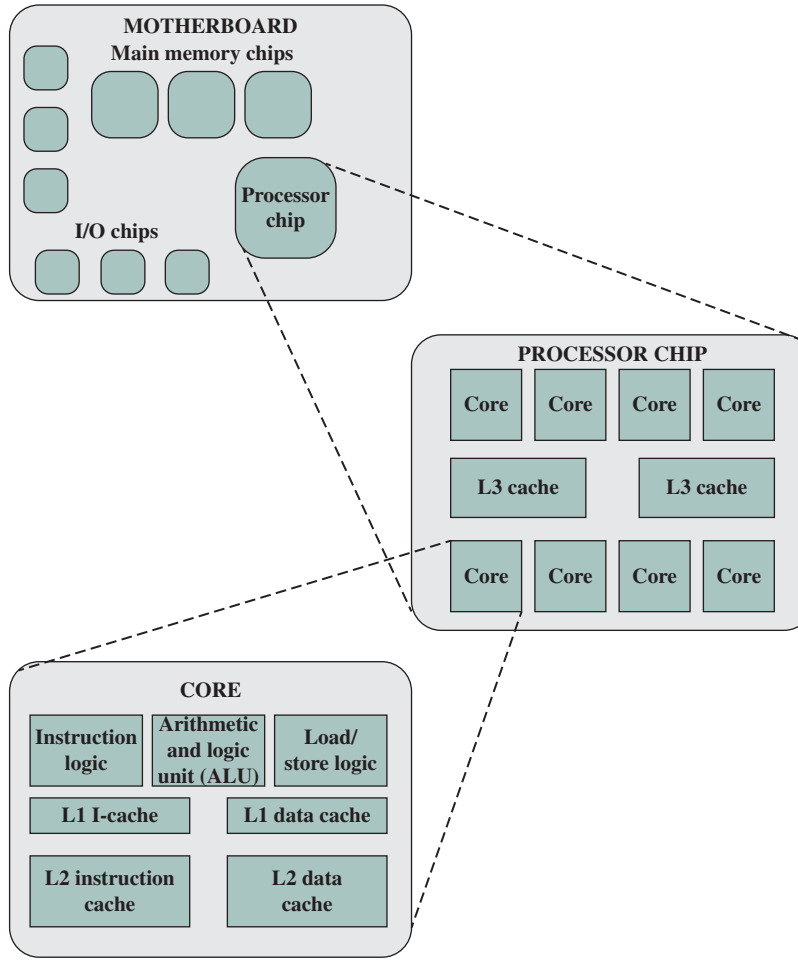


Figure 1.2 Simplified View of Major Elements of a Multicore Computer

the board. The main printed circuit board in a computer is called a system board or **motherboard**, while smaller ones that plug into the slots in the main board are called expansion boards.

The most prominent elements on the motherboard are the chips. A **chip** is a single piece of semiconducting material, typically silicon, upon which electronic circuits and logic gates are fabricated. The resulting product is referred to as an **integrated circuit**.

The motherboard contains a slot or socket for the processor chip, which typically contains multiple individual cores, in what is known as a *multicore processor*. There are also slots for memory chips, I/O controller chips, and other key computer components. For desktop computers, expansion slots enable the inclusion of more components on expansion boards. Thus, a modern motherboard connects only a few individual chip components, with each chip containing from a few thousand up to hundreds of millions of transistors.

Figure 1.2 shows a processor chip that contains eight cores and an L3 cache. Not shown is the logic required to control operations between the cores and the cache and between the cores and the external circuitry on the motherboard. The figure indicates that the L3 cache occupies two distinct portions of the chip surface. However, typically, all cores have access to the entire L3 cache via the aforementioned control circuits. The processor chip shown in Figure 1.2 does not represent any specific product, but provides a general idea of how such chips are laid out.

Next, we zoom in on the structure of a single core, which occupies a portion of the processor chip. In general terms, the functional elements of a core are:

- **Instruction logic:** This includes the tasks involved in fetching instructions, and decoding each instruction to determine the instruction operation and the memory locations of any operands.
- **Arithmetic and logic unit (ALU):** Performs the operation specified by an instruction.
- **Load/store logic:** Manages the transfer of data to and from main memory via cache.

The core also contains an L1 cache, split between an instruction cache (I-cache) that is used for the transfer of instructions to and from main memory, and an L1 data cache, for the transfer of operands and results. Typically, today's processor chips also include an L2 cache as part of the core. In many cases, this cache is also split between instruction and data caches, although a combined, single L2 cache is also used.

Keep in mind that this representation of the layout of the core is only intended to give a general idea of internal core structure. In a given product, the functional elements may not be laid out as the three distinct elements shown in Figure 1.2, especially if some or all of these functions are implemented as part of a microprogrammed control unit.

EXAMPLES It will be instructive to study some real-world examples that illustrate the hierarchical structure of computers. Let us take as an example the motherboard for a computer built around two Intel Quad-Core Xeon processor chips.

Activity: The most important elements of this motherboard, in addition to the processor sockets are listed below. Research online and find motherboards that match this description.

- PCI-Express slots for a high-end display adapter and for additional peripherals (Section 3.6 describes PCIe).
- Ethernet controller and Ethernet ports for network connections.
- USB sockets for peripheral devices.
- Serial ATA (SATA) sockets for connection to disk memory (Section 7.7 discusses Ethernet, USB, and SATA).
- Interfaces for DDR (double data rate) main memory chips (Section 5.3 discusses DDR).
- Intel 3420 chipset is an I/O controller for direct memory access operations between peripheral devices and main memory (Section 7.5 discusses DDR).

Following our top-down strategy, as illustrated in Figures 1.1 and 1.2, we can now zoom in and look at the internal structure of a processor chip, referred to as a processor unit (PU). For variety, we look at an IBM chip instead of the Intel processor chip. Figure 1.3 is a to-scale layout of the processor chip for the IBM z13 mainframe computer [LASC16]. This chip has 3.99 billion transistors. The superimposed labels indicate how the silicon surface area of the chip is allocated. We see that this chip has eight cores, or processors. In addition, a substantial portion of the chip is devoted to the L3 cache, which is shared by all eight cores. The L3 control logic controls traffic between the L3 cache and the cores and between the L3 cache and the external environment. Additionally, there is storage control (SC) logic between the cores and the L3 cache. The memory controller (MC) function controls access to memory external to the chip. The GX I/O bus controls the interface to the channel adapters accessing the I/O.

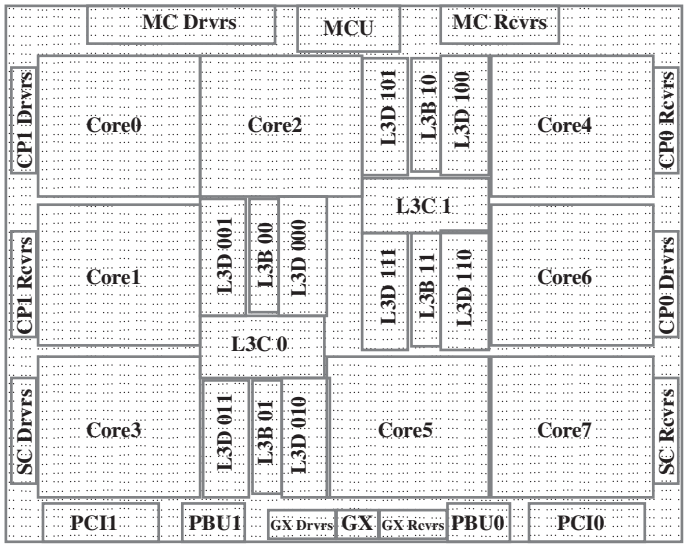


Figure 1.3 IBM z13 Processor Unit (PU) Chip Diagram

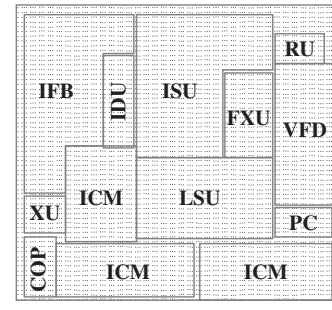


Figure 1.4 IBM z13 Core Layout

Going down one level deeper, we examine the internal structure of a single core, as shown in the photograph of Figure 1.4. The core implements the z13 instruction set architecture, referred to as the z/Architecture. Keep in mind that this is a portion of the silicon surface area making up a single-processor chip. The main subareas within this core area are the following:

- **ISU (instruction sequence unit):** Determines the sequence in which instructions are executed in what is referred to as a superscalar architecture. It enables the out-of-order (OOO) pipeline. It tracks register names, OOO instruction dependency, and handling of instruction resource dispatch. These concepts are discussed in Chapter 16.
- **IFB (instruction fetch and branch) and ICM (instruction cache and merge)** These two subunits contain the 128-kB[1] instruction cache, branch prediction logic, instruction fetching controls, and buffers. The relative size of these subunits is the result of the elaborate branch prediction design.
- **IDU (instruction decode unit):** The IDU is fed from the IFU buffers, and is responsible for the parsing and decoding of all z/Architecture operation codes.
- **LSU (load-store unit):** The LSU contains the 96-kB L1 data cache, and manages data traffic between the L2 data cache and the functional execution units. It is responsible for handling all types of operand accesses of all lengths, modes, and formats as defined in the z/Architecture.
- **XU (translation unit):** This unit translates logical addresses from instructions into physical addresses in main memory. The XU also contains a translation

[1]kB = kilobyte = 1048 bytes. Numerical prefixes are explained in a document under the "Other Useful" tab at ComputerScienceStudent.com.

lookaside buffer (TLB) used to speed up memory access. TLBs are discussed in Chapter 8.

- **PC (core pervasive unit):** Used for instrumentation and error collection.
- **FXU (fixed-point unit):** The FXU executes fixed-point arithmetic operations.
- **VFU (vector and floating-point units):** The binary floating-unit part handles all binary and hexadecimal floating-point operations, as well as fixed-point multiplication operations. The decimal floating-unit part handles both fixed-point and floating-point operations on numbers that are stored as decimal digits. The vector execution part handles vector operations.
- **RU (recovery unit):** The RU keeps a copy of the complete state of the system that includes all registers, collects hardware fault signals, and manages the hardware recovery actions.
- **COP (dedicated co-processor):** The COP is responsible for data compression and encryption functions for each core.
- **L2D:** A 2-MB L2 data cache for all memory traffic other than instructions.
- **L2I:** A 2-MB L2 instruction cache.

As we progress through the book, the concepts introduced in this section will become clearer.

1.3 THE IAS COMPUTER

The first generation of computers used vacuum tubes for digital logic elements and memory. A number of research and then commercial computers were built using vacuum tubes. For our purposes, it will be instructive to examine perhaps the most famous first-generation computer, known as the IAS computer. This example illustrates many of the fundamental concepts found in all computer systems.

A fundamental design approach first implemented in the IAS computer is known as the *stored-program concept*. This idea is usually attributed to the mathematician John von Neumann. Alan Turing developed the idea at about the same time. The first publication of the idea was in a 1945 proposal by von Neumann for a new computer, the EDVAC (Electronic Discrete Variable Computer).[2]

In 1946, von Neumann and his colleagues began the design of a new stored-program computer, referred to as the IAS computer, at the Princeton Institute for Advanced Studies. The IAS computer, although not completed until 1952, is the prototype of all subsequent general-purpose computers.[3]

Figure 1.5 shows the structure of the IAS computer (compare with Figure 1.1). It consists of

- A **main memory**, which stores both data and instructions[4]
- An **arithmetic and logic unit (ALU)** capable of operating on binary data

[2]The 1945 report on EDVAC is available at box.com/COA11e.

[3]A 1954 report [GOLD54] describes the implemented IAS machine and lists the final instruction set. It is available at box.com/COA11e.

[4]In this book, unless otherwise noted, the term *instruction* refers to a machine instruction that is directly interpreted and executed by the processor, in contrast to a statement in a high-level language, such as Ada or C++, which must first be compiled into a series of machine instructions before being executed.

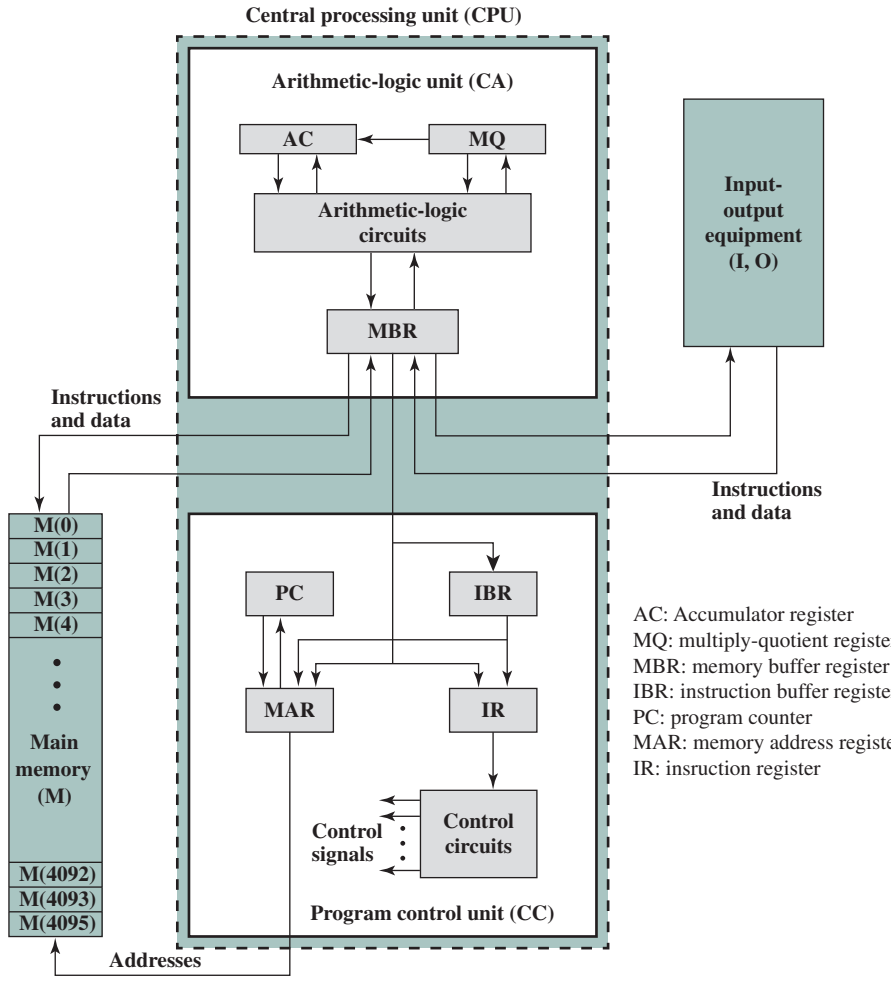


Figure 1.5 IAS Structure

- A **control unit**, which interprets the instructions in memory and causes them to be executed
- **Input–output (I/O)** equipment operated by the control unit

This structure was outlined in von Neumann's earlier proposal, which is worth quoting in part at this point [VONN45]:

> 2.2 **First:** Since the device is primarily a computer, it will have to perform the elementary operations of arithmetic most frequently. These are addition, subtraction, multiplication, and division. It is therefore reasonable that it should contain specialized organs for just these operations.
>
> It must be observed, however, that while this principle as such is probably sound, the specific way in which it is realized requires close scrutiny. At any rate a *central arithmetical* part of the device will probably have to exist, and this constitutes *the first specific part: CA*.

2.3 **Second:** The logical control of the device, that is, the proper sequencing of its operations, can be most efficiently carried out by a central control organ. If the device is to be *elastic*, that is, as nearly as possible *all purpose*, then a distinction must be made between the specific instructions given for and defining a particular problem, and the general control organs that see to it that these instructions—no matter what they are—are carried out. The former must be stored in some way; the latter are represented by definite operating parts of the device. By the *central control* we mean this latter function only, and the organs that perform it form *the second specific part: CC*.

2.4 **Third:** Any device that is to carry out long and complicated sequences of operations (specifically of calculations) must have a considerable memory . . .

The instructions which govern a complicated problem may constitute considerable material, particularly so if the code is circumstantial (which it is in most arrangements). This material must be remembered.

At any rate, the total *memory* constitutes *the third specific part of the device: M*.

2.6 The three specific parts CA, CC (together C), and M correspond to the *associative* neurons in the human nervous system. It remains to discuss the equivalents of the *sensory* or *afferent* and the *motor* or *efferent* neurons. These are the *input* and *output* organs of the device.

The device must be endowed with the ability to maintain input and output (sensory and motor) contact with some specific medium of this type. The medium will be called the *outside recording medium of the device: R*.

2.7 **Fourth:** The device must have organs to transfer information from R into its specific parts C and M. These organs form its *input*, the *fourth specific part: I*. It will be seen that it is best to make all transfers from R (by I) into M and never directly from C.

2.8 **Fifth:** The device must have organs to transfer from its specific parts C and M into R. These organs form its *output*, *the fifth specific part: O*. It will be seen that it is again best to make all transfers from M (by O) into R, and never directly from C.

With rare exceptions, all of today's computers have this same general structure and function and are thus referred to as *von Neumann machines*. Thus, it is worthwhile at this point to describe briefly the operation of the IAS computer [BURK46, GOLD54]. Following [HAYE98], the terminology and notation of von Neumann are changed in the following to conform more closely to modern usage; the examples accompanying this discussion are based on that latter text.

The memory of the IAS consists of 4,096 storage locations, called *words*, of 40 binary digits (bits) each.[5] Both data and instructions are stored there. Numbers

[5]There is no universal definition of the term *word*. In general, a word is an ordered set of bytes or bits that is the normal unit in which information may be stored, transmitted, or operated on within a given computer. Typically, if a processor has a fixed-length instruction set, then the instruction length equals the word length.

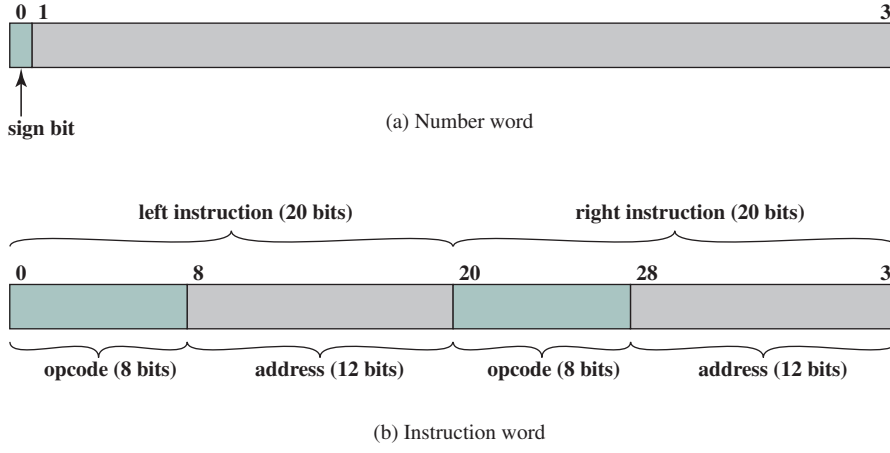


Figure 1.6 IAS Memory Formats

are represented in binary form, and each instruction is a binary code. Figure 1.6 illustrates these formats. Each number is represented by a sign bit and a 39-bit value. A word may alternatively contain two 20-bit instructions, with each instruction consisting of an 8-bit operation code (opcode) specifying the operation to be performed and a 12-bit address designating one of the words in memory (numbered from 0 to 999).

The control unit operates the IAS by fetching instructions from memory and executing them one at a time. We explain these operations with reference to Figure 1.7. This figure reveals that both the control unit and the ALU contain storage locations, called *registers*, defined as follows:

- **Memory buffer register (MBR):** Contains a word to be stored in memory or sent to the I/O unit, or is used to receive a word from memory or from the I/O unit.
- **Memory address register (MAR):** Specifies the address in memory of the word to be written from or read into the MBR.
- **Instruction register (IR):** Contains the 8-bit opcode instruction being executed.
- **Instruction buffer register (IBR):** Employed to hold temporarily the right-hand instruction from a word in memory.
- **Program counter (PC):** Contains the address of the next instruction pair to be fetched from memory.
- **Accumulator (AC) and multiplier quotient (MQ):** Employed to hold temporarily operands and results of ALU operations. For example, the result of multiplying two 40-bit numbers is an 80-bit number; the most significant 40 bits are stored in the AC and the least significant in the MQ.

The IAS operates by repetitively performing an *instruction cycle*, as shown in Figure 1.7. Each instruction cycle consists of two subcycles. During the *fetch cycle*, the opcode of the next instruction is loaded into the IR and the address portion is loaded into the MAR. This instruction may be taken from the IBR, or it can be

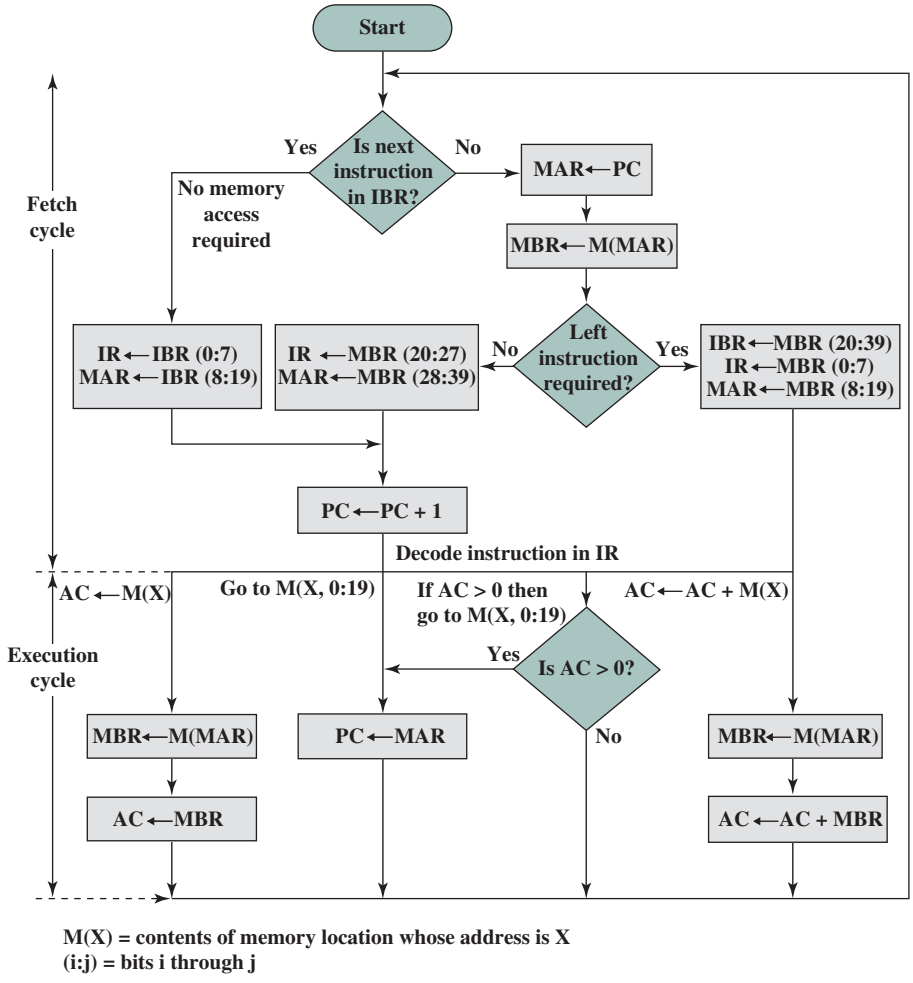


Figure 1.7 Partial Flowchart of IAS Operation

obtained from memory by loading a word into the MBR, and then down to the IBR, IR, and MAR.

Why the indirection? These operations are controlled by electronic circuitry and result in the use of data paths. To simplify the electronics, there is only one register that is used to specify the address in memory for a read or write and only one register used for the source or destination.

Once the opcode is in the IR, the *execute cycle* is performed. Control circuitry interprets the opcode and executes the instruction by sending out the appropriate control signals to cause data to be moved or an operation to be performed by the ALU.

The IAS computer had a total of 21 instructions, which are listed in Table 1.1. These can be grouped as follows:

- **Data transfer:** Move data between memory and ALU registers or between two ALU registers.

Table 1.1 The IAS Instruction Set

Instruction Type	Opcode	Symbolic Representation	Description
Data transfer	00001010	LOAD MQ	Transfer contents of register MQ to the accumulator AC
	00001001	LOAD MQ,M(X)	Transfer contents of memory location X to MQ
	00100001	STOR M(X)	Transfer contents of accumulator to memory location X
	00000001	LOAD M(X)	Transfer M(X) to the accumulator
	00000010	LOAD –M(X)	Transfer –M(X) to the accumulator
	00000011	LOAD |M(X)|	Transfer absolute value of M(X) to the accumulator
	00000100	LOAD –|M(X)|	Transfer –|M(X)| to the accumulator
Unconditional branch	00001101	JUMP M(X,0:19)	Take next instruction from left half of M(X)
	00001110	JUMP M(X,20:39)	Take next instruction from right half of M(X)
Conditional branch	00001111	JUMP + M(X,0:19)	If number in the accumulator is nonnegative, take next instruction from left half of M(X)
	00010000	JUMP + M(X,20:39)	If number in the accumulator is nonnegative, take next instruction from right half of M(X)
Arithmetic	00000101	ADD M(X)	Add M(X) to AC; put the result in AC
	00000111	ADD |M(X)|	Add |M(X)| to AC; put the result in AC
	00000110	SUB M(X)	Subtract M(X) from AC; put the result in AC
	00001000	SUB |M(X)|	Subtract |M(X)| from AC; put the remainder in AC
	00001011	MUL M(X)	Multiply M(X) by MQ; put most significant bits of result in AC, put least significant bits in MQ
	00001100	DIV M(X)	Divide AC by M(X); put the quotient in MQ and the remainder in AC
	00010100	LSH	Multiply accumulator by 2; that is, shift left one bit position
	00010101	RSH	Divide accumulator by 2; that is, shift right one position
Address modify	00010010	STOR M(X,8:19)	Replace left address field at M(X) by 12 rightmost bits of AC
	00010011	STOR M(X,28:39)	Replace right address field at M(X) by 12 rightmost bits of AC

- **Unconditional branch:** Normally, the control unit executes instructions in sequence from memory. This sequence can be changed by a branch instruction, which facilitates repetitive operations.
- **Conditional branch:** The branch can be made dependent on a condition, thus allowing decision points.
- **Arithmetic:** Operations performed by the ALU.
- **Address modify:** Permits addresses to be computed in the ALU and then inserted into instructions stored in memory. This allows a program considerable addressing flexibility.

Table 1.1 presents instructions (excluding I/O instructions) in a symbolic, easy-to-read form. In binary form, each instruction must conform to the format of

Figure 1.6b. The opcode portion (first 8 bits) specifies which of the 21 instructions is to be executed. The address portion (remaining 12 bits) specifies which of the 4,096 memory locations is to be involved in the execution of the instruction.

Figure 1.7 shows several examples of instruction execution by the control unit. Note that each operation requires several steps, some of which are quite elaborate. The multiplication operation requires 39 suboperations, one for each bit position except that of the sign bit.

1.4 GATES, MEMORY CELLS, CHIPS, AND MULTICHIP MODULES

Gates and Memory Cells

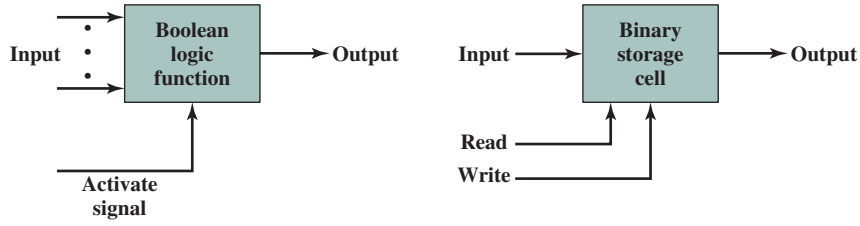
The basic elements of a digital computer, as we know, must perform data storage, movement, processing, and control functions. Only two fundamental types of components are required (Figure 1.8): gates and memory cells. A **gate** is a device that implements a simple Boolean or logical function. For example, an AND gate with inputs *A* and *B* and output *C* implements the expression IF *A* AND *B* ARE TRUE THEN *C* IS TRUE. Such devices are called gates because they control data flow in much the same way that canal gates control the flow of water. The **memory cell** is a device that can store one bit of data; that is, the device can be in one of two stable states at any time. By interconnecting large numbers of these fundamental devices, we can construct a computer. We can relate this to our four basic functions as follows:

- **Data storage:** Provided by memory cells.
- **Data processing:** Provided by gates.
- **Data movement:** The paths among components are used to move data from memory to memory and from memory through gates to memory.
- **Control:** The paths among components can carry control signals. For example, a gate will have one or two data inputs plus a control signal input that activates the gate. When the control signal is ON, the gate performs its function on the data inputs and produces a data output. Conversely, when the control signal is OFF, the output line is null, such as is produced by a high impedance state. Similarly, the memory cell will store the bit that is on its input lead when the WRITE control signal is ON and will place the bit that is in the cell on its output lead when the READ control signal is ON.

Thus, a computer consists of gates, memory cells, and interconnections among these elements. The gates and memory cells are, in turn, constructed of simple electronic components, such as transistors and capacitors.

Transistors

The fundamental building block of digital circuits used to construct processors, memories, and other digital logic devices is the transistor. The active part of the transistor is made of silicon or some other semiconductor material that can change its



(a) Gate (b) Memory cell

Figure 1.8 Fundamental Computer Elements

electrical state when pulsed. In its normal state, the material may be nonconductive or conductive, either impeding or allowing current flow. When voltage is applied to the gate, the transistor changes its state.

A single, self-contained transistor is called a *discrete component*. Throughout the 1950s and early 1960s, electronic equipment was composed largely of discrete components—transistors, resistors, capacitors, and so on. Discrete components were manufactured separately, packaged in their own containers, and soldered or wired together onto Masonite-like circuit boards, which were then installed in computers, oscilloscopes, and other electronic equipment. Whenever an electronic device called for a transistor, a little tube of metal containing a pinhead-sized piece of silicon had to be soldered to a circuit board. The entire manufacturing process, from transistor to circuit board, was expensive and cumbersome.

These facts of life were beginning to create problems in the computer industry. Early second-generation computers contained about 10,000 transistors. This figure grew to the hundreds of thousands, making the manufacture of newer, more powerful machines increasingly difficult.

Microelectronic Chips

Microelectronics means, literally, "small electronics." Since the beginning of digital electronics and the computer industry, there has been a consistent trend toward the reduction in size of digital electronic circuits. Before examining the implications and benefits of this trend, we need to say something about the nature of digital electronics. A more detailed discussion is found in Chapter 12.

The integrated circuit exploits the fact that such components as transistors, resistors, and conductors can be fabricated from a semiconductor such as silicon. It is merely an extension of the solid-state art to fabricate an entire circuit in a tiny piece of silicon rather than assemble discrete components made from separate pieces of silicon into the same circuit. Many transistors can be produced at the same time on a single wafer of silicon. Equally important, these transistors can be connected with a process of metallization to form circuits.

Figure 1.9 depicts the key concepts in an integrated circuit. A thin *wafer* of silicon is divided into a matrix of small areas, each a few millimeters square. The identical circuit pattern is fabricated in each area, and the wafer is broken up into *chips*. Each chip consists of many gates and/or memory cells plus a number of input and output attachment points. This chip is then packaged in housing that protects it and provides pins for attachment to devices beyond the chip. A number of these

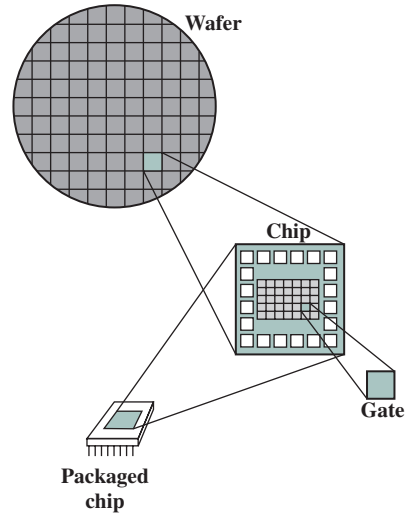


Figure 1.9 Relationship among Wafer, Chip, and Gate

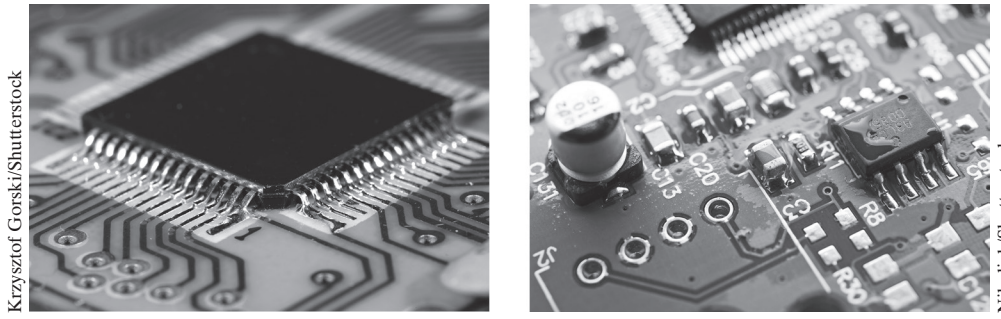
packages can then be interconnected on a printed circuit board to produce larger and more complex circuits. Figure 1.10a indicates what a packaged processor or memory chip looks like, and Figure 1.10b shows a packaged chip wired onto a motherboard.

Initially, only a few gates or memory cells could be reliably manufactured and packaged together. These early integrated circuits are referred to as *small-scale integration* (SSI). As time went on, it became possible to pack more and more components on the same chip. This growth in density is illustrated in Figure 1.11; it is one of the most remarkable technological trends ever recorded.[6] This figure reflects the famous Moore's law, which was propounded by Gordon Moore, cofounder of Intel, in 1965 [MOOR65]. Moore observed that the number of transistors that could be put on a single chip was doubling every year, and correctly predicted that this pace would continue into the near future. To the surprise of many, including Moore, the pace continued year after year and decade after decade. The pace slowed to a doubling every 18 months in the 1970s, but has sustained that rate ever since.

The consequences of Moore's law are profound:

1. The cost of a chip has remained virtually unchanged during this period of rapid growth in density. This means that the cost of computer logic and memory circuitry has fallen at a dramatic rate.
2. Because logic and memory elements are placed closer together on more densely packed chips, the electrical path length is shortened, increasing operating speed.

[6]Note that the vertical axis uses a log scale. A basic review of log scales is in the math refresher document at the Computer Science Student Resource Site at ComputerScienceStudent.com.



(a) Close-up of packaged chip (b) Chip on motherboard

Figure 1.10 Processor or Memory Chip on Motherboard

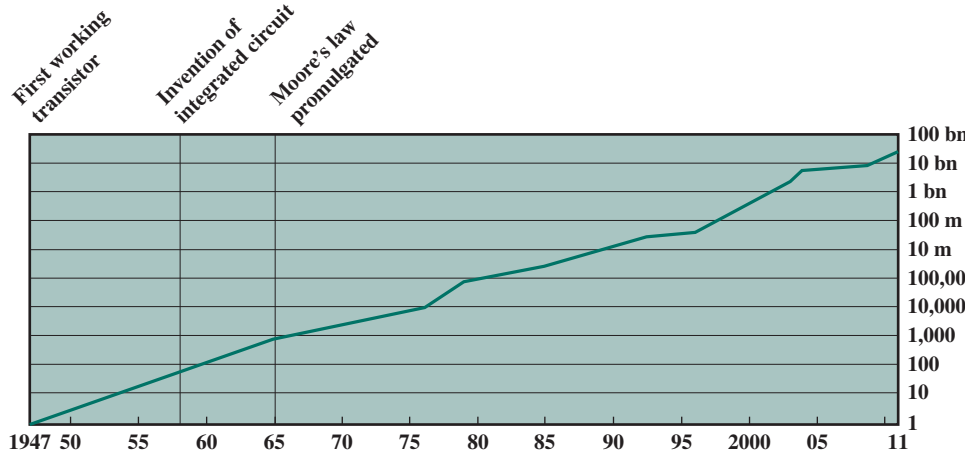


Figure 1.11 Growth in Transistor Count on Integrated Circuits

3. The computer becomes smaller, making it more convenient to place in a variety of environments.
4. There is a reduction in power requirements.
5. The interconnections on the integrated circuit are much more reliable than solder connections. With more circuitry on each chip, there are fewer interchip connections.

Multichip Module

The increasing requirements for denser and faster memories have led to efforts to further compact standard packaging approaches, with one of the most important and widely used being the multichip module. In traditional system design, each individual process or memory chip is packaged and then wired to a motherboard (see Figure 1.10).

The basic idea behind developing MCM technology is to decrease the average spacing between ICs in an electronic system. An MCM is a chip package that

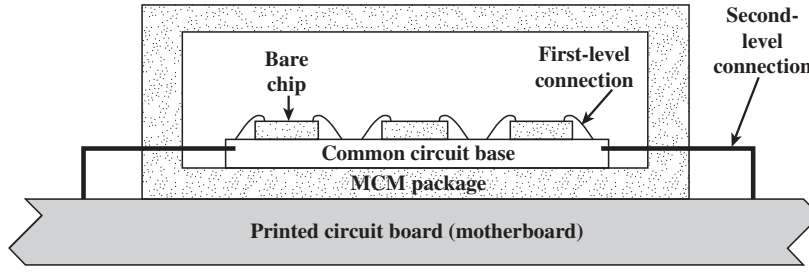


Figure 1.12 Multichip Module

contains several bare chips mounted close together on a substrate (base) of some kind and interconnected by conductors in that base. The short tracks between the chips increase performance and eliminate much of the noise that external tracks between individual chip packages can pick up.

MCMs are classified by substrate, which include the following types [BLUM99]:

- **MCM-L:** composed of metal traces on stacked organic laminate sheets.
- **MCM-C:** metal patterned and interconnected on co-fired ceramic layers.
- **MCM-D:** vapor-deposited, patterned metal layers alternating sequentially with spun-on or vapor-deposited dielectric thin films.

The basic architecture of an MCM is composed of (Figure 1.12):

- **Integrated circuits:** Bare chips mounted on/in the surface of the substrate.
- **Level-1 interconnections:** Connections between chips through paths in the substrate.
- **Substrate:** The common base that provides all the signal interconnections and the mechanical support for all chips
- **MCM package:** Provides a degree of protection to the circuits in addition to heat removal and interconnections.
- **Level-2 interconnections:** Provides the necessary interface to the printed circuit board on which the MCM is mounted.

Table 1.3 Evolution of Intel Microprocessors (page 1 of 2)

(a) 1970s Processors

	4004	8008	8080	8086	8088
Introduced	1971	1972	1974	1978	1979
Clock speeds	108 kHz	108 kHz	2 MHz	5 MHz, 8 MHz, 10 MHz	5 MHz, 8 MHz
Bus width	4 bits	8 bits	8 bits	16 bits	8 bits
Number of transistors	2,300	3,500	6,000	29,000	29,000
Feature size (μm)	10	8	6	3	6
Addressable memory	640 bytes	16 KB	64 KB	1 MB	1 MB

(b) 1980s Processors

	80286	386TM DX	386TM SX	486TM DX CPU
Introduced	1982	1985	1988	1989
Clock speeds	6–12.5 MHz	16–33 MHz	16–33 MHz	25–50 MHz
Bus width	16 bits	32 bits	16 bits	32 bits
Number of transistors	134,000	275,000	275,000	1.2 million
Feature size (μm)	1.5	1	1	0.8–1
Addressable memory	16 MB	4 GB	16 MB	4 GB
Virtual memory	1 GB	64 TB	64 TB	64 TB
Cache	—	—	—	8 kB

(c) 1990s Processors

	486TM SX	Pentium	Pentium Pro	Pentium II
Introduced	1991	1993	1995	1997
Clock speeds	16–33 MHz	60–166 MHz,	150–200 MHz	200–300 MHz
Bus width	32 bits	32 bits	64 bits	64 bits
Number of transistors	1.185 million	3.1 million	5.5 million	7.5 million
Feature size (μm)	1	0.8	0.6	0.35
Addressable memory	4 GB	4 GB	64 GB	64 GB
Virtual memory	64 TB	64 TB	64 TB	64 TB
Cache	8 kB	8 kB	512 kB L1 and 1 MB L2	512 kB L2

(d) Recent Processors

	Pentium III	Pentium 4	Core 2 Duo	Core i7 EE 4960X	Core i9-7900X
Introduced	1999	2000	2006	2013	2017
Clock speeds	450–660 MHz	1.3–1.8 GHz	1.06–1.2 GHz	4 GHz	4.3 GHz
Bus width	64 bits	64 bits	64 bits	64 bits	64 bits
Number of transistors	9.5 million	42 million	167 million	1.86 billion	7.2 billion
Feature size (nm)	250	180	65	22	14
Addressable memory	64 GB	64 GB	64 GB	64 GB	128 GB
Virtual memory	64 TB	64 TB	64 TB	64 TB	64 TB
Cache	512 kB L2	256 kB L2	2 MB L2	1.5 MB L2/ 15 MB L3	14 MB L3
Number of cores	1	1	2	6	10

1.5 THE EVOLUTION OF THE INTEL x86 ARCHITECTURE

Throughout this book, we rely on many concrete examples of computer design and implementation to illustrate concepts and to illuminate trade-offs. Numerous systems, both contemporary and historical, provide examples of important computer architecture design features. But the book relies principally on examples from two processor families: the Intel x86 and the ARM architectures. The current x86 offerings represent the results of decades of design effort on **complex instruction set computers (CISCs)**. The x86 incorporates the sophisticated design principles once found only on mainframes and supercomputers and serves as an excellent example of CISC design. An alternative approach to processor design is the **reduced instruction set computer (RISC)**. The ARM architecture is used in a wide variety of embedded systems and is one of the most powerful and best-designed RISC-based systems on the market. In this section and the next, we provide a brief overview of these two systems.

In terms of market share, Intel has ranked as the number one maker of microprocessors for non-embedded systems for decades, a position it seems unlikely to yield. The evolution of its flagship microprocessor product serves as a good indicator of the evolution of computer technology in general.

Table 1.3 shows that evolution. Interestingly, as microprocessors have grown faster and much more complex, Intel has actually picked up the pace. Intel used to develop microprocessors one after another, every four years. But Intel hopes to keep rivals at bay by trimming a year or two off this development time, and has done so with the most recent x86 generations.[7]

It is worthwhile to list some of the highlights of the evolution of the Intel product line:

- **8080:** The world's first general-purpose microprocessor. This was an 8-bit machine, with an 8-bit data path to memory. The 8080 was used in the first personal computer, the Altair.
- **8086:** A far more powerful, 16-bit machine. In addition to a wider data path and larger registers, the 8086 sported an instruction cache, or queue, that prefetches a few instructions before they are executed. A variant of this processor, the 8088, was used in IBM's first personal computer, securing the success of Intel. The 8086 is the first appearance of the x86 architecture.
- **80286:** This extension of the 8086 enabled addressing a 16-MB memory instead of just 1 MB.
- **80386:** Intel's first 32-bit machine, and a major overhaul of the product. With a 32-bit architecture, the 80386 rivaled the complexity and power of minicomputers and mainframes introduced just a few years earlier. This was the first Intel processor to support multitasking, meaning it could run multiple programs at the same time.
- **80486:** The 80486 introduced the use of much more sophisticated and powerful cache technology and sophisticated instruction pipelining. The 80486 also

[7]Intel refers to this as the *tick-tock model*. Using this model, Intel has successfully delivered next-generation silicon technology as well as new processor microarchitecture on alternating years for the past several years. See http://www.intel.com/content/www/us/en/silicon-innovations/intel-tick-tock-model-general.html.

offered a built-in math coprocessor, offloading complex math operations from the main CPU.

- **Pentium:** With the Pentium, Intel introduced the use of superscalar techniques, which allow multiple instructions to execute in parallel.
- **Pentium Pro:** The Pentium Pro continued the move into superscalar organization begun with the Pentium, with aggressive use of register renaming, branch prediction, data flow analysis, and speculative execution.
- **Pentium II:** The Pentium II incorporated Intel MMX technology, which is designed specifically to process video, audio, and graphics data efficiently.
- **Pentium III:** The Pentium III incorporates additional floating-point instructions: The Streaming SIMD Extensions (SSE) instruction set extension added 70 new instructions designed to increase performance when exactly the same operations are to be performed on multiple data objects. Typical applications are digital signal processing and graphics processing.
- **Pentium 4:** The Pentium 4 includes additional floating-point and other enhancements for multimedia.
- **Core:** This is the first Intel x86 microprocessor with a dual core, referring to the implementation of two cores on a single chip.
- **Core 2:** The Core 2 extends the Core architecture to 64 bits. The Core 2 Quad provides four cores on a single chip. More recent Core offerings have up to 10 cores per chip. An important addition to the architecture was the Advanced Vector Extensions instruction set that provided a set of 256-bit, and then 512-bit, instructions for efficient processing of vector data.

Almost 40 years after its introduction in 1978, the x86 architecture continues to dominate the processor market outside of embedded systems. Although the organization and technology of the x86 machines have changed dramatically over the decades, the instruction set architecture has evolved to remain backward compatible with earlier versions. Thus, any program written on an older version of the x86 architecture can execute on newer versions. All changes to the instruction set architecture have involved additions to the instruction set, with no subtractions. The rate of change has been the addition of roughly one instruction per month added to the architecture [ANTH08], so that there are now thousands of instructions in the instruction set.

The x86 provides an excellent illustration of the advances in computer hardware over the past 35 years. The 1978 8086 was introduced with a clock speed of 5 MHz and had 29,000 transistors. A six-core Core i7 EE 4960X introduced in 2013 operates at 4 GHz, a speedup of a factor of 800, and has 1.86 billion transistors, about 64,000 times as many as the 8086. Yet the Core i7 EE 4960X is in only a slightly larger package than the 8086 and has a comparable cost.

1.6 EMBEDDED SYSTEMS

The term *embedded system* refers to the use of electronics and software within a product, as opposed to a general-purpose computer, such as a laptop or desktop system. Millions of computers are sold every year, including laptops, personal

computers, workstations, servers, mainframes, and supercomputers. In contrast, billions of computer systems are produced each year that are embedded within larger devices. Today many, perhaps most, devices that use electric power have an embedded computing system. It is likely that in the near future virtually all such devices will have embedded computing systems.

Types of devices with embedded systems are almost too numerous to list. Examples include cell phones, digital cameras, video cameras, calculators, microwave ovens, home security systems, washing machines, lighting systems, thermostats, printers, various automotive systems (e.g., transmission control, cruise control, fuel injection, anti-lock brakes, and suspension systems), tennis rackets, toothbrushes, and numerous types of sensors and actuators in automated systems.

Often, embedded systems are tightly coupled to their environment. This can give rise to real-time constraints imposed by the need to interact with the environment. Constraints, such as required speeds of motion, required precision of measurement, and required time durations, dictate the timing of software operations. If multiple activities must be managed simultaneously, this imposes more complex real-time constraints.

Figure 1.13 shows in general terms an embedded system organization. In addition to the processor and memory, there are a number of elements that differ from the typical desktop or laptop computer:

- There may be a variety of interfaces that enable the system to measure, manipulate, and otherwise interact with the external environment. Embedded systems often interact (sense, manipulate, and communicate) with the external world through sensors and actuators, and hence are typically reactive systems; a reactive system is in continual interaction with the environment and executes at a pace determined by that environment.

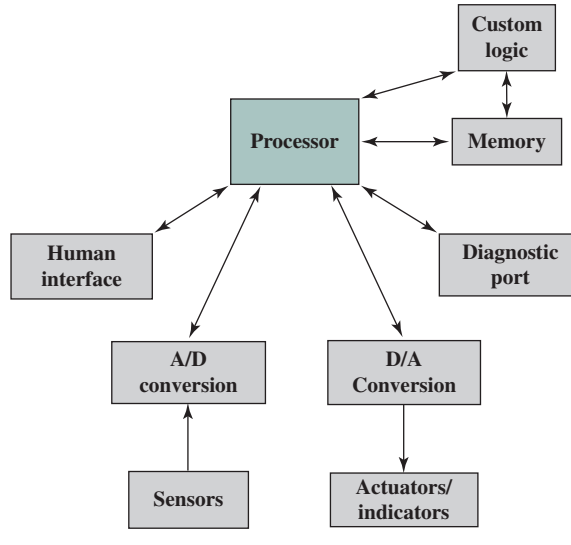


Figure 1.13 Possible Organization of an Embedded System

- The human interface may be as simple as a flashing light or as complicated as real-time robotic vision. In many cases, there is no human interface.
- The diagnostic port may be used for diagnosing the system that is being controlled—not just for diagnosing the computer.
- Special-purpose field programmable (FPGA), application-specific (ASIC), or even nondigital hardware may be used to increase performance or reliability.
- Software often has a fixed function and is specific to the application.
- Efficiency is of paramount importance for embedded systems. They are optimized for energy, code size, execution time, weight and dimensions, and cost.

There are several noteworthy areas of similarity to general-purpose computer systems as well:

- Even with nominally fixed function software, the ability to field upgrade to fix bugs, to improve security, and to add functionality, has become very important for embedded systems, and not just in consumer devices.
- One comparatively recent development has been of embedded system platforms that support a wide variety of apps. Good examples of this are smartphones and audio/visual devices, such as smart TVs.

The Internet of Things

It is worthwhile to separately call out one of the major drivers in the proliferation of embedded systems. The **Internet of things (IoT)** is a term that refers to the expanding interconnection of smart devices, ranging from appliances to tiny sensors. A dominant theme is the embedding of short-range mobile transceivers into a wide array of gadgets and everyday items, enabling new forms of communication between people and things, and between things themselves. The Internet now supports the interconnection of billions of industrial and personal objects, usually through cloud systems. The objects deliver sensor information, act on their environment, and, in some cases, modify themselves to create overall management of a larger system, like a factory or city.

The IoT is primarily driven by deeply embedded devices (defined below). These devices are low-bandwidth, low-repetition data-capture, and low-bandwidth data-usage appliances that communicate with each other and provide data via user interfaces. Embedded appliances, such as high-resolution video security cameras, video VoIP phones, and a handful of others, require high-bandwidth streaming capabilities. Yet countless products simply require packets of data to be intermittently delivered.

With reference to the end systems supported, the Internet has gone through roughly four generations of deployment culminating in the IoT:

1. **Information technology (IT):** PCs, servers, routers, firewalls, and so on, bought as IT devices by enterprise IT people and primarily using wired connectivity.
2. **Operational technology (OT):** Machines/appliances with embedded IT built by non-IT companies, such as medical machinery, SCADA (supervisory control and data acquisition), process control, and kiosks, bought as appliances by enterprise OT people and primarily using wired connectivity.

3. **Personal technology:** Smartphones, tablets, and eBook readers bought as IT devices by consumers (employees) exclusively using wireless connectivity and often multiple forms of wireless connectivity.
4. **Sensor/actuator technology:** Single-purpose devices bought by consumers, IT, and OT people exclusively using wireless connectivity, generally of a single form, as part of larger systems.

It is the fourth generation that is usually thought of as the IoT, and it is marked by the use of billions of embedded devices.

Embedded Operating Systems

There are two general approaches to developing an embedded operating system (OS). The first approach is to take an existing OS and adapt it for the embedded application. For example, there are embedded versions of Linux, Windows, and Mac, as well as other commercial and proprietary operating systems specialized for embedded systems. The other approach is to design and implement an OS intended solely for embedded use. An example of the latter is TinyOS, widely used in wireless sensor networks. This topic is explored in depth in [STAL18].

Application Processors versus Dedicated Processors

In this subsection, and the next two, we briefly introduce some terms commonly found in the literature on embedded systems. **Application processors** are defined by the processor's ability to execute complex operating systems, such as Linux, Android, and Chrome. Thus, the application processor is general-purpose in nature. A good example of the use of an embedded application processor is the smartphone. The embedded system is designed to support numerous apps and perform a wide variety of functions.

Most embedded systems employ a **dedicated processor**, which, as the name implies, is dedicated to one or a small number of specific tasks required by the host device. Because such an embedded system is dedicated to a specific task or tasks, the processor and associated components can be engineered to reduce size and cost.

Microprocessors versus Microcontrollers

As we have seen, early **microprocessor** chips included registers, an ALU, and some sort of control unit or instruction processing logic. As transistor density increased, it became possible to increase the complexity of the instruction set architecture, and ultimately to add memory and more than one processor. Contemporary microprocessor chips, as shown in Figure 1.2, include multiple cores and a substantial amount of cache memory.

A **microcontroller** chip makes a substantially different use of the logic space available. Figure 1.14 shows in general terms the elements typically found on a microcontroller chip. As shown, a microcontroller is a single chip that contains the processor, non-volatile memory for the program (ROM), volatile memory for input and output (RAM), a clock, and an I/O control unit. The processor portion of the microcontroller has a much lower silicon area than other microprocessors and much higher energy efficiency. We examine microcontroller organization in more detail in Section 1.7.

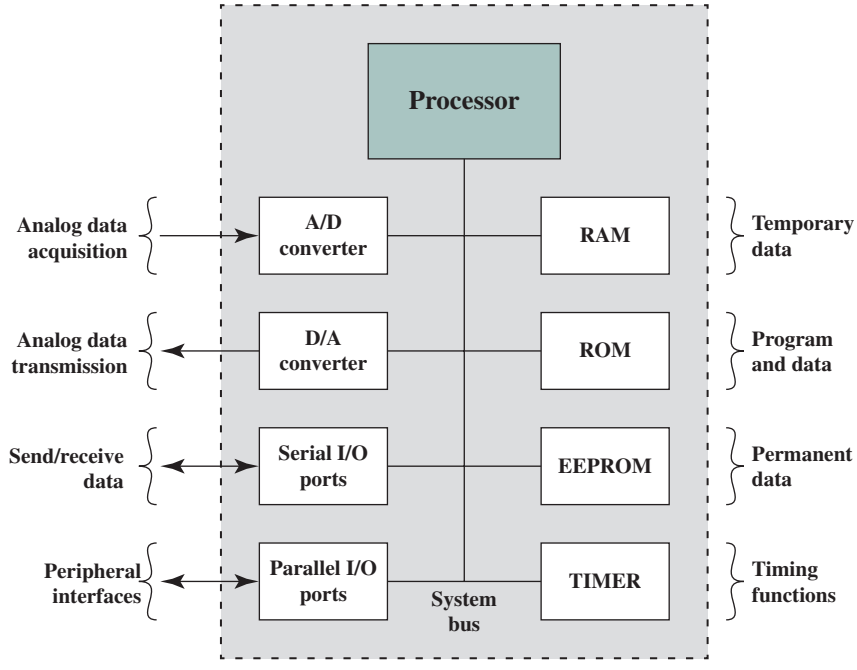


Figure 1.14 Typical Microcontroller Chip Elements

Also called a "computer on a chip," billions of microcontroller units are embedded each year in myriad products from toys to appliances to automobiles. For example, a single vehicle can use 70 or more microcontrollers. Typically, especially for the smaller, less expensive microcontrollers, they are used as dedicated processors for specific tasks. For example, microcontrollers are heavily utilized in automation processes. By providing simple reactions to input, they can control machinery, turn fans on and off, open and close valves, and so forth. They are integral parts of modern industrial technology and are among the most inexpensive ways to produce machinery that can handle extremely complex functionalities.

Microcontrollers come in a range of physical sizes and processing power. Processors range from 4-bit to 32-bit architectures. Microcontrollers tend to be much slower than microprocessors, typically operating in the MHz range rather than the GHz speeds of microprocessors. Another typical feature of a microcontroller is that it does not provide for human interaction. The microcontroller is programmed for a specific task, embedded in its device, and executes as and when required.

Embedded versus Deeply Embedded Systems

We have, in this section, defined the concept of an embedded system. A subset of embedded systems, and a quite numerous subset, is referred to as **deeply embedded systems**. Although this term is widely used in the technical and commercial literature, you will search the Internet in vain (or at least I did) for a straightforward definition. Generally, we can say that a deeply embedded system has a processor whose behavior is difficult to observe both by the programmer and the user.

A deeply embedded system uses a microcontroller rather than a microprocessor, is not programmable once the program logic for the device has been burned into ROM (read-only memory), and has no interaction with a user.

Deeply embedded systems are dedicated, single-purpose devices that detect something in the environment, perform a basic level of processing, and then do something with the results. Deeply embedded systems often have wireless capability and appear in networked configurations, such as networks of sensors deployed over a large area (e.g., factory, agricultural field). The Internet of things depends heavily on deeply embedded systems. Typically, deeply embedded systems have extreme resource constraints in terms of memory, processor size, time, and power consumption.

1.7 ARM ARCHITECTURE

The ARM architecture refers to a processor architecture that has evolved from RISC design principles and is used in embedded systems. Chapter 7 examines RISC design principles in detail. In this section, we give a brief overview of the ARM architecture.

ARM Evolution

ARM is a family of RISC-based microprocessors and microcontrollers designed by ARM Holdings, Cambridge, England. The company doesn't make processors but instead designs microprocessor and multicore architectures and licenses them to manufacturers. ARM Holdings has two types of licensable products: processors and processor architectures. For processors, the customer buys the rights to use ARM-supplied design in their own chips. For a processor architecture, the customer buys the rights to design their own processor compliant with ARM's architecture.

ARM chips are high-speed processors that are known for their small die size and low power requirements. They are widely used in smartphones and other handheld devices, including game systems, as well as a large variety of consumer products. ARM chips are the processors in Apple's popular iPod and iPhone devices, and are used in virtually all Android smartphones as well. ARM's partners shipped 16.7 billion ARM-based chips in 2016. ARM is probably the most widely used embedded processor architecture and indeed the most widely used processor architecture of any kind in the world [VANC14].

The origins of ARM technology can be traced back to the British-based Acorn Computers company. In the early 1980s, Acorn was awarded a contract by the British Broadcasting Corporation (BBC) to develop a new microcomputer architecture for the BBC Computer Literacy Project. The success of this contract enabled Acorn to go on to develop the first commercial RISC processor, the Acorn RISC Machine (ARM). The first version, ARM1, became operational in 1985 and was used for internal research and development as well as being used as a coprocessor in the BBC machine.

In this early stage, Acorn used the company VLSI Technology to do the actual fabrication of the processor chips. VLSI was licensed to market the chip on its own and had some success in getting other companies to use the ARM in their products, particularly as an embedded processor.

The ARM design matched a growing commercial need for a high-performance, low-power-consumption, small-size, and low-cost processor for embedded applications. But further development was beyond the scope of Acorn's capabilities. Accordingly, a new company was organized, with Acorn, VLSI, and Apple Computer as founding partners, known as ARM Ltd. The Acorn RISC Machine became Advanced RISC Machines.[8] ARM was acquired by Japanese telecommunications company SoftBank Group in 2016.

Instruction Set Architecture

The ARM instruction set is highly regular, designed for efficient implementation of the processor and efficient execution. All instructions are 32 bits long and follow a regular format. This makes the ARM ISA suitable for implementation over a wide range of products.

Augmenting the basic ARM ISA is the Thumb instruction set, which is a re-encoded subset of the ARM instruction set. Thumb is designed to increase the performance of ARM implementations that use a 16-bit or narrower memory data bus, and to allow better code density than provided by the ARM instruction set. The Thumb instruction set contains a subset of the ARM 32-bit instruction set recoded into 16-bit instructions. The current defined version is Thumb-2.

The ARM and Thumb-2 ISAs are discussed in Chapters 12 and 13.

ARM Products

ARM Holdings licenses a number of specialized microprocessors and related technologies, but the bulk of their product line is the Cortex family of microprocessor architectures. There are three Cortex architectures, conveniently labeled with the initials A, R, and M.

CORTEX-A The Cortex-A series of processors are application processors, intended for mobile devices such as smartphones and eBook readers, as well as consumer devices such as digital TV and home gateways (e.g., DSL and cable Internet modems). These processors run at higher clock frequency (over 1 GHz), and support a memory management unit (MMU), which is required for full feature OSs such as Linux, Android, MS Windows, and mobile OSs. An MMU is a hardware module that supports virtual memory and paging by translating virtual addresses into physical addresses; this topic is explored in Chapter 8.

The two architectures use both the ARM and Thumb-2 instruction. Some of the processors in this series are 32-bit machines and others are 64-bit machines.

CORTEX-R The Cortex-R is designed to support real-time applications, in which the timing of events needs to be controlled with rapid response to events. They can run at a fairly high clock frequency (e.g., 2 MHz to 4 MHz) and have very low response latency. The Cortex-R includes enhancements both to the instruction set and to the processor organization to support deeply embedded real-time devices. Most of these processors do not have MMU; the limited data requirements and the limited number

[8]The company dropped the designation *Advanced RISC Machines* in the late 1990s. It is now simply known as the ARM architecture.

of simultaneous processes eliminates the need for elaborate hardware and software support for virtual memory. The Cortex-R does have a Memory Protection Unit (MPU), cache, and other memory features designed for industrial applications. An MPU is a hardware module that prohibits one program in memory from accidentally accessing memory assigned to another active program. Using various methods, a protective boundary is created around the program, and instructions within the program are prohibited from referencing data outside of that boundary.

Examples of embedded systems that would use the Cortex-R are automotive braking systems, mass storage controllers, and networking and printing devices.

CORTEX–M Cortex-M series processors have been developed primarily for the microcontroller domain where the need for fast, highly deterministic interrupt management is coupled with the desire for extremely low gate count and lowest possible power consumption. As with the Cortex-R series, the Cortex-M architecture has an MPU but no MMU. The Cortex-M uses only the Thumb-2 instruction set. The market for the Cortex-M includes IoT devices, wireless sensor/actuator networks used in factories and other enterprises, automotive body electronics, and so on.

There are currently seven versions of the Cortex-M series:

- **Cortex-M0:** Designed for 8- and 16-bit applications, this model emphasizes low cost, ultra low power, and simplicity. It is optimized for small silicon die size (starting from 12k gates) and use in the lowest cost chips.
- **Cortex-M0+:** An enhanced version of the M0 that is more energy efficient.
- **Cortex-M3:** Designed for 16- and 32-bit applications, this model emphasizes performance and energy efficiency. It also has comprehensive debug and trace features to enable software developers to develop their applications quickly.
- **Cortex-M4:** This model provides all the features of the Cortex-M3, with additional instructions to support digital signal processing tasks.
- **Cortex-M7:** Provides higher performance than the M4. It is still primarily a 32-bit machine but uses 64-bit wide instruction and data buses.
- **Cortex-M23:** This model is similar to the M0+, and adds integer divide instructions and some security features.
- **Cortex-M33:** This model is similar to the M4, and adds some security features.

In this text, we will primarily use the ARM Cortex-M3 as our example embedded system processor. It is the best suited of all ARM models for general-purpose microcontroller use. The Cortex-M3 is used by a variety of manufacturers of microcontroller products. Initial microcontroller devices from lead partners already combine the Cortex-M3 processor with flash, SRAM, and multiple peripherals to provide a competitive offering at the price of just $1.

Figure 1.15 provides a block diagram of the EFM32 microcontroller from Silicon Labs. The figure also shows detail of the Cortex-M3 processor and core components. We examine each level in turn.

The **Cortex-M3 core** makes use of separate buses for instructions and data. This arrangement is sometimes referred to as a Harvard architecture, in contrast with the von Neumann architecture, which uses the same signal buses and memory

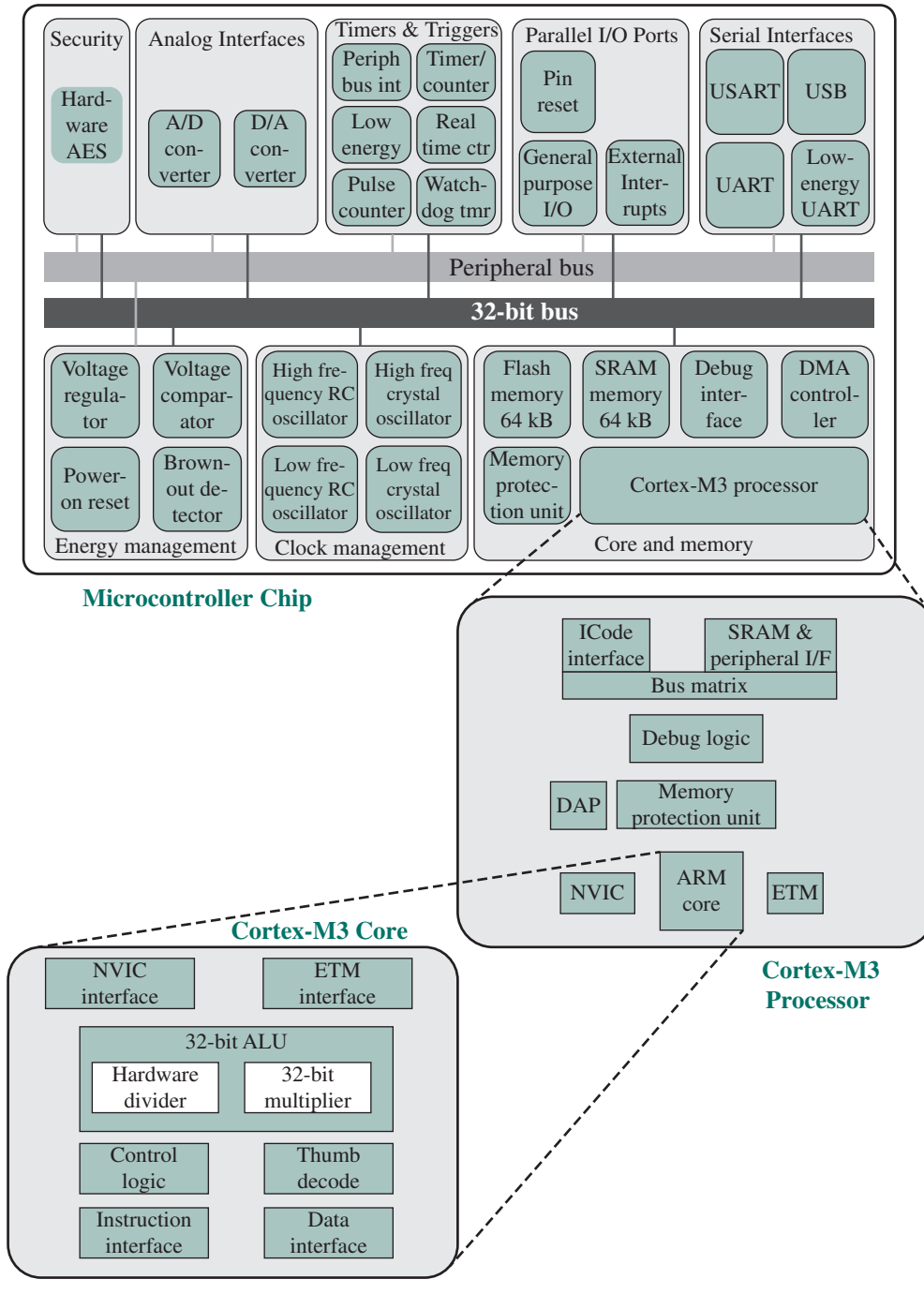


Figure 1.15 Typical Microcontroller Chip Based on Cortex-M3

for both instructions and data. By being able to read both an instruction and data from memory at the same time, the Cortex-M3 processor can perform many operations in parallel, speeding application execution. The core contains a decoder for Thumb instructions, an advanced ALU with support for hardware multiply and divide, control logic, and interfaces to the other components of the processor. In particular, there is an interface to the nested vector interrupt controller (NVIC) and the embedded trace macrocell (ETM) module.

The core is part of a module called the **Cortex-M3 processor**. This term is somewhat misleading, because typically in the literature, the terms core and processor are viewed as equivalent. In addition to the core, the processor includes the following elements:

- **NVIC:** Provides configurable interrupt handling abilities to the processor. It facilitates low-latency exception and interrupt handling, and controls power management.
- **ETM:** An optional debug component that enables reconstruction of program execution. The ETM is designed to be a high-speed, low-power debug tool that only supports instruction trace.
- **Debug access port (DAP):** This provides an interface for external debug access to the processor.
- **Debug logic:** Basic debug functionality includes processor halt, single-step, processor core register access, unlimited software breakpoints, and full system memory access.
- **ICode interface:** Fetches instructions from the code memory space.
- **SRAM & peripheral interface:** Read/write interface to data memory and peripheral devices.
- **Bus matrix:** Connects the core and debug interfaces to external buses on the microcontroller.
- **Memory protection unit:** Protects critical data used by the operating system from user applications, separating processing tasks by disallowing access to each other's data, disabling access to memory regions, allowing memory regions to be defined as read-only, and detecting unexpected memory accesses that could potentially break the system.

The upper part of Figure 1.15 shows the block diagram of a typical microcontroller built with the Cortex-M3, in this case the EFM32 microcontroller. This microcontroller is marketed for use in a wide variety of devices, including energy, gas, and water metering; alarm and security systems; industrial automation devices; home automation devices; smart accessories; and health and fitness devices. The silicon chip consists of 10 main areas:

- **Core and memory:** This region includes the Cortex-M3 processor, static RAM (SRAM) data memory,[9] and flash memory[10] for storing program instructions

[9]Static RAM (SRAM) is a form of random-access memory used for cache memory; see Chapter 6.

[10]Flash memory is a versatile form of memory used both in microcontrollers and as external memory; it is discussed in Chapter 7.

and nonvarying application data. Flash memory is nonvolatile (data is not lost when power is shut off) and so is ideal for this purpose. The SRAM stores variable data. This area also includes a debug interface, which makes it easy to reprogram and update the system in the field.

- **Parallel I/O ports:** Configurable for a variety of parallel I/O schemes.
- **Serial interfaces:** Supports various serial I/O schemes.
- **Analog interfaces:** Analog-to-digital and digital-to-analog logic to support sensors and actuators.
- **Timers and triggers:** Keeps track of timing and counts events, generates output waveforms, and triggers timed actions in other peripherals.
- **Clock management:** Controls the clocks and oscillators on the chip. Multiple clocks and oscillators are used to minimize power consumption and provide short startup times.
- **Energy management:** Manages the various low-energy modes of operation of the processor and peripherals to provide real-time management of the energy needs so as to minimize energy consumption.
- **Security:** The chip includes a hardware implementation of the Advanced Encryption Standard (AES).
- **32-bit bus:** Connects all of the components on the chip.
- **Peripheral bus:** A network which lets the different peripheral modules communicate directly with each other without involving the processor. This supports timing-critical operation and reduces software overhead.

Comparing Figure 1.15 with Figure 1.2, you will see many similarities and the same general hierarchical structure. Note, however, that the top level of a microcontroller computer system is a single chip, whereas for a multicore computer, the top level is a motherboard containing a number of chips. Another noteworthy difference is that there is no cache, either in the Cortex-M3 processor or in the microcontroller as a whole, which plays an important role if the code or data resides in external memory. Though the number of cycles to read the instruction or data varies depending on cache hit or miss, the cache greatly improves the performance when external memory is used. Such overhead is not needed for a microcontroller.

1.8 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

application processor	computer architecture	gate
arithmetic and logic unit (ALU)	computer organization	input–output (I/O)
ARM	control unit	instruction set architecture (ISA)
central processing unit (CPU)	core	integrated circuit
chip	dedicated processor	Intel x86
	deeply embedded system	Internet of things (IoT)
	embedded system	

main memory	microelectronics	processor
memory cell	microprocessor	registers
memory management unit (MMU)	motherboard	semiconductor
memory protection unit (MPU)	multichip module (MCM)	semiconductor memory
microcontroller	multicore	system bus
	multicore processor	system interconnection
	printed circuit board	transistor

Review Questions

1.1 Explain whether an ISA is an element of computer organization or computer architecture.

1.2 What, in general terms, is the distinction between a central processing unit and a core?

1.3 What are the three functional elements of a CPU core?

1.4 What are the main components of the IAS structure?

1.5 List and briefly define the main structural components of a core.

1.6 What is a stored program computer?

1.7 Explain the Internet of Things.

1.8 List and briefly discuss the four versions of the Cortex-M series.

Problems

1.1 You are to write an IAS program to compute the results of the following equation.

$$Y = \sum_{X=1}^{N} X$$

Assume that the computation does not result in an arithmetic overflow and that X, Y, and N are positive integers with $N \geq 1$. *Note*: The IAS did not have assembly language, only machine language.

a. Use the equation $\text{Sum}(Y) = \dfrac{N(N+1)}{2}$ when writing the IAS program.

b. Do it the "hard way," without using the equation from part (a).

1.2 **a.** On the IAS, what would the machine code instruction look like to store the contents of an accumulator to memory address 8?

b. On the IAS, what would the machine code instruction look like to add the contents of memory address 16 to the accumulator?

1.3 The IAS operates by repetitively performing an instruction cycle, which consists of two sub cycles: a fetch cycle and an execute cycle. On the IAS, describe in English the tasks accomplished during the fetch cycle and those accomplished during the execute cycle.

1.4 The instruction format of the IAS computer includes two instructions per word. Considering the design of the IAS computer, what benefits does this format provide?

1.5 During the fetch cycle in Figure 1.7, why is an instruction always taken from the IBR?

1.6 Under what conditions could the instruction format of the IAS computer be inefficient?

1.7 The relative performance of the IBM 360 Model 75 is 50 times that of the 360 Model 30, yet the instruction cycle time is only 5 times as fast. How do you account for this discrepancy?

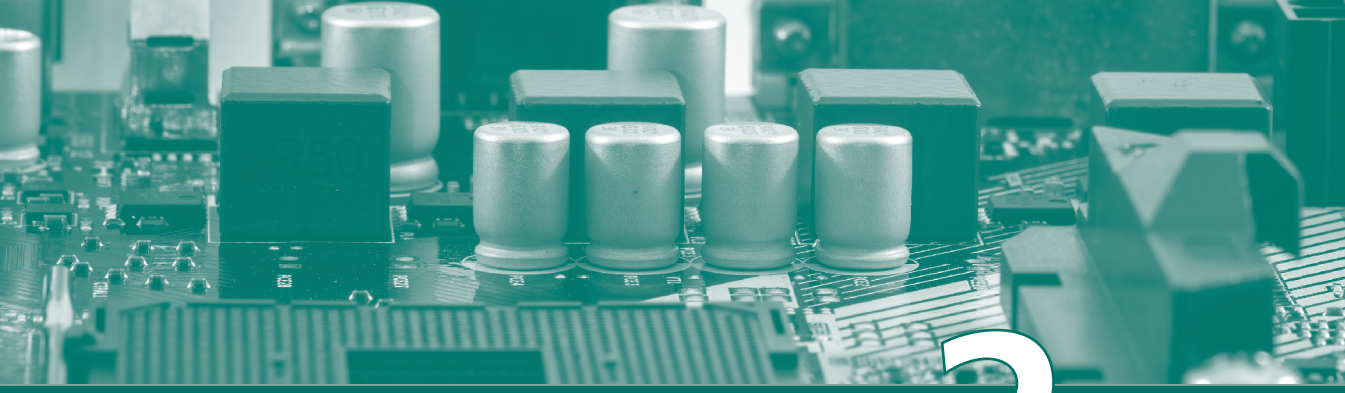
1.8 While browsing at Billy Bob's computer store, you overhear a customer asking Billy Bob what the fastest computer in the store is that he can buy. Billy Bob replies, "The fastest computer in the store is a Pentium 4. It has a clock speed of up to 1.8 GHz. The second best is the Core 2 Duo, but it only has a clock speed of 1.2 GHz." Is Billy Bob correct? What would you say to help this customer?

1.9 For each of the following examples, label them as either general-purpose systems, embedded systems, or deeply embedded systems.

a. university mainframe server
b. smart TV
c. smart fridge sensors
d. personal laptop
e. hearing aid
f. GPS Navigation Unit

1.10 The Cortex-R is an ARM CPU specifically designed for real-time or safety-critical systems. For each example in the list below, explain why the Cortex-R may or may not be appropriate.

a. self-driving car
b. environmental control system in an aircraft
c. blood gas analyzer
d. automated telephone communication switchboard
e. electrocardiogram machine
f. programmable keyboard
g. standard monitor for a desktop computer
h. video game console
i. electronic proximity key finder



CHAPTER 2

Performance Concepts

Learning Objectives

After studying this chapter, you should be able to:

- Understand the key performance issues that relate to computer design.
- Explain the reasons for the move to multicore organization, and understand the trade-off between cache and processor resources on a single chip.
- Distinguish among multicore, MIC, and GPGPU organizations.
- Summarize some of the issues in computer performance assessment.
- Discuss the SPEC benchmarks.
- Explain the differences among arithmetic, harmonic, and geometric means.

This chapter addresses the issue of computer system performance. We begin with a consideration of the need for balanced utilization of computer resources, which provides a perspective that is useful throughout the book. Next we look at contemporary computer organization designs intended to provide performance to meet current and projected demand. Finally, we look at tools and models that have been developed to provide a means of assessing comparative computer system performance.

2.1 DESIGNING FOR PERFORMANCE

Year by year, the cost of computer systems continues to drop dramatically, while the performance and capacity of those systems continue to rise equally dramatically. Today's laptops have the computing power of an IBM mainframe from 10 or 15 years ago. Thus, we have virtually "free" computer power. Processors are so inexpensive that we now have microprocessors we throw away. The digital pregnancy test is an example (used once and then thrown away). And this continuing technological revolution has enabled the development of applications of astounding complexity and power. For example, desktop applications that require the great power of today's microprocessor-based systems include:

- Image processing
- Three-dimensional rendering
- Speech recognition
- Videoconferencing
- Multimedia authoring
- Voice and video annotation of files
- Simulation modeling

Workstation systems now support highly sophisticated engineering and scientific applications and have the capacity to support image and video applications. In addition, businesses are relying on increasingly powerful servers to handle transaction and database processing and to support massive client/server networks that have replaced the huge mainframe computer centers of yesteryear. As well, cloud

service providers use massive high-performance banks of servers to satisfy high-volume, high-transaction-rate applications for a broad spectrum of clients.

What is fascinating about all this from the perspective of computer organization and architecture is that, on the one hand, the basic building blocks for today's computer miracles are virtually the same as those of the IAS computer from over 50 years ago, while on the other hand, the techniques for squeezing the maximum performance out of the materials at hand have become increasingly sophisticated.

This observation serves as a guiding principle for the presentation in this book. As we progress through the various elements and components of a computer, two objectives are pursued. First, the book explains the fundamental functionality in each area under consideration, and second, the book explores those techniques required to achieve maximum performance. In the remainder of this section, we highlight some of the driving factors behind the need to design for performance.

Microprocessor Speed

What gives Intel x86 processors or IBM mainframe computers such mind-boggling power is the relentless pursuit of speed by processor chip manufacturers. The evolution of these machines continues to bear out Moore's law, described in Chapter 1. So long as this law holds, chipmakers can unleash a new generation of chips every three years—with four times as many transistors. In memory chips, this has quadrupled the capacity of **dynamic random-access memory (DRAM)**, still the basic technology for computer main memory, every three years. In microprocessors, the addition of new circuits, and the speed boost that comes from reducing the distances between them, has improved performance four- or fivefold every three years or so since Intel launched its x86 family in 1978.

But the raw speed of the microprocessor will not achieve its potential unless it is fed a constant stream of work to do in the form of computer instructions. Anything that gets in the way of that smooth flow undermines the power of the processor. Accordingly, while the chipmakers have been busy learning how to fabricate chips of greater and greater density, the processor designers must come up with ever more elaborate techniques for feeding the monster. Among the techniques built into contemporary processors are the following:

- **Pipelining:** The execution of an instruction involves multiple stages of operation, including fetching the instruction, decoding the opcode, fetching operands, performing a calculation, and so on. Pipelining enables a processor to work simultaneously on multiple instructions by performing a different phase for each of the multiple instructions at the same time. The processor overlaps operations by moving data or instructions into a conceptual pipe with all stages of the pipe processing simultaneously. For example, while one instruction is being executed, the computer is decoding the next instruction. This is the same principle as seen in an assembly line.
- **Branch prediction:** The processor looks ahead in the instruction code fetched from memory and predicts which branches, or groups of instructions, are likely to be processed next. If the processor guesses right most of the time, it can prefetch the correct instructions and buffer them so that the processor is kept busy. The more sophisticated examples of this strategy predict not just

the next branch but multiple branches ahead. Thus, branch prediction potentially increases the amount of work available for the processor to execute.

- **Superscalar execution:** This is the ability to issue more than one instruction in every processor clock cycle. In effect, multiple parallel pipelines are used.
- **Data flow analysis:** The processor analyzes which instructions are dependent on each other's results, or data, to create an optimized schedule of instructions. In fact, instructions are scheduled to be executed when ready, independent of the original program order. This prevents unnecessary delay.
- **Speculative execution:** Using branch prediction and data flow analysis, some processors speculatively execute instructions ahead of their actual appearance in the program execution, holding the results in temporary locations. This enables the processor to keep its execution engines as busy as possible by executing instructions that are likely to be needed.

These and other sophisticated techniques are made necessary by the sheer power of the processor. Collectively they make it possible to execute many instructions per processor cycle, rather than to take many cycles per instruction.

Performance Balance

While processor power has raced ahead at breakneck speed, other critical components of the computer have not kept up. The result is a need to look for performance balance: an adjustment/tuning of the organization and architecture to compensate for the mismatch among the capabilities of the various components.

The problem created by such mismatches is particularly critical at the interface between processor and main memory. While processor speed has grown rapidly, the speed with which data can be transferred between main memory and the processor has lagged badly. The interface between processor and main memory is the most crucial pathway in the entire computer because it is responsible for carrying a constant flow of program instructions and data between memory chips and the processor. If memory or the pathway fails to keep pace with the processor's insistent demands, the processor stalls in a wait state, and valuable processing time is lost.

A system architect can attack this problem in a number of ways, all of which are reflected in contemporary computer designs. Consider the following examples:

- Increase the number of bits that are retrieved at one time by making DRAMs "wider" rather than "deeper" and by using wide bus data paths.
- Change the DRAM interface to make it more efficient by including a cache[1] or other buffering scheme on the DRAM chip.
- Reduce the frequency of memory access by incorporating increasingly complex and efficient cache structures between the processor and main memory. This includes the incorporation of one or more caches on the processor chip as well as on an off-chip cache close to the processor chip.

[1]A cache is a relatively small fast memory interposed between a larger, slower memory and the logic that accesses the larger memory. The cache holds recently accessed data and is designed to speed up subsequent access to the same data. Caches are discussed in Chapter 4.

- Increase the interconnect bandwidth between processors and memory by using higher-speed buses and a hierarchy of buses to buffer and structure data flow.

Another area of design focus is the handling of I/O devices. As computers become faster and more capable, more sophisticated applications are developed that support the use of peripherals with intensive I/O demands. Figure 2.1 gives some examples of typical peripheral devices in use on personal computers and workstations. These devices create tremendous data throughput demands. While the current generation of processors can handle the data pumped out by these devices, there remains the problem of getting that data moved between processor and peripheral. Strategies here include caching and buffering schemes plus the use of higher-speed interconnection buses and more elaborate interconnection structures. In addition, the use of multiple-processor configurations can aid in satisfying I/O demands.

The key in all this is balance. Designers constantly strive to balance the throughput and processing demands of the processor components, main memory, I/O devices, and the interconnection structures. This design must constantly be rethought to cope with two constantly evolving factors:

- The rate at which performance is changing in the various technology areas (processor, buses, memory, peripherals) differs greatly from one type of element to another.
- New applications and new peripheral devices constantly change the nature of the demand on the system in terms of typical instruction profile and the data access patterns.

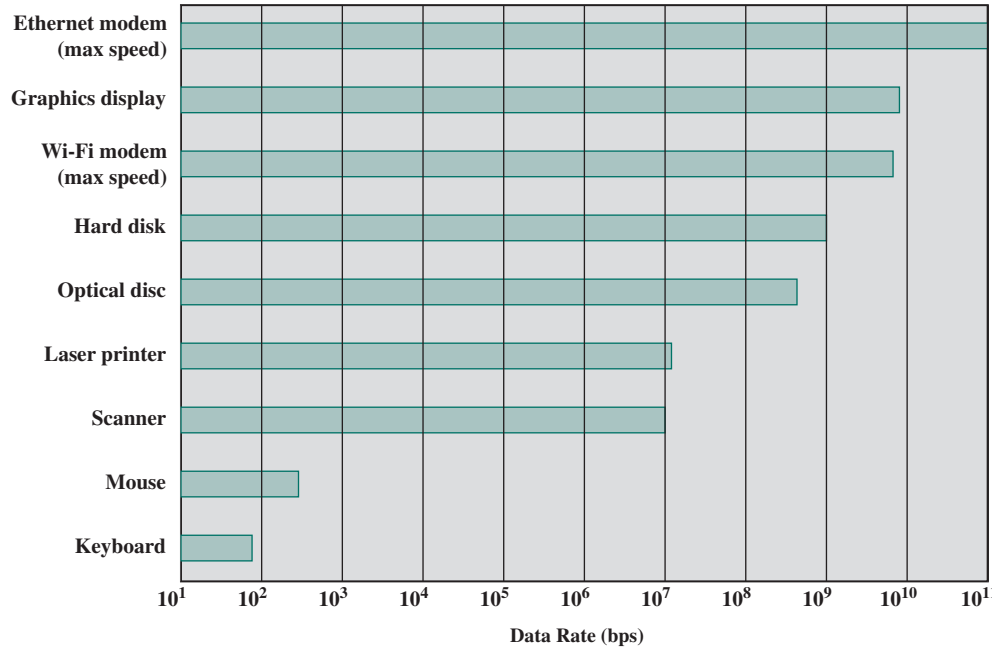


Figure 2.1 Typical I/O Device Data Rates

Thus, computer design is a constantly evolving art form. This book attempts to present the fundamentals on which this art form is based and to present a survey of the current state of that art.

Improvements in Chip Organization and Architecture

As designers wrestle with the challenge of balancing processor performance with that of main memory and other computer components, the need to increase processor speed remains. There are three approaches to achieving increased processor speed:

- Increase the hardware speed of the processor. This increase is fundamentally due to shrinking the size of the logic gates on the processor chip so that more gates can be packed together more tightly and to increasing the clock rate. With gates closer together, the propagation time for signals is significantly reduced, enabling a speeding up of the processor. An increase in clock rate means that individual operations are executed more rapidly.
- Increase the size and speed of caches that are interposed between the processor and main memory. In particular, by dedicating a portion of the processor chip itself to the cache, cache access times drop significantly.
- Make changes to the processor organization and architecture that increase the effective speed of instruction execution. Typically, this involves using parallelism in one form or another.

Traditionally, the dominant factor in performance gains has been increases in clock speed and logic density. However, as clock speed and logic density increase, a number of obstacles become more significant [INTE04]:

- **Power:** As the density of logic and the clock speed on a chip increase, so does the power density (Watts/cm^2). The difficulty of dissipating the heat generated on high-density, high-speed chips is becoming a serious design issue [GIBB04, BORK03].
- **RC delay:** The speed at which electrons can flow on a chip between transistors is limited by the resistance and capacitance of the metal wires connecting them; specifically, delay increases as the RC product increases. As components on the chip decrease in size, the wire interconnects become thinner, increasing resistance. Also, the wires are closer together, increasing capacitance.
- **Memory latency and throughput:** Memory access speed (latency) and transfer speed (throughput) lag processor speeds, as previously discussed.

Thus, there will be more emphasis on organization and architectural approaches to improving performance. These techniques are discussed in later chapters of the text.

Beginning in the late 1980s, and continuing for about 15 years, two main strategies have been used to increase performance beyond what can be achieved simply by increasing clock speed. First, there has been an increase in cache capacity. There are now typically two or three levels of cache between the processor and main memory. As chip density has increased, more of the cache memory has been incorporated on the chip, enabling faster cache access. For example, the original Pentium

chip devoted about 10% of on-chip area to a cache. Contemporary chips devote over half of the chip area to caches. And, typically, about three-quarters of the other half is for pipeline-related control and buffering.

Second, the instruction execution logic within a processor has become increasingly complex to enable parallel execution of instructions within the processor. Two noteworthy design approaches have been pipelining and superscalar. A pipeline works much like an assembly line in a manufacturing plant, enabling different stages of execution of different instructions to occur at the same time along the pipeline. A superscalar approach, in essence, allows multiple pipelines within a single processor, so that instructions that do not depend on one another can be executed in parallel.

By the mid to late 90s, both of these approaches were reaching a point of diminishing returns. The internal organization of contemporary processors is exceedingly complex and is able to squeeze a great deal of parallelism out of the instruction stream. It seems likely that further significant increases in this direction will be relatively modest [GIBB04]. With three levels of cache on the processor chip, each level providing substantial capacity, it also seems that the benefits from the cache are reaching a limit.

However, simply relying on increasing clock rate for increased performance runs into the power dissipation problem already referred to. The faster the clock rate, the greater the amount of power to be dissipated, and some fundamental physical limits are being reached.

Figure 2.2 illustrates the concepts we have been discussing.[2] The top line shows that, as per Moore's Law, the number of transistors on a single chip continues to

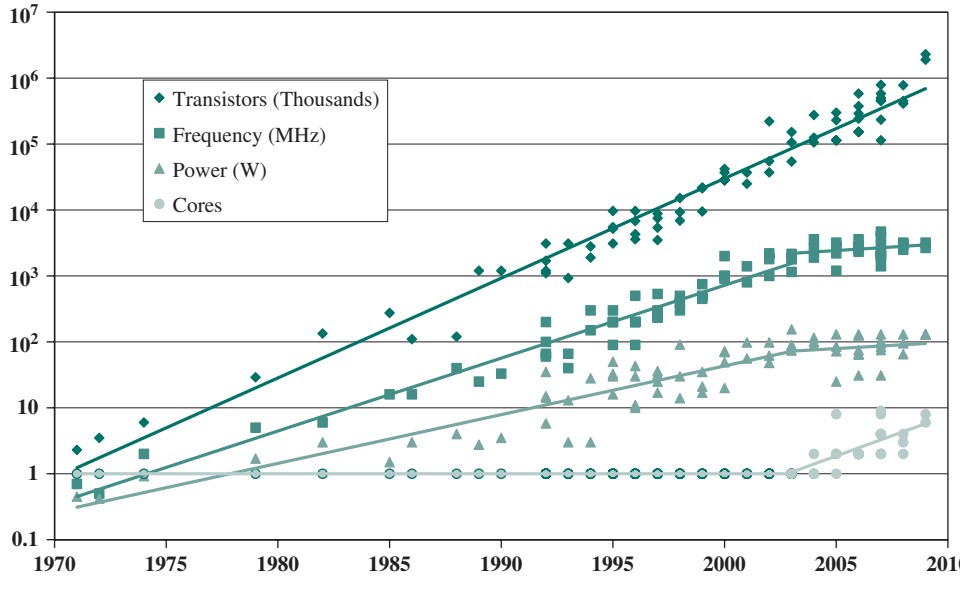


Figure 2.2 Processor Trends

Source: Graph provided by: Professor Kathy Yelick, Associate Laboratory Director for Computing Sciences Lawrence Berkeley National Laboratory, Computer Science Division University of California at Berkeley.

[2]I am grateful to Professor Kathy Yelick of UC Berkeley, who provided this graph.

grow exponentially.[3] Meanwhile, the clock speed has leveled off, in order to prevent a further rise in power. To continue increasing performance, designers have had to find ways of exploiting the growing number of transistors other than simply building a more complex processor. The response in recent years has been the development of the multicore computer chip.

2.2 MULTICORE, MICS, AND GPGPUS

With all of the difficulties cited in the preceding section in mind, designers have turned to a fundamentally new approach to improving performance: placing multiple processors on the same chip, with a large shared cache. The use of multiple processors on the same chip, also referred to as multiple cores, or **multicore**, provides the potential to increase performance without increasing the clock rate. Studies indicate that, within a processor, the increase in performance is roughly proportional to the square root of the increase in complexity [BORK03]. But if the software can support the effective use of multiple processors, then doubling the number of processors almost doubles performance. Thus, the strategy is to use two simpler processors on the chip, rather than one more complex processor.

In addition, with two processors larger caches are justified. This is important because the power consumption of memory logic on a chip is much less than that of processing logic.

As the logic density on chips continues to rise, the trend for both more cores and more cache on a single chip continues. Two-core chips were quickly followed by four-core chips, then 8, then 16, and so on. As the caches became larger, it made performance sense to create two and then three levels of cache on a chip, with the first-level cache initially dedicated to an individual processor, and levels two and three being shared by all the processors. It is now common for the second-level cache to also be private to each core.

Chip manufacturers are now in the process of making a huge leap forward in the number of cores per chip, with more than 50 cores per chip. The leap in performance as well as the challenges in developing software to exploit such a large number of cores has led to the introduction of a new term: **many integrated core (MIC)**.

The multicore and MIC strategy involves a homogeneous collection of general-purpose processors on a single chip. At the same time, chip manufacturers are pursuing another design option: a chip with multiple general-purpose processors plus **graphics processing units (GPUs)** and specialized cores for video processing and other tasks. In broad terms, a GPU is a core designed to perform parallel operations on graphics data. Traditionally found on a plug-in graphics card (display adapter), it is used to encode and render 2D and 3D graphics as well as process video.

Since GPUs perform parallel operations on multiple sets of data, they are increasingly being used as vector processors for a variety of applications that require repetitive computations. This blurs the line between the GPU and the CPU

[3]The observant reader will note that the transistor count values in this figure are significantly less than those of Figure 1.12. That latter figure shows the transistor count for a form of main memory known as DRAM (discussed in Chapter 5), which supports higher transistor density than processor chips.

[AROR12, FATA08, PROP11]. When a broad range of applications are supported by such a processor, the term **general-purpose computing on GPUs (GPGPU)** is used.

We explore design characteristics of multicore computers in Chapter 18 and GPGPUs in Chapter 19.

2.3 TWO LAWS THAT PROVIDE INSIGHT: AHMDAHL'S LAW AND LITTLE'S LAW

In this section, we look at two equations, called "laws." The two laws are unrelated, but both provide insight into the performance of parallel systems and multicore systems.

Amdahl's Law

Computer system designers look for ways to improve system performance by advances in technology or change in design. Examples include the use of parallel processors, the use of a memory cache hierarchy, and speedup in memory access time and I/O transfer rate due to technology improvements. In all of these cases, it is important to note that a speedup in one aspect of the technology or design does not result in a corresponding improvement in performance. This limitation is succinctly expressed by Amdahl's law.

Amdahl's law was first proposed by Gene Amdahl in 1967 ([AMDA67], [AMDA13]) and deals with the potential speedup of a program using multiple processors compared to a single processor. Consider a program running on a single processor such that a fraction $(1 - f)$ of the execution time involves code that is inherently sequential, and a fraction f that involves code that is infinitely parallelizable with no scheduling overhead. Let T be the total execution time of the program using a single processor. Then the speedup using a parallel processor with N processors that fully exploits the parallel portion of the program is as follows:

$$\text{Speedup} = \frac{\text{Time to execute program on a single processor}}{\text{Time to execute program on } N \text{ parallel processors}}$$

$$= \frac{T(1-f) + Tf}{T(1-f) + \frac{Tf}{N}} = \frac{1}{(1-f) + \frac{f}{N}}$$

This equation is illustrated in Figures 2.3 and 2.4. Two important conclusions can be drawn:

1. When f is small, the use of parallel processors has little effect.
2. As N approaches infinity, speedup is bound by $1/(1 - f)$, so that there are diminishing returns for using more processors.

These conclusions are too pessimistic, an assertion first put forward in [GUST88]. For example, a server can maintain multiple threads or multiple tasks to handle multiple clients and execute the threads or tasks in parallel up to the limit of the number of processors. Many database applications involve computations on massive amounts of data that can be split up into multiple parallel tasks.

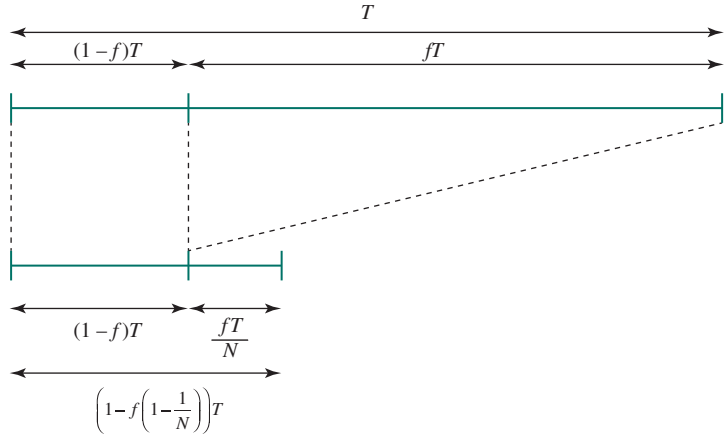


Figure 2.3 Illustration of Amdahl's Law

Nevertheless, Amdahl's law illustrates the problems facing industry in the development of multicore machines with an ever-growing number of cores: The software that runs on such machines must be adapted to a highly parallel execution environment to exploit the power of parallel processing.

Amdahl's law can be generalized to evaluate any design or technical improvement in a computer system. Consider any enhancement to a feature of a system that results in a speedup. The speedup can be expressed as

$$\text{Speedup} = \frac{\text{Performance after enhancement}}{\text{Performance before enhancement}} = \frac{\text{Execution time before enhancement}}{\text{Execution time after enhancement}} \tag{2.1}$$

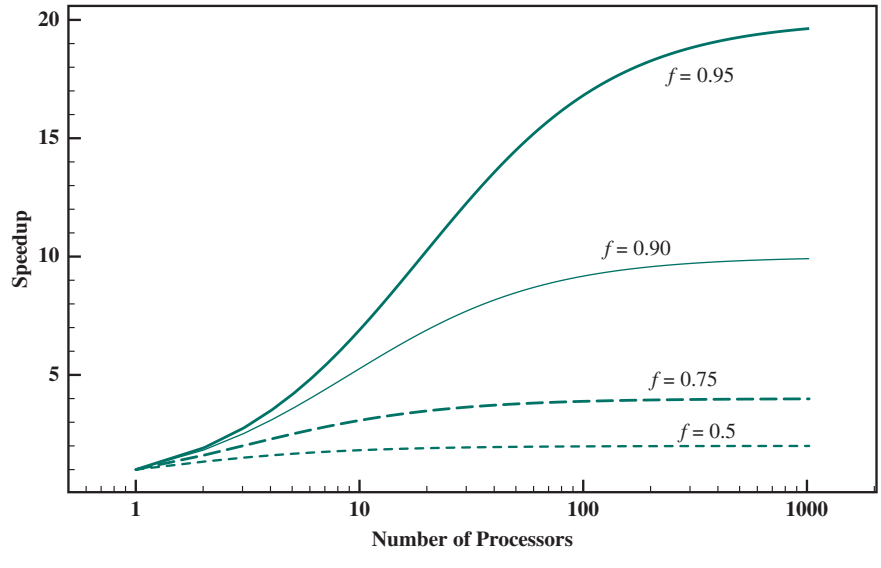


Figure 2.4 Amdahl's Law for Multiprocessors

Suppose that a feature of the system is used during execution a fraction of the time f, before enhancement, and that the speedup of that feature after enhancement is SU_f. Then the overall speedup of the system is

$$\text{Speedup} = \frac{1}{(1 - f) + \frac{f}{SU_f}}$$

EXAMPLE 2.1 Suppose that a task makes extensive use of floating-point operations, with 40% of the time consumed by floating-point operations. With a new hardware design, the floating-point module is sped up by a factor of K. Then the overall speedup is as follows:

$$\text{Speedup} = \frac{1}{0.6 + \frac{0.4}{K}}$$

Thus, independent of K, the maximum speedup is 1.67.

Little's Law

A fundamental and simple relation with broad applications is Little's Law [LITT61, LITT11].[4] We can apply it to almost any system that is statistically in steady state, and in which there is no leakage. Specifically, we have a steady state system to which items arrive at an average rate of λ items per unit time. The items stay in the system an average of W units of time. Finally, there is an average of L units in the system at any one time. Little's Law relates these three variables as $L = \lambda W$.

Using queuing theory terminology, Little's Law applies to a queuing system. The central element of the system is a server, which provides some service to items. Items from some population of items arrive at the system to be served. If the server is idle, an item is served immediately. Otherwise, an arriving item joins a waiting line, or queue. There can be a single queue for a single server, a single queue for multiple servers, or multiples queues, one for each of multiple servers. When a server has completed serving an item, the item departs. If there are items waiting in the queue, one is immediately dispatched to the server. The server in this model can represent anything that performs some function or service for a collection of items. Examples: A processor provides service to processes; a transmission line provides a transmission service to packets or frames of data; and an I/O device provides a read or write service for I/O requests.

[4]The second reference is a retrospective article on his law that Little wrote 50 years after his original paper. That must be unique in the history of the technical literature, although Amdahl comes close, with a 46-year gap between [AMDA67] and [AMDA13].

To understand Little's formula, consider the following argument, which focuses on the experience of a single item. When the item arrives, it will find on average L items ahead of it, one being serviced and the rest in the queue. When the item leaves the system after being serviced, it will leave behind on average the same number of items in the system, namely L, because L is defined as the average number of items waiting. Further, the average time that the item was in the system was W. Since items arrive at a rate of λ, we can reason that in the time W, a total of λW items must have arrived. Thus $L = \lambda W$.

To summarize, under steady state conditions, the average number of items in a queuing system equals the average rate at which items arrive multiplied by the average time that an item spends in the system. This relationship requires very few assumptions. We do not need to know what the service time distribution is, what the distribution of arrival times is, or the order or priority in which items are served. Because of its simplicity and generality, Little's Law is extremely useful and has experienced somewhat of a revival due to the interest in performance problems related to multicore computers.

A very simple example, from [LITT11], illustrates how Little's Law might be applied. Consider a multicore system, with each core supporting multiple threads of execution. At some level, the cores share a common memory. The cores share a common main memory and typically share a common cache memory as well. In any case, when a thread is executing, it may arrive at a point at which it must retrieve a piece of data from the common memory. The thread stops and sends out a request for that data. All such stopped threads are in a queue. If the system is being used as a server, an analyst can determine the demand on the system in terms of the rate of user requests, and then translate that into the rate of requests for data from the threads generated to respond to an individual user request. For this purpose, each user request is broken down into subtasks that are implemented as threads. We then have λ = the average rate of total thread processing required after all members' requests have been broken down into whatever detailed subtasks are required. Define L as the average number of stopped threads waiting during some relevant time. Then W = average response time. This simple model can serve as a guide to designers as to whether user requirements are being met and, if not, provide a quantitative measure of the amount of improvement needed.

2.4 BASIC MEASURES OF COMPUTER PERFORMANCE

In evaluating processor hardware and setting requirements for new systems, performance is one of the key parameters to consider, along with cost, size, security, reliability, and, in some cases, power consumption.

It is difficult to make meaningful performance comparisons among different processors, even among processors in the same family. Raw speed is far less important than how a processor performs when executing a given application. Unfortunately, application performance depends not just on the raw speed of the processor but also on the instruction set, choice of implementation language,

efficiency of the compiler, and skill of the programming done to implement the application.

In this section, we look at some traditional measures of processor speed. In the next section, we examine benchmarking, which is the most common approach to assessing processor and computer system performance. The following section discusses how to average results from multiple tests.

Clock Speed

Operations performed by a processor, such as fetching an instruction, decoding the instruction, performing an arithmetic operation, and so on, are governed by a system clock. Typically, all operations begin with the pulse of the clock. Thus, at the most fundamental level, the speed of a processor is dictated by the pulse frequency produced by the clock, measured in cycles per second, or Hertz (Hz).

Typically, clock signals are generated by a quartz crystal, which generates a constant sine wave while power is applied. This wave is converted into a digital voltage pulse stream that is provided in a constant flow to the processor circuitry (Figure 2.5). For example, a 1-GHz processor receives 1 billion pulses per second. The rate of pulses is known as the **clock rate**, or **clock speed**. One increment, or pulse, of the clock is referred to as a **clock cycle**, or a **clock tick**. The time between pulses is the **cycle time**.

The clock rate is not arbitrary, but must be appropriate for the physical layout of the processor. Actions in the processor require signals to be sent from one processor element to another. When a signal is placed on a line inside the processor, it takes some finite amount of time for the voltage levels to settle down so that an accurate value (logical 1 or 0) is available. Furthermore, depending on the physical layout of the processor circuits, some signals may change more rapidly than others. Thus, operations must be synchronized and paced so that the proper electrical signal (voltage) values are available for each operation.

The execution of an instruction involves a number of discrete steps, such as fetching the instruction from memory, decoding the various portions of the instruction, loading and storing data, and performing arithmetic and logical operations. Thus, most instructions on most processors require multiple clock cycles to complete. Some instructions may take only a few cycles, while others require dozens. In addition, when pipelining is used, multiple instructions are being executed simultaneously. Thus, a straight comparison of clock speeds on different processors does not tell the whole story about performance.

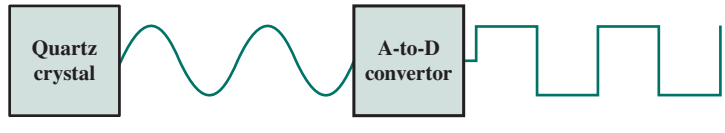


Figure 2.5 System Clock

Instruction Execution Rate

A processor is driven by a clock with a constant frequency f or, equivalently, a constant cycle time τ, where $\tau = 1/f$. Define the instruction count, I_c, for a program as the number of machine instructions executed for that program until it runs to completion or for some defined time interval. Note that this is the number of instruction executions, not the number of instructions in the object code of the program. An important parameter is the average cycles per instruction (*CPI*) for a program. If all instructions required the same number of clock cycles, then *CPI* would be a constant value for a processor. However, on any given processor, the number of clock cycles required varies for different types of instructions, such as load, store, branch, and so on. Let CPI_i be the number of cycles required for instruction type i, and I_i be the number of executed instructions of type i for a given program. Then we can calculate an overall *CPI* as follows:

$$CPI = \frac{\sum_{i=1}^{n}(CPI_i \times I_i)}{I_c} \tag{2.2}$$

The processor time T needed to execute a given program can be expressed as

$$T = I_c \times CPI \times \tau$$

We can refine this formulation by recognizing that during the execution of an instruction, part of the work is done by the processor, and part of the time a word is being transferred to or from memory. In this latter case, the time to transfer depends on the memory cycle time, which may be greater than the processor cycle time. We can rewrite the preceding equation as

$$T = I_c \times [p + (m \times k)] \times \tau$$

where p is the number of processor cycles needed to decode and execute the instruction, m is the number of memory references needed, and k is the ratio between memory cycle time and processor cycle time. The five performance factors in the preceding equation (I_c, p, m, k, τ) are influenced by four system attributes: the design of the instruction set (known as *instruction set architecture*); compiler technology (how effective the compiler is in producing an efficient machine language program from a high-level language program); processor implementation; and cache and memory hierarchy. Table 2.1 is a matrix in which one dimension shows the five performance factors and the other dimension shows the four system attributes. An X in a cell indicates a system attribute that affects a performance factor.

Table 2.1 Performance Factors and System Attributes

	I_c	p	m	k	τ
Instruction set architecture	X	X			
Compiler technology	X	X	X		
Processor implementation		X			X
Cache and memory hierarchy				X	X

A common measure of performance for a processor is the rate at which instructions are executed, expressed as millions of instructions per second (MIPS), referred to as the **MIPS rate**. We can express the MIPS rate in terms of the clock rate and *CPI* as follows:

$$\text{MIPS rate} = \frac{I_c}{T \times 10^6} = \frac{f}{CPI \times 10^6} \tag{2.3}$$

EXAMPLE 2.2 Consider the execution of a program that results in the execution of 2 million instructions on a 400-MHz processor. The program consists of four major types of instructions. The instruction mix and the *CPI* for each instruction type are given below, based on the result of a program trace experiment:

Instruction Type	*CPI*	Instruction Mix (%)
Arithmetic and logic	1	60
Load/store with cache hit	2	18
Branch	4	12
Memory reference with cache miss	8	10

The average *CPI* when the program is executed on a uniprocessor with the above trace results is $CPI = 0.6 + (2 \times 0.18) + (4 \times 0.12) + (8 \times 0.1) = 2.24$. The corresponding MIPS rate is $(400 \times 10^6)/(2.24 \times 10^6) \approx 178$.

Another common performance measure deals only with floating-point instructions. These are common in many scientific and game applications. Floating-point performance is expressed as millions of floating-point operations per second (MFLOPS), defined as follows:

$$\text{MFLOPS rate} = \frac{\text{Number of executed floating} - \text{point operations in a program}}{\text{Execution time} \times 10^6}$$

2.5 CALCULATING THE MEAN

In evaluating some aspect of computer system performance, it is often the case that a single number, such as execution time or memory consumed, is used to characterize performance and to compare systems. Clearly, a single number can provide only a very simplified view of a system's capability. Nevertheless, and especially in the field of benchmarking, single numbers are typically used for performance comparison [SMIT88].

As is discussed in Section 2.6, the use of benchmarks to compare systems involves calculating the mean value of a set of data points related to execution time. It turns out that there are multiple alternative algorithms that can be used for calculating a mean value, and this has been the source of some controversy in

the benchmarking field. In this section, we define these alternative algorithms and comment on some of their properties. This prepares us for a discussion in the next section of mean calculation in benchmarking.

The three common formulas used for calculating a mean are arithmetic, geometric, and harmonic. Given a set of n real numbers $(x_1, x_2, \ldots, x_n)$, the three means are defined as follows:

Arithmetic mean

$$AM = \frac{x_1 + \cdots + x_n}{n} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{2.4}$$

Geometric mean

$$GM = \sqrt[n]{x_1 \times \cdots \times x_n} = \left(\prod_{i=1}^{n} x_i\right)^{1/n} = e^{\left(\frac{1}{n}\sum_{i=1}^{n} \ln(x_i)\right)} \tag{2.5}$$

Harmonic mean

$$HM = \frac{n}{\left(\frac{1}{x_1}\right) + \cdots + \left(\frac{1}{x_n}\right)} = \frac{n}{\sum_{i=1}^{n}\left(\frac{1}{x_i}\right)} \qquad x_i > 0 \tag{2.6}$$

It can be shown that the following inequality holds:

$$AM \geq GM \geq HM$$

The values are equal only if $x_1 = x_2 = \ldots x_n$.

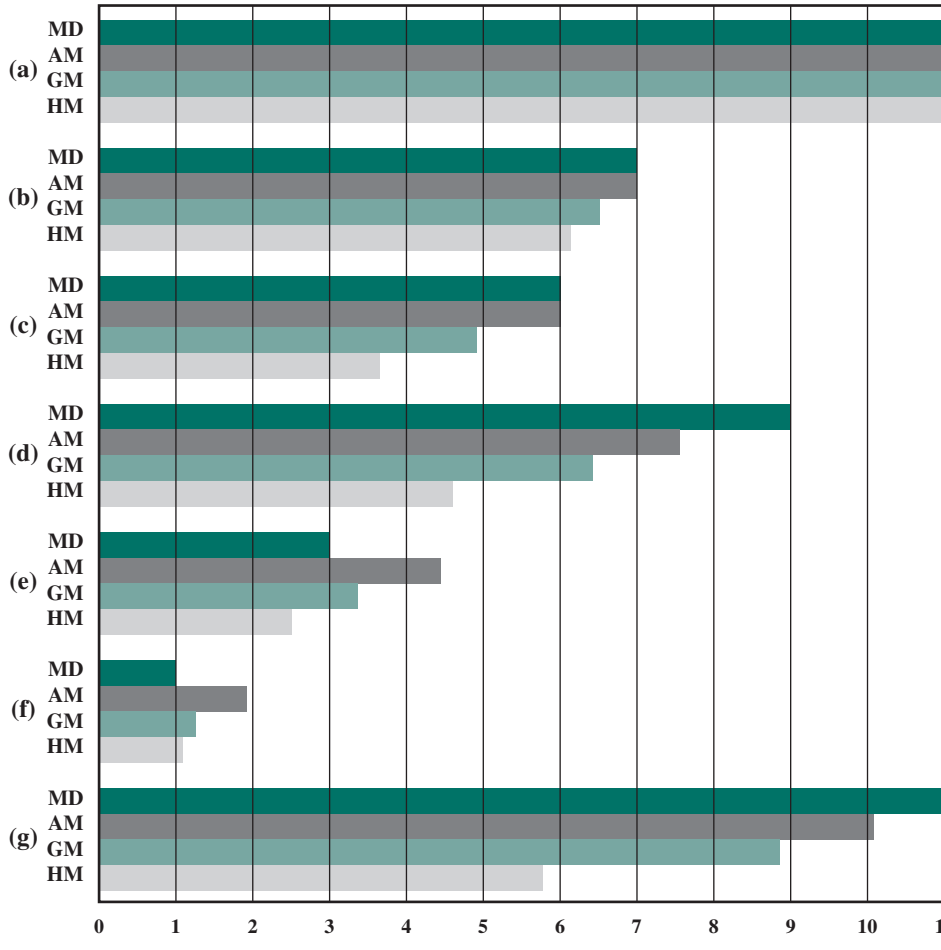
We can get a useful insight into these alternative calculations by defining the functional mean. Let $f(x)$ be a continuous monotonic function defined in the interval $0 \leq y < \infty$. The functional mean with respect to the function $f(x)$ for n positive real numbers $(x_1, x_2, \ldots, x_n)$ is defined as

Functional mean $$FM = f^{-1}\left(\frac{f(x_1) + \cdots + f(x_n)}{n}\right) = f^{-1}\left(\frac{1}{n}\sum_{i=1}^{n} f(x_i)\right)$$

where $f^{-1}(x)$ is the inverse of $f(x)$. The mean values defined in Equations (2.1) through (2.3) are special cases of the functional mean, as follows:

- AM is the FM with respect to $f(x) = x$
- GM is the FM with respect to $f(x) = \ln x$
- HM is the FM with respect to $f(x) = 1/x$

EXAMPLE 2.3 Figure 2.6 illustrates the three means applied to various data sets, each of which has eleven data points and a maximum data point value of 11. The median value is also included in the chart. Perhaps what stands out the most in this figure is that the HM has a tendency to produce a misleading result when the data is skewed to larger values or when there is a small-value outlier.



(a) Constant (11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11)
(b) Clustered around a central value (3, 5, 6, 6, 7, 7, 7, 8, 8, 9, 11)
(c) Uniform distribution (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
(d) Large-number bias (1, 4, 4, 7, 7, 9, 9, 10, 10, 11, 11)
(e) Small-number bias(1, 1, 2, 2, 3, 3, 5, 5, 8, 8, 11)
(f) Upper outlier (11, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
(g) Lower outlier (1, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11)

MD = median
AM = arithmetic mean
GM = geometric mean
HM = harmonic mean

Figure 2.6 Comparison of Means on Various Data Sets (each set has a maximum data point value of 11)

Let us now consider which of these means are appropriate for a given performance measure. As a preface to these remarks, it should be noted that a number of papers ([CITR06], [FLEM86], [GILA95], [JACO95], [JOHN04], [MASH04], [SMIT88]) and books ([HENN12], [HWAN93], [JAIN91], [LILJ00]) over the years have argued the pros and cons of the three means for performance analysis and come to conflicting conclusions. To simplify a complex controversy, we just note that the conclusions reached depend very much on the examples chosen and the way in which the objectives are stated.

Arithmetic Mean

An AM is an appropriate measure if the sum of all the measurements is a meaningful and interesting value. The AM is a good candidate for comparing the execution time performance of several systems. For example, suppose we were interested in using a system for large-scale simulation studies and wanted to evaluate several alternative products. On each system we could run the simulation multiple times with different input values for each run, and then take the average execution time across all runs. The use of multiple runs with different inputs should ensure that the results are not heavily biased by some unusual feature of a given input set. The AM of all the runs is a good measure of the system's performance on simulations, and a good number to use for system comparison.

The AM used for a time-based variable (e.g., seconds), such as program execution time, has the important property that it is directly proportional to the total time. So, if the total time doubles, the mean value doubles.

Harmonic Mean

For some situations, a system's execution rate may be viewed as a more useful measure of the value of the system. This could be either the instruction execution rate, measured in MIPS or MFLOPS, or a program execution rate, which measures the rate at which a given type of program can be executed. Consider how we wish the calculated mean to behave. It makes no sense to say that we would like the mean rate to be proportional to the total rate, where the total rate is defined as the sum of the individual rates. The sum of the rates would be a meaningless statistic. Rather, we would like the mean to be inversely proportional to the total execution time. For example, if the total time to execute all the benchmark programs in a suite of programs is twice as much for system C as for system D, we would want the mean value of the execution rate to be half as much for system C as for system D.

Let us look at a basic example and first examine how the AM performs. Suppose we have a set of n benchmark programs and record the execution times of each program on a given system as $t_1, t_2, \ldots, t_n$. For simplicity, let us assume that each program executes the same number of operations Z; we could weight the individual programs and calculate accordingly, but this would not change the conclusion of our argument. The execution rate for each individual program is $R_i = Z/t_i$. We use the AM to calculate the average execution rate.

$$AM = \frac{1}{n}\sum_{i=1}^{n} R_i = \frac{1}{n}\sum_{i=1}^{n} \frac{Z}{t_i} = \frac{Z}{n}\sum_{i=1}^{n} \frac{1}{t_i}$$

We see that the AM execution rate is proportional to the sum of the inverse execution times, which is not the same as being inversely proportional to the sum of the execution times. Thus, the AM does not have the desired property.

The HM yields the following result.

$$HM = \frac{n}{\sum_{i=1}^{n}\left(\frac{1}{R_i}\right)} = \frac{n}{\sum_{i=1}^{n}\left(\frac{1}{Z/t_i}\right)} = \frac{nZ}{\sum_{i=1}^{n} t_i}$$

The HM is inversely proportional to the total execution time, which is the desired property.

EXAMPLE 2.4 A simple numerical example will illustrate the difference between the two means in calculating a mean value of the rates, shown in Table 2.2. The table compares the performance of three computers on the execution of two programs. For simplicity, we assume that the execution of each program results in the execution of 10^8 floating-point operations. The left half of the table shows the execution times for each computer running each program, the total execution time, and the AM of the execution times. Computer A executes in less total time than B, which executes in less total time than C, and this is reflected accurately in the AM.

The right half of the table provides a comparison in terms of rates, expressed in MFLOPS. The rate calculation is straightforward. For example, program 1 executes 100 million floating-point operations. Computer A takes 2 seconds to execute the program for a MFLOPS rate of $100/2 = 50$. Next, consider the AM of the rates. The greatest value is for computer A, which suggests that A is the fastest computer. In terms of total execution time, A has the minimum time, so it is the fastest computer of the three. But the AM of rates shows B as slower than C, whereas in fact B is faster than C. Looking at the HM values, we see that they correctly reflect the speed ordering of the computers. This confirms that the HM is preferred when calculating rates.

The reader may wonder why go through all this effort. If we want to compare execution times, we could simply compare the total execution times of the three systems. If we want to compare rates, we could simply take the inverse of the total execution time, as shown in the table. There are two reasons for doing the individual calculations rather than only looking at the aggregate numbers:

Table 2.2 A Comparison of Arithmetic and Harmonic Means for Rates

	Computer A time (secs)	Computer B time (secs)	Computer C time (secs)	Computer A rate (MFLOPS)	Computer B rate (MFLOPS)	Computer C rate (MFLOPS)
Program 1 (10^8 FP ops)	2.0	1.0	0.75	50	100	133.33
Program 2 (10^8 FP ops)	0.75	2.0	4.0	133.33	50	25
Total execution time	2.75	3.0	4.75	—	—	—
Arithmetic mean of times	1.38	1.5	2.38	—	—	—
Inverse of total execution time (1/sec)	0.36	0.33	0.21	—	—	—
Arithmetic mean of rates	—	—	—	91.67	75.00	79.17
Harmonic mean of rates	—	—	—	72.72	66.67	42.11

1. A customer or researcher may be interested not only in the overall average performance but also performance against different types of benchmark programs, such as business applications, scientific modeling, multimedia applications, and systems programs. Thus, a breakdown by type of benchmark is needed, as well as a total.
2. Usually, the different programs used for evaluation are weighted differently. In Table 2.2, it is assumed that the two test programs execute the same number of operations. If that is not the case, we may want to weight accordingly. Or different programs could be weighted differently to reflect importance or priority.

Let us see what the result is if test programs are weighted proportional to the number of operations. Following the preceding notation, each program i executes Zi instructions in a time ti. Each rate is weighted by the instructions count. The weighted HM is therefore:

$$WHM = \frac{1}{\sum_{i=1}^{n}\left(\left(\frac{Z_i}{\sum_{j=1}^{n} Z_j}\right)\left(\frac{1}{R_i}\right)\right)} = \frac{n}{\sum_{i=1}^{n}\left(\left(\frac{Z_i}{\sum_{j=1}^{n} Z_j}\right)\left(\frac{t_i}{Z_i}\right)\right)} = \frac{\sum_{j=1}^{n} Z_j}{\sum_{i=1}^{n} t_i} \quad \textbf{(2.7)}$$

We see that the weighted HM is the quotient of the sum of the operation count divided by the sum of the execution times.

Geometric Mean

Looking at the equations for the three types of means, it is easier to get an intuitive sense of the behavior of the AM and the HM than that of the GM. Several observations from [FEIT15] may be helpful in this regard. First, we note that with respect to changes in values, the GM gives equal weight to all of the values in the data set. For example, suppose the set of data values to be averaged includes a few large values and more small values. Here, the AM is dominated by the large values. A change of 10% in the largest value will have a noticeable effect, while a change in the smallest value by the same factor will have a negligible effect. In contrast, a change in value by 10% of any of the data values results in the same change in the GM: $\sqrt[n]{1.1}$.

EXAMPLE 2.5 This point is illustrated by data set (e) in Figure 2.6. Here are the effects of increasing either the maximum or the minimum value in the data set by 10%:

	Geometric Mean	Arithmetic Mean
Original value	3.37	4.45
Increase max value from 11 to 12.1 (+10%)	3.40 (+ 0.87%)	4.55 (+ 2.24%)
Increase min value from 1 to 1.1 (+10%)	3.40 (+ 0.87%)	4.46 (+ 0.20%)

A second observation is that for the GM of a ratio, the GM of the ratios equals the ratio of the GMs:

$$GM = \left(\prod_{i=1}^{n} \frac{Z_i}{t_i}\right)^{1/n} = \frac{\left(\prod_{i=1}^{n} Z_i\right)^{1/n}}{\left(\prod_{i=1}^{n} t_i\right)^{1/n}} \tag{2.8}$$

Compare this with Equation 2.4.

For use with execution times, as opposed to rates, one drawback of the GM is that it may be non-monotonic relative to the more intuitive AM. In other words there may be cases where the AM of one data set is larger than that of another set, but the GM is smaller.

EXAMPLE 2.6 In Figure 2.6, the AM for data set d is larger than the AM for data set c, but the opposite is true for the GM.

	Data set c	Data set d
Arithmetic mean	7.00	7.55
Geometric mean	6.68	6.42

One property of the GM that has made it appealing for benchmark analysis is that it provides consistent results when measuring the relative performance of machines. This is in fact what benchmarks are primarily used for: to compare one machine with another in terms of performance metrics. The results, as we have seen, are expressed in terms of values that are normalized to a reference machine.

EXAMPLE 2.7 A simple example will illustrate the way in which the GM exhibits consistency for normalized results. In Table 2.3, we use the same performance results as were used in Table 2.2. In Table 2.3a, all results are normalized to Computer A, and the means are calculated on the normalized values. Based on total execution time, A is faster than B, which is faster than C. Both the AMs and GMs of the normalized times reflect this. In Table 2.3b, the systems are now normalized to B. Again the GMs correctly reflect the relative speeds of the three computers, but now the AM produces a different ordering.

Sadly, consistency does not always produce correct results. In Table 2.4, some of the execution times are altered. Once again, the AM reports conflicting results for the two normalizations. The GM reports consistent results, but the result is that B is faster than A and C, which are equal.

It is examples like this that have fueled the "benchmark means wars" in the citations listed earlier. It is safe to say that no single number can provide all the information that one needs for comparing performance across systems. However,

Table 2.3 A Comparison of Arithmetic and Geometric Means for Normalized Results

(a) Results normalized to Computer A

	Computer A time	Computer B time	Computer C time
Program 1	2.0 (1.0)	1.0 (0.5)	0.75 (0.38)
Program 2	0.75 (1.0)	2.0 (2.67)	4.0 (5.33)
Total execution time	2.75	3.0	4.75
Arithmetic mean of normalized times	1.00	1.58	2.85
Geometric mean of normalized times	1.00	1.15	1.41

(b) Results normalized to Computer B

	Computer A time	Computer B time	Computer C time
Program 1	2.0 (2.0)	1.0 (1.0)	0.75 (0.75)
Program 2	0.75 (0.38)	2.0 (1.0)	4.0 (2.0)
Total execution time	2.75	3.0	4.75
Arithmetic mean of normalized times	1.19	1.00	1.38
Geometric mean of normalized times	0.87	1.00	1.22

Table 2.4 Another Comparison of Arithmetic and Geometric Means for Normalized Results

(a) Results normalized to Computer A

	Computer A time	Computer B time	Computer C time
Program 1	2.0 (1.0)	1.0 (0.5)	0.20 (0.1)
Program 2	0.4 (1.0)	2.0 (5.0)	4.0 (10.0)
Total execution time	2.4	3.00	4.2
Arithmetic mean of normalized times	1.00	2.75	5.05
Geometric mean of normalized times	1.00	1.58	1.00

(b) Results normalized to Computer B

	Computer A time	Computer B time	Computer C time
Program 1	2.0 (2.0)	1.0 (1.0)	0.20 (0.2)
Program 2	0.4 (0.2)	2.0 (1.0)	4.0 (2.0)
Total execution time	2.4	3.00	4.2
Arithmetic mean of normalized times	1.10	1.00	1.10
Geometric mean of normalized times	0.63	1.00	0.63

despite the conflicting opinions in the literature, SPEC has chosen to use the GM, for several reasons:

1. As mentioned, the GM gives consistent results regardless of which system is used as a reference. Because benchmarking is primarily a comparison analysis, this is an important feature.
2. As documented in [MCMA93], and confirmed in subsequent analyses by SPEC analysts [MASH04], the GM is less biased by outliers than the HM or AM.
3. [MASH04] demonstrates that distributions of performance ratios are better modeled by lognormal distributions than by normal ones, because of the generally skewed distribution of the normalized numbers. This is confirmed in [CITR06]. And, as shown in Equation (2.5), the GM can be described as the back-transformed average of a lognormal distribution.

2.6 BENCHMARKS AND SPEC

Benchmark Principles

Measures such as MIPS and MFLOPS have proven inadequate to evaluating the performance of processors. Because of differences in instruction sets, the instruction execution rate is not a valid means of comparing the performance of different architectures.

EXAMPLE 2.8 Consider this high-level language statement:

```
A = B + C   /* assume all quantities in main memory */
```

With a traditional instruction set architecture, referred to as a complex instruction set computer (CISC), this instruction can be compiled into one processor instruction:

```
add   mem(B), mem(C), mem (A)
```

On a typical RISC machine, the compilation would look something like this:

```
load  mem(B), reg(1);
load  mem(C), reg(2);
add   reg(1), reg(2), reg(3);
store reg(3), mem (A)
```

Because of the nature of the RISC architecture (discussed in Chapter 15), both machines may execute the original high-level language instruction in about the same time. If this example is representative of the two machines, then if the CISC machine is rated at 1 MIPS, the RISC machine would be rated at 4 MIPS. But both do the same amount of high-level language work in the same amount of time.

Another consideration is that the performance of a given processor on a given program may not be useful in determining how that processor will perform on a very different type of application. Accordingly, beginning in the late 1980s and early 1990s, industry and academic interest shifted to measuring the performance of systems using a set of benchmark programs. The same set of programs can be run on

different machines and the execution times compared. Benchmarks provide guidance to customers trying to decide which system to buy, and can be useful to vendors and designers in determining how to design systems to meet benchmark goals.

[WEIC90] lists the following as desirable characteristics of a benchmark program:

1. It is written in a high-level language, making it portable across different machines.
2. It is representative of a particular kind of programming domain or paradigm, such as systems programming, numerical programming, or commercial programming.
3. It can be measured easily.
4. It has wide distribution.

SPEC Benchmarks

The common need in industry and academic and research communities for generally accepted computer performance measurements has led to the development of standardized benchmark suites. A benchmark suite is a collection of programs, defined in a high-level language, that together attempt to provide a representative test of a computer in a particular application or system programming area. The best known such collection of benchmark suites is defined and maintained by the Standard Performance Evaluation Corporation (SPEC), an industry consortium. This organization defines several benchmark suites aimed at evaluating computer systems. SPEC performance measurements are widely used for comparison and research purposes.

The best known of the SPEC benchmark suites is SPEC CPU2017. This is the industry standard suite for processor-intensive applications. That is, SPEC CPU2017 is appropriate for measuring performance for applications that spend most of their time doing computation rather than I/O.

Other SPEC suites include the following:

- **SPEC Cloud_IaaS:** Benchmark addresses the performance of infrastructure-as-a-service (IaaS) public or private cloud platforms.
- **SPECviewperf:** Standard for measuring 3D graphics performance based on professional applications.
- **SPECwpc:** benchmark to measure all key aspects of workstation performance based on diverse professional applications, including media and entertainment, product development, life sciences, financial services, and energy.
- **SPECjvm2008:** Intended to evaluate performance of the combined hardware and software aspects of the Java Virtual Machine (JVM) client platform.
- **SPECjbb2015 (Java Business Benchmark):** A benchmark for evaluating server-side Java-based electronic commerce applications.
- **SPECsfs2014:** Designed to evaluate the speed and request-handling capabilities of file servers.
- **SPECvirt_sc2013:** Performance evaluation of datacenter servers used in virtualized server consolidation. Measures the end-to-end performance of all system components including the hardware, virtualization platform, and the virtualized guest operating system and application software. The benchmark supports hardware virtualization, operating system virtualization, and hardware partitioning schemes.

The CPU2017 suite is based on existing applications that have already been ported to a wide variety of platforms by SPEC industry members. In order to make the benchmark results reliable and realistic, the CPU2017 benchmarks are drawn from real-life applications, rather than using artificial loop programs or synthetic benchmarks. The suite consists of 20 integer benchmarks and 23 floating-point benchmarks written in C, C++, and Fortran (Table 2.5). For all of the integer benchmarks

Table 2.5 SPEC CPU2017 Benchmarks

(a) Integer

Rate	Speed	Language	Kloc	Application Area
500.perlbench_r	600.perlbench_s	C	363	Perl interpreter
502.gcc_r	602.gcc_s	C	1304	GNU C compiler
505.mcf_r	605.mcf_s	C	3	Route planning
520.omnetpp_r	620.omnetpp_s	C++	134	Discrete event simulation - computer network
523.xalancbmk_r	623.xalancbmk_s	C++	520	XML to HTML conversion via XSLT
525.x264_r	625.x264_s	C	96	Video compression
531.deepsjeng_r	631.deepsjeng_s	C++	10	AI: alpha-beta tree search (chess)
541.leela_r	641.leela_s	C++	21	AI: Monte Carlo tree search (Go)
548.exchange2_r	648.exchange2_s	Fortran	1	AI: recursive solution generator (Sudoku)
557.xz_r	657.xz_s	C	33	General data compression

(b) Floating Point

503.bwaves_r	603.bwaves_s	Fortran	1	Explosion modeling
507.cactuBSSN_r	607.cactuBSSN_s	C++, C, Fortran	257	Physics; relativity
508.namd_r		C++, C	8	Molecular dynamics
510.parest_r		C++	427	Biomedical imaging; optical tomography with finite elements
511.povray_r		C++	170	Ray tracing
519.ibm_r	619.ibm_s	C	1	Fluid dynamics
521.wrf_r	621.wrf_s	Fortran, C	991	Weather forecasting
526.blender_r		C++	1577	3D rendering and animation
527.cam4_r	627.cam4_s	Fortran, C	407	Atmosphere modeling
	628.pop2_s	Fortran, C	338	Wide-scale ocean modeling (climate level)
538.imagick_r	638.imagick_s	C	259	Image manipulation
544.nab_r	644.nab_s	C	24	Molecular dynamics
549.fotonik3d_r	649.fotonik3d_s	Fortran	14	Computational electromagnetics
554.roms_r	654.roms_s	Fortran	210	Regional ocean modeling.

Kloc = line count (including comments/whitespace) for source files used in a build/1000

and most of the floating-point benchmarks, there are both rate and speed benchmark programs. The differences between corresponding rate and speed benchmarks include workload sizes, compile flags, and run rules. The suite contains over 11 million lines of code. This is the sixth generation of processor-intensive suites from SPEC; the fifth generation was CPU2006. CPU2017 is designed to provide a contemporary set of benchmarks that reflect the dramatic changes in workload and performance requirements in the 11 years since CPU2006 [MOOR17].

To better understand published results of a system using CPU2017, we define the following terms used in the SPEC documentation:

- **Benchmark:** A program written in a high-level language that can be compiled and executed on any computer that implements the compiler.
- **System under test:** This is the system to be evaluated.
- **Reference machine:** This is a system used by SPEC to establish a baseline performance for all benchmarks. Each benchmark is run and measured on this machine to establish a reference time for that benchmark. A system under test is evaluated by running the CPU2017 benchmarks and comparing the results for running the same programs on the reference machine.
- **Base metric:** These are required for all reported results and have strict guidelines for compilation. In essence, the standard compiler with more or less default settings should be used on each system under test to achieve comparable results.
- **Peak metric:** This enables users to attempt to optimize system performance by optimizing the compiler output. For example, different compiler options may be used on each benchmark, and feedback-directed optimization is allowed.
- **Speed metric:** This is simply a measurement of the time it takes to execute a compiled benchmark. The speed metric is used for comparing the ability of a computer to complete single tasks.
- **Rate metric:** This is a measurement of how many tasks a computer can accomplish in a certain amount of time; this is called a **throughput**, capacity, or rate measure. The rate metric allows the system under test to execute simultaneous tasks to take advantage of multiple processors.

SPEC uses a historical Sun system, the "Ultra Enterprise 2," which was introduced in 1997, as the reference machine. The reference machine uses a 296-MHz UltraSPARC II processor. It takes about 12 days to do a rule-conforming run of the base metrics for CINT2017 and CFP2017 on the CPU2017 reference machine. Tables 2.5 and 2.6 show the amount of time to run each benchmark using the reference machine. The tables also show the dynamic instruction counts on the reference machine, as reported in [PHAN07]. These values are the actual number of instructions executed during the run of each program.

We now consider the specific calculations that are done to assess a system. We consider the integer benchmarks; the same procedures are used to create a floating-point benchmark value. For the integer benchmarks, there are 12 programs in the test suite. Calculation is a three-step process (Figure 2.7):

1. The first step in evaluating a system under test is to compile and run each program on the system three times. For each program, the runtime is measured and the median value is selected. The reason to use three runs and take the median value is to account for variations in execution time that are not intrinsic to the program, such as disk access time variations, and OS kernel execution variations from one run to another.
2. Next, each of the 12 results is normalized by calculating the runtime ratio of the reference run time to the system run time. The ratio is calculated as follows:

$$r_i = \frac{Tref_i}{Tsut_i} \tag{2.9}$$

where $Tref_i$ is the execution time of benchmark program i on the reference system and $Tsut_i$ is the execution time of benchmark program i on the system under test. Thus, ratios are higher for faster machines.

3. Finally, the geometric mean of the 12 runtime ratios is calculated to yield the overall metric:

$$r_G = \left(\prod_{i=1}^{12} r_i\right)^{1/12}$$

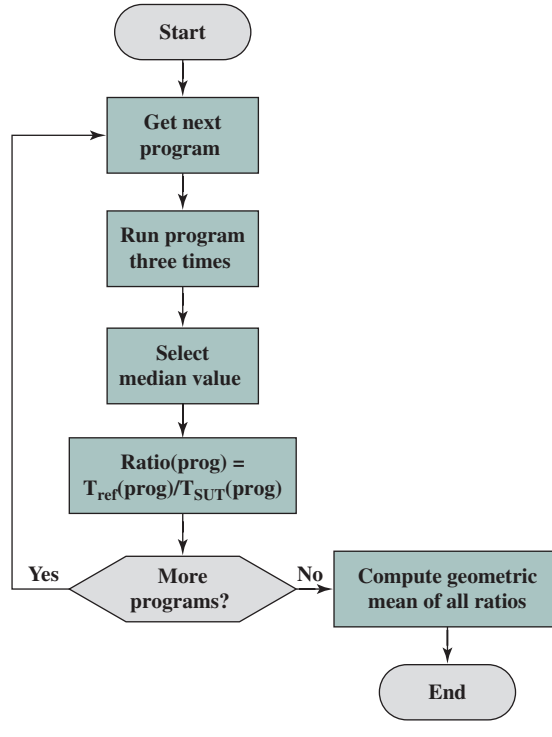


Figure 2.7 SPEC Evaluation Flowchart

For the integer benchmarks, four separate metrics can be calculated:

- **SPECspeed2017_int_base:** The geometric mean of 12 normalized ratios when the benchmarks are compiled with base tuning.
- **SPECspeed2017_int_peak:** The geometric mean of 12 normalized ratios when the benchmarks are compiled with peak tuning.
- **SPECrate2017_int_base:** The geometric mean of 12 normalized throughput ratios when the benchmarks are compiled with base tuning.
- **SPECrate2017_int_peak:** The geometric mean of 12 normalized throughput ratios when the benchmarks are compiled with peak tuning.

Table 2.6 shows the CPU2017 integer benchmarks reported for the HP Integrity Superdome X.

Table 2.6 SPEC CPU2017 Integer Benchmarks for HP Integrity Superdome X

(a) Rate Result (768 copies)

	Base		Peak	
Benchmark	**Seconds**	**Rate**	**Seconds**	**Rate**
500.perlbench_r	1141	1070	933	1310
502.gcc_r	1303	835	1276	852
505.mcf_r	1433	866	1378	901
520.omnetpp_r	1664	606	1634	617
523.xalancbmk_r	722	1120	713	1140
525.x264_r	655	2053	661	2030
531.deepsjeng_r	604	1460	597	1470
541.leela_r	892	1410	896	1420
548.exchange2_r	833	2420	770	2610
557.xz_r	870	953	863	961

(b) Speed Result (384 threads)

	Base		Peak	
Benchmark	**Seconds**	**Ratio**	**Seconds**	**Ratio**
600.perlbench_s	358	4.96	295	6.01
602.gcc_s	546	7.29	535	7.45
605.mcf_s	866	5.45	700	6.75
620.omnetpp_s	276	5.90	247	6.61
623.xalancbmk_s	188	7.52	179	7.91
625.x264_s	283	6.23	271	6.51
631.deepsjeng_s	407	3.52	343	4.18
641.leela_s	469	3.63	439	3.88
648.exchange2_s	329	8.93	299	9.82
657.xz_s	2164	2.86	2119	2.92

EXAMPLE 2.9 One of the SPEC CPU2017 integer speed benchmarks is 625.x264_s. This is an implementation of H.264/AVC (Advanced Video Coding), the commonly used video compression standard. The reference machine Sun Fire V490 executes this program in a median time of 1764 seconds for the base speed metric. The HP Integrity Superdome X requires 283 seconds. The ratio is calculated as: $1764/283 = 6.23$. Similar calculations are done to determine the ratios for the other benchmark programs. The SPECspeed2017_int_base speed metric is calculated by taking the tenth root of the product of the ratios:

$$(4.96 \times 7.29 \times 5.45 \times 5.90 \times 7.52 \times 6.23 \times 3.52 \times 3.63 \times 8.93 \times 2.86)^{1/10} = 5.31$$

The rate metrics take into account a system with multiple processors. To test a machine, a number of copies N is selected—usually this is equal to the number of processors or the number of simultaneous threads of execution on the test system. Each individual test program's rate is determined by taking the median of three runs. Each run consists of N copies of the program running simultaneously on the test system. The execution time is the time it takes for all the copies to finish (i.e., the time from when the first copy starts until the last copy finishes). The rate metric for that program is calculated by the following formula:

$$rate_i = N \times \frac{Tref_i}{Tsut_i}$$

The rate score for the system under test is determined from a geometric mean of rates for each program in the test suite.

EXAMPLE 2.10 The results for the HP Integrity Superdome X are shown in Table 2.6a. This system has 16 processor chips, with 24 cores per chip, for a total of 384 cores. Two threads are run per core so that a total of 768 copies of a program are run simultaneously. To get the rate metric, each benchmark program is executed simultaneously on all threads, with the execution time being the time from the start of all 768 copies to the end of the slowest run. The speed ratio is calculated as before, and the rate value is simply 384 times the speed ratio. For example, for the integer rate benchmark SPECrate2017_int_base, the reference machine report a speed of 1751 seconds, and the system under test reports a speed of 655 seconds. The rate is calculated as $768 \times (1751/655) = 2053$. The final rate metric is found by taking the geometric mean of the rate values:

$$(1070 \times 835 \times 866 \times 606 \times 1120 \times 2053 \times 1460 \times 1410 \times 2420 \times 953)^{1/10} = 1223$$

SPEC CPU2017 introduces an additional, experimental, metric that enables measurement of power consumption while running the benchmark, giving users insight into the relationship between performance and power. A vendor can measure and report power statistics, including maximum power (W), average power (W), and total energy used (kJ) and compare these to the reference machine. The results for the reference machine are shown in Table 2.7.

Table 2.7 SPECspeed2017_int_base Benchmark Results for Reference Machine (1 thread)

Benchmark	Seconds	Energy (kJ)	Average Power (W)	Maximum Power (W)
600.perlbench_s	1774	1920	1080	1090
602.gcc_s	3981	4330	1090	1110
605.mcf_s	4721	5150	1090	1120
620.omnetpp_s	1630	1770	1090	1090
623.xalancbmk_s	1417	1540	1090	1090
625.x264_s	1764	1920	1090	1100
631.deepsjeng_s	1432	1560	1090	1130
641.leela_s	1706	1850	1090	1090
648.exchange2_s	2939	3200	1080	1090
657.xz_s	6182	6730	1090	1140

2.7 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

Amdahl's law	functional mean (FM)	microprocessor
arithmetic mean (AM)	general-purpose computing on GPU (GPGPU)	MIPS rate
base metric	geometric mean (GM)	multicore
benchmark	graphics processing unit (GPU)	peak metric
clock cycle	harmonic mean (HM)	rate metric
clock cycle time	instruction execution rate	reference machine
clock rate	Little's law	speed metric
clock speed	many integrated core (MIC)	SPEC
clock tick		system under test
cycles per instruction (*CPI*)		throughput

Review Questions

2.1 List and briefly discuss the obstacles that arise when clock speed and logic density increase.

2.2 What are the advantages of using a cache?

2.3 Briefly describe some of the methods used to increase processor speed.

2.4 What obstacles become significant in CPU performance gains as clock speed and logic density increase?

2.5 Define clock rate. Is it similar to clock speed?

2.6 Why are MIPS and FLOPS inadequate for evaluating processor performance?

2.7 When is the harmonic mean an appropriate measure of the value of a system?

2.8 Explain each variable that is related to Little's Law.

2.9 Why are geometric means superior to arithmetic means for benchmarking?

2.10 While evaluating a system's performance using SPEC benchmarks, why is each program compiled and run three times?