WILLIAM STALLINGS

COMPUTER ORGANIZATION AND ARCHITECTURE

Designing for Performance

Pearson

Eleventh Edition

Computer Organization and Architecture

Designing for Performance

Eleventh Edition

Computer Organization and Architecture

Designing for Performance

Eleventh Edition

William Stallings

330 Hudson Street, New York, NY 10013

Senior Vice President Courseware Portfolio Management: *Marcia J. Horton*

Director, Portfolio Management: Engineering, Computer Science & Global Editions: *Julian Partridge*

Executive Portfolio Manager: *Tracy Johnson*

Portfolio Management Assistant: *Meghan Jacoby*

Managing Content Producer: *Scott Disanno*

Content Producer: *Amanda Brands*

R&P Manager: *Ben Ferrini*

Manufacturing Buyer, Higher Ed, Lake Side Communications, Inc. (LSC): *Maura Zaldivar-Garcia*

Inventory Manager: *Bruce Boundy*

Field Marketing Manager: *Demetrius Hall*

Product Marketing Manager: *Yvonne Vannatta*

Marketing Assistant: *Jon Bryant*

Cover Designer: *Black Horse Designs*

Cover Art: *Shuttersstock/Shimon Bar*

Full-Service Project Management: *Kabilan Selvakumar, SPi Global*

Printer/Binder: *LSC Communications, Inc.*

Library of Congress Cataloging-in-Publication Data

Names: Stallings, William, author.

Title: Computer organization and architecture : designing for performance / William Stallings.

Description: Eleventh edition. | Hoboken : Pearson Education, 2019. | Includes bibliographical references and index.

Identifiers: LCCN 0134997190 | ISBN 9780134997193

Subjects: LCSH: Computer organization. | Computer architecture.

Classification: LCC QA76.9.C643 S73 2018 | DDC 004.2/2—dc23 LC record available at **https://lccn.loc.gov/**

1 18

ISBN-10: 0-13-499719-0

ISBN-13: 978-0-13-499719-3

To Tricia my loving wife, the kindest and gentlest person

Contents

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**
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-	—**Chapter 1** —**Chapter 8** —**Chapter 13** —**Chapter**

	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 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	—Chapter 16 —Chapter

	• Instruction pipelining • Introduction to instruction-level parallelism (ILP)	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.pearson.com/stallings**. To gain access to the IRC, please contact your local Pearson sales representative via **www.pearson.com/replocator**. 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.pearson.com/stallings**, includes a list of relevant links organized by chapter and an errata sheet for the book. 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.

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.

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.

Acronyms

ACM	Association for Computing Machinery
ALU	Arithmetic Logic Unit
ANSI	American National Standards Institute
ASCII	American Standards Code for Information Interchange
BCD	Binary Coded Decimal
CD	Compact Disk
CD-ROM	Compact Disk Read-Only Memory
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
DRAM	Dynamic Random-Access Memory
DMA	Direct Memory Access
DVD	Digital Versatile Disk
EEPROM	Electrically Erasable Programmable Read-Only Memory
EPIC	Explicitly Parallel Instruction Computing
EPROM	Erasable Programmable Read-Only Memory
HLL	High-Level Language

I/O	Input/Output
IAR	Instruction Address Register
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
ILP	Instruction-Level Parallelism
IR	Instruction Register
LRU	Least Recently Used
LSI	Large-scale Integration
MAR	Memory Address Register
MBR	Memory Buffer Register
MESI	Modify-Exclusive-Shared-Invalid
MIC	Many Integrated Core
MMU	Memory Management Unit
MSI	Medium-Scale Integration
NUMA	Nonuniform Memory Access
OS	Operating System
PC	Program Counter

PCB	Process Control Block
PCI	Peripheral Component Interconnect
PROM	Programmable Read-Only Memory
PSW	Processor Status Word
RAID	Redundant Array of Independent Disks
RALU	Register/Arithmetic-Logic Unit
RAM	Random-Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
SCSI	Small Computer System Interface
SMP	Symmetric Multiprocessors
SRAM	Static Random-Access Memory
SSI	Small-Scale Integration
ULSI	Ultra Large-Scale Integration
VLIW	Very Long Instruction Word
VLSI	Very Large-Scale Integration

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 text 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:

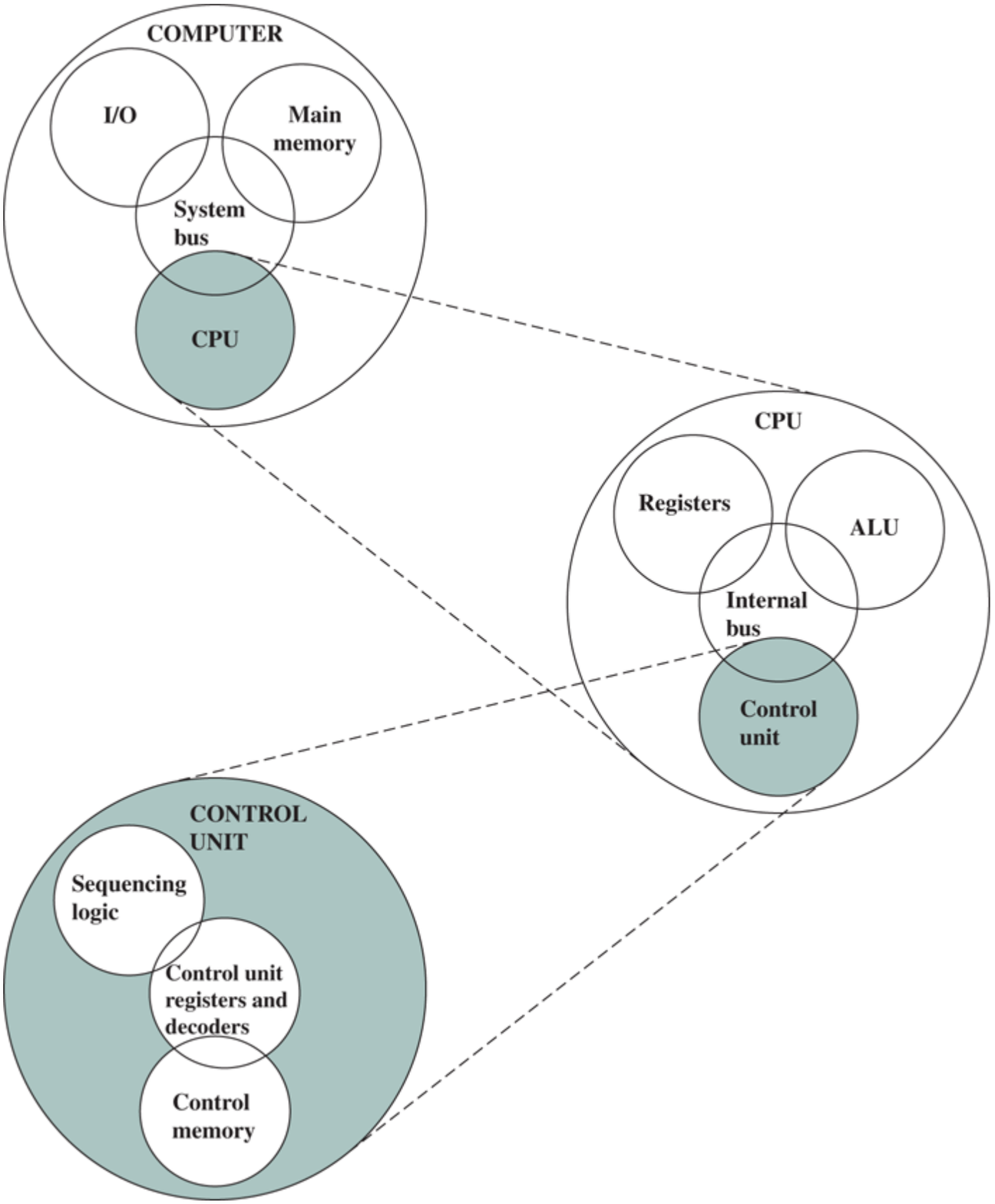


Figure 1.1 The Computer: Top-Level Structure

- **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 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 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.

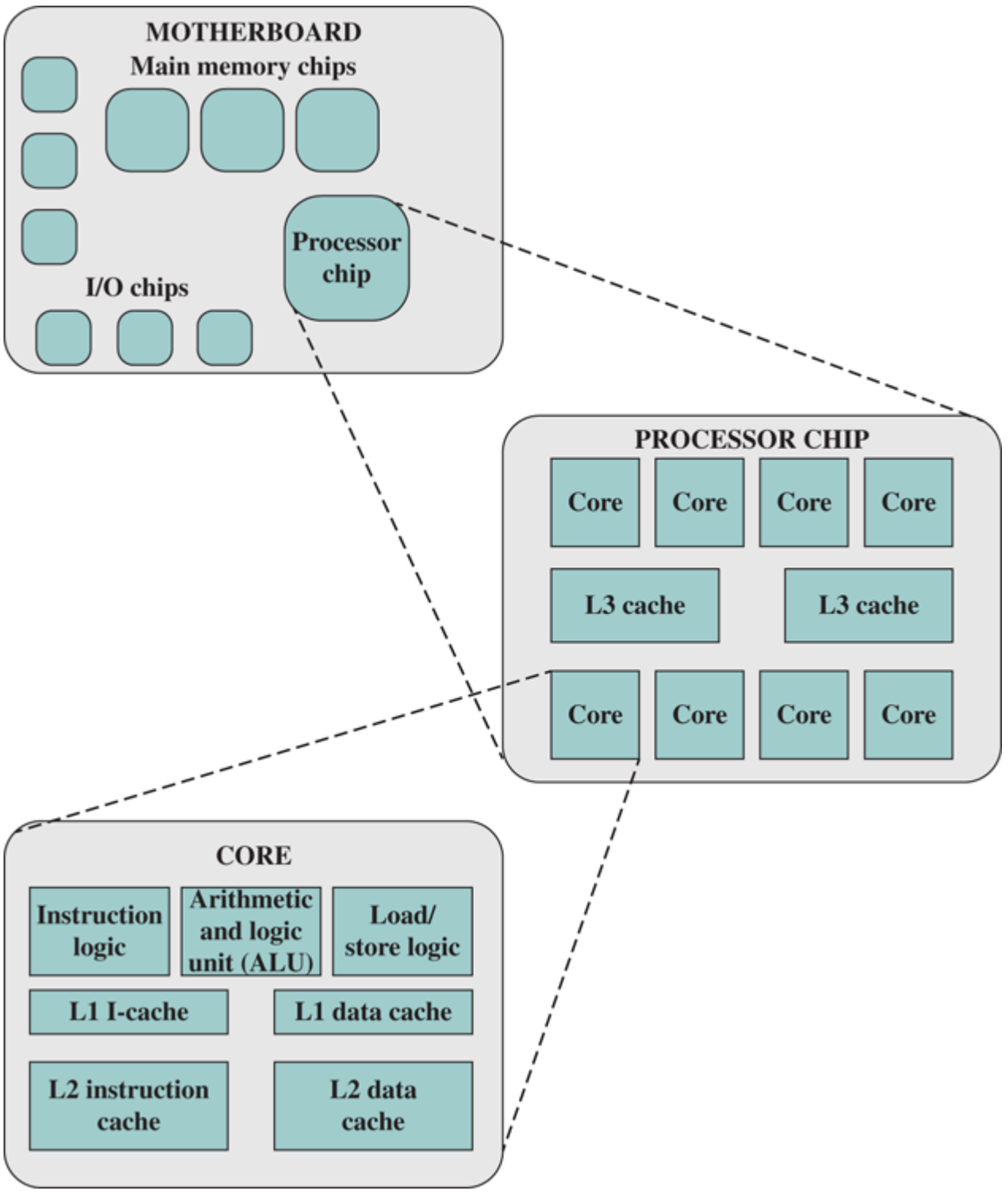


Figure 1.2 Simplified View of Major Elements of a Multicore Computer

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 look at some real-world examples that illustrate the hierarchical structure of computers. **Figure 1.3** is a photograph of the motherboard for a computer built around two Intel Quad-Core Xeon processor chips. Many of the elements labeled on the photograph are discussed subsequently in this book. Here, we mention the most important, in addition to the processor sockets:

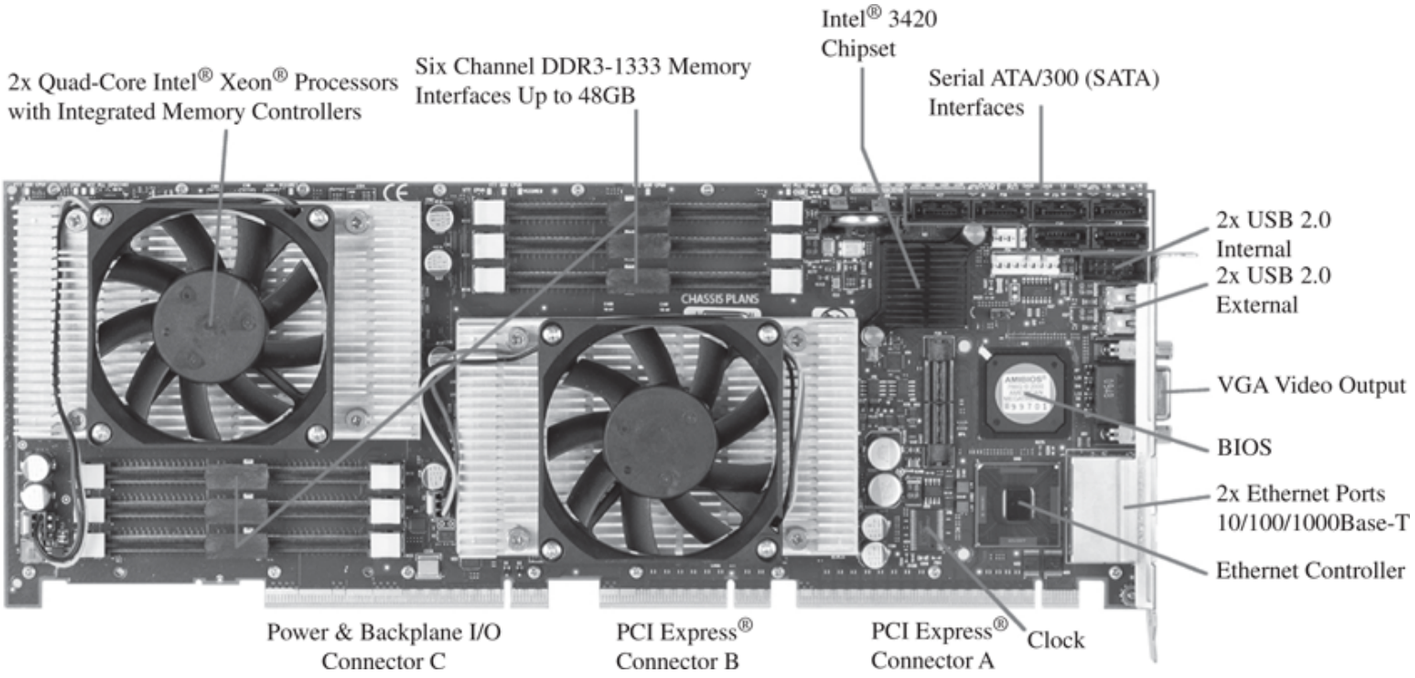


Figure 1.3 Motherboard with Two Intel Quad-Core Xeon Processors

Source: Courtesy of Chassis Plans Rugged Rackmount Computers

- 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 1.4** 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.

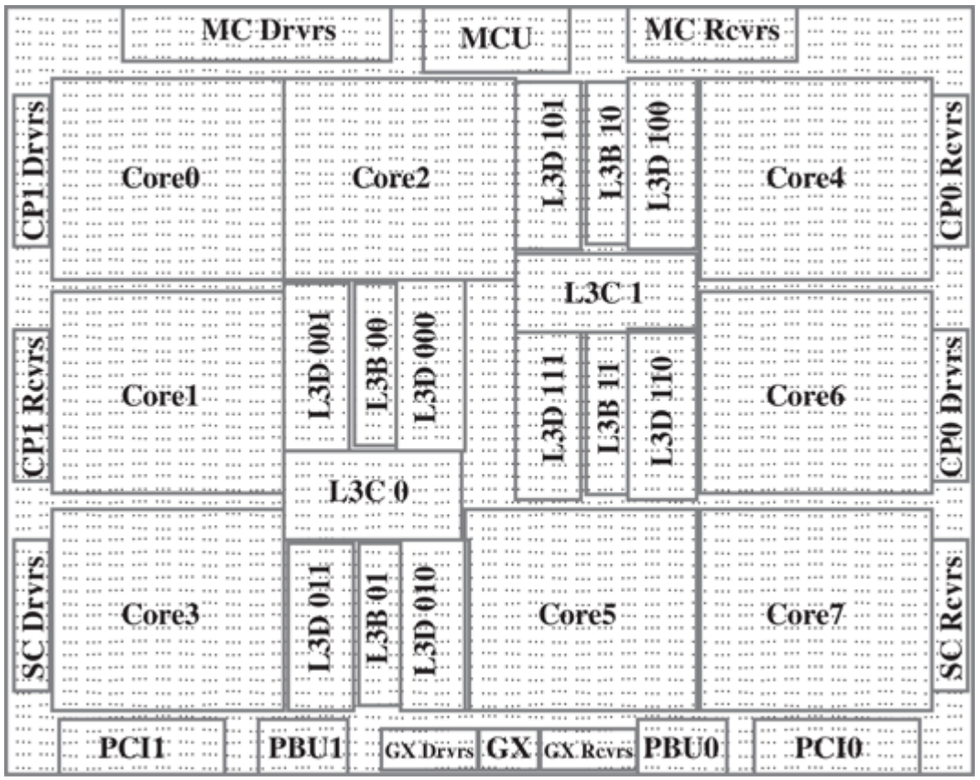


Figure 1.4 IBM z13 Processor Unit (PU) Chip Diagram

Going down one level deeper, we examine the internal structure of a single core, as shown in the photograph of **Figure 1.5**. 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 sub-areas within this core area are the following:

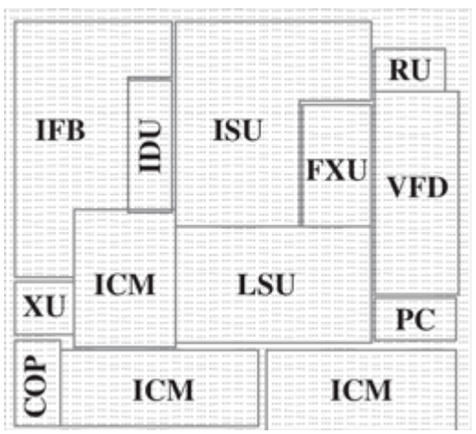


Figure 1.5 IBM z13 Core Layout

- **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.

[1]

kB = kilobyte = 1048 bytes. Numerical prefixes are explained in a document under the “Other Useful” tab at ComputerScienceStudent.com.

- **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 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]

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

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]

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

Figure 1.6 shows the structure of the IAS computer (compare with **Figure 1.1**). It consists of

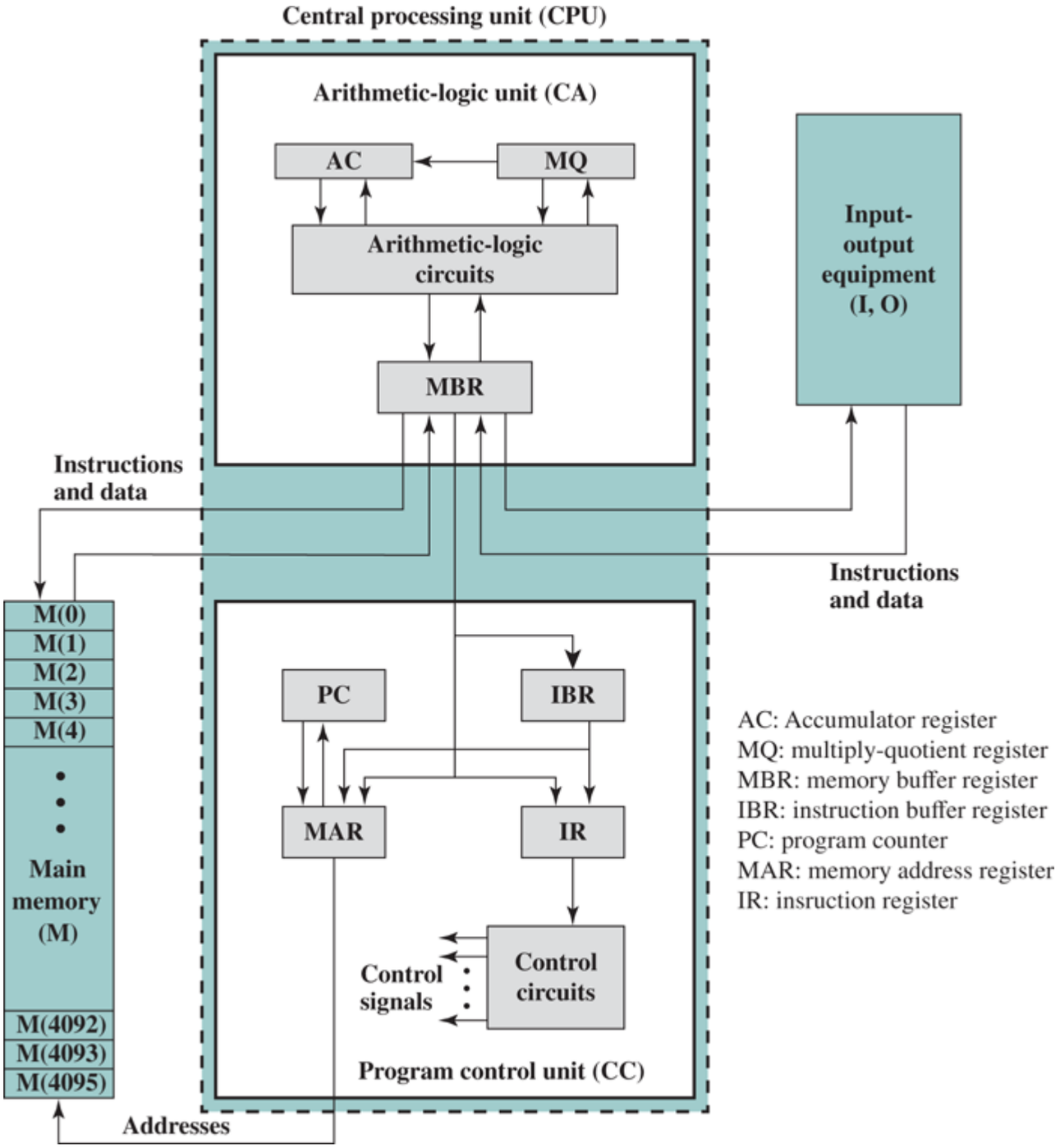


Figure 1.6 IAS Structure

- A **main memory** , which stores both data and instructions[4]

[4] In this book text, 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.

- An **arithmetic and logic unit (ALU)** capable of operating on binary data
- 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 are represented in binary form, and each instruction is a binary code. **Figure 1.7** 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).

[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.

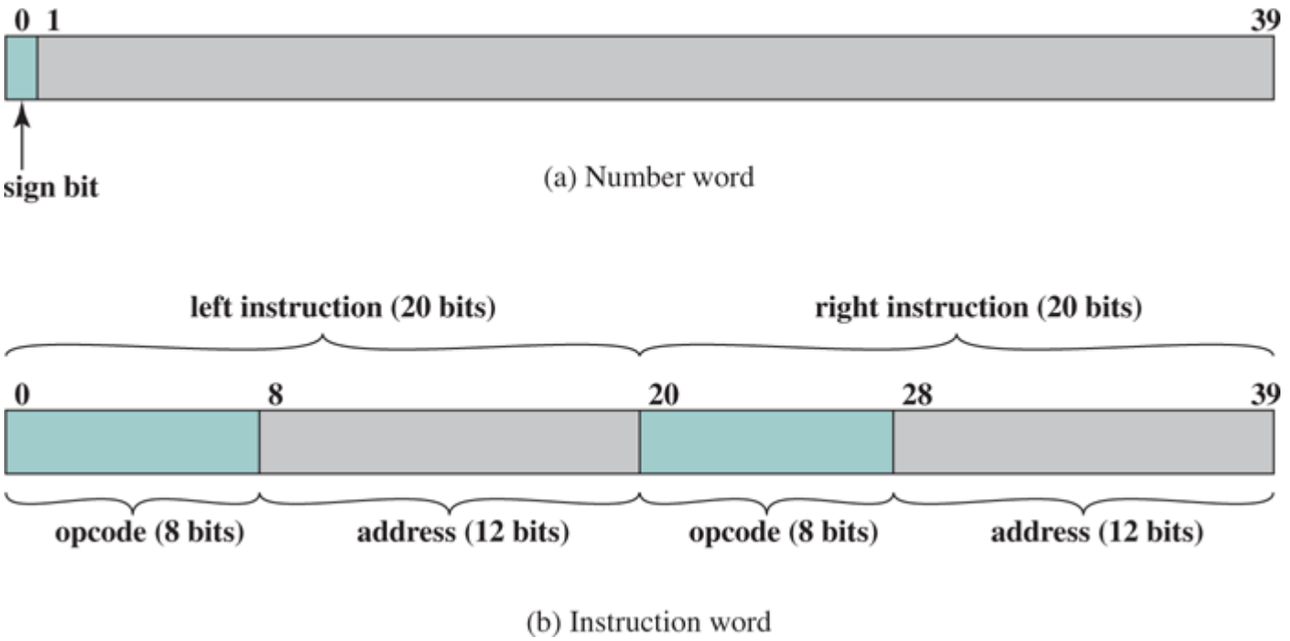


Figure 1.7 IAS Memory Formats

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 1.6**. 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 1.8**. 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 obtained from memory by loading a word into the MBR, and then down to the IBR, IR, and MAR.

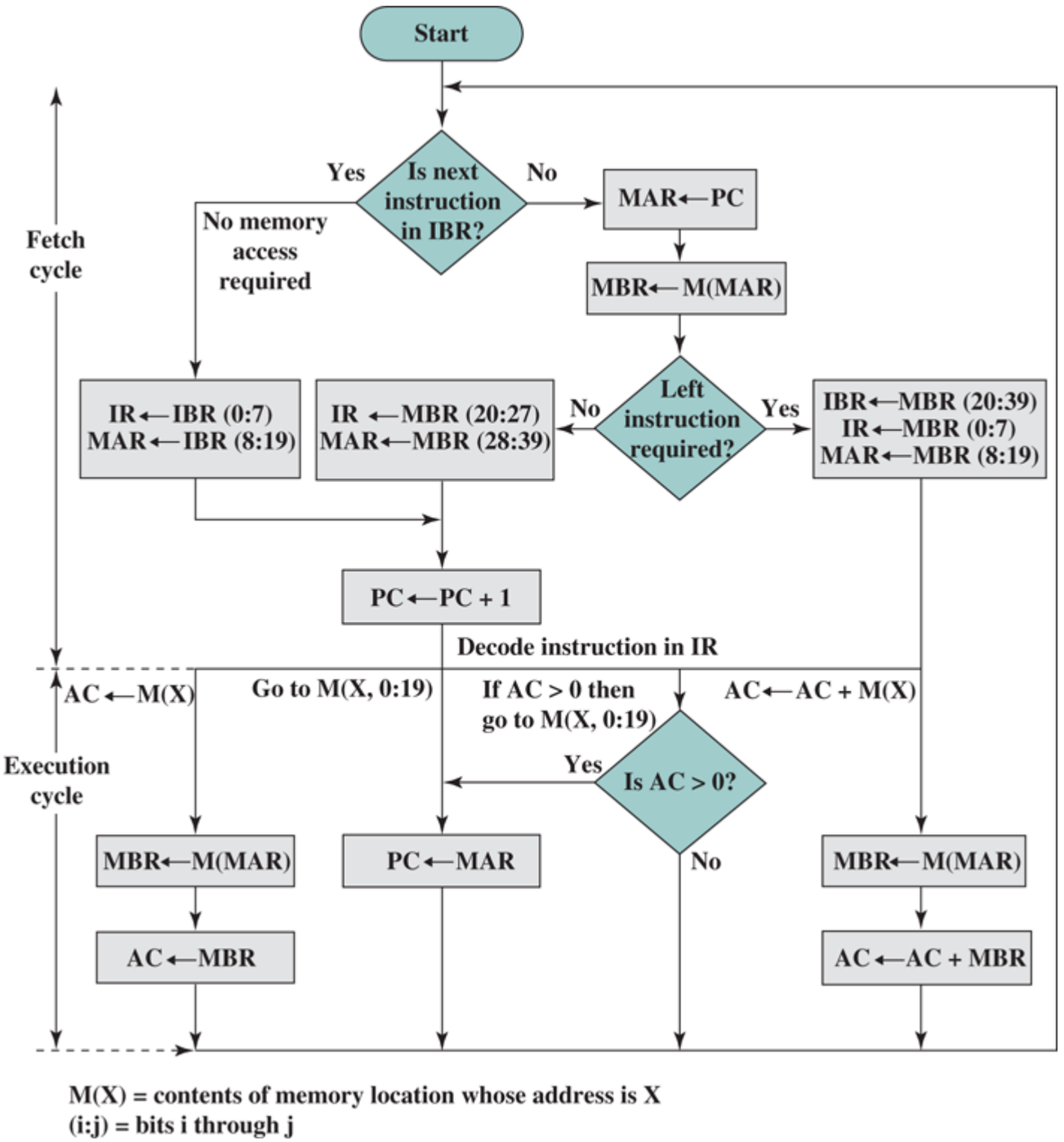


Figure 1.8 Partial Flowchart of IAS Operation

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:

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	$\text{JUMP} + \text{M}(\text{X}, 0:19)$	If number in the accumulator is nonnegative, take next instruction from left half of M(X)
	00010000	$\text{JUMP} + \text{M}(\text{X}, 20:39)$	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

- **Data transfer:** Move data between memory and ALU registers or between two ALU registers.
- **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 1.7**b. 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 1.8 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 1.9**): 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.

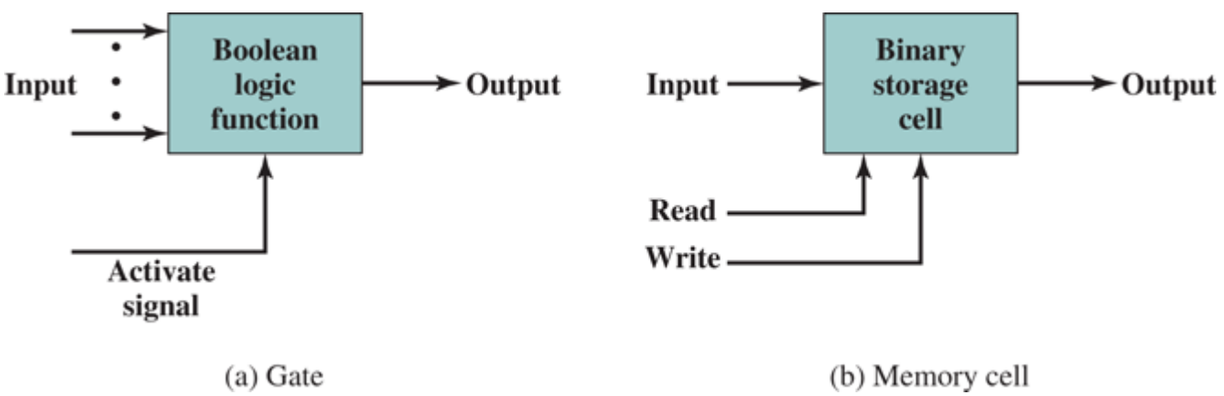


Figure 1.9 Fundamental Computer Elements

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 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.

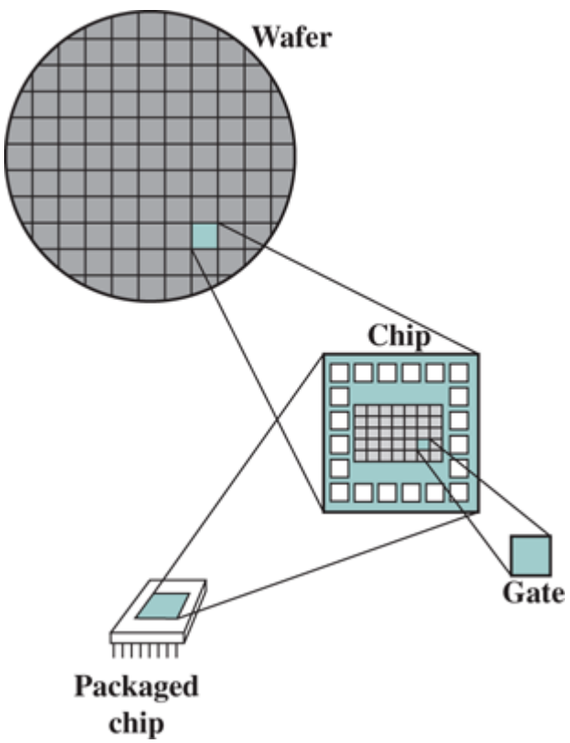
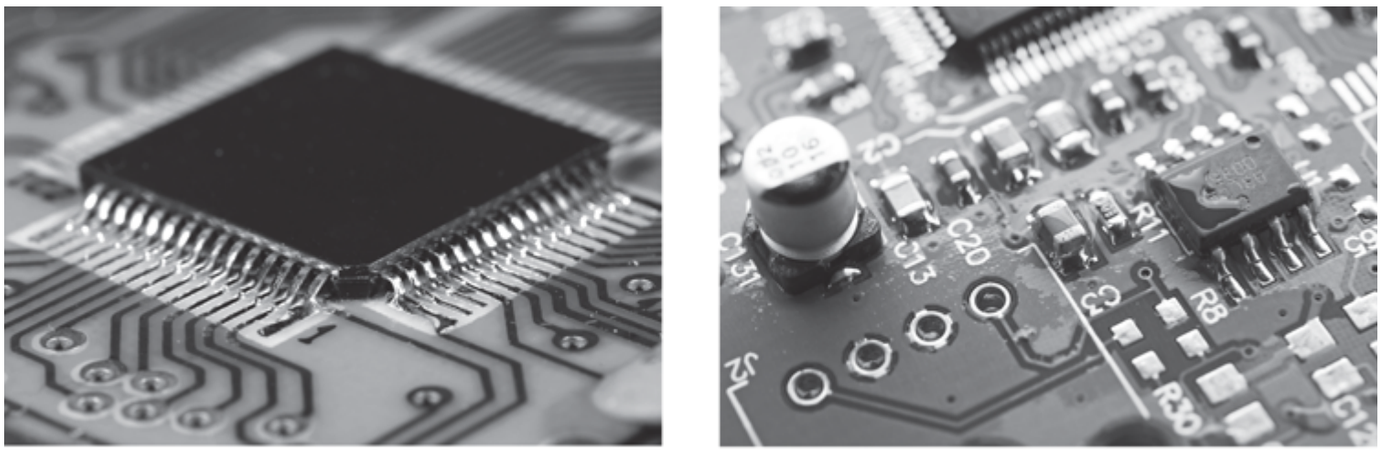


Figure 1.10 Relationship among Wafer, Chip, and Gate

Figure 1.10 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 packages can then be interconnected on a printed circuit board to produce larger and more complex circuits. **Figure 1.11a** indicates what a packaged processor or memory chip looks like, and **Figure 1.11b** shows a packaged chip wired onto a motherboard.



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

Figure 1.11 Processor or Memory Chip on Motherboard

Krzysztof Gorski/Shutterstock

Nikolich/Shutterstock

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 1.12**; 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.

[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.

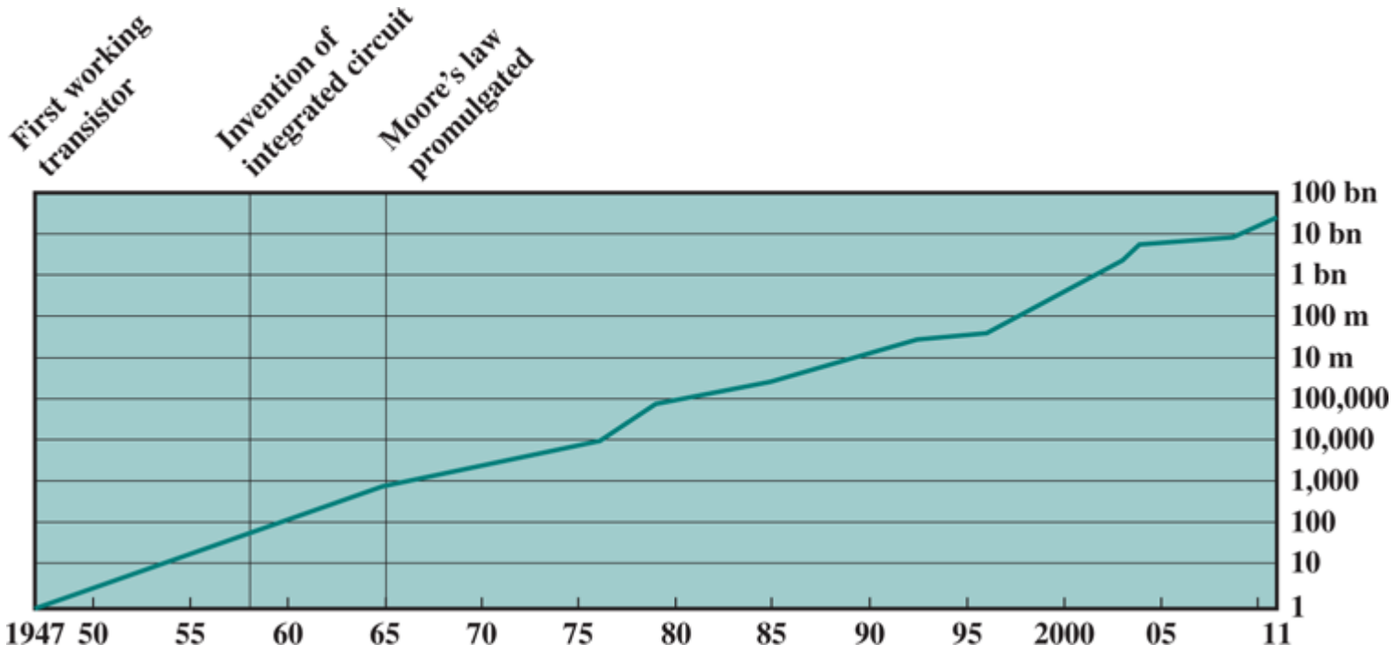


Figure 1.12 Growth in Transistor Count on Integrated Circuits

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.
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 1.11**).

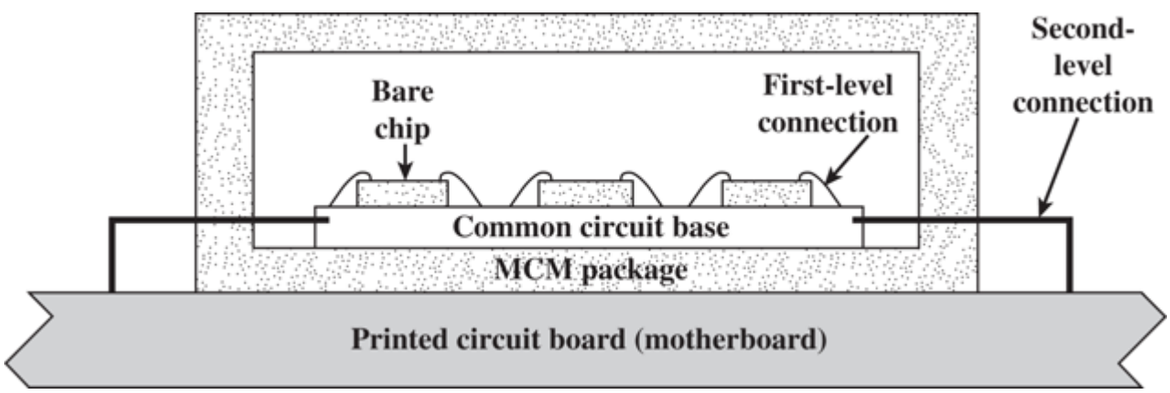


Figure 1.13 Multichip Module

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 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 1.13**):

- **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.

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]

[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.

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

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 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 1.14 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:

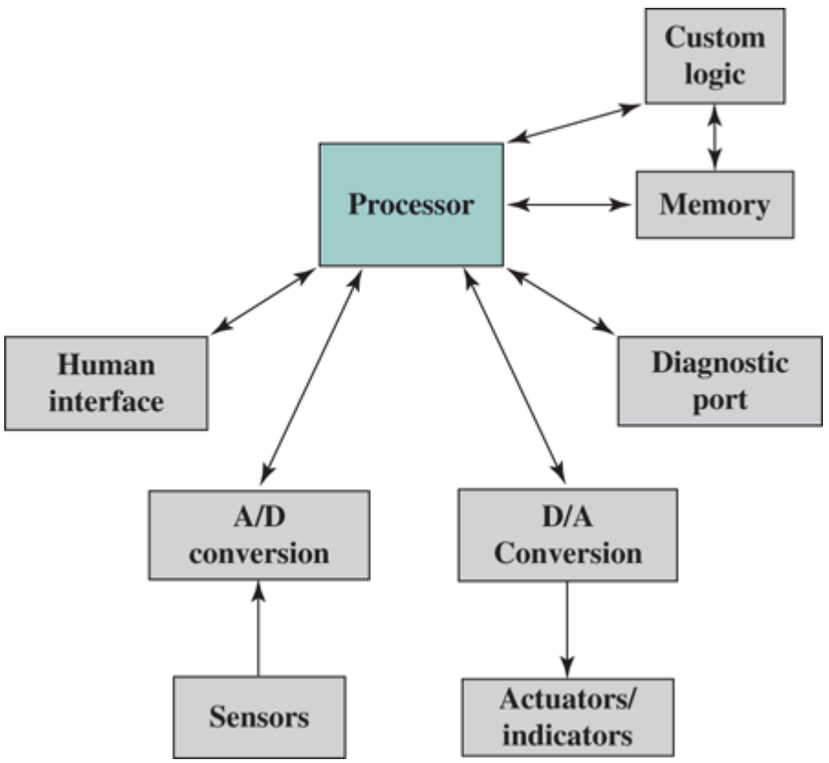


Figure 1.14 Possible Organization of an Embedded System

- 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.
- 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 1.15** 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**.

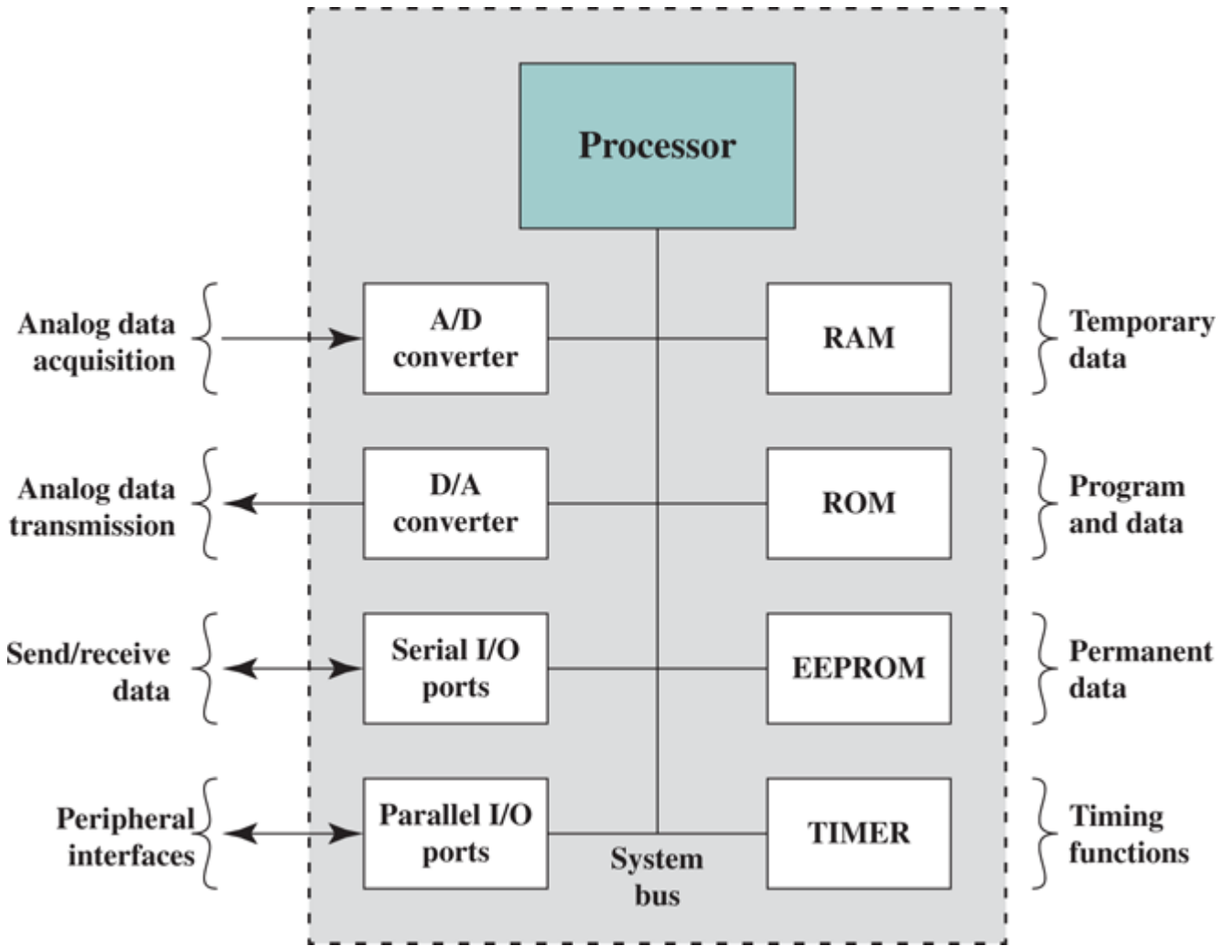


Figure 1.15 Typical Microcontroller Chip Elements

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.

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

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 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

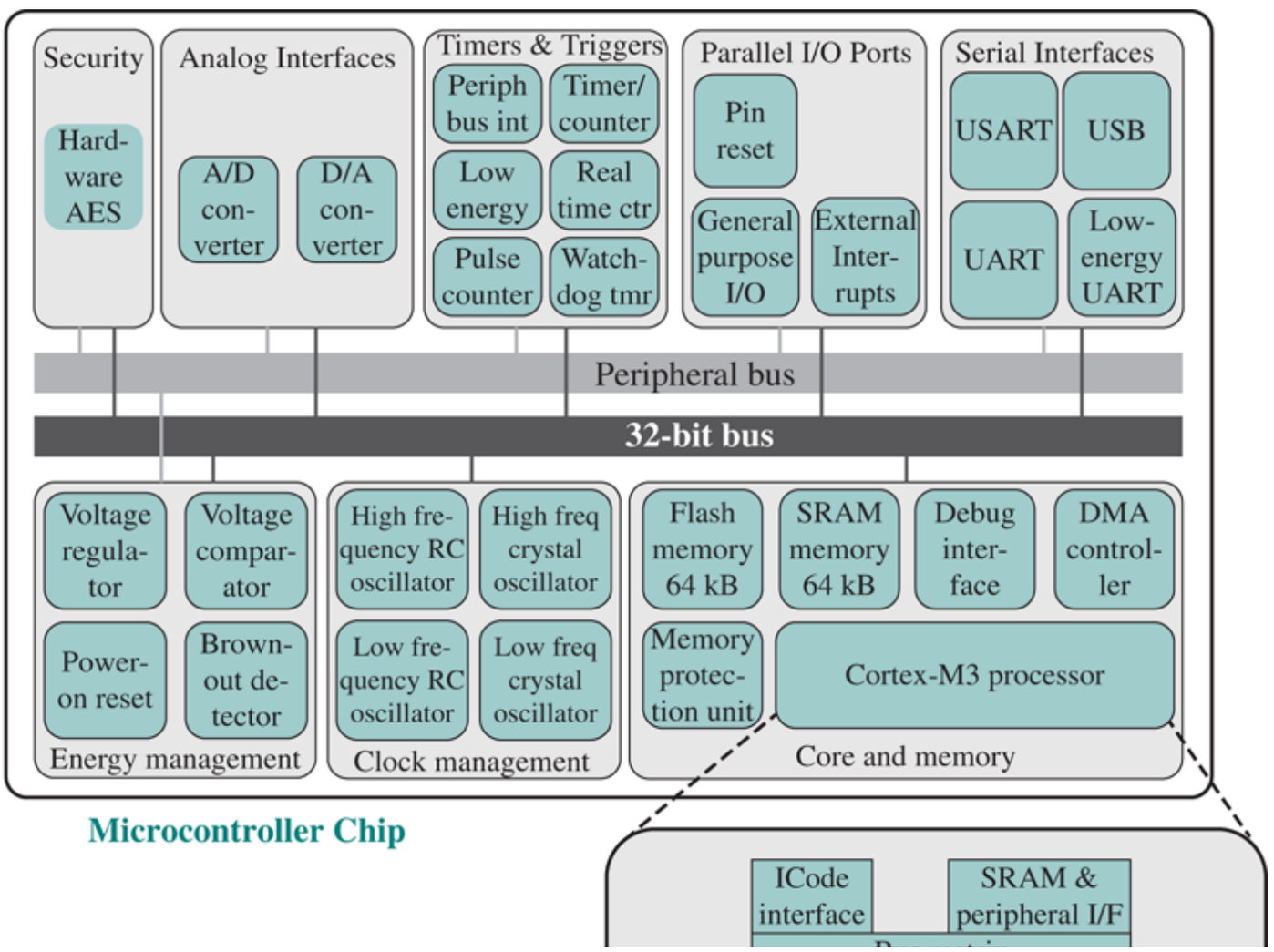
Cortex-M series processors have been developed primarily for the microcontroller domain where the need for fast, highly deterministic interrupt management is coupled with the desire for extremely low gate count and lowest possible power consumption. As with the Cortex-R series, the Cortex-M architecture 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 1.16 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.



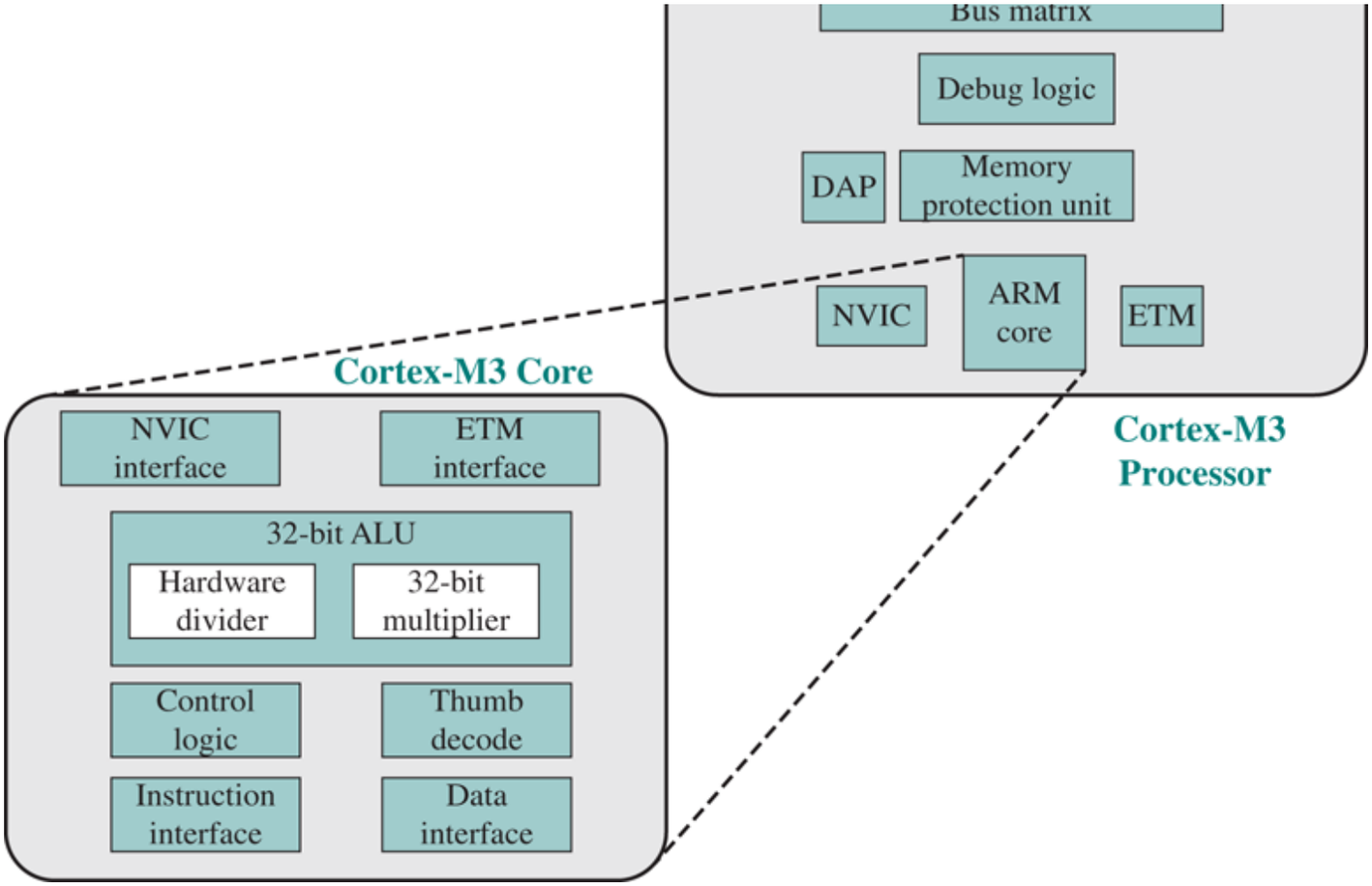


Figure 1.16 Typical Microcontroller Chip Based on Cortex-M3

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 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 1.16** 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 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.

[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**.

- **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 1.16** 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

arithmetic and logic unit (ALU)

ARM

central processing unit (CPU)

chip

computer architecture

computer organization

control unit

core

dedicated processor

deeply embedded system

embedded system

gate

input–output (I/O)

instruction set architecture (ISA)

integrated circuit

Intel x86

Internet of things (IoT)

main memory

memory cell

memory management unit (MMU)

memory protection unit (MPU)

microcontroller

microelectronics

microprocessor

motherboard

multichip module (MCM)

multicore

multicore processor

printed circuit board

processor

registers

semiconductor

semiconductor memory

system bus

system interconnection

transistor

Review Questions

1.1 What, in general terms, is the distinction between computer organization and computer architecture?
1.2 What, in general terms, is the distinction between computer structure and computer function?
1.3 What are the four main functions of a computer?
1.4 List and briefly define the main structural components of a computer.
1.5 List and briefly define the main structural components of a processor.
1.6 What is a stored program computer?
1.7 Explain Moore's law.
1.8 What is the key distinguishing feature of a microprocessor?

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 $\text{Sum}(Y) = \frac{N(N+1)}{2}$ 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 load the contents of memory address 2 to the accumulator?
b. How many trips to memory does the CPU need to make to complete this instruction during the instruction cycle?

1.3 On the IAS, describe in English the process that the CPU must undertake to read a value from memory and to write a value to memory in terms of what is put into the MAR, MBR, address bus, data bus, and control bus.
1.4 Given the memory contents of the IAS computer shown below,

Address	Contents

08A	010FA210FB
08B	010FA0F08D
08C	020FA210FB

show the assembly language code for the program, starting at address 08A. Explain what this program does.

1.5 In **Figure 1.6** , indicate the width, in bits, of each data path (e.g., between AC and ALU).

1.6 In the IBM 360 Models 65 and 75, addresses are staggered in two separate main memory units (e.g., all even-numbered words in one unit and all odd-numbered words in another). What might be the purpose of this technique?

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 is the fastest computer in the store that he can buy. Billy Bob replies, "You're looking at our Macintoshes. The fastest Mac we have runs at a clock speed of 1.2 GHz. If you really want the fastest machine, you should buy our 2.4-GHz Intel Pentium IV instead." Is Billy Bob correct? What would you say to help this customer?

1.9 The ENIAC, a precursor to the ISA machine, was a decimal machine, in which each register was represented by a ring of 10 vacuum tubes. At any time, only one vacuum tube was in the ON state, representing one of the 10 decimal digits. Assuming that ENIAC had the capability to have multiple vacuum tubes in the ON and OFF state simultaneously, why is this representation "wasteful" and what range of integer values could we represent using the 10 vacuum tubes?

1.10 For each of the following examples, determine whether this is an embedded system, explaining why or why not.

a. Are programs that understand physics and/or hardware embedded? For example, one that uses finite-element methods to predict fluid flow over airplane wings?
b. Is the internal microprocessor controlling a disk drive an example of an embedded system?
c. I/O drivers control hardware, so does the presence of an I/O driver imply that the computer executing the driver is embedded?
d. Is a PDA (Personal Digital Assistant) an embedded system?
e. Is the microprocessor controlling a cell phone an embedded system?
f. Are the computers in a big phased-array radar considered embedded? These radars are 10-story buildings with one to three 100-foot diameter radiating patches on the sloped sides of the building.
g. Is a traditional flight management system (FMS) built into an airplane cockpit considered embedded?
h. Are the computers in a hardware-in-the-loop (HIL) simulator embedded?
i. Is the computer controlling a pacemaker in a person's chest an embedded computer?
j. Is the computer controlling fuel injection in an automobile engine embedded?

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.

[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**.

- 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.
- 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.

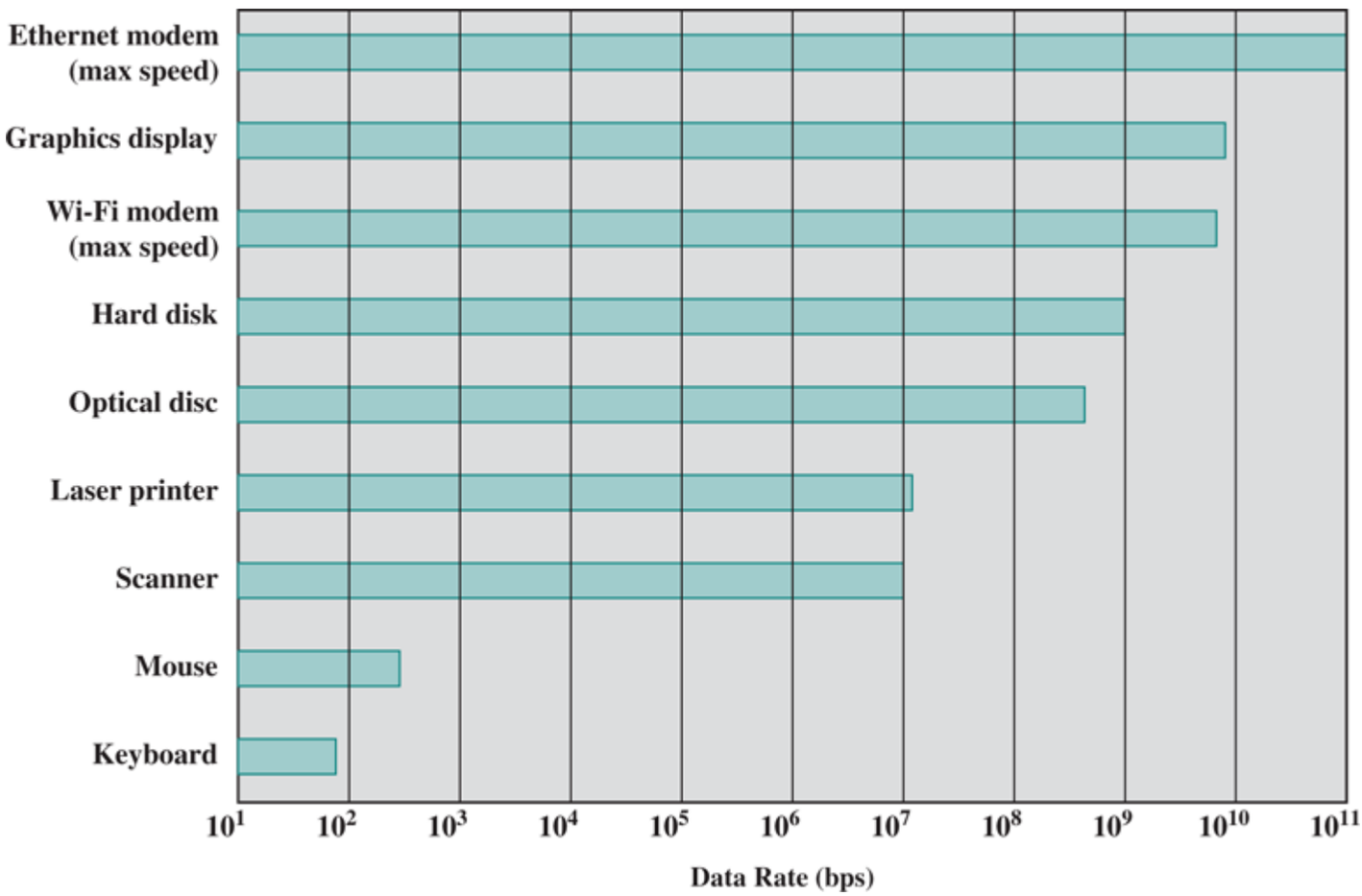


Figure 2.1 Typical I/O Device Data Rates

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.

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 $(\text{Watts}/\text{cm}^2)$. 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 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] I am grateful to Professor Kathy Yelick of UC Berkeley, who provided this graph.

[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.

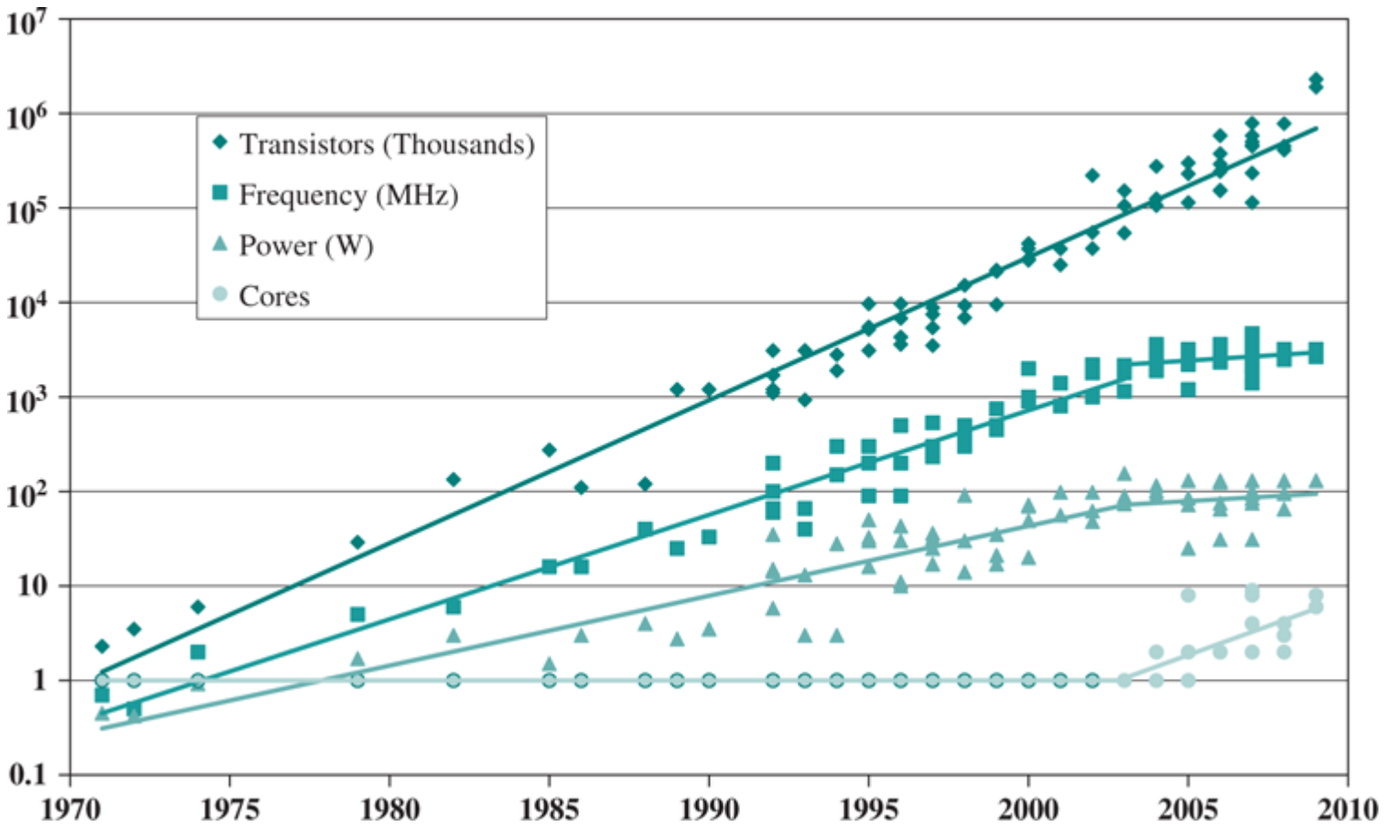


Figure 2.2 Processor Trends

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.

2.2 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 [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:

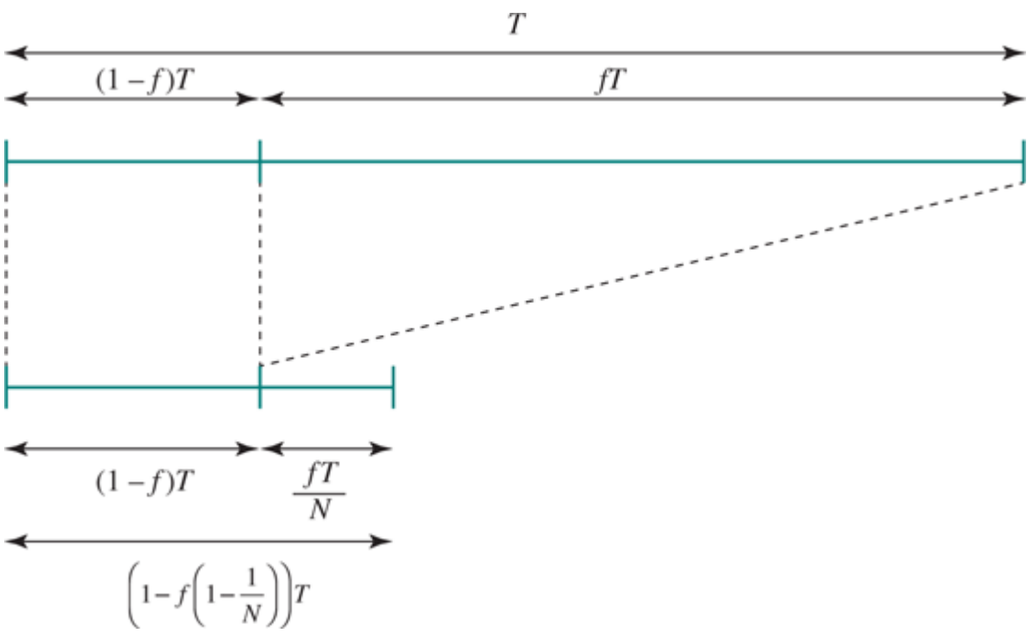


Figure 2.3 Illustration of Amdahl's Law

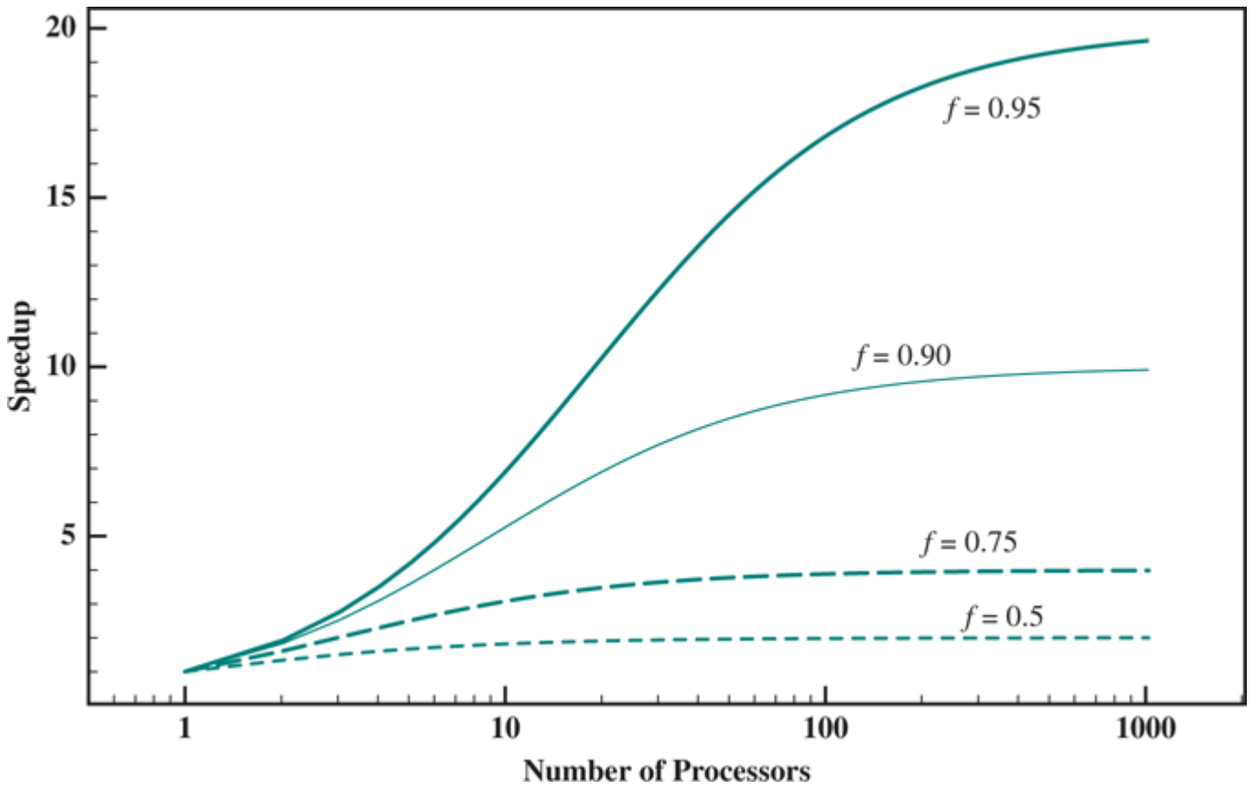


Figure 2.4 Amdahl's Law for Multiprocessors

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. 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

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

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 $L = \lambda W$.

[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].

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.

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 $\lambda =$ 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**.

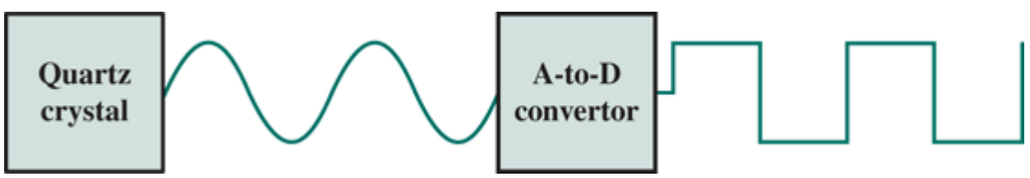


Figure 2.5 System Clock

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.

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:

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

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

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

Harmonic mean

$$HM = \frac{n}{\frac{1}{x_1} + \cdots + \frac{1}{x_n}} = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}} \quad x_i > 0 \tag{2.6}$$

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

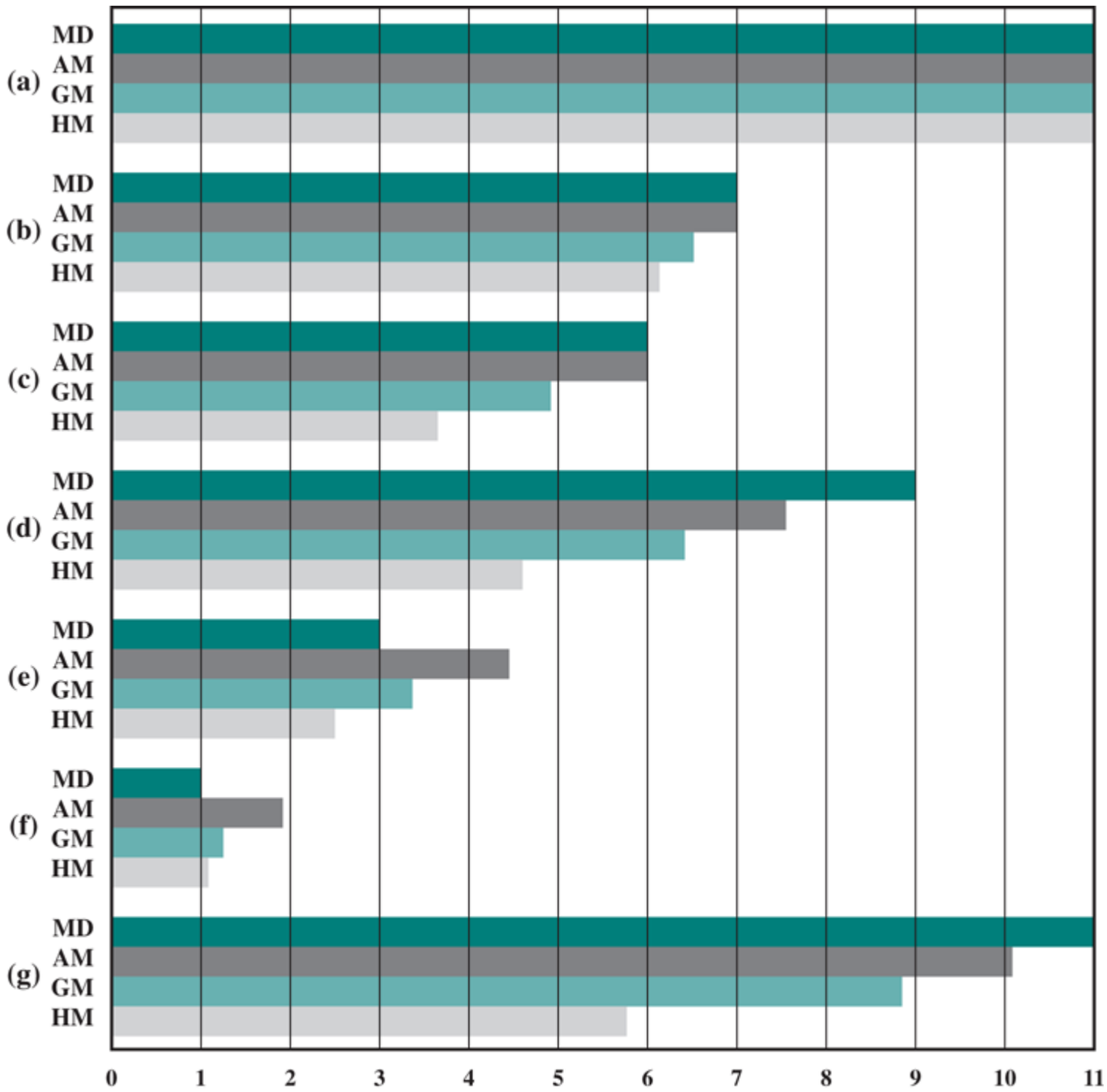
$$FM = f^{-1}\left(\frac{f(x_1) + \cdots + f(x_n)}{n}\right) = f^{-1}\left(\frac{1}{n}\sum_{i=1}^{n} f(x_i)\right)$$

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.

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

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.

$$HM = \frac{n}{\sum_{i=1} \frac{1}{R_i}} = \frac{n}{\sum_{i=1} \frac{1}{Z / t_i}} = \frac{nZ}{\sum_{i=1} t_i}$$

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.

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 (108 FP ops)	2.0	1.0	0.75	50	100	133.33
Program 2 (108 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

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:

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:

$$WHM = \frac{1}{\sum_{i=1}^{n} \frac{Z_i}{\sum_{j=1}^{n} Z_j} \frac{1}{R_i}} = \frac{n}{\sum_{i=1}^{n} \frac{Z_i}{\sum_{j=1}^{n} Z_j} \frac{t_i}{Z_i}} = \frac{\sum_{j=1}^{n} Z_j}{\sum_{i=1}^{n} t_i} \tag{2.7}$$

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
	3.40 (+0.87%)	4.55 (+2.24%)

Increase max value from 11 to 12.1 (+10%)	3.40 (+0.87%)	4.46 (+0.20%)
Increase min value from 1 to 1.1 (+10%)		

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

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

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.

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

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.

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

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, 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 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].

Table 2.5 SPEC CPU2017 Benchmarks

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

(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.

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.

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

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**):

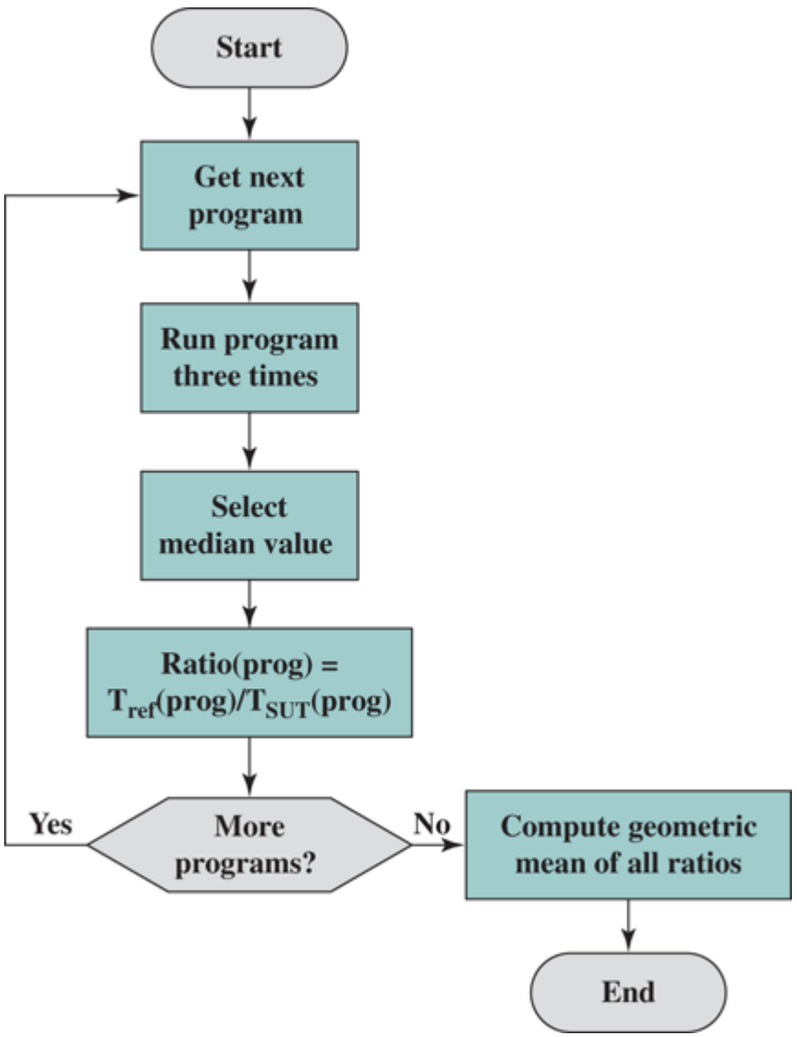


Figure 2.7 SPEC Evaluation Flowchart

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 *Trefi* is the execution time of benchmark program *i* on the reference system and *Tsuti* 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:

$$r_G = \prod_{i=1}^{12} r_i^{1/12}$$

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.

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

arithmetic mean (AM)

base metric

benchmark

clock cycle

clock cycle time

clock rate

clock speed

clock tick

cycles per instruction (CPI)

functional mean (FM)

general-purpose computing on GPU (GPGPU)

geometric mean (GM)

graphics processing unit (GPU)

harmonic mean (HM)

instruction execution rate

Little's law

many integrated core (MIC)

microprocessor

MIPS rate

multicore

peak metric

rate metric

reference machine

speed metric

SPEC

system under test

throughput

Review Questions

2.1 List and briefly define some of the techniques used in contemporary processors to increase speed.
2.2 Explain the concept of performance balance.
2.3 Explain the differences among multicore systems, MICs, and GPGPUs.
2.4 Briefly characterize Amdahl's law.
2.5 Briefly characterize Little's law.
2.6 Define MIPS and FLOPS.
2.7 List and define three methods for calculating a mean value of a set of data values.
2.8 List the desirable characteristics of a benchmark program.
2.9 What are the SPEC benchmarks?
2.10 What are the differences among base metric, peak metric, speed metric, and rate metric?

Problems

2.1 A benchmark program is run on a 40 MHz processor. The executed program consists of 100,000 instruction executions, with the following instruction mix and clock cycle count:

Instruction Type	Instruction Count	Cycles per Instruction
Integer arithmetic	45,000	1
Data transfer	32,000	2
Floating point	15,000	2
Control transfer	8000	2

Determine the effective CPI, MIPS rate, and execution time for this program.
2.2 Consider two different machines, with two different instruction sets, both of which have a clock rate of 200 MHz. The following measurements are recorded on the two machines running a given set of benchmark programs:

Instruction Type	Instruction Count (millions)	Cycles per Instruction
Machine A		
Arithmetic and logic	8	1
Load and store	4	3
Branch	2	4
Others	4	3
Machine A		
Arithmetic and logic	10	1

Load and store	8	2
Branch	2	4
Others	4	3

a. Determine the effective CPI, MIPS rate, and execution time for each machine.
b. Comment on the results.

2.3 Early examples of CISC and RISC design are the VAX 11/780 and the IBM RS/6000, respectively. Using a typical benchmark program, the following machine characteristics result:

Processor	Clock Frequency (MHz)	Performance (MIPS)	CPU Time (secs)
VAX 11/780	5	1	12 *x*
IBM RS/6000	25	18	*x*

The final column shows that the VAX required 12 times longer than the IBM measured in CPU time.

a. What is the relative size of the instruction count of the machine code for this benchmark program running on the two machines?
b. What are the *CPI* values for the two machines?

2.4 Four benchmark programs are executed on three computers with the following results:

	Computer A	Computer B	Computer C
Program 1	1	10	20
Program 2	1000	100	20
Program 3	500	1000	50
Program 4	100	800	100

The table shows the execution time in seconds, with 100,000,000 instructions executed in each of the four programs. Calculate the MIPS values for each computer for each program. Then calculate the arithmetic and harmonic means assuming equal weights for the four programs, and rank the computers based on arithmetic mean and harmonic mean.

2.5 The following table, based on data reported in the literature [HEAT84], shows the execution times, in seconds, for five different benchmark programs on three machines.

Benchmark	Processor		
	R	M	Z
E	417	244	134

F	83	70	70
H	66	153	135
I	39,449	35,527	66,000
K	772	368	369

a. Compute the speed metric for each processor for each benchmark, normalized to machine R. That is, the ratio values for R are all 1.0. Other ratios are calculated using **Equation (2.5)** with R treated as the reference system. Then compute the arithmetic mean value for each system using **Equation (2.3)** . This is the approach taken in [HEAT84].
b. Repeat part (a) using M as the reference machine. This calculation was not tried in [HEAT84].
c. Which machine is the slowest based on each of the preceding two calculations?
d. Repeat the calculations of parts (a) and (b) using the geometric mean, defined in **Equation (2.6)** . Which machine is the slowest based on the two calculations?

2.6 To clarify the results of the preceding problem, we look at a simpler example.

Benchmark	Processor		
	X	Y	Z
1	20	10	40
2	40	80	20

a. Compute the arithmetic mean value for each system using X as the reference machine and then using Y as the reference machine. Argue that intuitively, the three machines have roughly equivalent performance and that the arithmetic mean gives misleading results.
b. Compute the geometric mean value for each system, using X as the reference machine and then using Y as the reference machine. Argue that the results are more realistic than with the arithmetic mean.

2.7 Consider the example in **Section 2.5** for the calculation of average CPI and MIPS rate, which yielded the result of $CPI = 2.24$ and MIPS $\text{rate} = 178$. Now assume that the program can be executed in eight parallel tasks or threads, with roughly equal number of instructions executed in each task. Execution is on an 8-core system, with each core (processor) having the same performance as the single processor originally used. Coordination and synchronization between the parts adds an extra 25,000 instruction executions to each task. Assume the same instruction mix as in the example for each task, but increase the CPI for memory reference with cache miss to 12 cycles due to contention for memory.

a. Determine the average CPI.
b. Determine the corresponding MIPS rate.

c. Calculate the speedup factor.
d. Compare the actual speedup factor with the theoretical speedup factor determined by Amdhal's law.

2.8 A processor accesses main memory with an average access time of T_2. A smaller cache memory is interposed between the processor and main memory. The cache has a significantly faster access time of $T_1 < T_2$. The cache holds, at any time, copies of some main memory words and is designed so that the words more likely to be accessed in the near future are in the cache. Assume that the probability that the next word accessed by the processor is in the cache is *H*, known as the hit ratio.

a. For any single memory access, what is the theoretical speedup of accessing the word in the cache rather than in main memory?
b. Let *T* be the average access time. Express *T* as a function of T_1, T_2, and *H*. What is the overall speedup as a function of *H*?
c. In practice, a system may be designed so that the processor must first access the cache to determine if the word is in the cache and, if it is not, then access main memory, so that on a miss (opposite of a hit), memory access time is $T_1 + T_2$. Express *T* as a function of T_1, T_2, and *H*. Now calculate the speedup and compare to the result produced in part (b).

2.9 The owner of a shop observes that on average 18 customers per hour arrive, and there are typically 8 customers in the shop. What is the average length of time each customer spends in the shop?

2.10 We can gain more insight into Little's law by considering **Figure 2.8a** . Over a period of time *T*, a total of *C* items arrive at a system, wait for service, and complete service. The upper solid line shows the time sequence of arrivals, and the lower solid line shows the time sequence of departures. The shaded area bounded by the two lines represents the total "work" done by the system in units of job-seconds; let *A* be the total work. We wish to derive the relationship among *L*, *W*, and λ.

a. **Figure 2.8b** divides the total area into horizontal rectangles, each with a height of one job. Picture sliding all these rectangles to the left so that their left edges line up at $t = 0$. Develop an equation that relates *A*, *C*, and *W*.
b. **Figure 2.8c** divides the total area into vertical rectangles, defined by the vertical transition boundaries indicated by the dashed lines. Picture sliding all these rectangles down so that their lower edges line up at $N(t) = 0$. Develop an equation that relates *A*, *T*, and *L*.
c. Finally, derive $L = \lambda W$ from the results of (a) and (b).

2.11 In **Figure 2.8a** , jobs arrive at times $t = 0, 1, 1.5, 3.25, 5.25,$ and 7.75. The corresponding completion times are $t = 2, 3, 3.5, 4.25, 8.25,$ and 8.75.

a. Determine the area of each of the six rectangles in **Figure 2.8b** and sum to get the total area *A*. Show your work.
b. Determine the area of each of the 10 rectangles in **Figure 2.8c** and sum to get the total area *A*. Show your work.

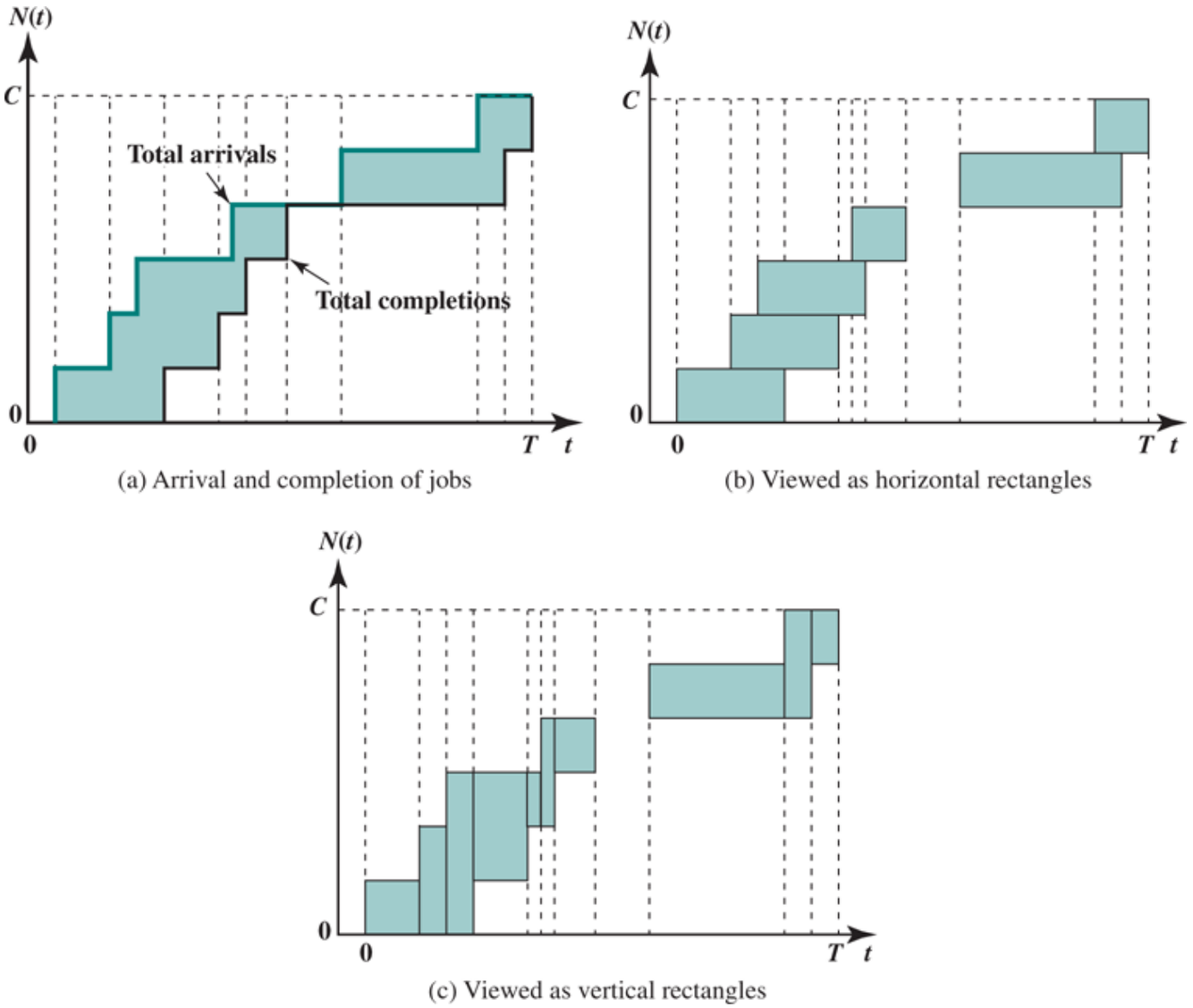


Figure 2.8 Illustration of Little's Law

2.12 In **Section 2.6** , we specified that the base ratio used for comparing a system under test to a reference system is:

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

a. The preceding equation provides a measure of the speedup of the system under test compared to the reference system. Assume that the number of floating-point operations executed in the test program is I_i. Now show the speedup as a function of the instruction execution rate $FLOPS_i$.
b. Another technique for normalizing performance is to express the performance of a system as a percent change relative to the performance of another system. Express this relative change first as a function of instruction execution rate, and then as a function of execution times.

2.13 Assume that a benchmark program executes in 480 seconds on a reference machine A. The same program executes on systems B, C, and D in 360, 540, and 210 seconds, respectively.

a. Show the speedup of each of the three systems under test relative to A.
b. Now show the relative speedup of the three systems. Comment on the three ways of comparing machines (execution time, speedup, relative speedup).

2.14 Repeat the preceding problem using machine D as the reference machine. How does this affect the relative rankings of the four systems?
2.15 Recalculate the results in **Table 2.2** using the computer time data of **Table 2.4** and comment on the results.
2.16 **Equation 2.5** shows two different formulations of the geometric mean, one using a product operator and one using a summation operator.
a. Show that the two formulas are equivalent.
b. Why would the summation formulation be preferred for calculating the geometric mean?

2.17 **Project.** **Section 2.5** lists a number of references that document the "benchmark means wars." All of the referenced papers are available at **box.com/COA10e**. Read these papers and summarize the case for and against the use of the geometric mean for SPEC calculations.

Part Two The Computer System

Chapter 3 A Top-Level View of Computer Function and Interconnection

Learning Objectives

After studying this chapter, you should be able to:

- Understand the basic elements of an instruction cycle and the role of interrupts.
- Describe the concept of interconnection within a computer system.
- Assess the relative advantages of point-to-point interconnection compared to bus interconnection.
- Present an overview of QPI.
- Present an overview of PCIe.

At a top level, a computer consists of CPU (central processing unit), memory, and I/O components, with one or more modules of each type. These components are interconnected in some fashion to achieve the basic function of the computer, which is to execute programs. Thus, at a top level, we can characterize a computer system by describing (1) the external behavior of each component, that

is, the data and control signals that it exchanges with other components, and (2) the interconnection structure and the controls required to manage the use of the interconnection structure.

This top-level view of structure and function is important because of its explanatory power in understanding the nature of a computer. Equally important is its use to understand the increasingly complex issues of performance evaluation. A grasp of the top-level structure and function offers insight into system bottlenecks, alternate pathways, the magnitude of system failures if a component fails, and the ease of adding performance enhancements. In many cases, requirements for greater system power and fail-safe capabilities are being met by changing the design rather than merely increasing the speed and reliability of individual components.

This chapter focuses on the basic structures used for computer component interconnection. As background, the chapter begins with a brief examination of the basic components and their interface requirements. Then a functional overview is provided. We are then prepared to examine the use of buses to interconnect system components.

3.1 Computer Components

As discussed in **Chapter 1**, virtually all contemporary computer designs are based on concepts developed by John von Neumann at the Institute for Advanced Studies, Princeton. Such a design is referred to as the *von Neumann architecture* and is based on three key concepts:

- Data and instructions are stored in a single read–write memory.
- The contents of this memory are addressable by location, without regard to the type of data contained there.
- Execution occurs in a sequential fashion (unless explicitly modified) from one instruction to the next.

The reasoning behind these concepts was discussed in **Chapter 2** but is worth summarizing here. There is a small set of basic logic components that can be combined in various ways to store binary data and perform arithmetic and logical operations on that data. If there is a particular computation to be performed, a configuration of logic components designed specifically for that computation could be constructed. We can think of the process of connecting the various components in the desired configuration as a form of programming. The resulting "program" is in the form of hardware and is termed a *hardwired program*.

Now consider this alternative. Suppose we construct a general-purpose configuration of arithmetic and logic functions. This set of hardware will perform various functions on data depending on control signals applied to the hardware. In the original case of customized hardware, the system accepts data and produces results (**Figure 3.1a**). With general-purpose hardware, the system accepts data and control signals and produces results. Thus, instead of rewiring the hardware for each new program, the programmer merely needs to supply a new set of control signals.

How shall control signals be supplied? The answer is simple but subtle. The entire program is actually a sequence of steps. At each step, some arithmetic or logical operation is performed on some data. For each step, a new set of control signals is needed. Let us provide a unique code for each possible set of control signals, and let us add to the general-purpose hardware a segment that can accept a code and generate control signals (**Figure 3.1b**).

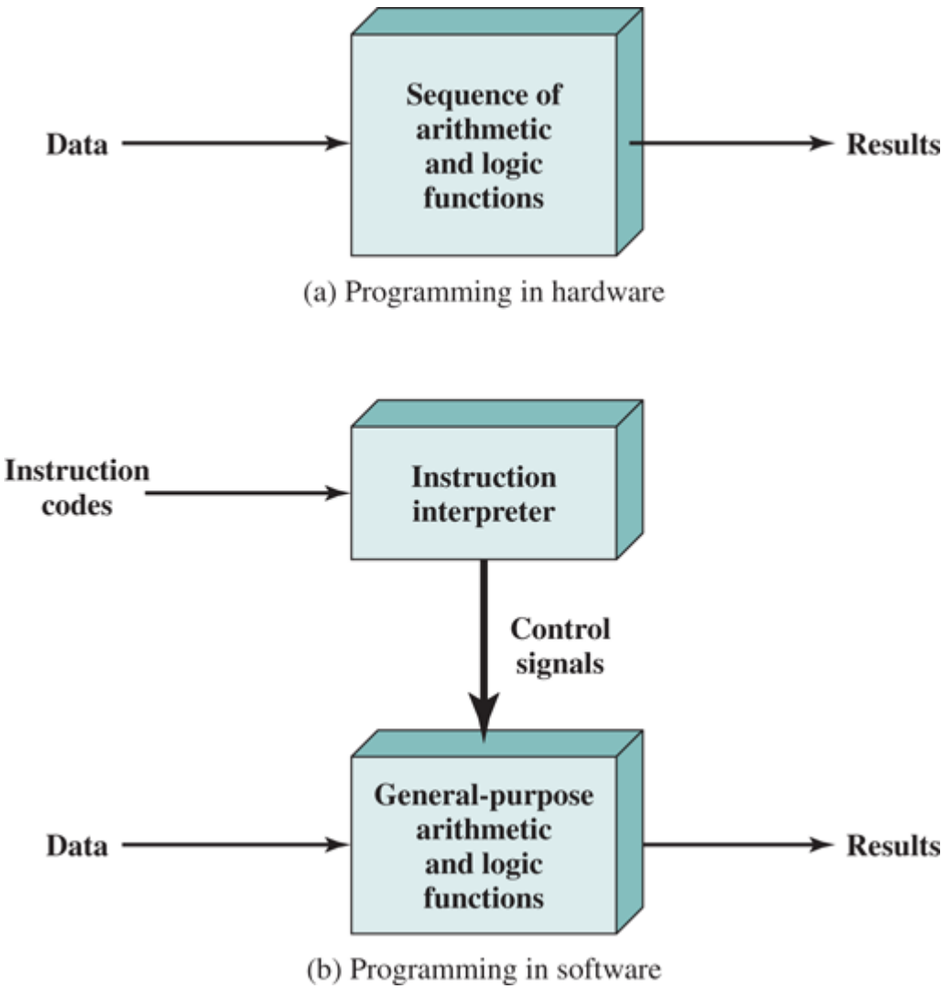


Figure 3.1 Hardware and Software Approaches

Programming is now much easier. Instead of rewiring the hardware for each new program, all we need to do is provide a new sequence of codes. Each code is, in effect, an instruction, and part of the hardware interprets each instruction and generates control signals. To distinguish this new method of programming, a sequence of codes or instructions is called *software*.

Figure 3.1b indicates two major components of the system: an instruction interpreter and a module of general-purpose arithmetic and logic functions. These two constitute the CPU. Several other components are needed to yield a functioning computer. Data and instructions must be put into the system. For this we need some sort of input module. This module contains basic components for accepting data and instructions in some form and converting them into an internal form of signals usable by the system. A means of reporting results is needed, and this is in the form of an output module. Taken together, these are referred to as *I/O components*.

One more component is needed. An input device will bring instructions and data in sequentially. But a program is not invariably executed sequentially; it may jump around (e.g., the IAS jump instruction). Similarly, operations on data may require access to more than just one element at a time in a predetermined sequence. Thus, there must be a place to temporarily store both instructions and data. That module is called *memory*, or *main memory*, to distinguish it from external storage or peripheral devices. Von Neumann pointed out that the same memory could be used to store both instructions and data.

Figure 3.2 illustrates these top-level components and suggests the interactions among them. The

CPU exchanges data with memory. For this purpose, it typically makes use of two internal (to the CPU) registers: a **memory address register (MAR)** , which specifies the address in memory for the next read or write, and a **memory buffer register (MBR)** , which contains the data to be written into memory or receives the data read from memory. Similarly, an I/O address register (I/OAR) specifies a particular I/O device. An I/O buffer register (I/OBR) is used for the exchange of data between an I/O module and the CPU.

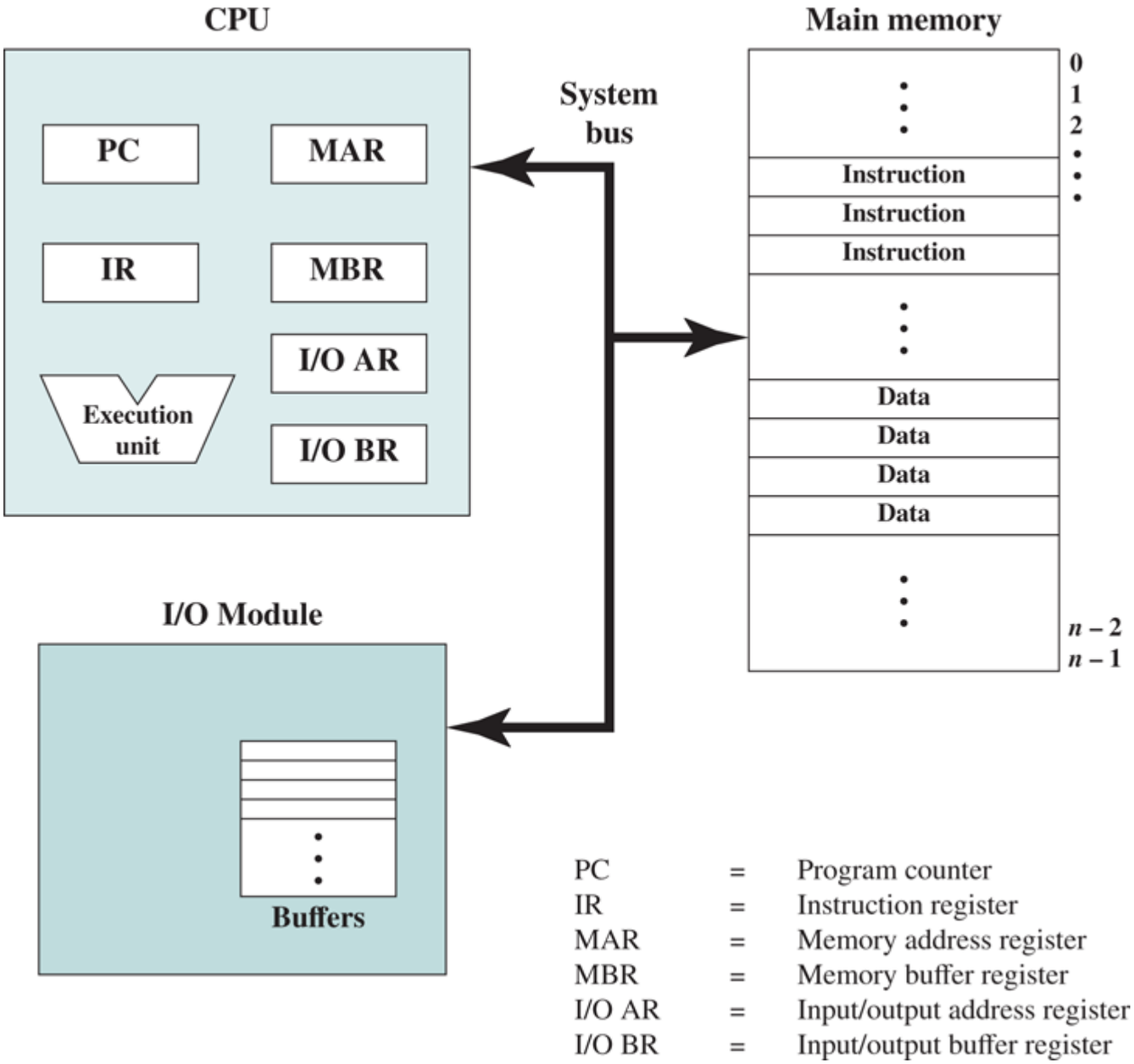


Figure 3.2 Computer Components: Top-Level View

A memory module consists of a set of locations, defined by sequentially numbered addresses. Each location contains a binary number that can be interpreted as either an instruction or data. An I/O module transfers data from external devices to CPU and memory, and vice versa. It contains internal buffers for temporarily holding these data until they can be sent on.

Having looked briefly at these major components, we now turn to an overview of how these components function together to execute programs.

3.2 Computer Function

The basic function performed by a computer is execution of a program, which consists of a set of instructions stored in memory. The processor does the actual work by executing instructions specified in the program. This section provides an overview of the key elements of program execution. In its simplest form, instruction processing consists of two steps. The processor reads (*fetches*) instructions from memory one at a time, then executes each instruction. Program execution consists of repeating the process of instruction fetch and instruction execution. The instruction execution may involve several operations and depends on the nature of the instruction (see, for example, the lower portion of **Figure 2.4**).

The processing required for a single instruction is called an **instruction cycle** . Using the simplified two-step description given previously, the instruction cycle is depicted in **Figure 3.3**. The two steps are referred to as the **fetch cycle** and the **execute cycle** . Program execution halts only if the machine is turned off, some sort of unrecoverable error occurs, or a program instruction that halts the computer is encountered.

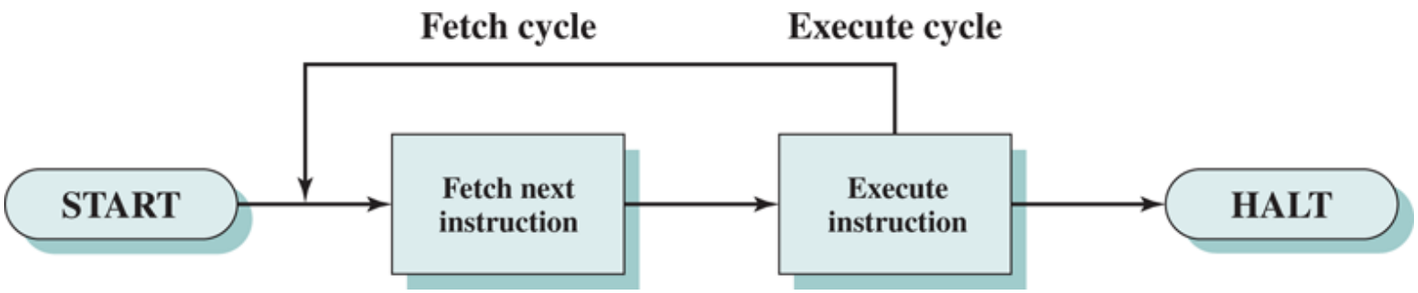


Figure 3.3 Basic Instruction Cycle

Instruction Fetch and Execute

At the beginning of each instruction cycle, the processor fetches an instruction from memory. In a typical processor, a register called the program counter (PC) holds the address of the instruction to be fetched next. Unless told otherwise, the processor always increments the PC after each instruction fetch so that it will fetch the next instruction in sequence (i.e., the instruction located at the next higher memory address). So, for example, consider a computer in which each instruction occupies one 16-bit word of memory. Assume that the program counter is set to memory location 300, where the location address refers to a 16-bit word. The processor will next fetch the instruction at location 300. On succeeding instruction cycles, it will fetch instructions from locations 301, 302, 303, and so on. This sequence may be altered, as explained presently.

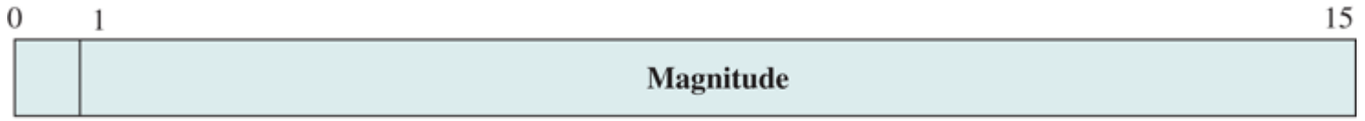
The fetched instruction is loaded into a register in the processor known as the instruction register (IR). The instruction contains bits that specify the action the processor is to take. The processor interprets the instruction and performs the required action. In general, these actions fall into four categories:

- **Processor-memory:** Data may be transferred from processor to memory or from memory to processor.
- **Processor-I/O:** Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module.
- **Data processing:** The processor may perform some arithmetic or logic operation on data.
- **Control:** An instruction may specify that the sequence of execution be altered. For example, the processor may fetch an instruction from location 149, which specifies that the next instruction be

from location 182. The processor will remember this fact by setting the program counter to 182. Thus, on the next fetch cycle, the instruction will be fetched from location 182 rather than 150. An instruction's execution may involve a combination of these actions.

Consider a simple example using a hypothetical machine that includes the characteristics listed in **Figure 3.4**. The processor contains a single data register, called an accumulator (AC). Both instructions and data are 16 bits long. Thus, it is convenient to organize memory using 16-bit words. The instruction format provides 4 bits for the opcode, so that there can be as many as $2^4 = 16$ different opcodes, and up to $2^{12} = 4096$ (4K) words of memory can be directly addressed.

(a) Instruction format



(b) Integer format

Program counter (PC) = Address of instruction
Instruction register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

(c) Internal CPU registers

0001 = Load AC from memory
0010 = Store AC to memory
0101 = Add to AC from memory

(d) Partial list of opcodes

Figure 3.4 Characteristics of a Hypothetical Machine

Figure 3.5 illustrates a partial program execution, showing the relevant portions of memory and processor registers.[1] The program fragment shown adds the contents of the memory word at address 940 to the contents of the memory word at address 941 and stores the result in the latter location. Three instructions, which can be described as three fetch and three execute cycles, are required:

[1] Hexadecimal notation is used, in which each digit represents 4 bits. This is the most convenient notation for representing the contents of memory and registers when the word length is a multiple of 4. See **Chapter 9** for a basic refresher on number systems (decimal, binary, hexadecimal)

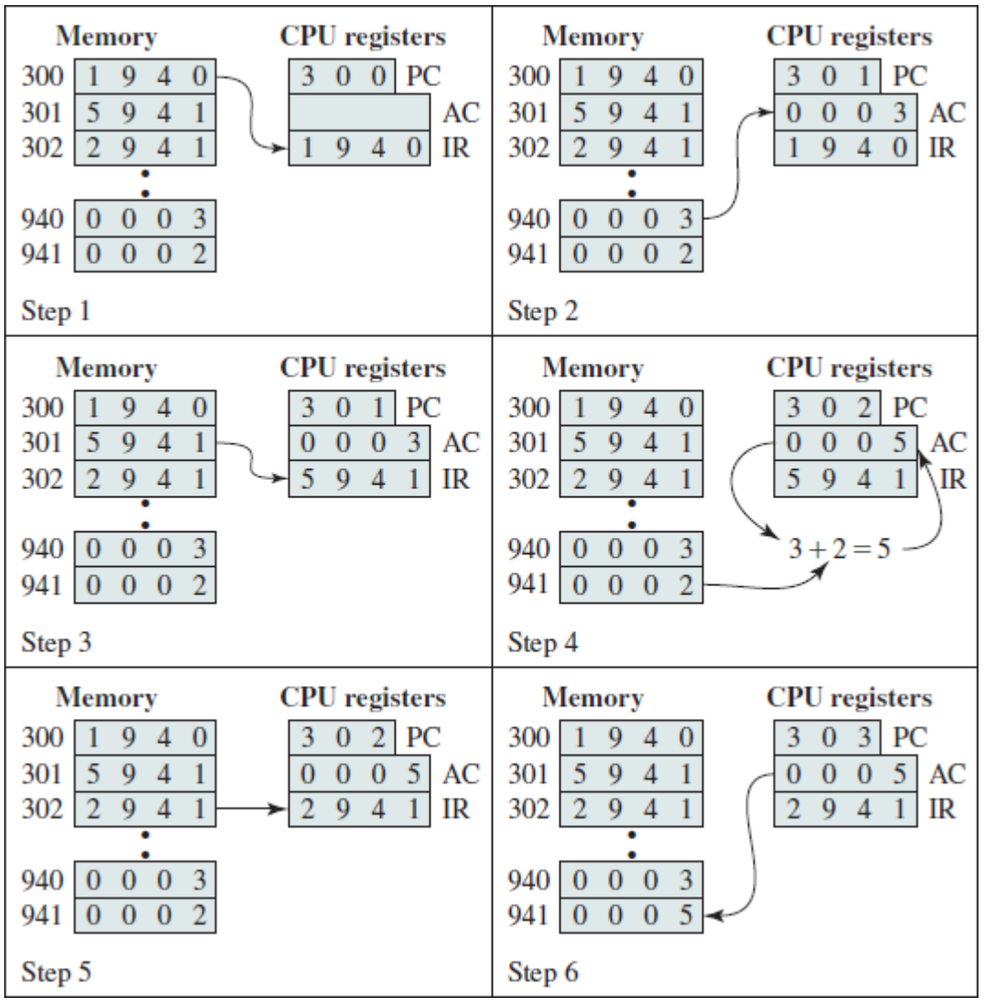


Figure 3.5 Example of Program Execution (contents of memory and registers in hexadecimal)

1. The PC contains 300, the address of the first instruction. This instruction (the value 1940 in hexadecimal) is loaded into the instruction register IR, and the PC is incremented. Note that this process involves the use of a memory address register and a memory buffer register. For simplicity, these intermediate registers are ignored.
2. The first 4 bits (first hexadecimal digit) in the IR indicate that the AC is to be loaded. The remaining 12 bits (three hexadecimal digits) specify the address (940) from which data are to be loaded.
3. The next instruction (5941) is fetched from location 301, and the PC is incremented.
4. The old contents of the AC and the contents of location 941 are added, and the result is stored in the AC.
5. The next instruction (2941) is fetched from location 302, and the PC is incremented.
6. The contents of the AC are stored in location 941.

In this example, three instruction cycles, each consisting of a fetch cycle and an execute cycle, are needed to add the contents of location 940 to the contents of 941. With a more complex set of instructions, fewer cycles would be needed. Some older processors, for example, included instructions that contain more than one memory address. Thus, the execution cycle for a particular instruction on such processors could involve more than one reference to memory. Also, instead of memory references, an instruction may specify an I/O operation.

For example, the PDP-11 processor includes an instruction, expressed symbolically as ADD B,A, that

stores the sum of the contents of memory locations B and A into memory location A. A single instruction cycle with the following steps occurs:

- Fetch the ADD instruction.
- Read the contents of memory location A into the processor.
- Read the contents of memory location B into the processor. In order that the contents of A are not lost, the processor must have at least two registers for storing memory values, rather than a single accumulator.
- Add the two values.
- Write the result from the processor to memory location A.

Thus, the execution cycle for a particular instruction may involve more than one reference to memory. Also, instead of memory references, an instruction may specify an I/O operation. With these additional considerations in mind, **Figure 3.6** provides a more detailed look at the basic instruction cycle of **Figure 3.3**. The figure is in the form of a state diagram. For any given instruction cycle, some states may be null and others may be visited more than once. The states can be described as follows:

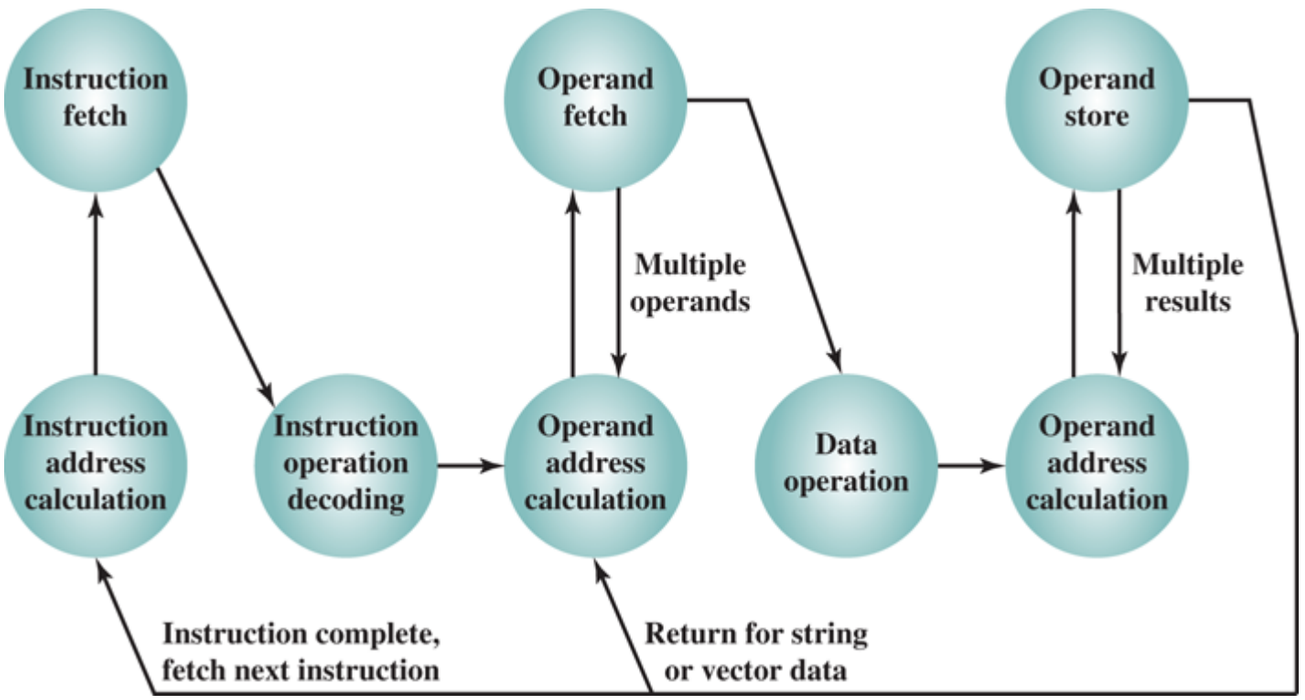


Figure 3.6 Instruction Cycle State Diagram

- **Instruction address calculation (iac):** Determine the address of the next instruction to be executed. Usually, this involves adding a fixed number to the address of the previous instruction. For example, if each instruction is 16 bits long and memory is organized into 16-bit words, then add 1 to the previous address. If instead memory is organized as individually addressable 8-bit bytes, then add 2 to the previous address.
- **Instruction fetch (if):** Read instruction from its memory location into the processor.
- **Instruction operation decoding (iod):** Analyze instruction to determine type of operation to be performed and operand(s) to be used.
- **Operand address calculation (oac):** If the operation involves reference to an operand in memory or available via I/O, then determine the address of the operand.
- **Operand fetch (of):** Fetch the operand from memory or read it in from I/O.
- **Data operation (do):** Perform the operation indicated in the instruction.
- **Operand store (os):** Write the result into memory or out to I/O.

States in the upper part of **Figure 3.6** involve an exchange between the processor and either memory

or an I/O module. States in the lower part of the diagram involve only internal processor operations. The oac state appears twice, because an instruction may involve a read, a write, or both. However, the action performed during that state is fundamentally the same in both cases, and so only a single state identifier is needed.

Also note that the diagram allows for multiple operands and multiple results, because some instructions on some machines require this. For example, the PDP-11 instruction ADD A,B results in the following sequence of states: iac, if, iod, oac, of, oac, of, do, oac, os.

Finally, on some machines, a single instruction can specify an operation to be performed on a vector (one-dimensional array) of numbers or a string (one-dimensional array) of characters. As **Figure 3.6** indicates, this would involve repetitive operand fetch and/or store operations.

Interrupts

Virtually all computers provide a mechanism by which other modules (I/O, memory) may **interrupt** the normal processing of the processor. **Table 3.1** lists the most common classes of interrupts. The specific nature of these interrupts is examined later in this book, especially in **Chapters 7** and **14**. However, we need to introduce the concept now to understand more clearly the nature of the instruction cycle and the implications of interrupts on the interconnection structure. The reader need not be concerned at this stage about the details of the generation and processing of interrupts, but only focus on the communication between modules that results from interrupts.

Table 3.1 Classes of Interrupts

Program	Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, or reference outside a user's allowed memory space.
Timer	Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.
I/O	Generated by an I/O controller, to signal normal completion of an operation, request service from the processor, or to signal a variety of error conditions.
Hardware Failure	Generated by a failure such as power failure or memory parity error.

Interrupts are provided primarily as a way to improve processing efficiency. For example, most external devices are much slower than the processor. Suppose that the processor is transferring data to a printer using the instruction cycle scheme of **Figure 3.3**. After each write operation, the processor must pause and remain idle until the printer catches up. The length of this pause may be on the order of many hundreds or even thousands of instruction cycles that do not involve memory. Clearly, this is a very wasteful use of the processor.

Figure 3.7a illustrates this state of affairs. The user program performs a series of WRITE calls interleaved with processing. Code segments 1, 2, and 3 refer to sequences of instructions that do not involve I/O. The WRITE calls are to an I/O program that is a system utility and that will perform the actual I/O operation. The I/O program consists of three sections:

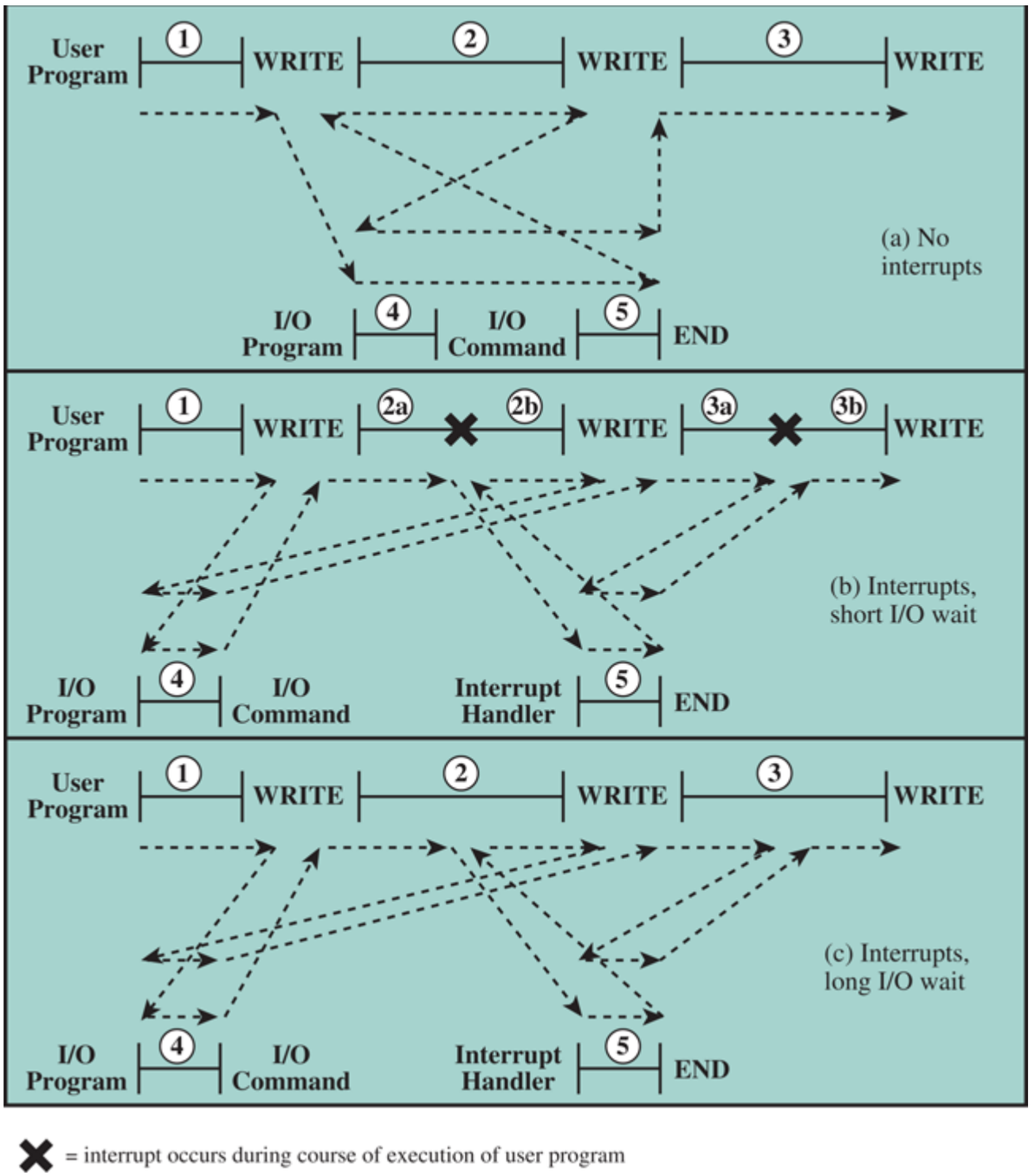


Figure 3.7 Program Flow of Control without and with Interrupts

- A sequence of instructions, labeled 4 in the figure, to prepare for the actual I/O operation. This may include copying the data to be output into a special buffer and preparing the parameters for a device command.
- The actual I/O command. Without the use of interrupts, once this command is issued, the program must wait for the I/O device to perform the requested function (or periodically poll the device). The program might wait by simply repeatedly performing a test operation to determine if the I/O operation is done.
- A sequence of instructions, labeled 5 in the figure, to complete the operation. This may include setting a flag indicating the success or failure of the operation.

Because the I/O operation may take a relatively long time to complete, the I/O program is hung up waiting for the operation to complete; hence, the user program is stopped at the point of the WRITE

call for some considerable period of time.

INTERRUPTS AND THE INSTRUCTION CYCLE

With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress. Consider the flow of control in **Figure 3.7b**. As before, the user program reaches a point at which it makes a system call in the form of a WRITE call. The I/O program that is invoked in this case consists only of the preparation code and the actual I/O command. After these few instructions have been executed, control returns to the user program. Meanwhile, the external device is busy accepting data from computer memory and printing it. This I/O operation is conducted concurrently with the execution of instructions in the user program.

When the external device becomes ready to be serviced—that is, when it is ready to accept more data from the processor—the I/O module for that external device sends an *interrupt request* signal to the processor. The processor responds by suspending operation of the current program, branching off to a program to service that particular I/O device, known as an **interrupt handler**, and resuming the original execution after the device is serviced. The points at which such interrupts occur are indicated by an asterisk in **Figure 3.7b**.

Let us try to clarify what is happening in **Figure 3.7**. We have a user program that contains two WRITE commands. There is a segment of code at the beginning, then one WRITE command, then a second segment of code, then a second WRITE command, then a third and final segment of code. The WRITE command invokes the I/O program provided by the OS. Similarly, the I/O program consists of a segment of code, followed by an I/O command, followed by another segment of code. The I/O command invokes a hardware I/O operation.

From the point of view of the user program, an interrupt is just that: an interruption of the normal sequence of execution. When the interrupt processing is completed, execution resumes (**Figure 3.8**). Thus, the user program does not have to contain any special code to accommodate interrupts; the processor and the operating system are responsible for suspending the user program and then resuming it at the same point.

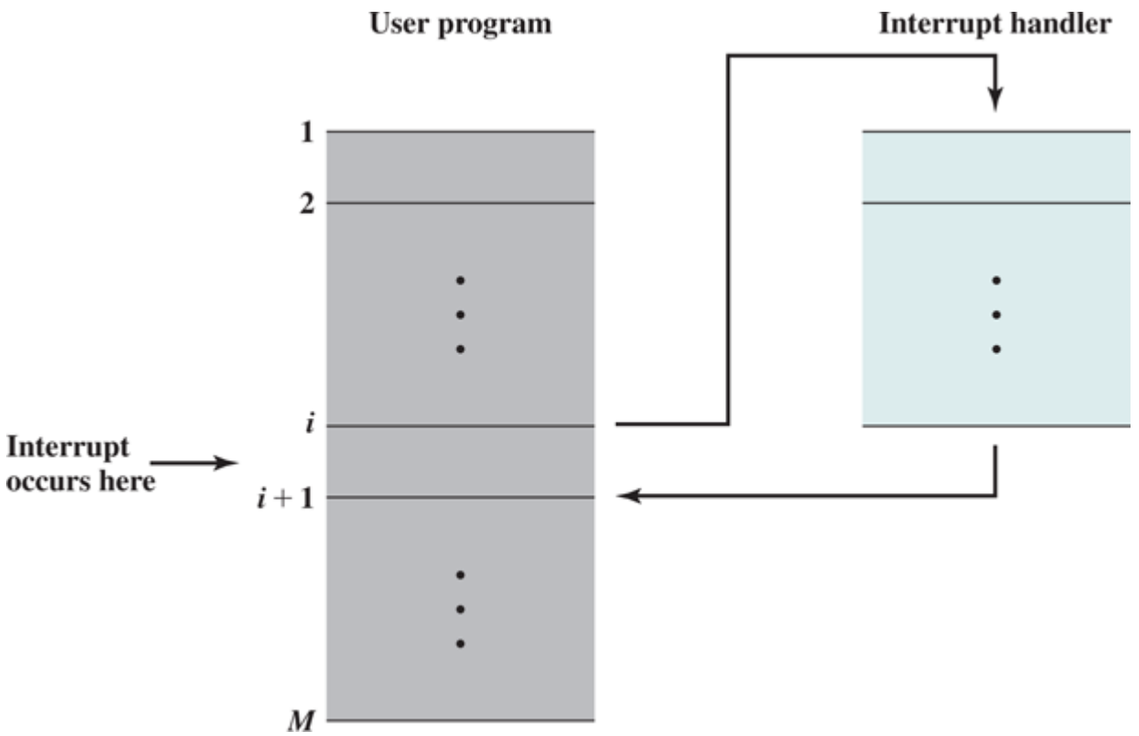


Figure 3.8 Transfer of Control via Interrupts

To accommodate interrupts, an *interrupt cycle* is added to the instruction cycle, as shown in **Figure 3.9**. In the interrupt cycle, the processor checks to see if any interrupts have occurred, indicated by the presence of an interrupt signal. If no interrupts are pending, the processor proceeds to the fetch cycle and fetches the next instruction of the current program. If an interrupt is pending, the processor does the following:

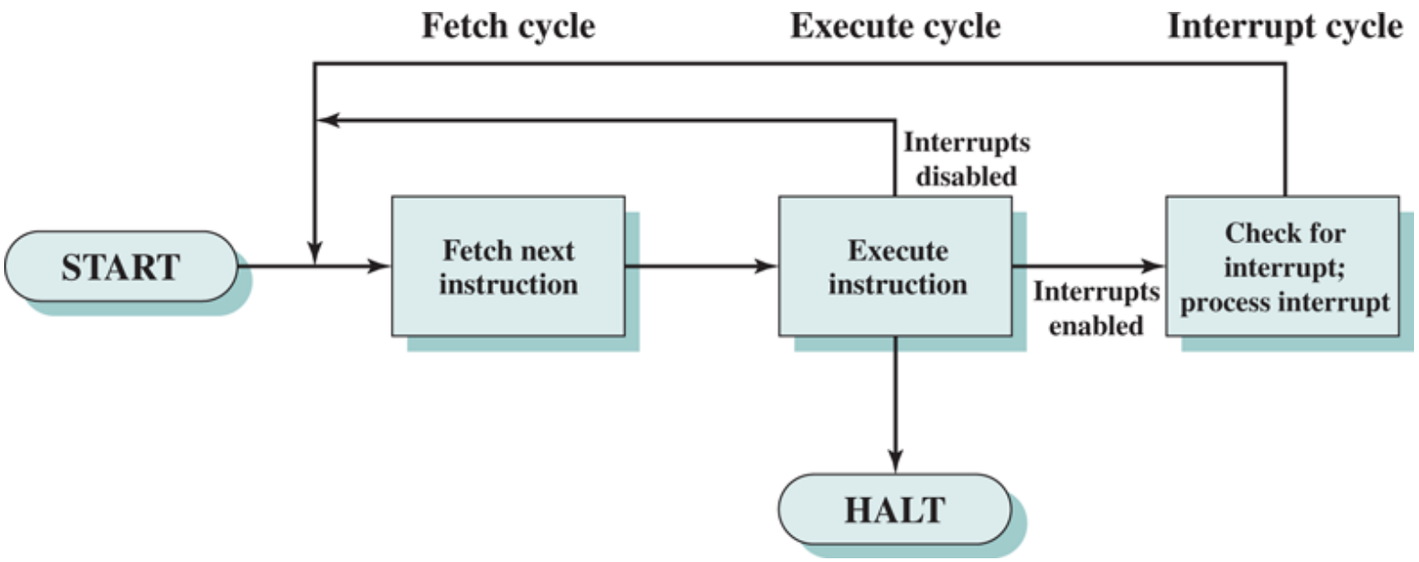


Figure 3.9 Instruction Cycle with Interrupts

- It suspends execution of the current program being executed and saves its context. This means saving the address of the next instruction to be executed (current contents of the program counter) and any other data relevant to the processor's current activity.
- It sets the program counter to the starting address of an *interrupt handler* routine.

The processor now proceeds to the fetch cycle and fetches the first instruction in the interrupt handler program, which will service the interrupt. The interrupt handler program is generally part of the operating system. Typically, this program determines the nature of the interrupt and performs whatever actions are needed. In the example we have been using, the handler determines which I/O module generated the interrupt and may branch to a program that will write more data out to that I/O module. When the interrupt handler routine is completed, the processor can resume execution of the user program at the point of interruption.

It is clear that there is some overhead involved in this process. Extra instructions must be executed (in the interrupt handler) to determine the nature of the interrupt and to decide on the appropriate action. Nevertheless, because of the relatively large amount of time that would be wasted by simply waiting on an I/O operation, the processor can be employed much more efficiently with the use of interrupts.

To appreciate the gain in efficiency, consider **Figure 3.10**, which is a timing diagram based on the flow of control in **Figures 3.7a** and **3.7b**. In this figure, user program code segments are shaded green, and I/O program code segments are shaded gray. **Figure 3.10a** shows the case in which interrupts are not used. The processor must wait while an I/O operation is performed.

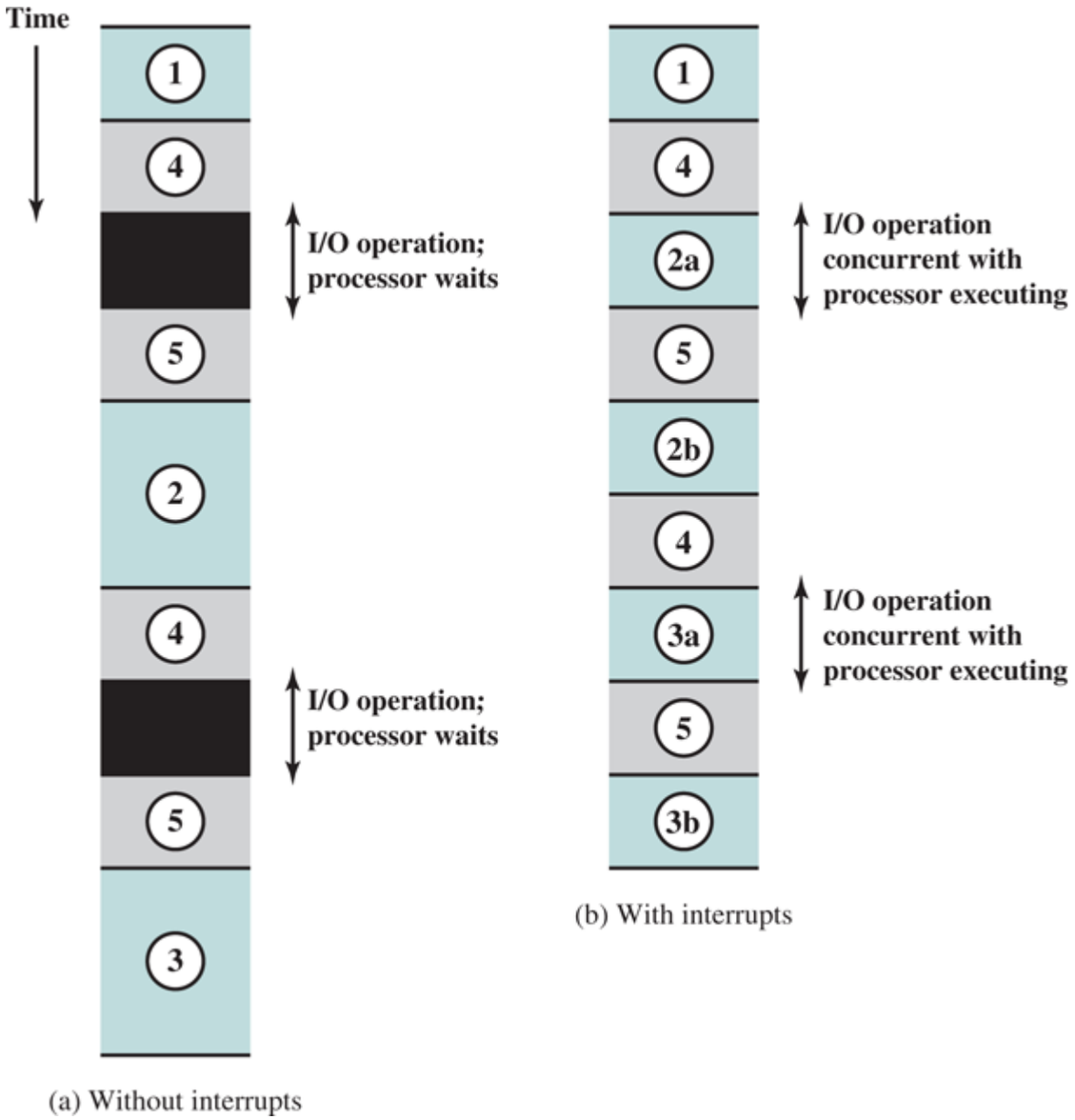


Figure 3.10 Program Timing: Short I/O Wait

Figures 3.7b and **3.10b** assume that the time required for the I/O operation is relatively short: less than the time to complete the execution of instructions between write operations in the user program. In this case, the segment of code labeled code segment 2 is interrupted. A portion of the code (2a) executes (while the I/O operation is performed) and then the interrupt occurs (upon the completion of the I/O operation). After the interrupt is serviced, execution resumes with the remainder of code segment 2 (2b).

The more typical case, especially for a slow device such as a printer, is that the I/O operation will take much more time than executing a sequence of user instructions. **Figure 3.7c** indicates this state of affairs. In this case, the user program reaches the second WRITE call before the I/O operation spawned by the first call is complete. The result is that the user program is hung up at that point. When the preceding I/O operation is completed, this new WRITE call may be processed, and a new I/O operation may be started. **Figure 3.11** shows the timing for this situation with and without the use of interrupts. We can see that there is still a gain in efficiency because part of the time during which the I/O operation is under way overlaps with the execution of user instructions.

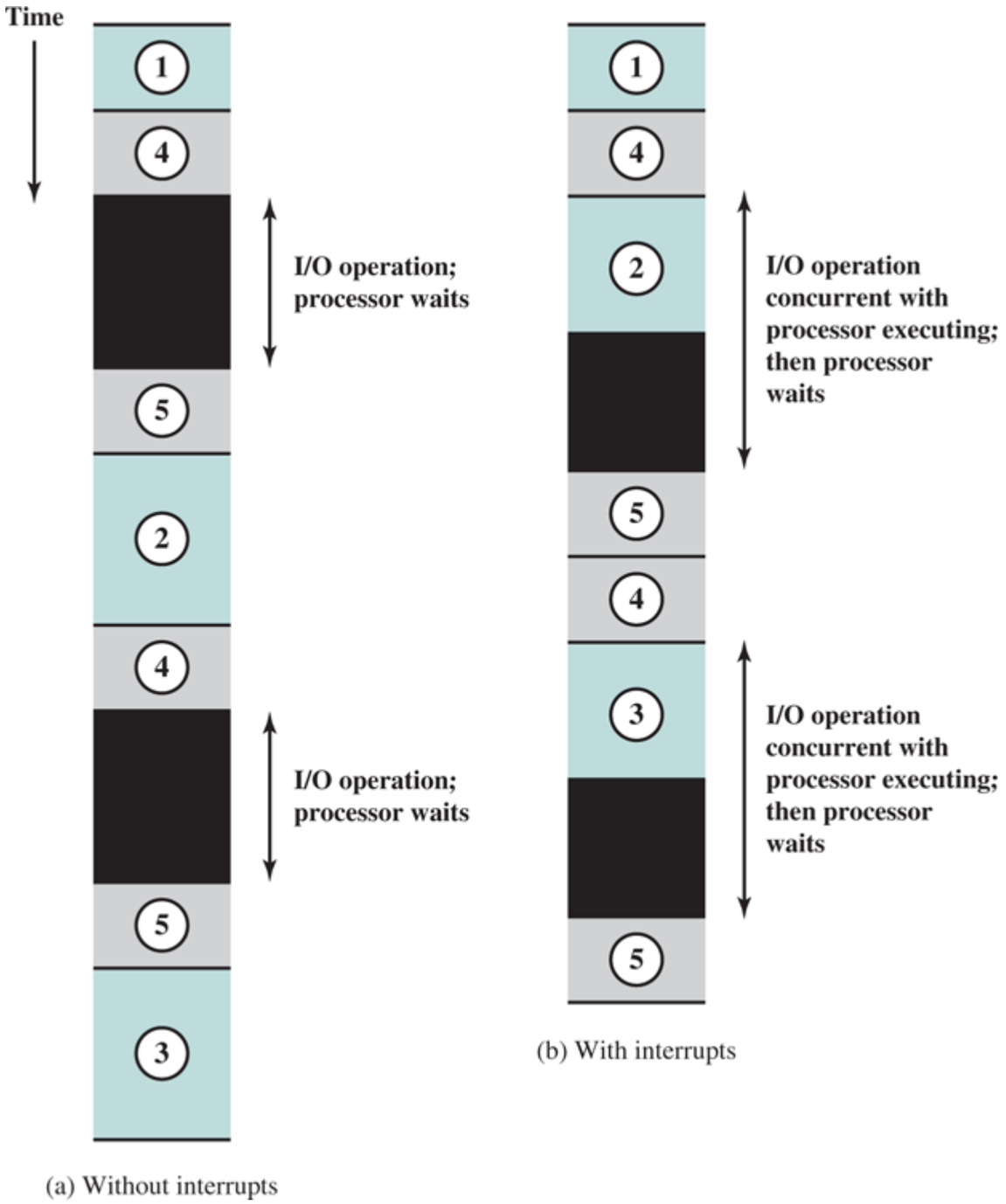


Figure 3.11 Program Timing: Long I/O Wait

Figure 3.12 shows a revised instruction cycle state diagram that includes interrupt cycle processing.

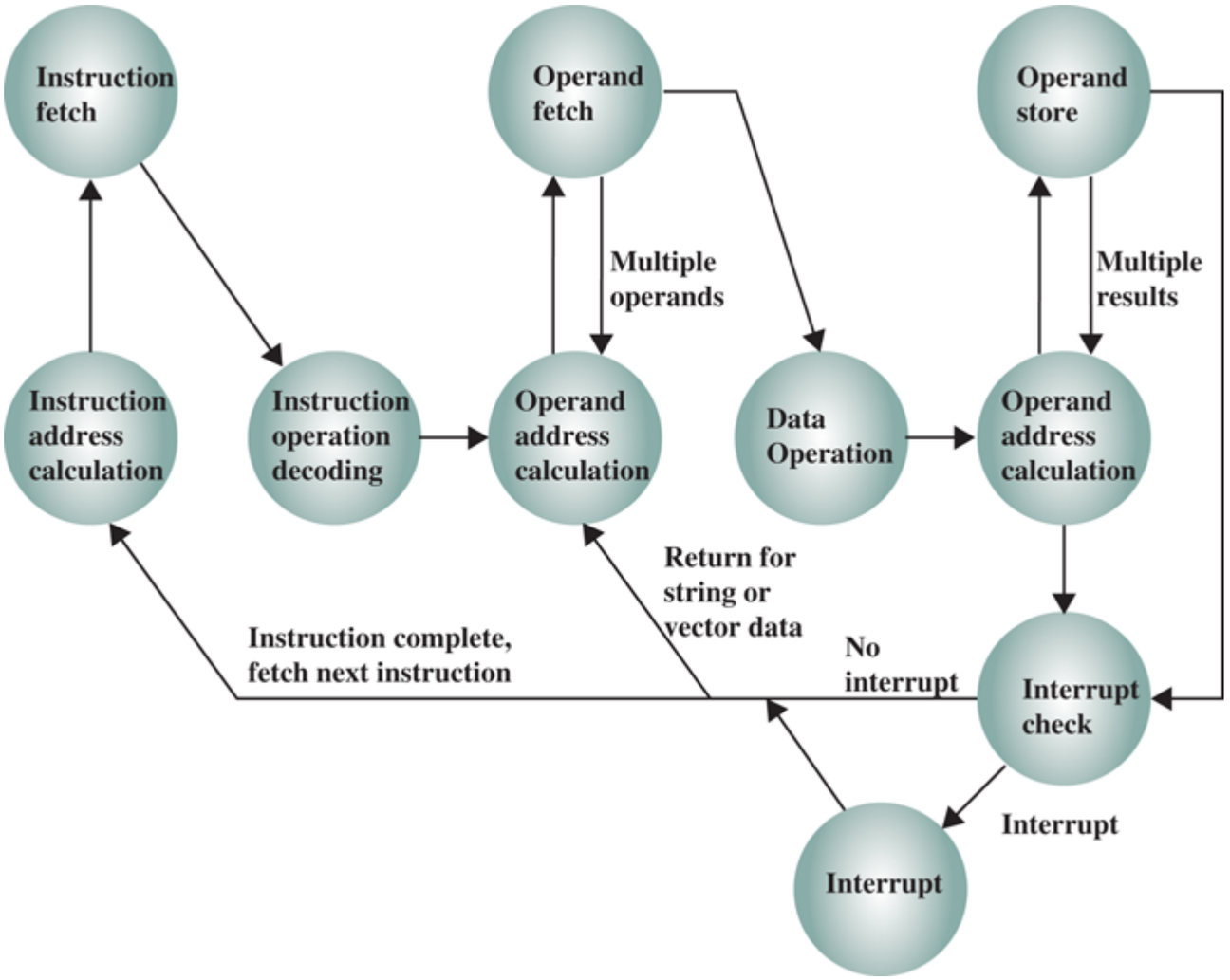
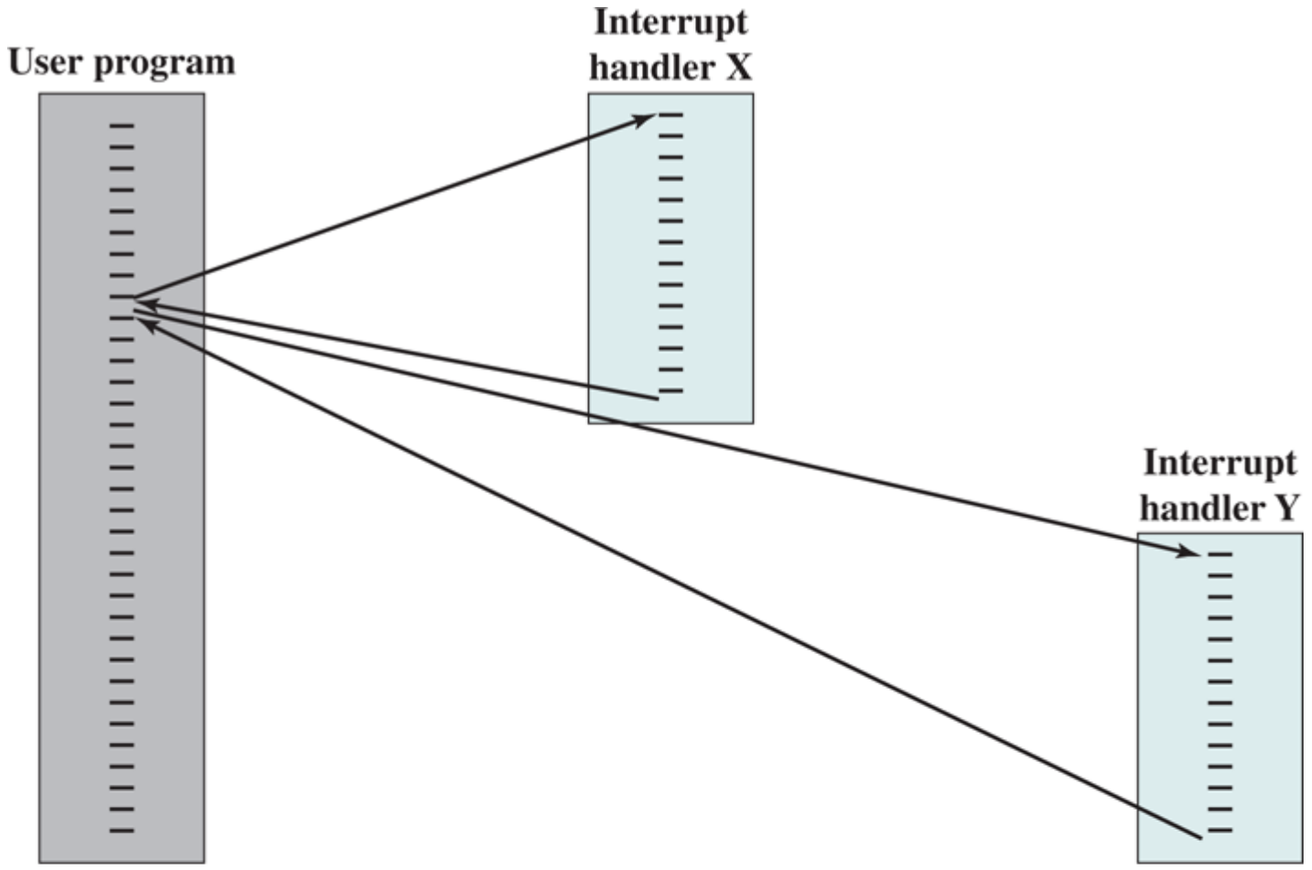


Figure 3.12 Instruction Cycle State Diagram, with Interrupts

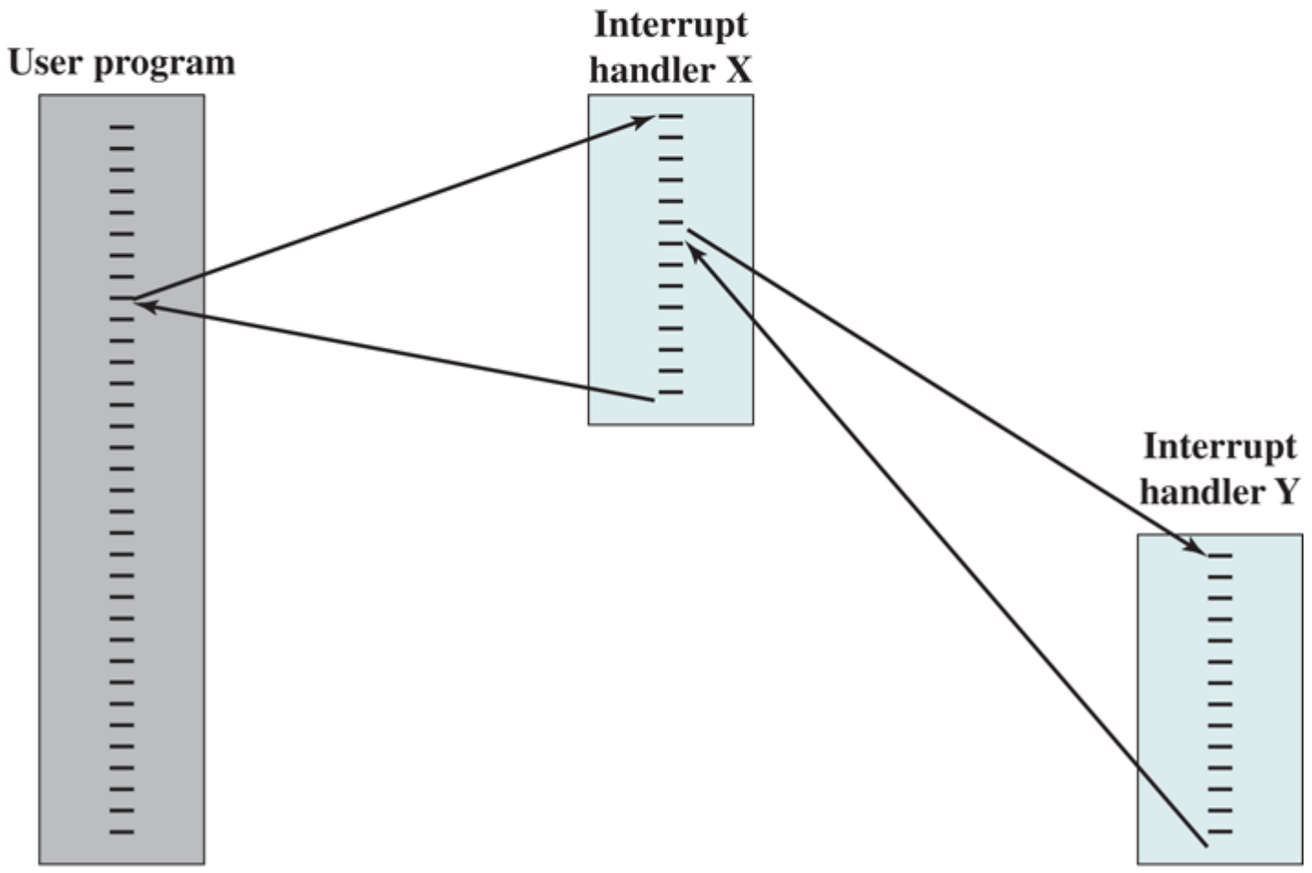
MULTIPLE INTERRUPTS

The discussion so far has focused only on the occurrence of a single interrupt. Suppose, however, that multiple interrupts can occur. For example, a program may be receiving data from a communications line and printing results. The printer will generate an interrupt every time it completes a print operation. The communication line controller will generate an interrupt every time a unit of data arrives. The unit could either be a single character or a block, depending on the nature of the communications discipline. In any case, it is possible for a communications interrupt to occur while a printer interrupt is being processed.

Two approaches can be taken to dealing with multiple interrupts. The first is to disable interrupts while an interrupt is being processed. A **disabled interrupt** simply means that the processor can and will ignore that interrupt request signal. If an interrupt occurs during this time, it generally remains pending and will be checked by the processor after the processor has enabled interrupts. Thus, when a user program is executing and an interrupt occurs, interrupts are disabled immediately. After the interrupt handler routine completes, interrupts are enabled before resuming the user program, and the processor checks to see if additional interrupts have occurred. This approach is nice and simple, as interrupts are handled in strict sequential order (**Figure 3.13a**).



(a) Sequential interrupt processing



(b) Nested interrupt processing

Figure 3.13 Transfer of Control with Multiple Interrupts

The drawback to the preceding approach is that it does not take into account relative priority or time-critical needs. For example, when input arrives from the communications line, it may need to be absorbed rapidly to make room for more input. If the first batch of input has not been processed before the second batch arrives, data may be lost.

A second approach is to define priorities for interrupts and to allow an interrupt of higher priority to cause a lower-priority interrupt handler to itself be interrupted (**Figure 3.13b**). As an example of this second approach, consider a system with three I/O devices: a printer, a disk, and a communications line, with increasing priorities of 2, 4, and 5, respectively. **Figure 3.14** illustrates a possible sequence. A user program begins at $t = 0$. At $t = 10$, a printer interrupt occurs; user information is placed on the system stack and execution continues at the printer **interrupt service routine (ISR)**. While this routine is still executing, at $t = 15$, a communications interrupt occurs. Because the communications line has higher priority than the printer, the interrupt is honored. The printer ISR is interrupted, its state is pushed onto the stack, and execution continues at the communications ISR. While this routine is executing, a disk interrupt occurs $(t = 20)$. Because this interrupt is of lower priority, it is simply held, and the communications ISR runs to completion.

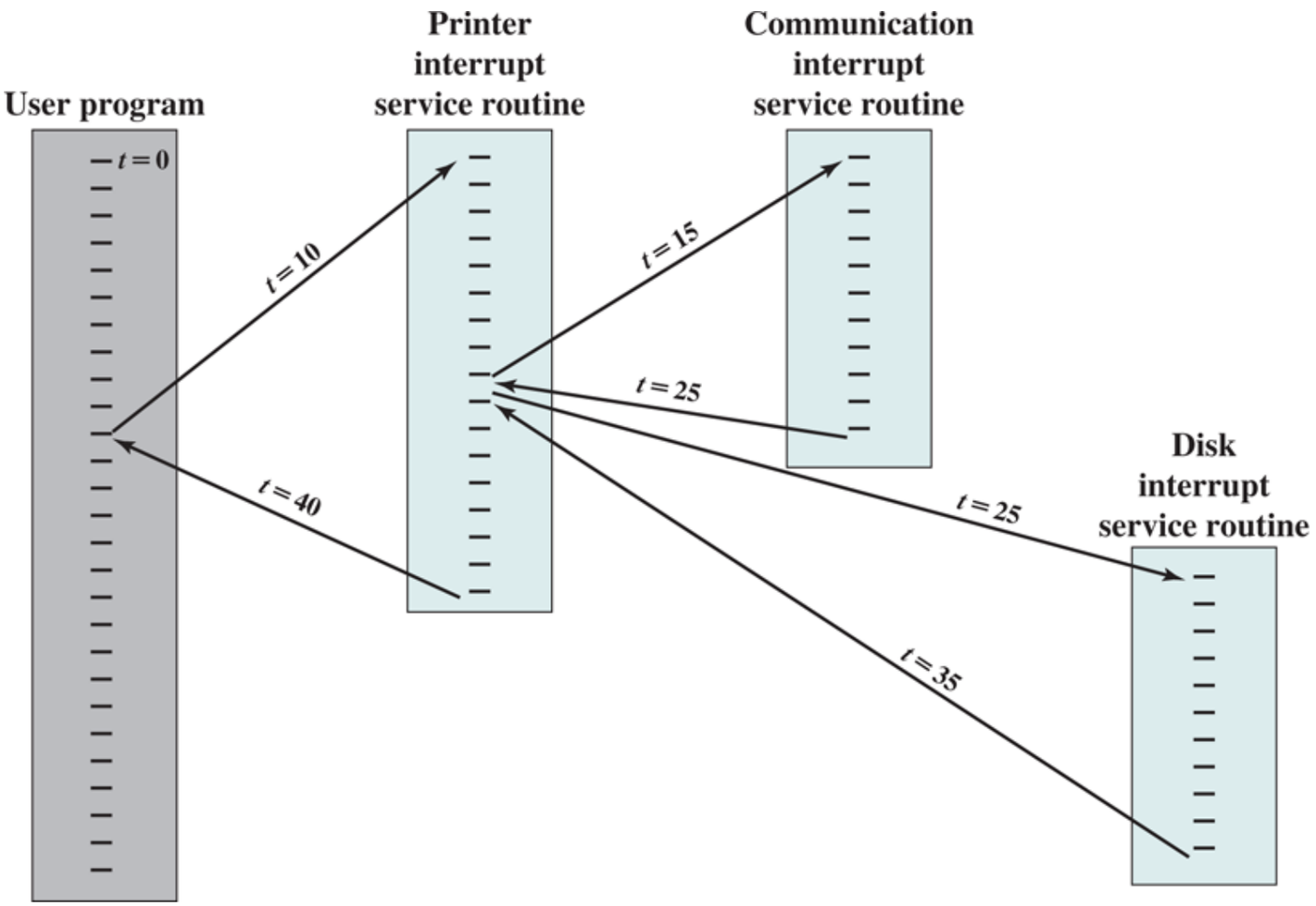


Figure 3.14 Example Time Sequence of Multiple Interrupts

When the communications ISR is complete $(t = 25)$, the previous processor state is restored, which is

the execution of the printer ISR. However, before even a single instruction in that routine can be executed, the processor honors the higher-priority disk interrupt and control transfers to the disk ISR. Only when that routine is complete $(t = 35)$ is the printer ISR resumed. When that routine completes $(t = 40)$, control finally returns to the user program.

I/O Function

Thus far, we have discussed the operation of the computer as controlled by the processor, and we have looked primarily at the interaction of processor and memory.

The discussion has only alluded to the role of the I/O component. This role is discussed in detail in **Chapter 7**, but a brief summary is in order here.

An I/O module (e.g., a disk controller) can exchange data directly with the processor. Just as the processor can initiate a read or write with memory, designating the address of a specific location, the processor can also read data from or write data to an I/O module. In this latter case, the processor identifies a specific device that is controlled by a particular I/O module. Thus, an instruction sequence similar in form to that of **Figure 3.5** could occur, with I/O instructions rather than memory-referencing instructions.

In some cases, it is desirable to allow I/O exchanges to occur directly with memory. In such a case, the processor grants to an I/O module the authority to read from or write to memory, so that the I/O-memory transfer can occur without tying up the processor. During such a transfer, the I/O module issues read or write commands to memory, relieving the processor of responsibility for the exchange. This operation is known as direct memory access (DMA) and is examined in **Chapter 7**.

3.3 Interconnection Structures

A computer consists of a set of components or modules of three basic types (processor, memory, I/O) that communicate with each other. In effect, a computer is a network of basic modules. Thus, there must be paths for connecting the modules.

The collection of paths connecting the various modules is called the *interconnection structure*. The design of this structure will depend on the exchanges that must be made among modules.

Figure 3.15 suggests the types of exchanges that are needed by indicating the major forms of input and output for each module type[2]:

[2] The wide arrows represent multiple signal lines carrying multiple bits of information in parallel. Each narrow arrow represents a single signal line.

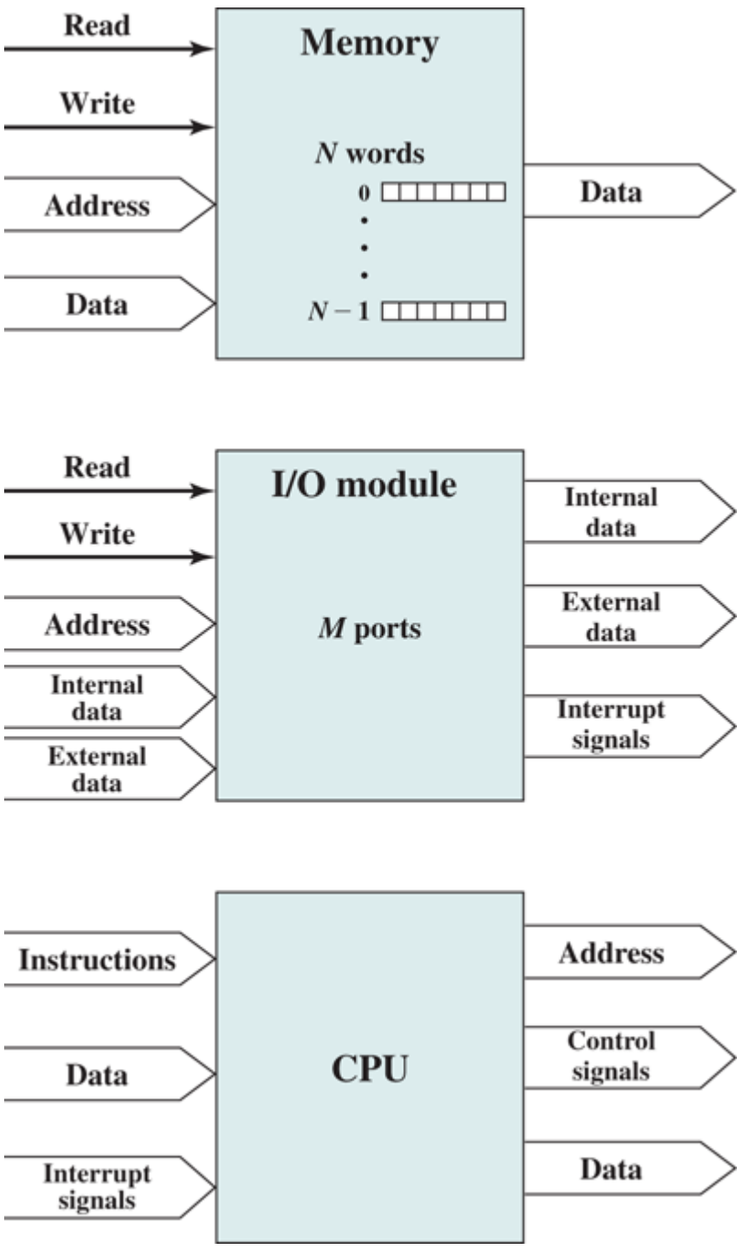


Figure 3.15 Computer Modules

- **Memory:** Typically, a memory module will consist of *N* words of equal length. Each word is assigned a unique numerical address $(0, 1, \ldots, N-1)$. A word of data can be read from or written into the memory. The nature of the operation is indicated by read and write control signals. The location for the operation is specified by an address.
- **I/O module:** From an internal (to the computer system) point of view, I/O is functionally similar to memory. There are two operations; read and write. Further, an I/O module may control more than one external device. We can refer to each of the interfaces to an external device as a *port* and give each a unique address $(\text{e.}g., 0, 1, \ldots, M-1)$. In addition, there are external data paths for the input and output of data with an external device. Finally, an I/O module may be able to send interrupt signals to the processor.
- **Processor:** The processor reads in instructions and data, writes out data after processing, and uses control signals to control the overall operation of the system. It also receives interrupt signals.

The preceding list defines the data to be exchanged. The interconnection structure must support the following types of transfers:

- **Memory to processor:** The processor reads an instruction or a unit of data from memory.
- **Processor to memory:** The processor writes a unit of data to memory.
- **I/O to processor:** The processor reads data from an I/O device via an I/O module.
- **Processor to I/O:** The processor sends data to the I/O device.
- **I/O to or from memory:** For these two cases, an I/O module is allowed to exchange data directly with memory, without going through the processor, using direct memory access.

Over the years, a number of interconnection structures have been tried. By far the most common are (1) the **bus** and various multiple-bus structures, and (2) point-to-point interconnection structures with packetized data transfer. We devote the remainder of this chapter to a discussion of these structures.

3.4 Bus Interconnection

The bus was the dominant means of computer system component interconnection for decades. For general-purpose computers, it has gradually given way to various point-to-point interconnection structures, which now dominate computer system design. However, bus structures are still commonly used for embedded systems, particularly microcontrollers. In this section, we give a brief overview of bus structure. Appendix A provides more detail.

A bus is a communication pathway connecting two or more devices. A key characteristic of a bus is that it is a shared transmission medium. Multiple devices connect to the bus, and a signal transmitted by any one device is available for reception by all other devices attached to the bus. If two devices transmit during the same time period, their signals will overlap and become garbled. Thus, only one device at a time can successfully transmit.

Typically, a bus consists of multiple communication pathways, or lines. Each line is capable of transmitting signals representing binary 1 and binary 0. Over time, a sequence of binary digits can be transmitted across a single line. Taken together, several lines of a bus can be used to transmit binary digits simultaneously (in parallel). For example, an 8-bit unit of data can be transmitted over eight bus lines.

Computer systems contain a number of different buses that provide pathways between components at various levels of the computer system hierarchy. A bus that connects major computer components (processor, memory, I/O) is called a **system bus** . The most common computer interconnection structures are based on the use of one or more system buses.

A system bus consists, typically, of from about fifty to hundreds of separate lines. Each line is assigned a particular meaning or function. Although there are many different bus designs, on any bus the lines can be classified into three functional groups (**Figure 3.16**): data, address, and control lines. In addition, there may be power distribution lines that supply power to the attached modules.

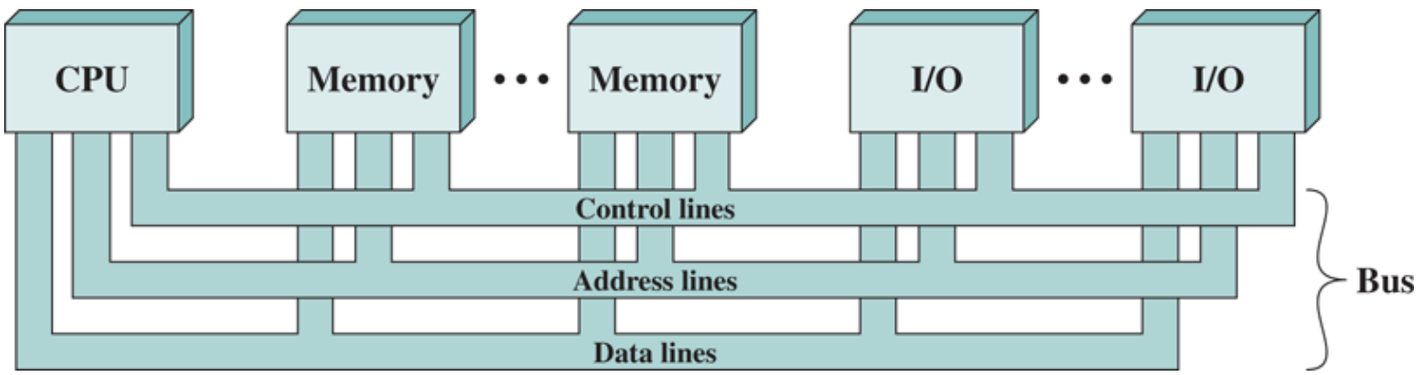


Figure 3.16 Bus Interconnection Scheme

The **data lines** provide a path for moving data among system modules. These lines, collectively, are called the **data bus** . The data bus may consist of 32, 64, 128, or even more separate lines, the number of lines being referred to as the *width* of the data bus. Because each line can carry only one bit at a time, the number of lines determines how many bits can be transferred at a time. The width of the data bus is a key factor in determining overall system performance. For example, if the data bus is 32 bits wide and each instruction is 64 bits long, then the processor must access the memory module twice during each instruction cycle.

The **address lines** are used to designate the source or destination of the data on the data bus. For example, if the processor wishes to read a word (8, 16, or 32 bits) of data from memory, it puts the address of the desired word on the address lines. Clearly, the width of the **address bus** determines the maximum possible memory capacity of the system. Furthermore, the address lines are generally also used to address I/O ports. Typically, the higher-order bits are used to select a particular module on the bus, and the lower-order bits select a memory location or I/O port within the module. For example, on an 8-bit address bus, address 01111111 and below might reference locations in a memory module (module 0) with 128 words of memory, and address 10000000 and above refer to devices attached to an I/O module (module 1).

The **control lines** are used to control the access to and the use of the data and address lines. Because the data and address lines are shared by all components, there must be a means of controlling their use. Control signals transmit both command and timing information among system modules. Timing signals indicate the validity of data and address information. Command signals specify operations to be performed. Typical control lines include:

- **Memory write:** causes data on the bus to be written into the addressed location.
- **Memory read:** causes data from the addressed location to be placed on the bus.
- **I/O write:** causes data on the bus to be output to the addressed I/O port.
- **I/O read:** causes data from the addressed I/O port to be placed on the bus.
- **Transfer ACK:** indicates that data have been accepted from or placed on the bus.
- **Bus request:** indicates that a module needs to gain control of the bus.
- **Bus grant:** indicates that a requesting module has been granted control of the bus.
- **Interrupt request:** indicates that an interrupt is pending.
- **Interrupt ACK:** acknowledges that the pending interrupt has been recognized.
- **Clock:** is used to synchronize operations.
- **Reset:** initializes all modules.

The operation of the bus is as follows. If one module wishes to send data to another, it must do two things: (1) obtain the use of the bus, and (2) transfer data via the bus. If one module wishes to request data from another module, it must (1) obtain the use of the bus, and (2) transfer a request to the other module over the appropriate control and address lines. It must then wait for that second module to send the data.

3.5 Point-to-Point Interconnect

The shared bus architecture was the standard approach to interconnection between the processor and other components (memory, I/O, and so on) for decades. But contemporary systems increasingly rely on point-to-point interconnection rather than shared buses.

The principal reason driving the change from bus to point-to-point interconnect was the electrical constraints encountered with increasing the frequency of wide synchronous buses. At higher and higher data rates, it becomes increasingly difficult to perform the synchronization and **arbitration** functions in a timely fashion. Further, with the advent of multicore chips, with multiple processors and significant memory on a single chip, it was found that the use of a conventional shared bus on the same chip magnified the difficulties of increasing bus data rate and reducing bus latency to keep up with the processors. Compared to the shared bus, the point-to-point interconnect has lower latency, higher data rate, and better scalability.

In this section, we look at an important and representative example of the point-to-point interconnect approach: Intel's **QuickPath Interconnect (QPI)**, which was introduced in 2008.

The following are significant characteristics of QPI and other point-to-point interconnect schemes:

- **Multiple direct connections:** Multiple components within the system enjoy direct pairwise connections to other components. This eliminates the need for arbitration found in shared transmission systems.
- **Layered protocol architecture:** As found in network environments, such as TCP/IP-based data networks, these processor-level interconnects use a layered protocol architecture, rather than the simple use of control signals found in shared bus arrangements.
- **Packetized data transfer:** Data are not sent as a raw bit stream. Rather, data are sent as a sequence of packets, each of which includes control headers and error control codes.

Figure 3.17 illustrates a typical use of QPI on a multicore computer. The QPI links (indicated by the green arrow pairs in the figure) form a switching fabric that enables data to move throughout the network. Direct QPI connections can be established between each pair of core processors. If core A in **Figure 3.17** needs to access the memory controller in core D, it sends its request through either cores B or C, which must in turn forward that request on to the memory controller in core D. Similarly, larger systems with eight or more processors can be built using processors with three links and routing traffic through intermediate processors.

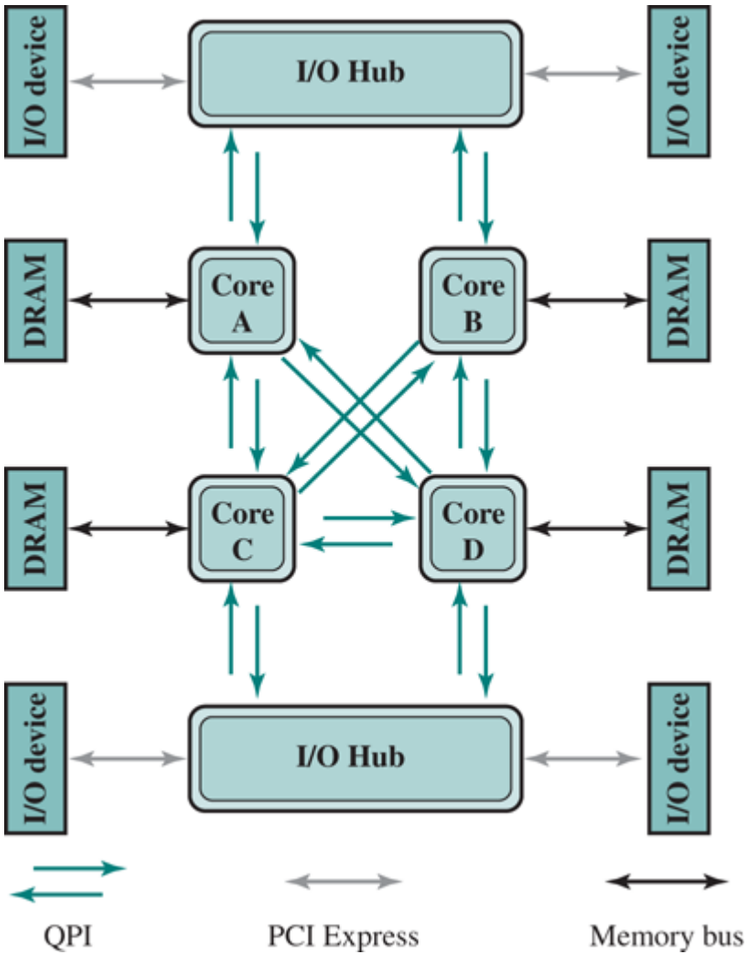


Figure 3.17 Multicore Configuration Using QPI

In addition, QPI is used to connect to an I/O module, called an I/O hub (IOH). The IOH acts as a switch directing traffic to and from I/O devices. Typically in newer systems, the link from the IOH to the I/O device controller uses an interconnect technology called PCI Express (PCIe), described later in this chapter. The IOH translates between the QPI protocols and formats and the PCIe protocols and formats. A core also links to a main memory module (typically the memory uses dynamic access random memory (DRAM) technology) using a dedicated memory bus.

QPI is defined as a four-layer protocol architecture, encompassing the following layers (**Figure 3.18**):

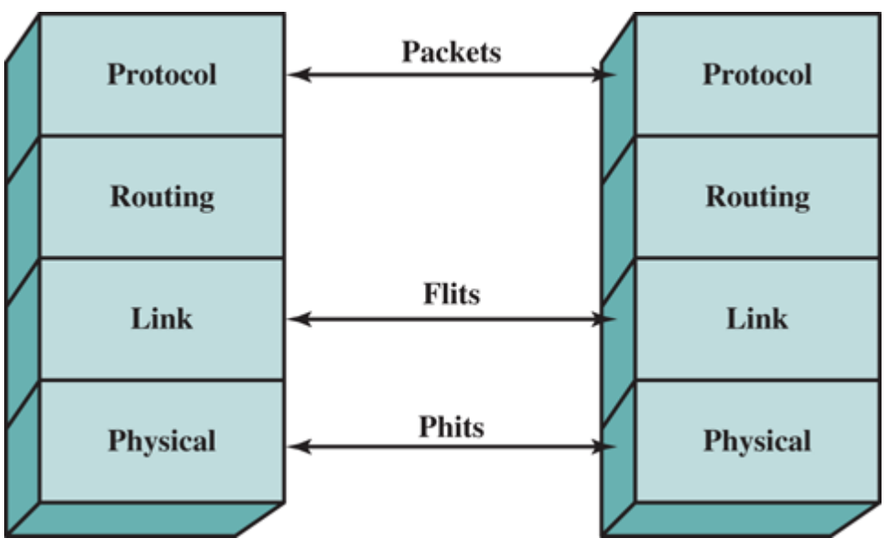


Figure 3.18 QPI Layers

- **Physical:** Consists of the actual wires carrying the signals, as well as circuitry and logic to support ancillary features required in the transmission and receipt of the 1s and 0s. The unit of transfer at the Physical layer is 20 bits, which is called a **Phit** (physical unit).
- **Link:** Responsible for reliable transmission and flow control. The Link layer's unit of transfer is an 80-bit **Flit** (flow control unit).
- **Routing:** Provides the framework for directing packets through the fabric.
- **Protocol:** The high-level set of rules for exchanging **packets** of data between devices. A packet is comprised of an integral number of Flits.

QPI Physical Layer

Figure 3.19 shows the physical architecture of a QPI port. The QPI port consists of 84 individual links grouped as follows. Each data path consists of a pair of wires that transmits data one bit at a time; the pair is referred to as a **lane**. There are 20 data lanes in each direction (transmit and receive), plus a clock lane in each direction. Thus, QPI is capable of transmitting 20 bits in parallel in each direction. The 20-bit unit is referred to as a *phit*. Typical signaling speeds of the link in current products calls for operation at 6.4 GT/s (transfers per second). At 20 bits per transfer, that adds up to 16 GB/s, and since QPI links involve dedicated bidirectional pairs, the total capacity is 32 GB/s.

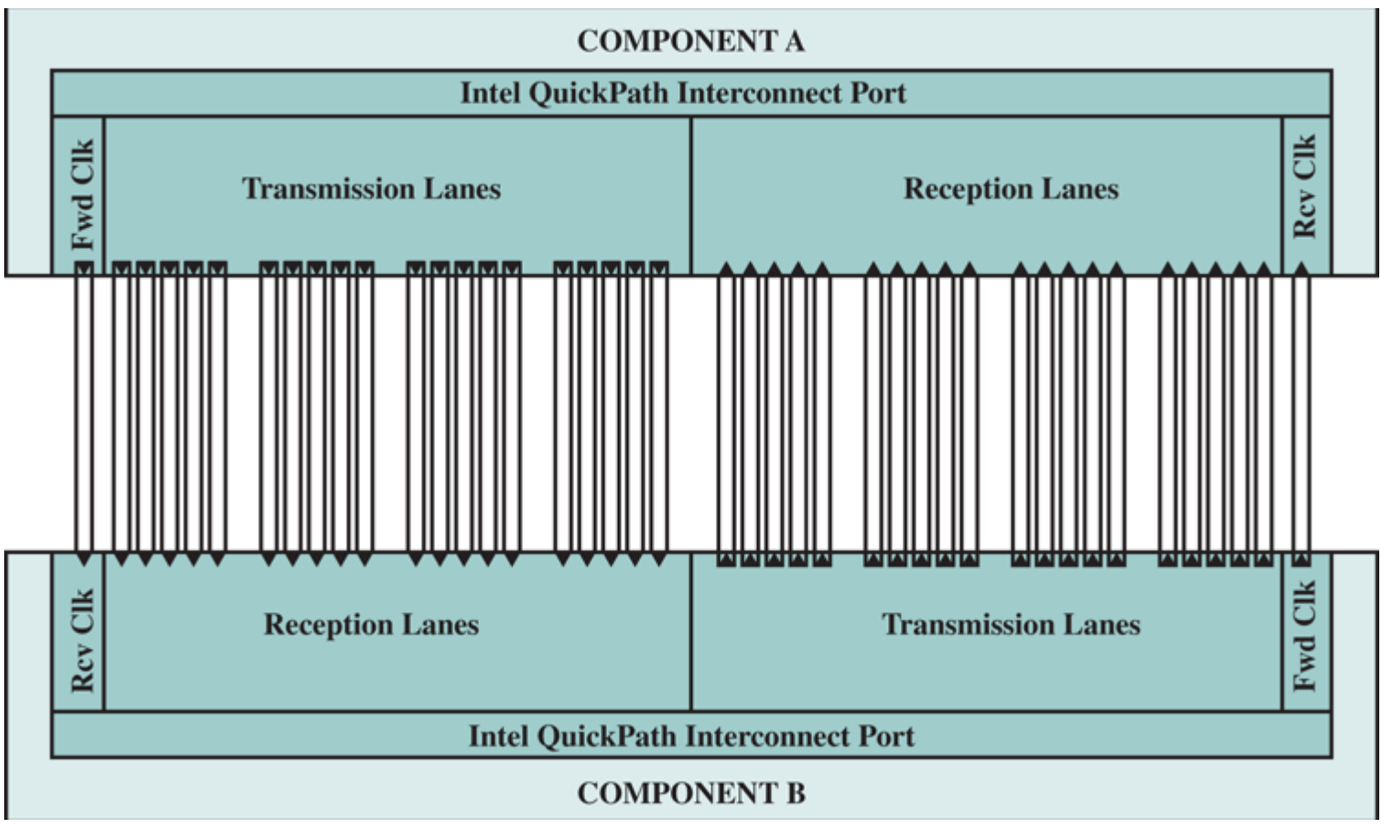


Figure 3.19 Physical Interface of the Intel QPI Interconnect

The lanes in each direction are grouped into four quadrants of 5 lanes each. In some applications, the link can also operate at half or quarter widths in order to reduce power consumption or work around failures.

The form of transmission on each lane is known as **differential signaling**, or **balanced transmission**. With balanced transmission, signals are transmitted as a current that travels down one conductor and returns on the other. The binary value depends on the voltage difference. Typically, one line has a positive voltage value and the other line has zero voltage, and one line is associated with binary 1 and the other is associated with binary 0. Specifically, the technique used by QPI is known as *low-voltage differential signaling* (LVDS). In a typical implementation, the transmitter injects a small current into one wire or the other, depending on the logic level to be sent. The current passes through a resistor at the receiving end, and then returns in the opposite direction along the other wire. The receiver senses the polarity of the voltage across the resistor to determine the logic level.

Another function performed by the physical layer is that it manages the translation between 80-bit flits and 20-bit phits using a technique known as **multilane distribution**. The flits can be considered as a bit stream that is distributed across the data lanes in a round-robin fashion (first bit to first lane, second bit to second lane, etc.), as illustrated in **Figure 3.20**. This approach enables QPI to achieve very high data rates by implementing the physical link between two ports as multiple parallel channels.

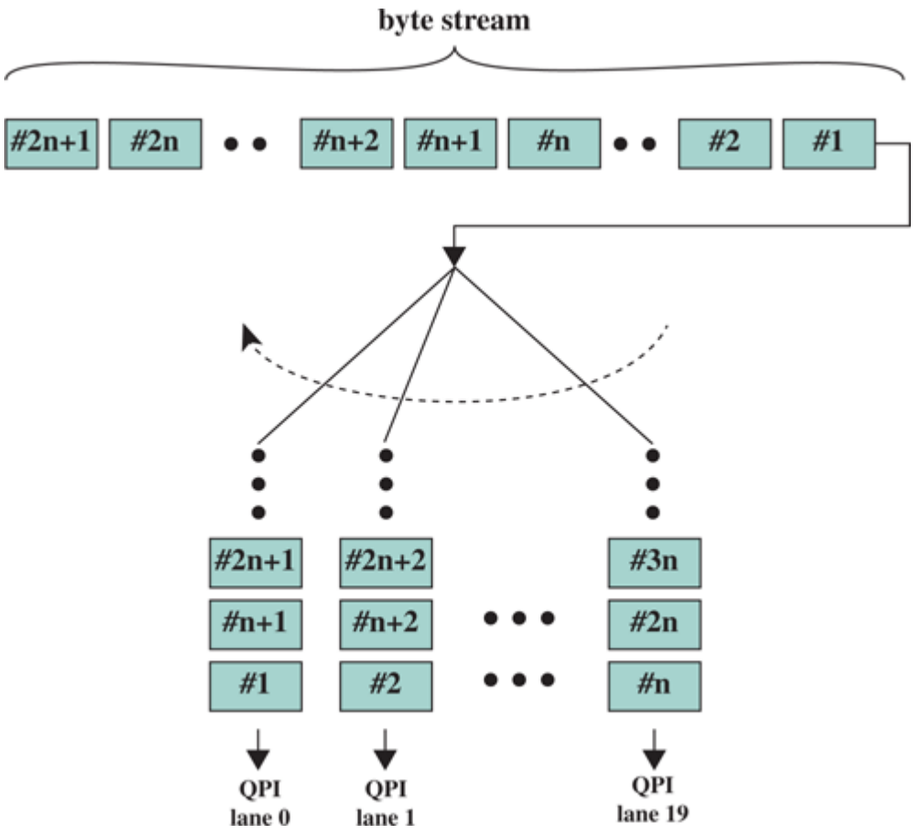


Figure 3.20 QPI Multilane Distribution

QPI Link Layer

The QPI link layer performs two key functions: flow control and error control. These functions are performed as part of the QPI link layer protocol, and operate on the level of the flit (flow control unit). Each flit consists of a 72-bit message payload and an 8-bit error control code called a cyclic redundancy check (CRC). We discuss error control codes in **Chapter 5**.

A flit payload may consist of data or message information. The data flits transfer the actual bits of data between cores or between a core and an IOH. The message flits are used for such functions as flow control, error control, and cache coherence. We discuss cache coherence in **Chapters 5** and **17**.

The **flow control function** is needed to ensure that a sending QPI entity does not overwhelm a receiving QPI entity by sending data faster than the receiver can process the data and clear buffers for more incoming data. To control the flow of data, QPI makes use of a credit scheme. During initialization, a sender is given a set number of credits to send flits to a receiver. Whenever a flit is sent to the receiver, the sender decrements its credit counters by one credit. Whenever a buffer is freed at the receiver, a credit is returned to the sender for that buffer. Thus, the receiver controls that pace at which data is transmitted over a QPI link.

Occasionally, a bit transmitted at the physical layer is changed during transmission, due to noise or some other phenomenon. The **error control function** at the link layer detects and recovers from such bit errors, and so isolates higher layers from experiencing bit errors. The procedure works as follows for a flow of data from system A to system B:

1. As mentioned, each 80-bit flit includes an 8-bit CRC field. The CRC is a function of the value of the remaining 72 bits. On transmission, A calculates a CRC value for each flit and inserts that value into the flit.
2. When a flit is received, B calculates a CRC value for the 72-bit payload and compares this value with the value of the incoming CRC value in the flit. If the two CRC values do not match, an error has been detected.
3. When B detects an error, it sends a request to A to retransmit the flit that is in error. However, because A may have had sufficient credit to send a stream of flits, so that additional flits have been transmitted after the flit in error and before A receives the request to retransmit. Therefore, the request is for A to back up and retransmit the damaged flit plus all subsequent flits.

QPI Routing Layer

The routing layer is used to determine the course that a packet will traverse across the available system interconnects. Routing tables are defined by firmware and describe the possible paths that a packet can follow. In small configurations, such as a two-socket platform, the routing options are limited and the routing tables quite simple. For larger systems, the routing table options are more complex, giving the flexibility of routing and rerouting traffic depending on how (1) devices are populated in the platform, (2) system resources are partitioned, and (3) reliability events result in mapping around a failing resource.

QPI Protocol Layer

In this layer, the packet is defined as the unit of transfer. The packet contents definition is standardized with some flexibility allowed to meet differing market segment requirements. One key function performed at this level is a cache coherency protocol, which deals with making sure that main memory values held in multiple caches are consistent. A typical data packet payload is a block of data being sent to or from a cache.

3.6 PCI Express

The **peripheral component interconnect (PCI)** is a popular high-bandwidth, processor-independent bus that can function as a mezzanine or peripheral bus. Compared with other common bus specifications, PCI delivers better system performance for high-speed I/O subsystems (e.g., graphic display adapters, network interface controllers, and disk controllers).

Intel began work on PCI in 1990 for its Pentium-based systems. Intel soon released all the patents to the public domain and promoted the creation of an industry association, the PCI Special Interest Group (SIG), to develop further and maintain the compatibility of the PCI specifications. The result is that PCI has been widely adopted and is finding increasing use in personal computer, workstation, and server systems. Because the specification is in the public domain and is supported by a broad cross-section of the microprocessor and peripheral industry, PCI products built by different vendors are compatible.

As with the system bus discussed in the preceding sections, the bus-based PCI scheme has not been able to keep pace with the data rate demands of attached devices. Accordingly, a new version, known as **PCI Express (PCIe)** has been developed. PCIe, as with QPI, is a point-to-point interconnect scheme intended to replace bus-based schemes such as PCI.

A key requirement for PCIe is high capacity to support the needs of higher data rate I/O devices, such as Gigabit Ethernet. Another requirement deals with the need to support time-dependent data streams. Applications such as video-on-demand and audio redistribution are putting real-time constraints on servers too. Many communications applications and embedded PC control systems also process data in real-time. Today's platforms must also deal with multiple concurrent transfers at ever-increasing data rates. It is no longer acceptable to treat all data as equal—it is more important, for example, to process streaming data first since late real-time data is as useless as no data. Data needs to be tagged so that an I/O system can prioritize its flow throughout the platform.

PCI Physical and Logical Architecture

Figure 3.21 shows a typical configuration that supports the use of PCIe. A **root complex** device, also referred to as a *chipset* or a *host bridge*, connects the processor and memory subsystem to the PCI Express switch fabric comprising one or more PCIe and PCIe switch devices. The root complex acts as a buffering device, to deal with differences in data rates between I/O controllers and memory and processor components. The root complex also translates between PCIe transaction formats and the processor and memory signal and control requirements. The chipset will typically support multiple PCIe ports, some of which attach directly to a PCIe device, and one or more that attach to a switch that manages multiple PCIe streams. PCIe links from the chipset may attach to the following kinds of devices that implement PCIe:

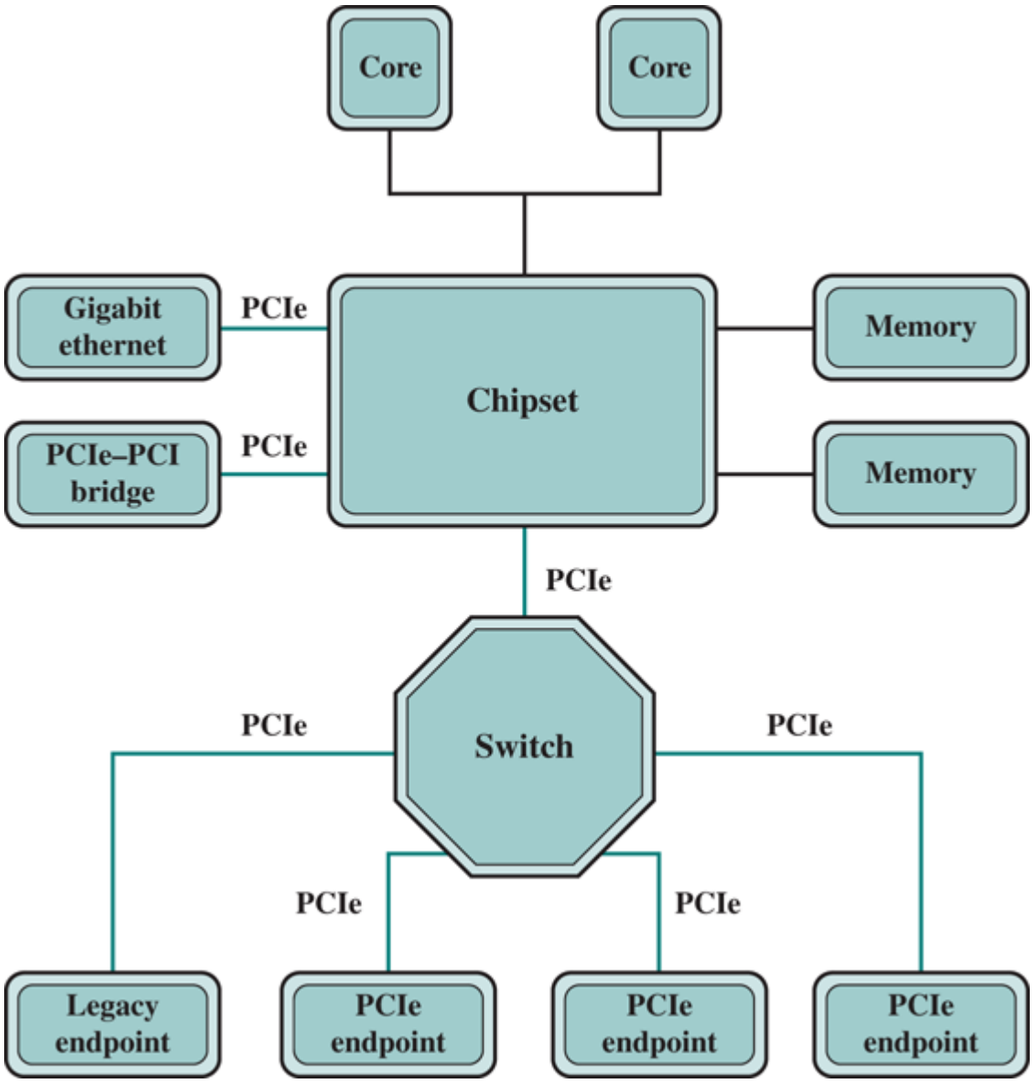


Figure 3.21 Typical Configuration Using PCIe

- **Switch:** The switch manages multiple PCIe streams.
- **PCIe endpoint:** An I/O device or controller that implements PCIe, such as a Gigabit ethernet switch, a graphics or video controller, disk interface, or a communications controller.
- **Legacy endpoint:** Legacy endpoint category is intended for existing designs that have been migrated to PCI Express, and it allows legacy behaviors such as use of I/O space and locked transactions. PCI Express endpoints are not permitted to require the use of I/O space at runtime and must not use locked transactions. By distinguishing these categories, it is possible for a system designer to restrict or eliminate legacy behaviors that have negative impacts on system performance and robustness.
- **PCIe/PCI bridge:** Allows older PCI devices to be connected to PCIe-based systems.

As with QPI, PCIe interactions are defined using a protocol architecture. The PCIe protocol architecture encompasses the following layers (**Figure 3.22**):

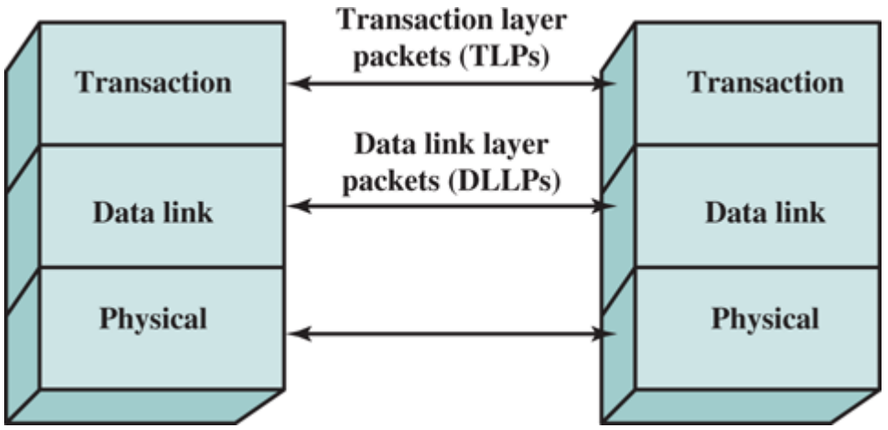


Figure 3.22 PCIe Protocol Layers

- **Physical:** Consists of the actual wires carrying the signals, as well as circuitry and logic to support ancillary features required in the transmission and receipt of the 1s and 0s.
- **Data link:** Is responsible for reliable transmission and flow control. Data packets generated and consumed by the DLL are called Data Link Layer Packets (DLLPs).
- **Transaction:** Generates and consumes data packets used to implement load/store data transfer mechanisms and also manages the flow control of those packets between the two components on a link. Data packets generated and consumed by the TL are called Transaction Layer Packets (TLPs).

Above the TL are software layers that generate read and write requests that are transported by the transaction layer to the I/O devices using a packet-based transaction protocol.

PCIe Physical Layer

Similar to QPI, PCIe is a point-to-point architecture. Each PCIe port consists of a number of bidirectional lanes (note that in QPI, the lane refers to transfer in one direction only). Transfer in each direction in a lane is by means of differential signaling over a pair of wires. A PCI port can provide 1, 4, 6, 16, or 32 lanes. In what follows, we refer to the PCIe 3.0 specification, introduced in late 2010.

As with QPI, PCIe uses a multilane distribution technique. **Figure 3.23** shows an example for a PCIe port consisting of four lanes. Data are distributed to the four lanes 1 byte at a time using a simple round-robin scheme. At each physical lane, data are buffered and processed 16 bytes (128 bits) at a time. Each block of 128 bits is encoded into a unique 130-bit codeword for transmission; this is referred to as 128b/130b encoding. Thus, the effective data rate of an individual lane is reduced by a factor of 128/130.

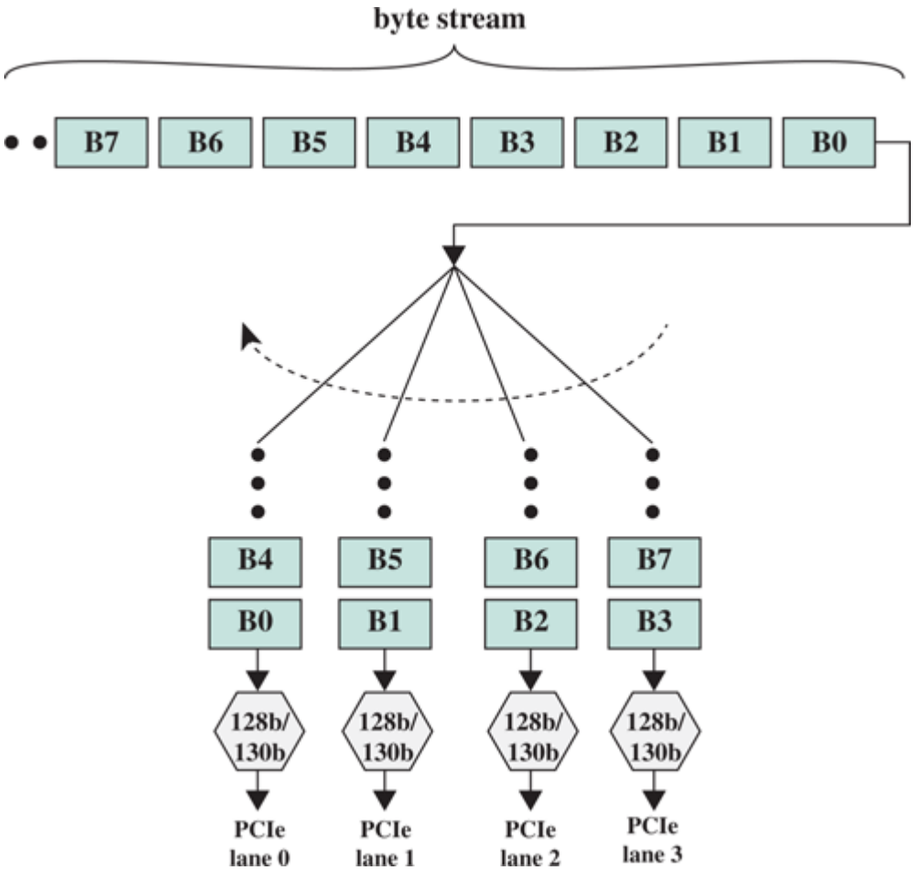


Figure 3.23 PCIe Multilane Distribution

To understand the rationale for the 128b/130b encoding, note that unlike QPI, PCIe does not use its clock line to synchronize the bit stream. That is, the clock line is not used to determine the start and end point of each incoming bit; it is used for other signaling purposes only. However, it is necessary for the receiver to be synchronized with the transmitter, so that the receiver knows when each bit begins and ends. If there is any drift between the clocks used for bit transmission and reception of the transmitter and receiver, errors may occur. To compensate for the possibility of drift, PCIe relies on the receiver synchronizing with the transmitter based on the transmitted signal. As with QPI, PCIe uses differential signaling over a pair of wires. Synchronization can be achieved by the receiver looking for transitions in the data and synchronizing its clock to the transition. However, consider that with a long string of 1s or 0s using differential signaling, the output is a constant voltage over a long period of time. Under these circumstances, any drift between the clocks of the transmitter and receiver will result in loss of synchronization between the two.

A common approach, and the one used in PCIe 3.0, to overcoming the problem of a long string of bits of one value is scrambling. Scrambling, which does not increase the number of bits to be transmitted, is a mapping technique that tends to make the data appear more random. At the receiving end, a descrambling algorithm recovers the original data sequence. The scrambling tends to spread out the number of transitions so that they appear at the receiver more uniformly spaced, which is good for synchronization. Also, other transmission properties, such as spectral properties, are enhanced if the data are more nearly of a random nature rather than constant or repetitive.

Another technique that can aid in synchronization is encoding, in which additional bits are inserted into the bit stream to force transitions. For PCIe 3.0, each group of 128 bits of input is mapped into a 130-bit block by adding a 2-bit block sync header. The value of the header is 10 for a data block and 01 for what is called an *ordered set block*, which refers to a link-level information block.

Figure 3.24 illustrates the use of scrambling and encoding. Data to be transmitted are fed into a scrambler. The scrambled output is then fed into a 128b/130b encoder, which buffers 128 bits and then maps the 128-bit block into a 130-bit block. This block then passes through a parallel-to-serial converter and is transmitted one bit at a time using differential signaling.

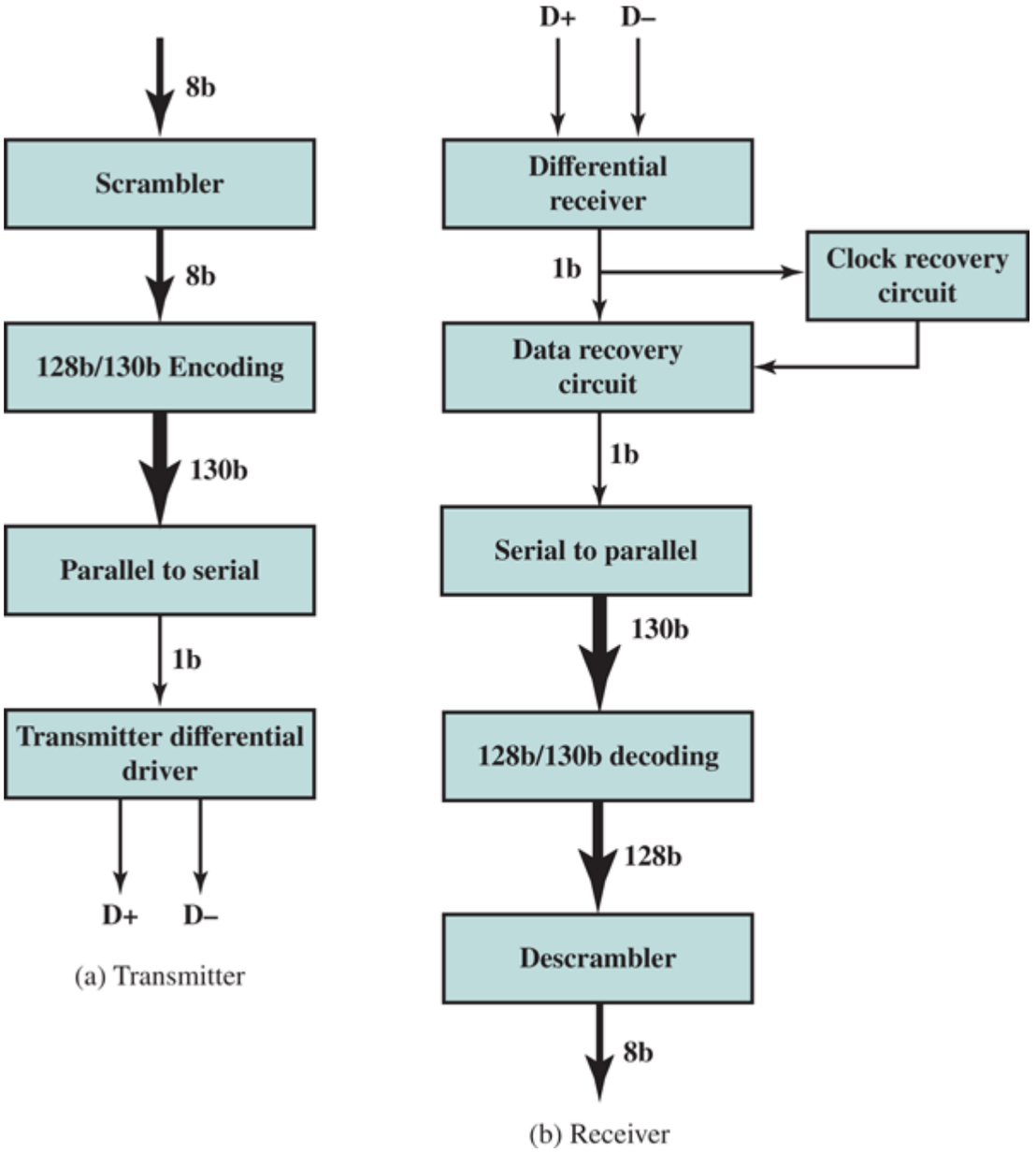


Figure 3.24 PCIe Transmit and Receive Block Diagrams

At the receiver, a clock is synchronized to the incoming data to recover the bit stream. This then passes through a serial-to-parallel converter to produce a stream of 130-bit blocks. Each block is passed through a 128b/130b decoder to recover the original scrambled bit pattern, which is then descrambled to produce the original bit stream.

Using these techniques, a data rate of 16 GB/s can be achieved. One final detail to mention; each transmission of a block of data over a PCI link begins and ends with an 8-bit framing sequence intended to give the receiver time to synchronize with the incoming physical layer bit stream.

PCIe Transaction Layer

The transaction layer (TL) receives read and write requests from the software above the TL and creates request packets for transmission to a destination via the link layer. Most transactions use a *split transaction* technique, which works in the following fashion. A request packet is sent out by a source PCIe device, which then waits for a response, called a *completion* packet. The completion following a request is initiated by the completer only when it has the data and/or status ready for delivery. Each packet has a unique identifier that enables completion packets to be directed to the correct originator. With the split transaction technique, the completion is separated in time from the request, in contrast to a typical bus operation in which both sides of a transaction must be available to seize and use the bus. Between the request and the completion, other PCIe traffic may use the link.

TL messages and some write transactions are *posted transactions*, meaning that no response is expected.

The TL packet format supports 32-bit memory addressing and extended 64-bit memory addressing. Packets also have attributes such as “no-snoop,” “relaxed-ordering,” and “priority,” which may be used to optimally route these packets through the I/O subsystem.

ADDRESS SPACES AND TRANSACTION TYPES

The TL supports four address spaces:

- **Memory:** The memory space includes system main memory. It also includes PCIe I/O devices. Certain ranges of memory addresses map into I/O devices.
- **I/O:** This address space is used for legacy PCI devices, with reserved memory address ranges used to address legacy I/O devices.
- **Configuration:** This address space enables the TL to read/write configuration registers associated with I/O devices.
- **Message:** This address space is for control signals related to interrupts, error handling, and power management.

Table 3.2 shows the transaction types provided by the TL. For memory, I/O, and configuration address spaces, there are read and write transactions. In the case of memory transactions, there is also a read lock request function. Locked operations occur as a result of device drivers requesting atomic access to registers on a PCIe device. A device driver, for example, can atomically read, modify, and then write to a device register. To accomplish this, the device driver causes the processor to execute an instruction or set of instructions. The root complex converts these processor instructions into a sequence of PCIe transactions, which perform individual read and write requests for the device driver. If these transactions must be executed atomically, the root complex locks the PCIe link while executing the transactions. This locking prevents transactions that are not part of the sequence from occurring. This sequence of transactions is called a locked operation. The particular set of processor instructions that can cause a locked operation to occur depends on the system chip set and processor architecture.

Table 3.2 PCIe TLP Transaction Types

Address Space	TLP Type	Purpose
Memory	Memory Read Request	Transfer data to or from a location in the system memory map.
	Memory Read Lock Request	

	Memory Write Request	
I/O	I/O Read Request	Transfer data to or from a location in the system memory map for legacy devices.
	I/O Write Request	
Configuration	Config Type 0 Read Request	Transfer data to or from a location in the configuration space of a PCIe device.
	Config Type 0 Write Request	
	Config Type 1 Read Request	
	Config Type 1 Write Request	
Message	Message Request	Provides in-band messaging and event reporting.
	Message Request with Data	
Memory, I/O, Configuration	Completion	Returned for certain requests.
	Completion with Data	
	Completion Locked	
	Completion Locked with Data	

To maintain compatibility with PCI, PCIe supports both Type 0 and Type 1 configuration cycles. A Type 1 cycle propagates downstream until it reaches the bridge interface hosting the bus (link) that the target device resides on. The configuration transaction is converted on the destination link from Type 1 to Type 0 by the bridge.

Finally, completion messages are used with split transactions for memory, I/O, and configuration transactions.

TLP PACKET ASSEMBLY

PCIe transactions are conveyed using transaction layer packets, which are illustrated in **Figure 3.25a**.

A TLP originates in the transaction layer of the sending device and terminates at the transaction layer of the receiving device.

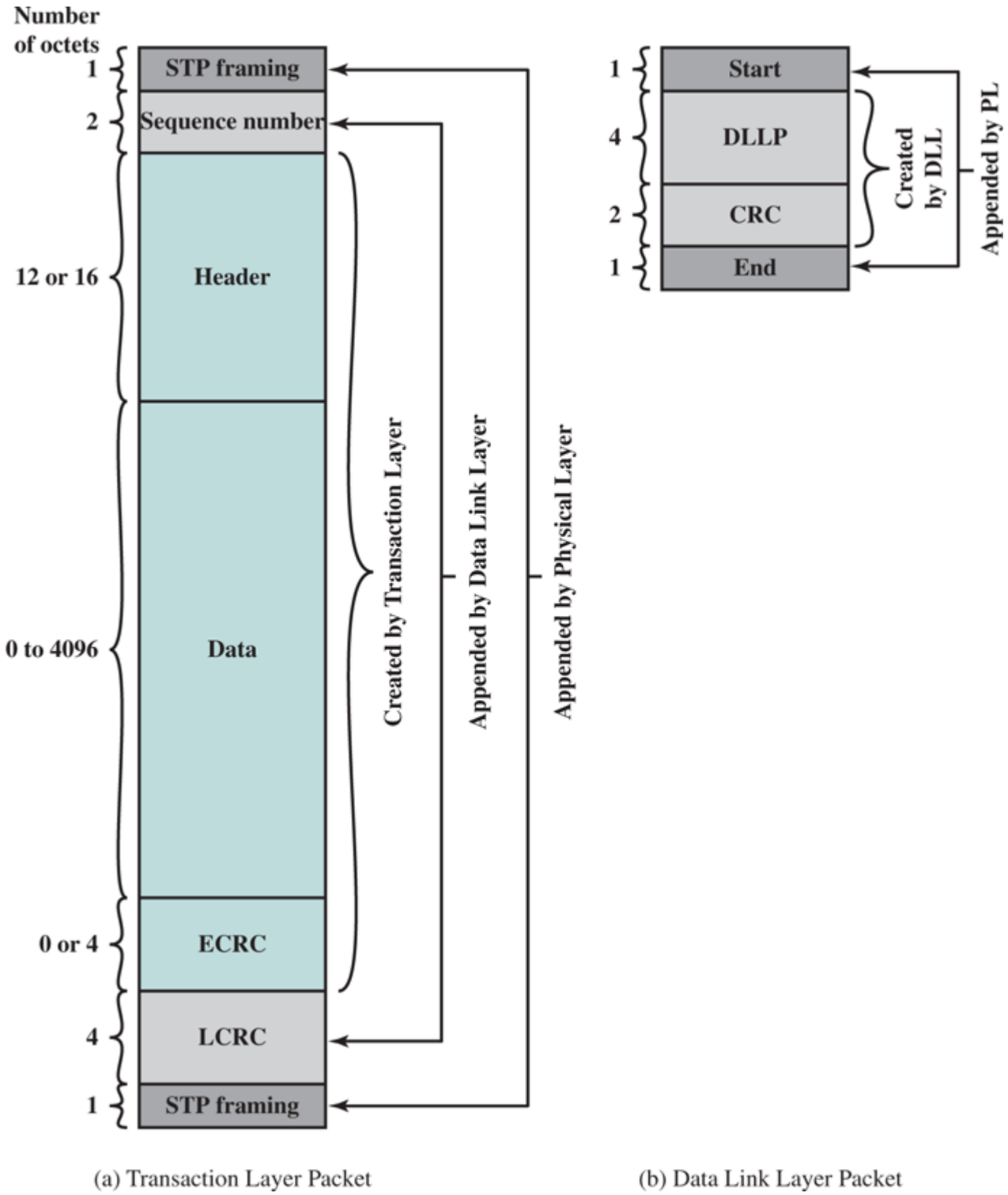


Figure 3.25 PCIe Protocol Data Unit Format

Upper layer software sends to the TL the information needed for the TL to create the core of the TLP, which consists of the following fields:

- **Header:** The header describes the type of packet and includes information needed by the receiver to process the packet, including any needed routing information. The internal header format is

discussed subsequently.
- **Data:** A data field of up to 4096 bytes may be included in the TLP. Some TLPs do not contain a data field.
- **ECRC:** An optional end-to-end CRC field enables the destination TL layer to check for errors in the header and data portions of the TLP.

PCIe Data Link Layer

The purpose of the PCIe data link layer is to ensure reliable delivery of packets across the PCIe link. The DLL participates in the formation of TLPs and also transmits DLLPs.

DATA LINK LAYER PACKETS

Data link layer packets originate at the data link layer of a transmitting device and terminate at the DLL of the device on the other end of the link. **Figure 3.25b** shows the format of a DLLP. There are three important groups of DLLPs used in managing a link: flow control packets, power management packets, and TLP ACK and NAK packets. Power management packets are used in managing power platform budgeting. Flow control packets regulate the rate at which TLPs and DLLPs can be transmitted across a link. The ACK and NAK packets are used in TLP processing, discussed in the following paragraphs.

TRANSACTION LAYER PACKET PROCESSING

The DLL adds two fields to the core of the TLP created by the TL (**Figure 3.25a**): a 16-bit sequence number and a 32-bit link-layer CRC (LCRC). Whereas the core fields created at the TL are only used at the destination TL, the two fields added by the DLL are processed at each intermediate node on the way from source to destination.

When a TLP arrives at a device, the DLL strips off the sequence number and LCRC fields and checks the LCRC. There are two possibilities:

1. If no errors are detected, the core portion of the TLP is handed up to the local transaction layer. If this receiving device is the intended destination, then the TL processes the TLP. Otherwise, the TL determines a route for the TLP and passes it back down to the DLL for transmission over the next link on the way to the destination.
2. If an error is detected, the DLL schedules an NAK DLL packet to return back to the remote transmitter. The TLP is eliminated.

When the DLL transmits a TLP, it retains a copy of the TLP. If it receives a NAK for the TLP with this sequence number, it retransmits the TLP. When it receives an ACK, it discards the buffered TLP.

3.7 Key Terms, Review Questions, and Problems

Key Terms

address bus

address lines

arbitration

balanced transmission

bus

control lines

data bus

data lines

differential signaling

disabled interrupt

distributed arbitration

error control function

execute cycle

fetch cycle

flit

flow control function

instruction cycle

interrupt

interrupt handler

interrupt service routine (ISR)

lane

memory address register (MAR)

memory buffer register (MBR)

multilane distribution

packets

PCI Express (PCIe)

peripheral component interconnect (PCI)

phit

QuickPath Interconnect (QPI)

root complex

system bus

Review Questions

3.1 What general categories of functions are specified by computer instructions?
3.2 List and briefly define the possible states that define an instruction execution.
3.3 List and briefly define two approaches to dealing with multiple interrupts.
3.4 What types of transfers must a computer's interconnection structure (e.g., bus) support?
3.5 List and briefly define the QPI protocol layers.
3.6 List and briefly define the PCIe protocol layers.

Problems

3.1 The hypothetical machine of **Figure 3.4** also has two I/O instructions:

0011 = Load AC from I/O
0111 = Store AC to I/O

In these cases, the 12-bit address identifies a particular I/O device. Show the program execution (using the format of **Figure 3.5**) for the following program:

1. Load AC from device 5.
2. Add contents of memory location 940.
3. Store AC to device 6.

Assume that the next value retrieved from device 5 is 3 and that location 940 contains a value of 2.
3.2 The program execution of **Figure 3.5** is described in the text using six steps. Expand this description to show the use of the MAR and MBR.
3.3 Consider a hypothetical 32-bit microprocessor having 32-bit instructions composed of two fields: the first byte contains the opcode and the remainder the immediate operand or an operand address.

a. What is the maximum directly addressable memory capacity (in bytes)?
b. Discuss the impact on the system speed if the microprocessor bus has:
    1. 32-bit local address bus and a 16-bit local data bus, or
    2. 16-bit local address bus and a 16-bit local data bus.
c. How many bits are needed for the program counter and the instruction register?

3.4 Consider a hypothetical microprocessor generating a 16-bit address (for example, assume that the program counter and the address registers are 16 bits wide) and having a 16-bit data bus.

a. What is the maximum memory address space that the processor can access directly if it is connected to a "16-bit memory"?
b. What is the maximum memory address space that the processor can access directly if it is connected to an "8-bit memory"?
c. What architectural features will allow this microprocessor to access a separate "I/O space"?
d. If an input and an output instruction can specify an 8-bit I/O port number, how many 8-bit I/O ports can the microprocessor support? How many 16-bit I/O ports? Explain.

3.5 Consider a 32-bit microprocessor, with a 16-bit external data bus, driven by an 8-MHz input clock. Assume that this microprocessor has a bus cycle whose minimum duration equals four input clock cycles. What is the maximum data transfer rate across the bus that this microprocessor can sustain, in bytes/sec? To increase its performance, would it be better to make its external data bus 32 bits or to double the external clock frequency supplied to the microprocessor? State any other assumptions you make, and explain. *Hint:* Determine the number of bytes that can be transferred per bus cycle.

3.6 Consider a computer system that contains an I/O module controlling a simple keyboard/printer teletype. The following registers are contained in the processor and connected directly to the system bus:

INPR: Input Register, 8 bits

OUTR: Output Register, 8 bits

FGI: Input Flag, 1 bit

FGO: Output Flag, 1 bit

IEN: Interrupt Enable, 1 bit

Keystroke input from the teletype and printer output to the teletype are controlled by the I/O module. The teletype is able to encode an alphanumeric symbol to an 8-bit word and decode an 8-bit word into an alphanumeric symbol.

a. Describe how the processor, using the first four registers listed in this problem, can achieve I/O with the teletype.
b. Describe how the function can be performed more efficiently by also employing IEN.

3.7 Consider two microprocessors having 8- and 16-bit-wide external data buses, respectively. The two processors are identical otherwise and their bus cycles take just as long.

a. Suppose all instructions and operands are two bytes long. By what factor do the maximum data transfer rates differ?
b. Repeat assuming that half of the operands and instructions are one byte long.

3.8 **Figure 3.26** indicates a distributed arbitration scheme that can be used with an obsolete bus scheme known as Multibus I. Agents are daisy-chained physically in priority order. The left-most agent in the diagram receives a constant *bus priority in* (BPRN) signal indicating that no higher-priority agent desires the bus. If the agent does not require the bus, it asserts its *bus priority out* (BPRO) line. At the beginning of a clock cycle, any agent can request control of the bus by lowering its BPRO line. This lowers the BPRN line of the next agent in the chain, which is in turn required to lower its BPRO line. Thus, the signal is propagated the length of the chain. At the end of this chain reaction, there should be only one agent whose BPRN is asserted and whose BPRO is not. This agent has priority. If, at the beginning of a bus cycle, the bus is not busy (BUSY inactive), the agent that has priority may seize control of the bus by asserting the BUSY line.

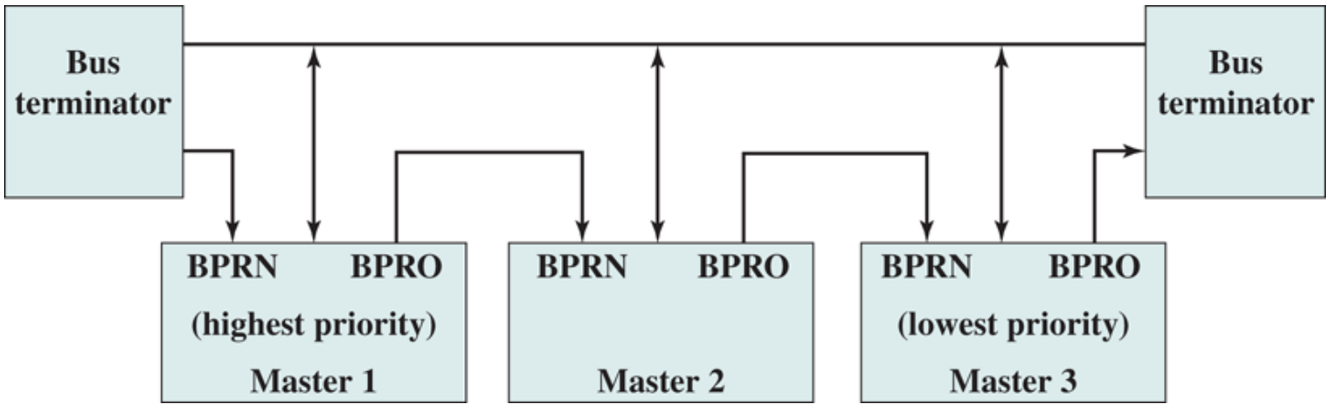


Figure 3.26 Multibus I Distributed Arbitration

It takes a certain amount of time for the BPR signal to propagate from the highest-priority agent to the lowest. Must this time be less than the clock cycle? Explain.

3.9 The VAX SBI bus uses a distributed, synchronous arbitration scheme. Each SBI device (i.e., processor, memory, I/O module) has a unique priority and is assigned a unique transfer request (TR) line. The SBI has 16 such lines (TR0, TR1, . . ., TR15), with TR0 having the highest priority. When a device wants to use the bus, it places a reservation for a future time slot by asserting its TR line during the current time slot. At the end of the current time slot, each device with a pending reservation examines the TR lines; the highest-priority device with a reservation uses the next time slot.

A maximum of 17 devices can be attached to the bus. The device with priority 16 has no TR line. Why not?

3.10 On the VAX SBI, the lowest-priority device usually has the lowest average wait time. For this reason, the processor is usually given the lowest priority on the SBI. Why does the priority 16 device usually have the lowest average wait time? Under what circumstances would this not be true?

3.11 For a synchronous read operation (Figure C.3 in **Appendix C**), the memory module must place the data on the bus sufficiently ahead of the falling edge of the Read signal to allow for signal settling. Assume a microprocessor bus is clocked at 10 MHz and that the Read signal begins to fall in the middle of the second half of T_3.

a. Determine the length of the memory read instruction cycle.
b. When, at the latest, should memory data be placed on the bus? Allow 20 ns for the settling of data lines.

3.12 Consider a microprocessor that has a memory read timing (Figure C.3 in **Appendix C**). After some analysis, a designer determines that the memory falls short of providing read data on time by about 180 ns.

a. How many wait states (clock cycles) need to be inserted for proper system operation if the bus clocking rate is 8 MHz?
b. To enforce the wait states, a Ready status line is employed. Once the processor has issued a Read command, it must wait until the Ready line is asserted before attempting to read data. At what time interval must we keep the Ready line low in order to force the processor to insert the required number of wait states?

3.13 A microprocessor has a memory write timing Figure A.3 in **Appendix A** . Its manufacturer specifies that the width of the Write signal can be determined by $T - 50$, where *T* is the clock period in ns.

a. What width should we expect for the Write signal if bus clocking rate is 5 MHz?
b. The data sheet for the microprocessor specifies that the data remain valid for 20 ns after the falling edge of the Write signal. What is the total duration of valid data presentation to memory?
c. How many wait states should we insert if memory requires valid data presentation for at least 190 ns?

3.14 A microprocessor has an increment memory direct instruction, which adds 1 to the value in a memory location. The instruction has five stages: fetch opcode (four bus clock cycles), fetch operand address (three cycles), fetch operand (three cycles), add 1 to operand (three cycles), and store operand (three cycles).

a. By what amount (in percent) will the duration of the instruction increase if we have to insert two bus wait states in each memory read and memory write operation?
b. Repeat assuming that the increment operation takes 13 cycles instead of 3 cycles.

3.15 The Intel 8088 microprocessor has a read bus timing similar to that of Figure C.3, but requires four processor clock cycles. The valid data is on the bus for an amount of time that extends into the fourth processor clock cycle. Assume a processor clock rate of 8 MHz.

a. What is the maximum data transfer rate?
b. Repeat, but assume the need to insert one wait state per byte transferred.

3.16 The Intel 8086 is a 16-bit processor similar in many ways to the 8-bit 8088. The 8086 uses a 16-bit bus that can transfer 2 bytes at a time, provided that the lower-order byte has an even address. However, the 8086 allows both even- and odd-aligned word operands. If an odd-aligned word is referenced, two memory cycles, each consisting of four bus cycles, are required to transfer the word. Consider an instruction on the 8086 that involves two 16-bit operands. How long does it take to fetch the operands? Give the range of possible answers. Assume a clocking rate of 4 MHz and no wait states.

3.17 Consider a 32-bit microprocessor whose bus cycle is the same duration as that of a 16-bit microprocessor. Assume that, on average, 20% of the operands and instructions are 32 bits long, 40% are 16 bits long, and 40% are only 8 bits long. Calculate the improvement achieved when fetching instructions and operands with the 32-bit microprocessor.

3.18 The microprocessor of Problem 3.14 initiates the fetch operand stage of the increment memory direct instruction at the same time that a keyboard activates an interrupt request line. After how long does the processor enter the interrupt processing cycle? Assume a bus clocking rate of 10 MHz.

Chapter 4 The Memory Hierarchy: Locality and Performance

Learning Objectives

After studying this chapter, you should be able to:

- Present an overview of the principle of locality.
- Describe key characteristics of a memory system.
- Discuss how locality influences the development of a memory hierarchy.
- Understand the performance implications of multiple levels of memory.

Although seemingly simple in concept, computer memory exhibits perhaps the widest range of type, technology, organization, performance, and cost of any feature of a computer system. No single technology is optimal in satisfying the memory requirements for a computer system. As a consequence, the typical computer system is equipped with a hierarchy of memory subsystems, some internal to the system (directly accessible by the processor) and some external (accessible by the processor via an I/O module).

This chapter focuses on the performance factors that drive the development of a computer memory system with multiple levels using different technologies. ***Section 4.1*** *introduces the key concept of locality of reference, which has a profound influence on both the organization of memory and on operating system memory management software. Following a brief discussion of key characteristics of memory systems, the chapter turns to a presentation of the concept of a memory hierarchy and indicates the typical components in contemporary systems. Finally,* ***Section 4.4*** *develops a simple but illuminating model of memory access performance.*

The next three chapters look at specific aspects of memory systems, using the insights provided in this chapter. Chapter 5 examines an essential element of all

modern computer systems: cache memory. ***Chapter 6*** *then looks at the technology options for internal memory, including cache and main memory.* ***Chapter 7*** *is devoted to external memory.*

4.1 Principle Of Locality

One of the most important concepts related to computer systems is **principle of locality** [DENN05], also referred to as the **locality of reference.** The principle reflects the observation that during the course of execution of a program, memory references by the processor, for both instructions and data, tend to cluster. Programs typically contain a number of iterative loops and subroutines. Once a loop or subroutine is entered, there are repeated references to a small set of instructions. Similarly, operations on tables and arrays involve access to a clustered set of data words. Over a long period of time, the clusters in use change, but over a short period of time, the processor is primarily working with fixed clusters of memory references.

We can put these observations more specifically. As we discuss in **Section 4.3**, for different types of memory, memory is accessed and retrieved in units of different sizes, ranging from individual words to large blocks of cache memory to much larger segments of disk memory. Denning observed that locality is based on three assertions [DENN72]:

1. During any interval of time, a program references memory locations non-uniformly. That is, some units of memory are more likely to be accessed than others.
2. As a function of time, the probability that a given unit of memory is referenced tends to change slowly. Put another way, the probability distribution of memory references across the entire memory space tends to change slowly over time.
3. The correlation between immediate past and immediate future memory reference patterns is high, and tapers off as the time interval increases.

Intuitively, the principle of locality makes sense. Consider the following line of reasoning:

1. Except for branch and call instructions, which constitute only a small fraction of all program instructions, program execution is sequential. Hence, in most cases, the next instruction to be fetched immediately follows the last instruction fetched.
2. It is rare to have a long uninterrupted sequence of procedure calls followed by the corresponding sequence of returns. Rather, a program remains confined to a rather narrow window of procedure-invocation depth. Thus, over a short period of time, references to instructions tend to be localized to a few procedures.
3. Most iterative constructs consist of a relatively small number of instructions repeated many times. For the duration of the iteration, computation is therefore confined to a small contiguous portion of a program.
4. In many programs, much of the computation involves processing data structures, such as arrays or sequences of records. In many cases, successive references to these data structures will be to closely located data items.

Numerous studies, stretching back to the early 1970s, confirm these observations. [FEIT15] provides a summary of many of these studies.

A distinction is made in the literature between two forms of locality:

1. **Temporal locality:** Refers to the tendency of a program to reference in the near future those units of memory referenced in the recent past. For example, when an iteration loop is executed, the processor executes the same set of instructions repeatedly. Constants, temporary variables, and working stacks are also constructs that lead to this principle.
2. **Spatial locality:** Refers to the tendency of a program to reference units of memory whose addresses are near one another. That is, if a unit of memory *x* is referenced at time *t*, it is likely

that units in the range $x - k$ through $x + k$ will be referenced in the near future, for a relatively small value of *k*. This reflects the tendency of a processor to access instructions sequentially. Spatial location also reflects the tendency of a program to access data locations sequentially, such as when processing a table of data.

A crude analogy may help illuminate the distinction between these two concepts (**Figure 4.1**). Suppose that Bob is working in an office and spends much of his time dealing with documents in file folders. Thousand of folders are stored in file cabinets in the next room, and for convenience Bob has a file organizer on his desk that can hold a few dozen files. When Bob is working on a file and temporarily is finished, it may be likely that he will need to read or write one of the documents in that file in the near future, so he keeps it in his desk organizer. This is an example of exploiting temporal locality. Bob also observes that when he retrieves a folder from the filing cabinets, it is likely that in the near future he will need access to some of the nearby folders as well, so he retrieves the folder he needs plus a few folders on either side at the same time. This is an example of exploiting spatial locality. Of course, Bob's desktop file organizer soon fills up, so that when he goes to retrieve a folder from the file cabinets, he needs to return folders from his desk. Bob needs some policy for replacing folders. If he focuses on temporal locality, Bob could choose to replace only one folder at a time, on the reasoning that he might need any of the folders currently on his desk in the near future. So Bob could replace perhaps the folder that had been on the desk the longest or the one that had be the least recently used. If Bob focuses on spatial locality, when he needs a folder not on his desk, he could return and refile all the folders on his desk and retrieve a batch of contiguous folders that includes the one he needs plus other nearby folders sufficient to fill up his desktop organizer. It is likely that neither policy is optimal. In the first case, he might have to make frequent trips to the next room to get one folder he doesn't have but which is near one he does have. In the second case, he might have to make frequent trips to the next room to get a folder that he had just recently put away. So perhaps a policy of returning and retrieving in batches equal to 10% or 20% of his desktop capacity would be closer to optimal.

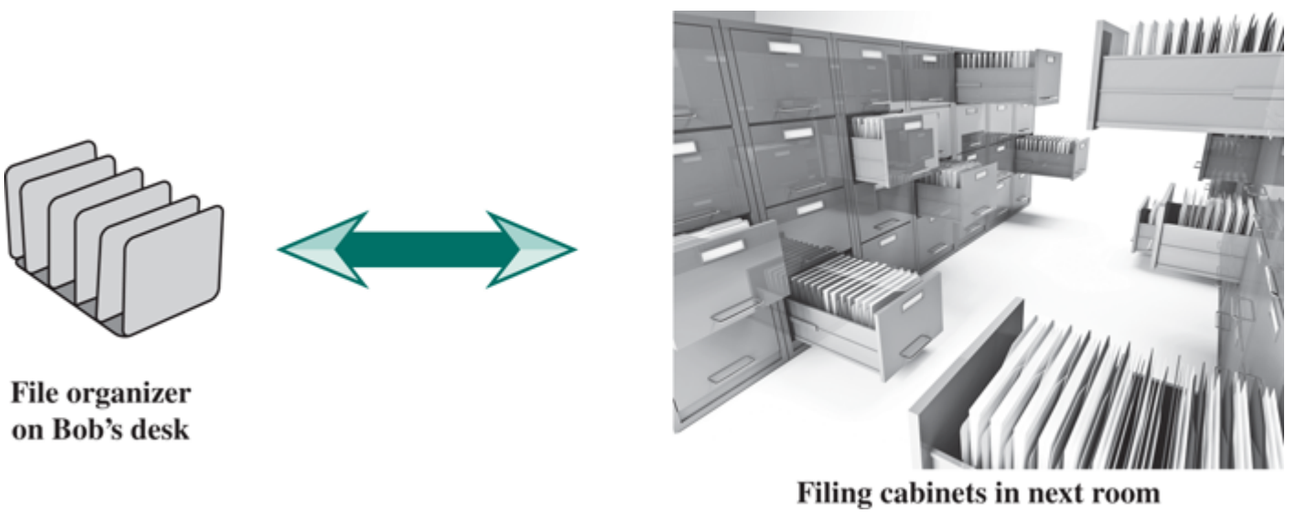


Figure 4.1 Moving File Folders Between Smaller, Faster-Access Storage and Larger, Slower-Access Storage

Gualtiero boffi/Shutterstock

For cache memory, temporal locality is traditionally exploited by keeping recently used instruction and data values in cache memory and by exploiting a cache hierarchy. Spatial locality is generally exploited by using larger cache blocks and by incorporating prefetching mechanisms (fetching items of anticipated use) into the cache control logic. Over the years, there has been considerable research on

refining these techniques to achieve greater performance, but the basic strategies remain the same.

Figure 4.2 provides a rough depiction of the behavior of programs that exhibit temporal locality. For a unit of memory accessed at time *t*, the figure shows the distribution of probability of the time of the next access to the same memory unit. Similarly, **Figure 4.3** provides a rough depiction of the behavior of programs that exhibit spatial locality. For spatial locality, the probability distribution curve is symmetrical around the location of the most recent memory access address.

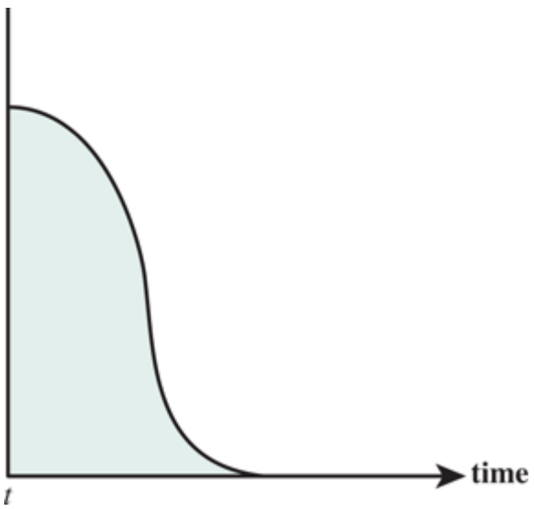


Figure 4.2 Idealized Temporal Locality Behavior: Probability Distribution for Time of Next Memory Access to Memory Unit Accessed at Time t

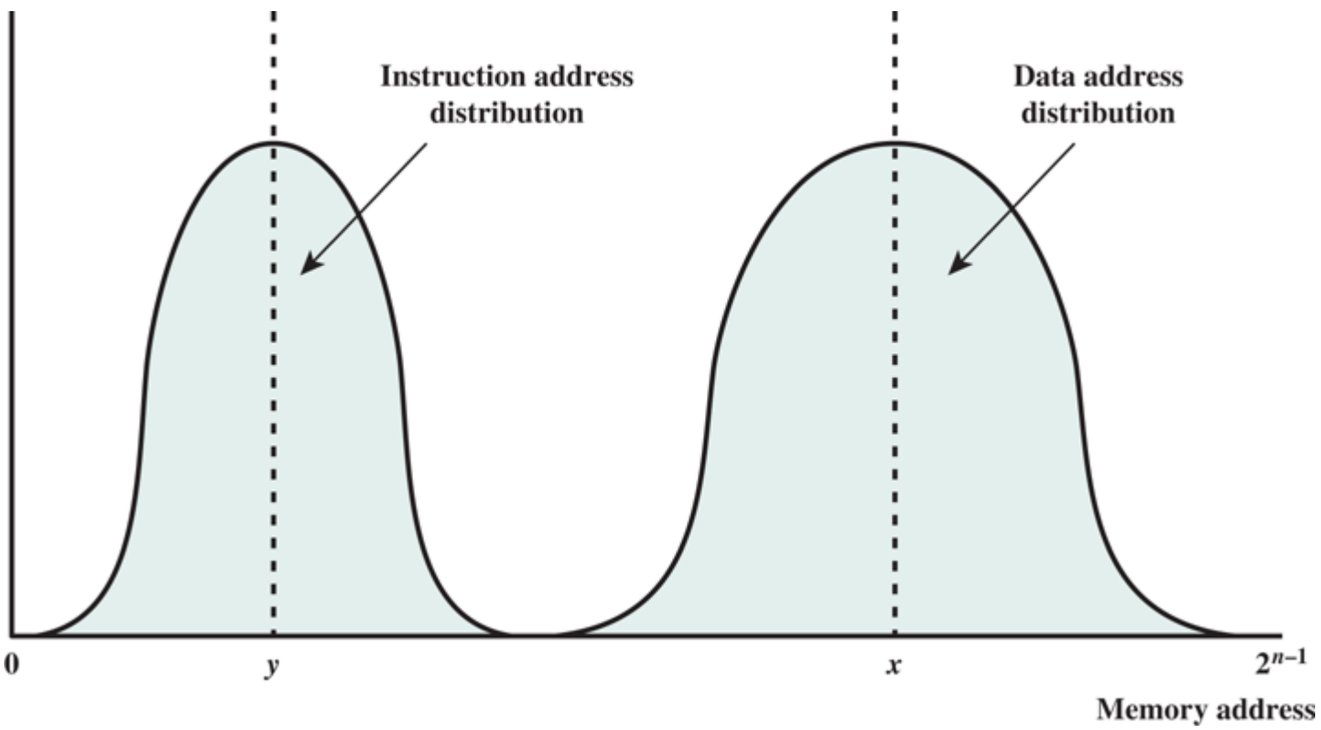


Figure 4.3 Idealized Spatial Locality Behavior: Probability Distribution for Next Memory Access (most recent data memory access at location x; most recent instruction fetch at location y)

Many programs exhibit both temporal and spatial locality for both instruction and data access. It has been found that data access patterns generally show a greater variance than instruction access

patterns [AHO07]. **Figure 4.3** suggests this distinction between the distribution of data location accesses (read or write) and instruction fetch addresses. Typically, each instruction execution involves fetching the instruction from memory and, during execution, accessing one or more data operands from one or more regions of memory. Thus, there is a dual locality of **data spatial locality** and **instruction spatial locality**. And, of course, temporal locality exhibits this same dual behavior: **data temporal locality** and **instruction temporal locality**. That is, when an instruction is fetched from a unit of memory, it is likely that in the near future, additional instructions will be fetched from that same memory unit; and when a data location is accessed, it is likely that in the near future, additional instructions will be fetched from that same memory unit.

An example of data locality is illustrated in **Figure 4.4** [BAEN97]. This shows the results of a study of Web-based document access patterns, where the documents are distributed among a number of servers. In this case, the unit of access is a single document and temporal locality is measured. The access scheme makes use of a document cache at the browser that can temporarily retain a small number of documents to facilitate reuse. The study covered 220,000 documents distributed over 11,000 servers. As shown in **Figure 4.4**, only a very small subset of pages incorporates a high number of references while most documents are accessed relatively infrequently.

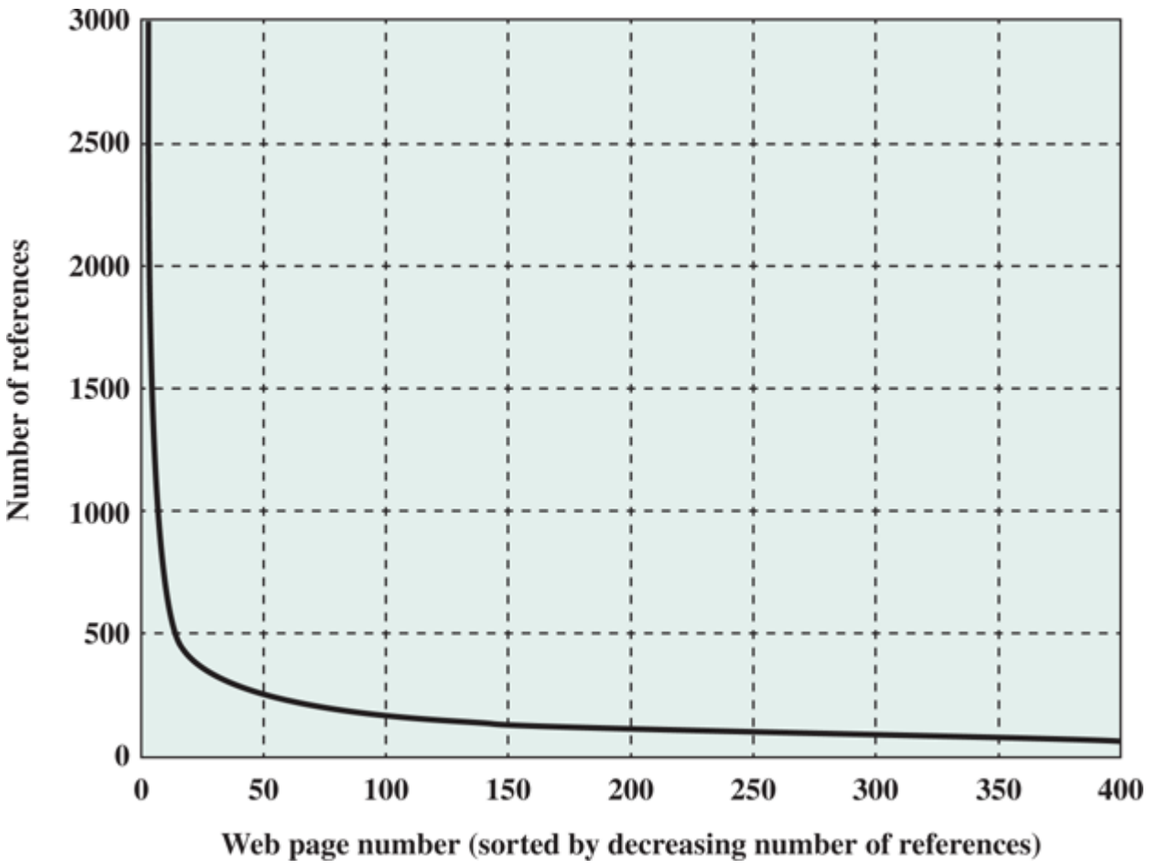


Figure 4.4 Data Locality of Reference for Web-Based Document Access Application

Figure 4.5 shows an example of instruction locality based on executing the integer benchmark programs in the SPEC CPU2006 benchmark suite; similar results were obtained for the floating-point programs. The following terms are used in the plot:

1. **Static instruction:** An instruction that exists in the code to be executed.
2. **Dynamic instruction:** Instructions that appear in the execution trace of a program.

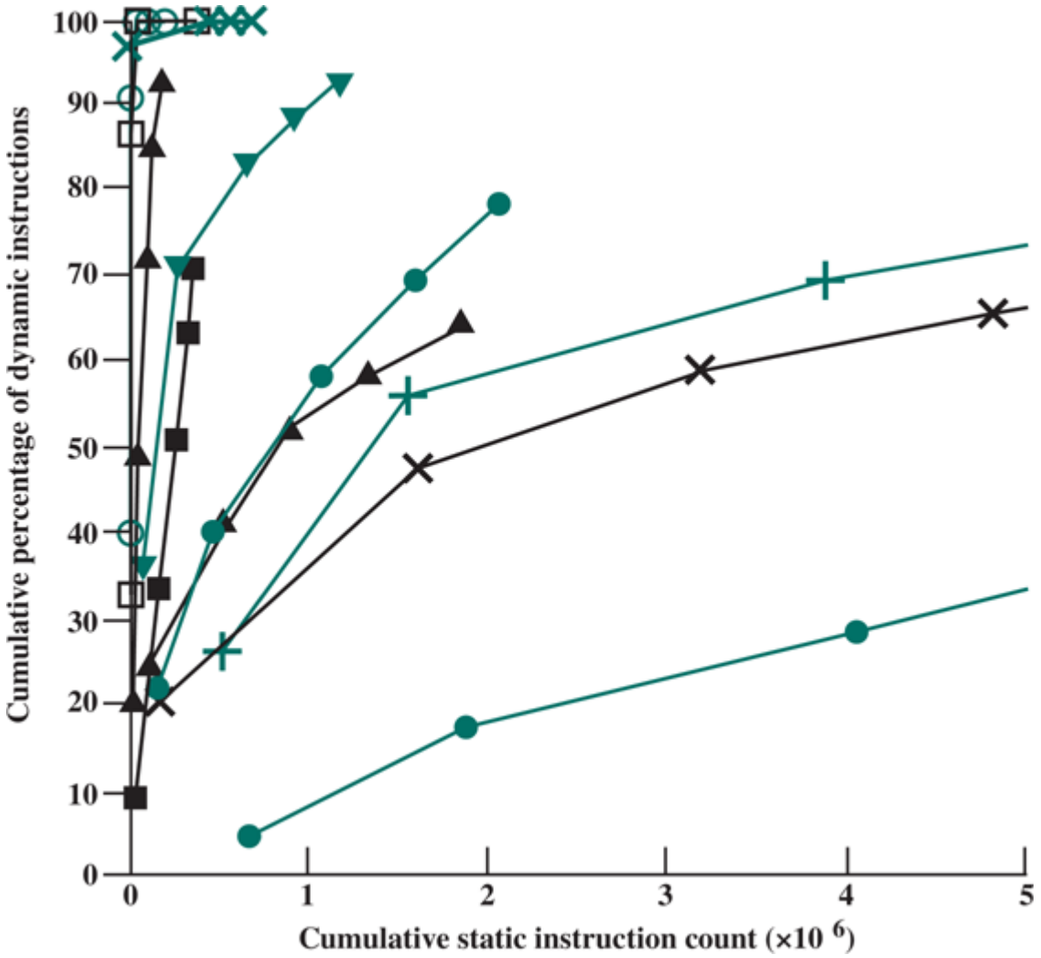


Figure 4.5 Instruction Locality Based on Code Reuse in Eleven Benchmark Programs in SPEC CPU2006

Thus, each static instruction is represented by zero or more instances of dynamic instructions when the program is executed. Each line in the graph represents a separate benchmark program. In the figure, the cumulative percentage of dynamic instructions executed by a program is shown on the *y*-axis and the cumulative count of static instructions is shown on the *x*-axis. The first point in the line plot for each benchmark represents the most frequently called subroutine, with the *x* coordinate showing the number of static instructions in the routine and *y* coordinate showing the percentage of dynamic instructions that it represents. The second, third, fourth, and fifth points respectively represent the top 5, 10, 15, and 20 most frequently called subroutines. Many programs initially show a steep upward climb as the static instruction count increases, which suggests very good instruction locality.

4.2 Characteristics Of Memory Systems

The complex subject of computer memory is made more manageable if we classify memory systems according to their key characteristics. The most important of these are listed in **Table 4.1**.

Table 4.1 Key Characteristics of Computer Memory Systems

Location	**Performance**
Internal (e.g., processor registers, cache, main memory)	Access time
External (e.g., optical disks, magnetic disks, tapes)	Cycle time
Capacity	Transfer rate
Number of words	**Physical Type**
Number of bytes	Semiconductor
Unit of Transfer	Magnetic
Word	Optical
Block	Magneto-optical
Access Method	**Physical Characteristics**
Sequential	Volatile/nonvolatile
Direct	Erasable/nonerasable
Random	**Organization**
Associative	Memory modules

The term **location** in **Table 4.1** refers to whether memory is internal or external to the computer. Internal memory is often equated with main memory, but there are other forms of internal memory. The processor requires its own local memory, in the form of registers (e.g., see **Figure 2.3**). Further, as we will see, the control unit portion of the processor may also require its own internal memory. We will defer discussion of these latter two types of internal memory to later chapters. Cache is another form of internal memory. External memory consists of peripheral storage devices, such as disk and tape, that are accessible to the processor via I/O controllers.

An obvious characteristic of memory is its **capacity**. For internal memory, this is typically expressed in terms of bytes $(1\text{byte} = 8\text{bits})$ or words. Common word lengths are 8, 16, and 32 bits. External memory capacity is typically expressed in terms of bytes.

A related concept is the **unit of transfer**. For internal memory, the unit of transfer is equal to the number of electrical lines into and out of the memory module. This may be equal to the word length,

but is often larger, such as 64, 128, or 256 bits. To clarify this point, consider three related concepts for internal memory:

- **Word:** The "natural" unit of organization of memory. The size of a word is typically equal to the number of bits used to represent an integer and to the instruction length. Unfortunately, there are many exceptions. For example, the CRAY C90 (an older model CRAY supercomputer) has a 64-bit word length but uses a 46-bit integer representation. The Intel x86 architecture has a wide variety of instruction lengths, expressed as multiples of bytes, and a word size of 32 bits.
- **Addressable units:** In some systems, the addressable unit is the word. However, many systems allow addressing at the byte level. In any case, the relationship between the length in bits *A* of an address and the number *N* of addressable units is $2^A = N$.
- **Unit of transfer:** For main memory, this is the number of bits read out of or written into memory at a time. The unit of transfer need not equal a word or an addressable unit. For external memory, data are often transferred in much larger units than a word, and these are referred to as blocks.

Another distinction among memory types is the **method of accessing** units of data. These include the following:

- **Sequential access:** Memory is organized into units of data, called records. Access must be made in a specific linear sequence. Stored addressing information is used to separate records and assist in the retrieval process. A shared read–write mechanism is used, and this must be moved from its current location to the desired location, passing and rejecting each intermediate record. Thus, the time to access an arbitrary record is highly variable. Tape units, discussed in **Chapter 7**, are sequential access.
- **Direct access:** As with sequential access, direct access involves a shared read–write mechanism. However, individual blocks or records have a unique address based on physical location. Access is accomplished by direct access to reach a general vicinity plus sequential searching, counting, or waiting to reach the final location. Again, access time is variable. Disk units, discussed in **Chapter 6**, are direct access.
- **Random access:** Each addressable location in memory has a unique, physically wired-in addressing mechanism. The time to access a given location is independent of the sequence of prior accesses and is constant. Thus, any location can be selected at random and directly addressed and accessed. Main memory and some cache systems are random access.
- **Associative:** This is a random access type of memory that enables one to make a comparison of desired bit locations within a word for a specified match, and to do this for all words simultaneously. Thus, a word is retrieved based on a portion of its contents rather than its address. As with ordinary random-access memory, each location has its own addressing mechanism, and retrieval time is constant independent of location or prior access patterns. Cache memories may employ associative access.

From a user's point of view, the two most important characteristics of memory are capacity and **performance**. Three performance parameters are used:

- **Access time (latency):** For random-access memory, this is the time it takes to perform a read or write operation, that is, the time from the instant that an address is presented to the memory to the instant that data have been stored or made available for use. For non-random-access memory, access time is the time it takes to position the read–write mechanism at the desired location.
- **Memory cycle time:** This concept is primarily applied to random-access memory and consists of the access time plus any additional time required before a second access can commence. This additional time may be required for transients to die out on signal lines or to regenerate data if they are read destructively. Note that memory cycle time is concerned with the system bus, not the processor.
- **Transfer rate:** This is the rate at which data can be transferred into or out of a memory unit. For random-access memory, it is equal to 1/(cycle time). For non-random-access memory, the

following relationship holds:

$$T_n = T_A + \frac{n}{R} \tag{4.1}$$

where

T_n = Average time to read or write n bits
T_A = Average access time
n = Number of bits
R = Transfer rate, in bits per second (bps)

A variety of **physical types** of memory have been employed. The most common today are semiconductor memory, magnetic surface memory, used for disk and tape, and optical and magneto-optical.

Several **physical characteristics** of data storage are important. In a volatile memory, information decays naturally or is lost when electrical power is switched off. In a nonvolatile memory, information once recorded remains without deterioration until deliberately changed; no electrical power is needed to retain information. Magnetic-surface memories are nonvolatile. Semiconductor memory (memory on integrated circuits) may be either volatile or nonvolatile. Nonerasable memory cannot be altered, except by destroying the storage unit. Semiconductor memory of this type is known as *read-only memory* (ROM). Of necessity, a practical nonerasable memory must also be nonvolatile.

For random-access memory, the **organization** is a key design issue. In this context, *organization* refers to the physical arrangement of bits to form words. The obvious arrangement is not always used, as is explained in **Chapter 6**.

4.3 The Memory Hierarchy

The design constraints on a computer's memory can be summed up by three questions: How much? How fast? How expensive?

The question of how much is somewhat open ended. If the capacity is there, applications will likely be developed to use it. The question of how fast is, in a sense, easier to answer. To achieve greatest performance, the memory must be able to keep up with the processor. That is, as the processor is executing instructions, we would not want it to have to pause waiting for instructions or operands. The final question must also be considered. For a practical system, the cost of memory must be reasonable in relationship to other components.

As might be expected, there is a trade-off among the three key characteristics of memory: capacity, access time, and cost. A variety of technologies are used to implement memory systems, and across this spectrum of technologies, the following relationships hold:

- Faster access time, greater cost per bit
- Greater capacity, smaller cost per bit
- Greater capacity, slower access time

The dilemma facing the designer is clear. The designer would like to use memory technologies that provide for large-capacity memory, both because the capacity is needed and because the cost per bit is low. However, to meet performance requirements, the designer needs to use expensive, relatively lower-capacity memories with short access times.

Cost and Performance Characteristics

The way out of this dilemma is not to rely on a single memory component or technology, but to employ a **memory hierarchy**. A typical hierarchy is illustrated in **Figure 4.6**. As one goes down the hierarchy, the following occur:

a. Decreasing cost per bit
b. Increasing capacity
c. Increasing access time
d. Decreasing frequency of access of the memory by the processor

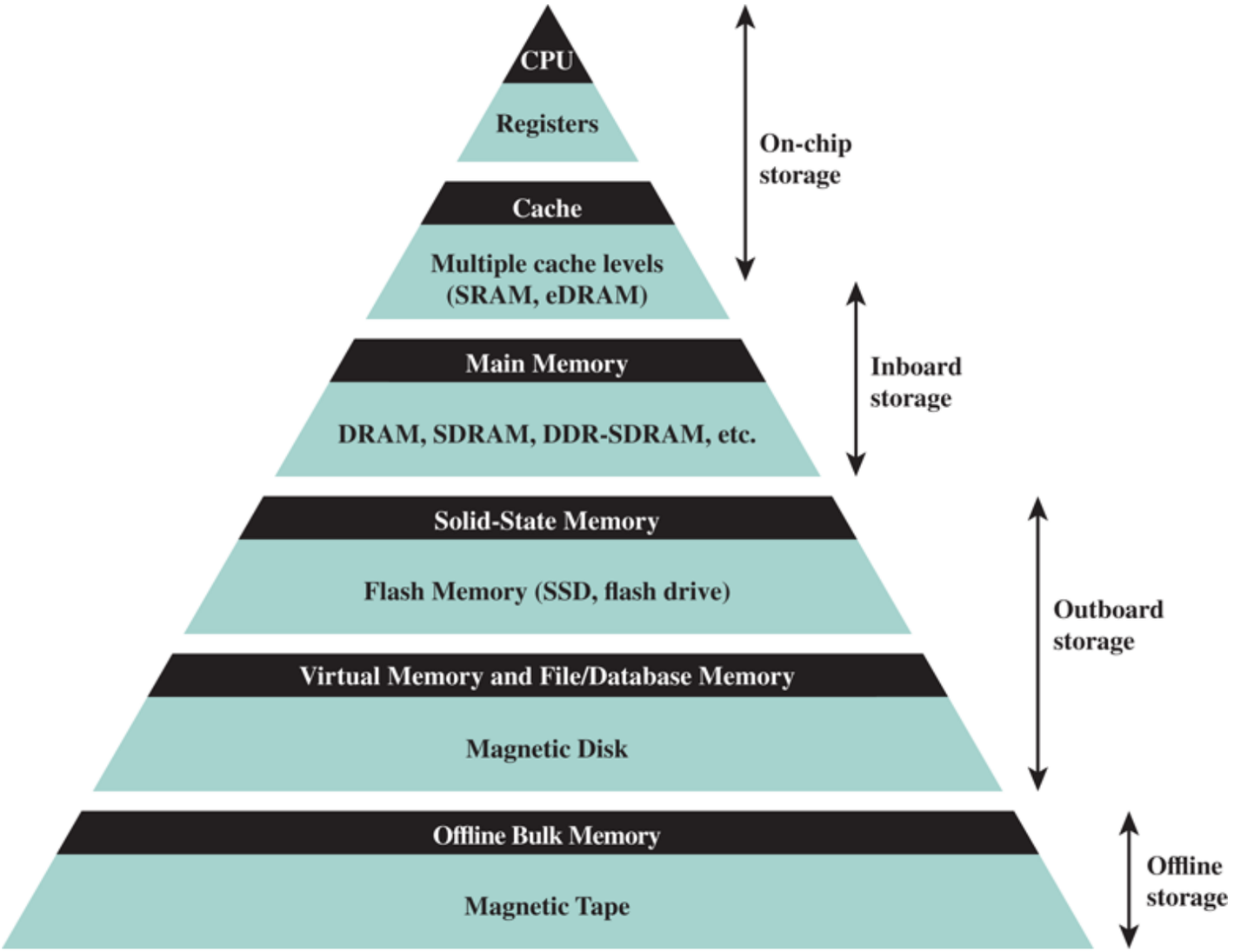


Figure 4.6 The Memory Hierarchy

Let us label the memory at level *i* of the memory hierarchy M_i, such that M_i is closer to the processor than M_{i+1}. If C_i, T_i, R_i, and S_i are respectively the cost per byte, average access time, average data transfer rate, and total memory size at level *i*, then the following relationships typically hold between levels *i* and $i+1$:

$$\begin{aligned} C_i &> C_{i+1} \\ T_i &< T_{i+1} \\ R_i &> R_{i+1} \\ S_i &< S_{i+1} \end{aligned}$$

Figure 4.7 (in a general way and not to scale) illustrates these relationships across the memory hierarchy.

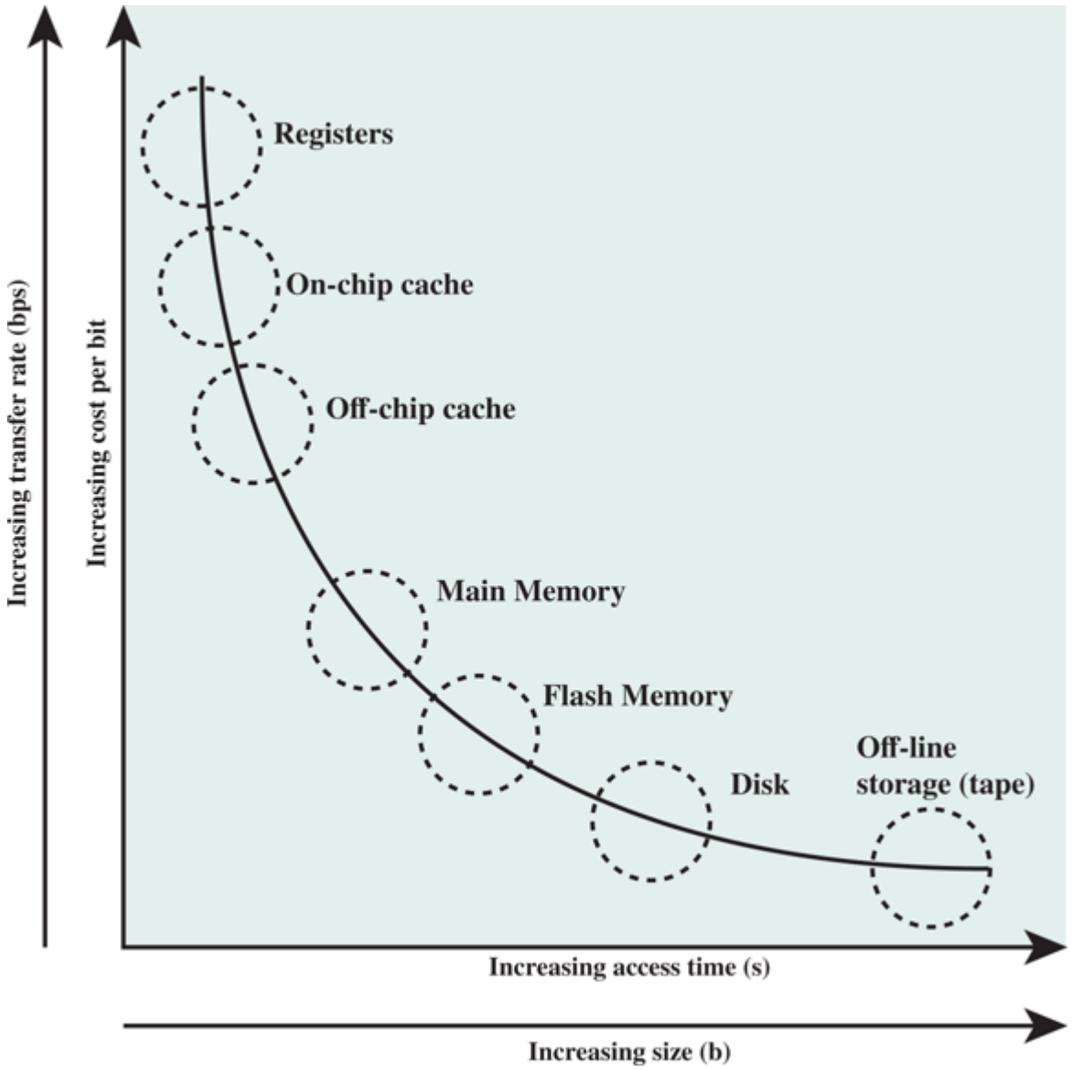


Figure 4.7 Relative Cost, Size, and Speed Characteristics Across the Memory Hierarchy

Thus, smaller, more expensive, faster memories are supplemented by larger, cheaper, slower memories. The key to the success of this organization is item (d): decreasing frequency of access, which can be achieved by exploiting the principle of locality, described in **Section 4.1**. We discuss techniques for exploiting locality in the treatment of cache, in **Chapter 5**, and virtual memory, in **Chapter 9**. A general discussion is provided at this point.

It is possible to organize data across the hierarchy such that the percentage of accesses to each successively lower level is substantially less than that of the level above. Consider the following example:

EXAMPLE 4.1

Suppose that the processor has access to two levels of memory. Level 1 contains X words and has an access time of $0.01\mu s$; level 2 contains $1000 \times X$ words and has an access time of $0.1\mu s$.

Assume that if a word to be accessed is in level 1, then the processor accesses it directly. If it is in level 2, then the word is first transferred to level 1 and then accessed by the processor. For simplicity, we ignore the time required for the processor to determine whether the word is in level 1 or level 2. **Figure 4.8** shows the general shape of the curve that covers this situation. The figure shows the average access time to a two-level memory as a function of the hit ratio H, where H is defined as the fraction of all memory accesses that are found in the faster memory (e.g., the

cache), T_1 is the access time to level 1, and T_2 is the access time to level 2.[1] As can be seen, for high percentages of level 1 access, the average total access time is much closer to that of level 1 than that of level 2.

[1] If the accessed word is found in the faster memory, that is defined as a **hit**. A **miss** occurs if the accessed word is not found in the faster memory.

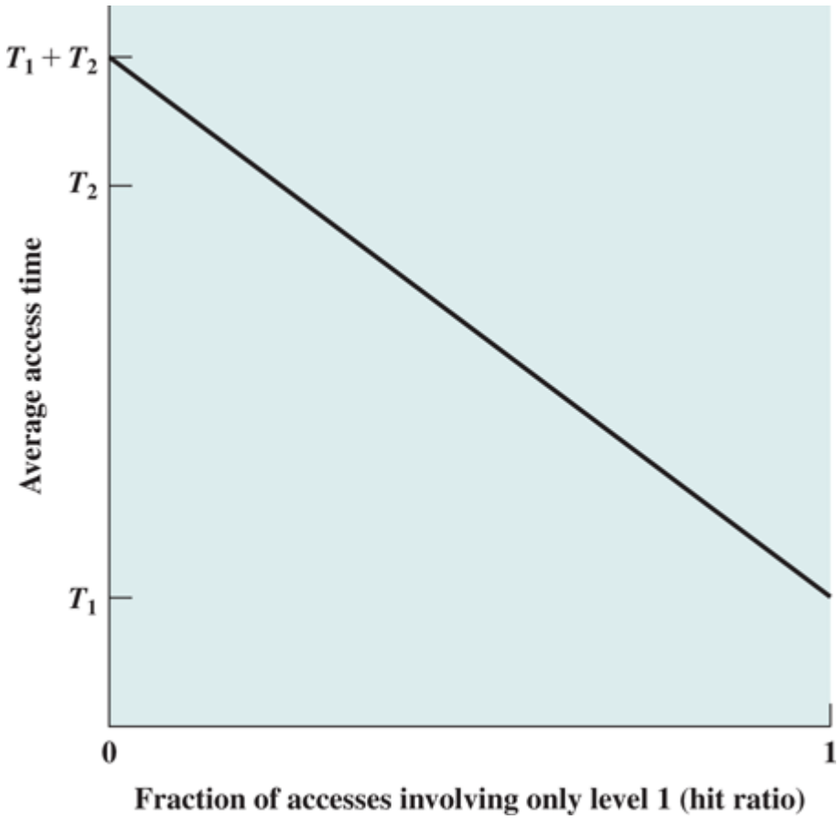


Figure 4.8 Performance of Accesses Involving only Level 1 (hit ratio)

In our example, suppose that 95% of the memory accesses are found in Level 1. Then the average time to access a word can be expressed as

$$(0.95)\,(0.01\mu s) + (0.05)(0.01\mu s + 0.1\mu s) = 0.0095 + 0.0055 = 0.015\mu s$$

The average access time is much closer to $0.01\mu s$ than to $0.1\mu s$, as desired.

Let level 2 memory contain all program instructions and data. Currently used clusters can be temporarily placed in level 1. From time to time, one of the clusters in level 1 will have to be swapped back to level 2 to make room for a new cluster coming in to level 1. On average, however, most references will be to instructions and data contained in level 1.

The use of two levels of memory to reduce average access time works in principle, but only if conditions (a) through (d) apply. By employing a variety of technologies, a spectrum of memory systems exists that satisfies conditions (a) through (c). Fortunately, condition (d) is also generally valid due to the principle of locality.

This principle can be applied across multiple levels of memory, as suggested by the hierarchy shown

in **Figure 4.6**. In practice, the dynamic movement of chunks of data between levels during program execution involves registers, one or more levels of cache, main memory, and virtual memory stored on disk. This is shown in **Figure 4.9**, with an indication of the size of the chunks of data exchanged between levels.

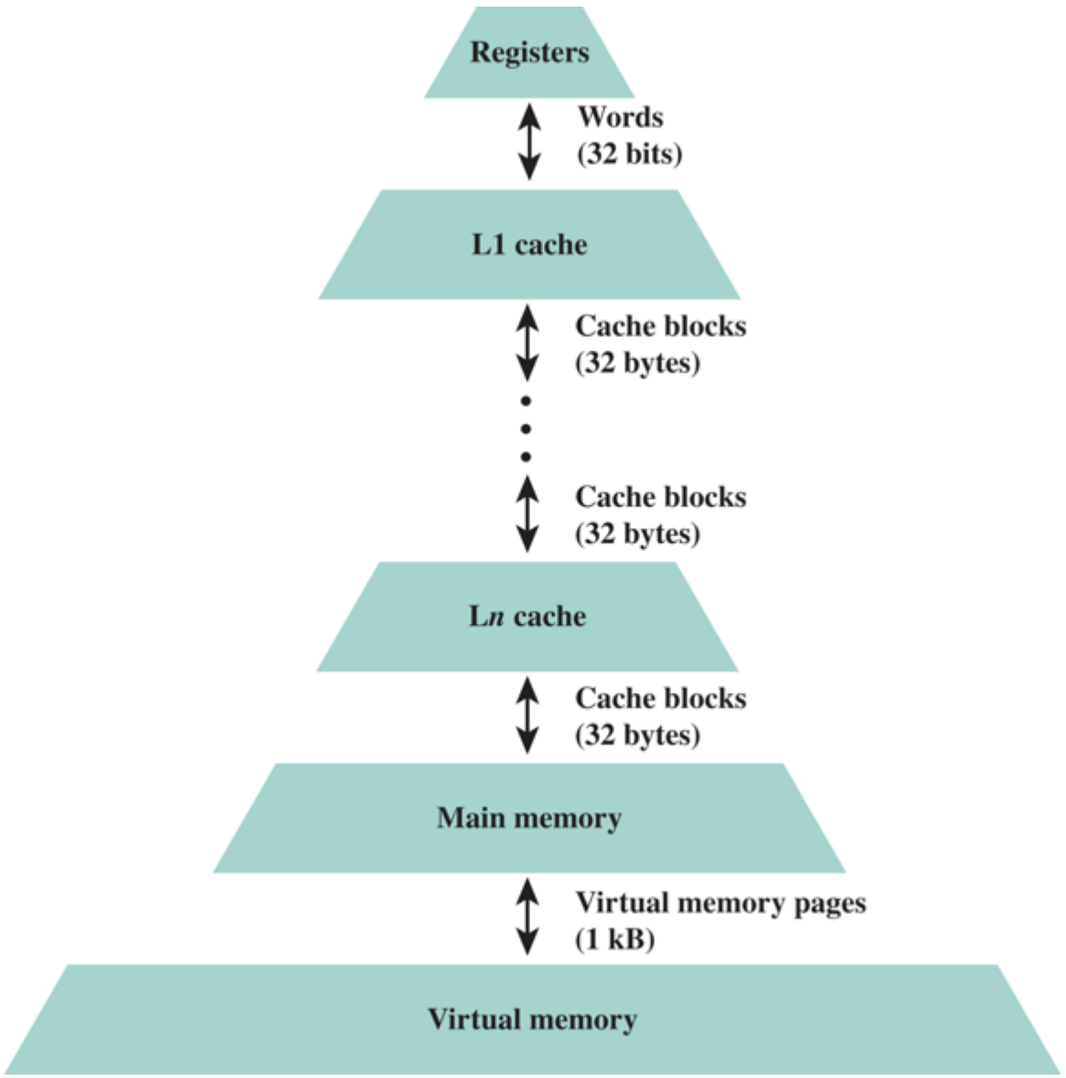


Figure 4.9 Exploiting Locality in the Memory Hierarchy (with typical transfer size)

Typical Members of the Memory Hierarchy

Table 4.2 lists some characteristics of key elements of the memory hierarchy. The fastest, smallest, and most expensive type of memory consists of the registers internal to the processor. Typically, a processor will contain a few dozen such registers, although some machines contain hundreds of registers. Next will be typically multiple layers of cache. Level 1 cache **(L1 cache)**, closest to the processor registers, is almost always divided into an instruction cache and a data cache. This split is also common for **L2 caches**. Most contemporary machines also have an **L3 cache** and some have an **L4 cache**; these two caches generally are not split between instruction and data and may be shared by multiple processors. Traditionally, cache memory has been constructed using a technology called static random access memory (SRAM). More recently, higher levels of cache on many systems have been implemented using embedded dynamic RAM (eDRAM), which is slower than SRAM but faster than the DRAM used to implement the main memory of the computer.

Table 4.2 Characteristics of Memory Devices in a Memory Architecture

Memory level	Typical technology	Unit of transfer with next larger level (typical size)	Managed by
Registers	CMOS	Word (32 bits)	Compiler
Cache	Static RAM (SRAM); Embedded dynamic RAM (eDRAM)	Cache block (32 bytes)	Processor hardware
Main memory	DRAM	Virtual memory page (1 kB)	Operating system (OS)
Secondary memory	Magnetic disk	Disk sector (512 bytes)	OS/user
Offline bulk memory	Magnetic tape		OS/User

Main memory is the principal internal memory system of the computer. Each location in main memory has a unique address. Main memory is visible to the programmer, whereas cache memory is not. The various levels of cache are controlled by hardware and are used for staging the movement of data between main memory and processor registers to improve performance.

The three forms of memory just described are, typically, volatile and employ semiconductor technology. The use of three levels exploits the fact that semiconductor memory comes in a variety of types, which differ in speed and cost. Data are stored more permanently on external mass storage devices, of which the most common are hard disk and removable media, such as removable magnetic disk, tape, and optical storage. External, nonvolatile memory is also referred to as **secondary memory** or **auxiliary memory**. These are used to store program and data files and are usually visible to the programmer only in terms of files and records, as opposed to individual bytes or words. Disk is also used to provide an extension to main memory known as virtual memory, which is discussed in **Chapter 9**. Other forms of secondary memory include optical disks and flash memory.

The IBM z13 Memory Hierarchy

Figure 4.10 illustrates the memory hierarchy for the IBM z13 mainframe computer [LASC16]. It consists of the following levels:

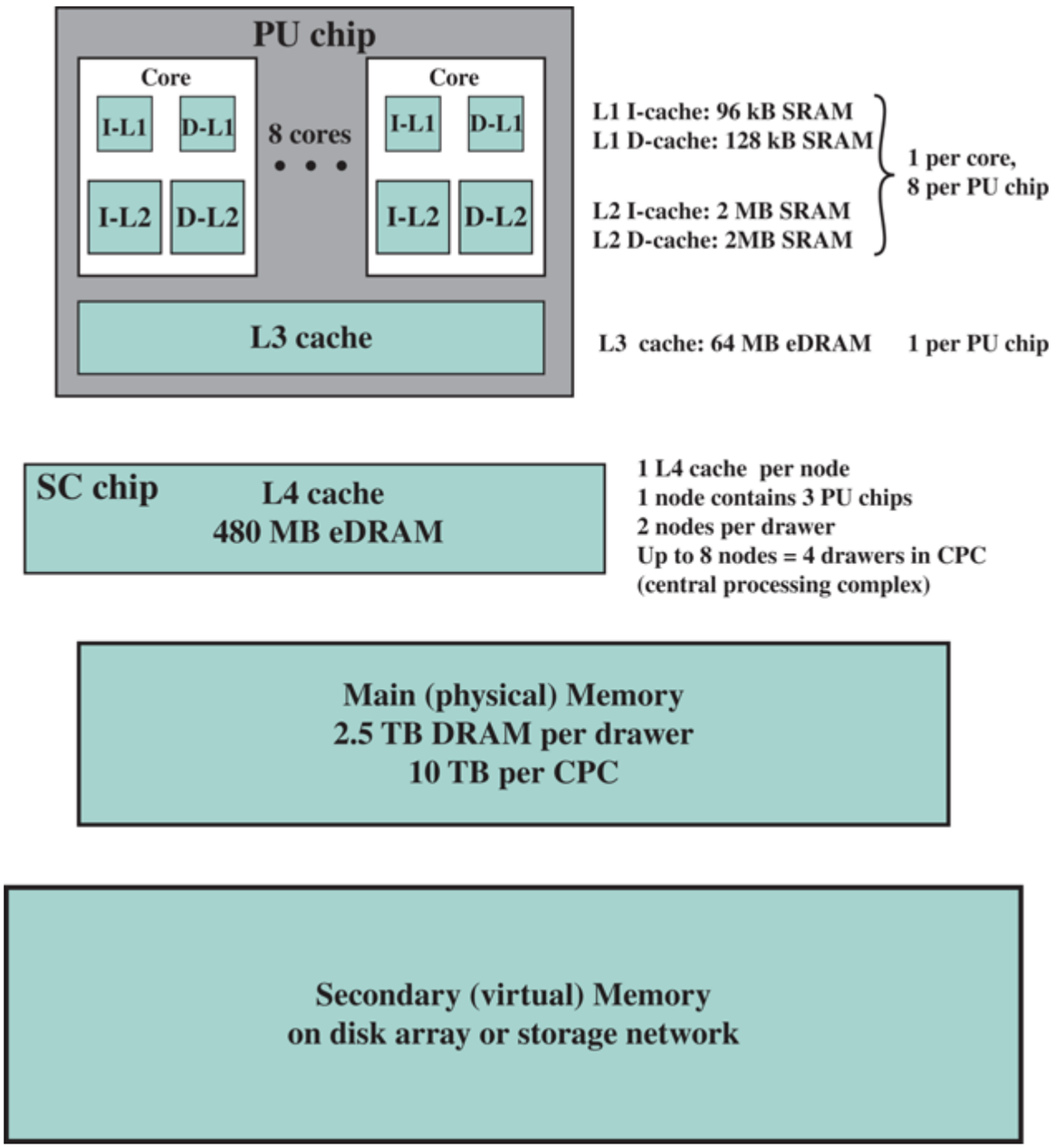


Figure 4.10 IBM z13 Memory Hierarchy

- L1 and L2 caches use SRAM, and are private for each core (**Figure 1.5**).
- L3 cache uses eDRAM and is shared by all eight cores within the PU chip (**Figure 1.4**). Each CPC drawer has six L3 caches. A four-CPC drawer system therefore has 24 of them, resulting in 1536 MB $(24 \times 64\text{MB})$ of this shared PU chip-level cache.
- L4 cache also uses eDRAM, and is shared by all PU chips on the node of a CPC drawer. Each L4 cache has 480 MB for previously owned and some least recently used (LRU) L3-owned lines and 224 MB for a non-data inclusive coherent (NIC) directory that points to L3 owned lines that have not been included in L4 cache. A four-CPC drawer system has 3840 MB $(4 \times 2 \times 384\text{MB})$ of shared L4 cache and 1792 MB $(4 \times 2 \times 224\text{MB})$ of NIC directory.
- Main storage has up to 2.5 TB addressable memory per CPC drawer, using DRAM. A four-CPC drawer system can have up to 10 TB of main storage.
- Secondary memory holds virtual memory and is stored in disks accessed by various I/O technologies.

Design Principles for a Memory Hierarchy

Three principles guide the design of a memory hierarchy and the supporting memory management hardware and software:

1. **Locality:** Locality is the principle that makes effective use of a memory hierarchy possible.
2. **Inclusion:** This principle dictates that all information items are originally stored in level M_n, where *n* is the level most remote from the processor. During the processing, subsets of M_n are copied into M_{n-1}. similarity, subsets of M_{n-1} are copied into M_{n-2}, and so on. This is expressed concisely as $M_i \subseteq M_{i+1}$. Thus, this is in contrast to our simple example of **Figure 4.1**, where Bob moved a folder from the file cabinet to his desk. With the memory hierarchy, units of data are copied rather than moved, so that the data unit that is moved to M_i remains in M_{i+1}. Thus, if a word is found in M_i, then copies of the same word also exist in all subsequent layers $M_{i+1}, M_{i+2}, \ldots, M_n$.
3. **Coherence:** Copies of the same data unit in adjacent memory levels must be consistent. If a word is modified in the cache, copies of that word must be updated immediately or eventually at all higher levels.

Coherence has both vertical and horizontal implications, and is required because multiple memories at one level may share the same memory at the next higher (greater value of *i*) level. For example, for the IBMz13, eight L2 caches share the same L3 cache, and three L3 caches share the same L4 cache. This leads to two requirements:

- **Vertical coherence:** If one core makes a change to a cache block of data at L2, that update must be returned to L3 before another L2 retrieves that block.
- **Horizontal coherence:** If two L2 caches that share the same L3 cache have copies of the same block of data, then if the block in one L2 cache is updated, the other L2 cache must be alerted that its copy is obsolete. The topic of coherence is discussed in future chapters.

4.4 Performance Modeling Of A Multilevel Memory Hierarchy

This section provides an overview of performance characteristics of memory access in a multilevel memory hierarchy. To gain insight, we begin with a look at the simplest case of two levels, and then develop models for multiple levels.

Two-Level Memory Access

In this chapter, reference is made to a cache that acts as a buffer between main memory and processor, creating a two-level internal memory. In the simplest case, rarely implemented in modern systems, there is a single level of cache to interact with main memory. This two-level architecture exploits locality to provide improved performance over a comparable one-level memory.

The main memory cache mechanism is part of the computer architecture, implemented in hardware and typically invisible to the operating system. There are two other instances of a two-level memory approach that also exploit locality and that are, at least partially, implemented in the operating system: virtual memory and the disk cache. Virtual memory is explored in **Chapter 9**; disk cache is beyond the scope of this book but is examined in [STAL18]. In this subsection, we look at some of the performance characteristics of two-level memories that are common to all three approaches.

OPERATION OF TWO-LEVEL MEMORY

The locality property can be exploited in the formation of a two-level memory. The upper-level memory (M1) is smaller, faster, and more expensive (per bit) than the lower-level memory (M2). M1 is used as a temporary store for part of the contents of the larger M2. When a memory reference is made, an attempt is made to access the item in M1. If this succeeds, then a quick access is made. If not, then a block of memory locations is copied from M2 to M1 and the access then takes place via M1. Because of locality, once a block is brought into M1, there should be a number of accesses to locations in that block, resulting in fast overall service.

To express the average time to access an item, we must consider not only the speeds of the two levels of memory, but also the probability that a given reference can be found in M1. We have

$$\begin{aligned} T_s &= H \times T_1 + (1 - H) \times (T_1 + T_2) \\ &= T_1 + (1 - H) \times T_2 \end{aligned} \tag{4.2}$$

where

$$\begin{aligned} T_s &= \text{average (system) access time} \\ T_1 &= \text{access time of M1 (e.g., } cache\text{)} \\ T_2 &= \text{access time of M2 (e.g., } main\ memory\text{)} \\ H &= \text{hit ratio (fraction of time reference is found in M1)} \end{aligned}$$

Figure 4.8 shows average access time as a function of hit ratio. As can be seen, for a high percentage of hits, the average total access time is much closer to that of M1 than M2.

PERFORMANCE

Let us look at some of the parameters relevant to an assessment of a two-level memory mechanism. First consider cost. We have

$$C_1 S_1 + C_2 S_2$$

$$C_s = \frac{C_1 S_1 + C_2 S_2}{S_1 + S_2} \tag{4.3}$$

where

C_s = average cost per bit for the combined two-level memory
C_1 = average cost per bit of upper-level memory M1
C_2 = average cost per bit of lower-level memory M2
S_1 = size of M1
S_2 = size of M2

We would like $C_s \approx C_2$. Given that $C_1 \gg C_2$, this requires $S_1 < S_2$. **Figure 4.11** shows the relationship.

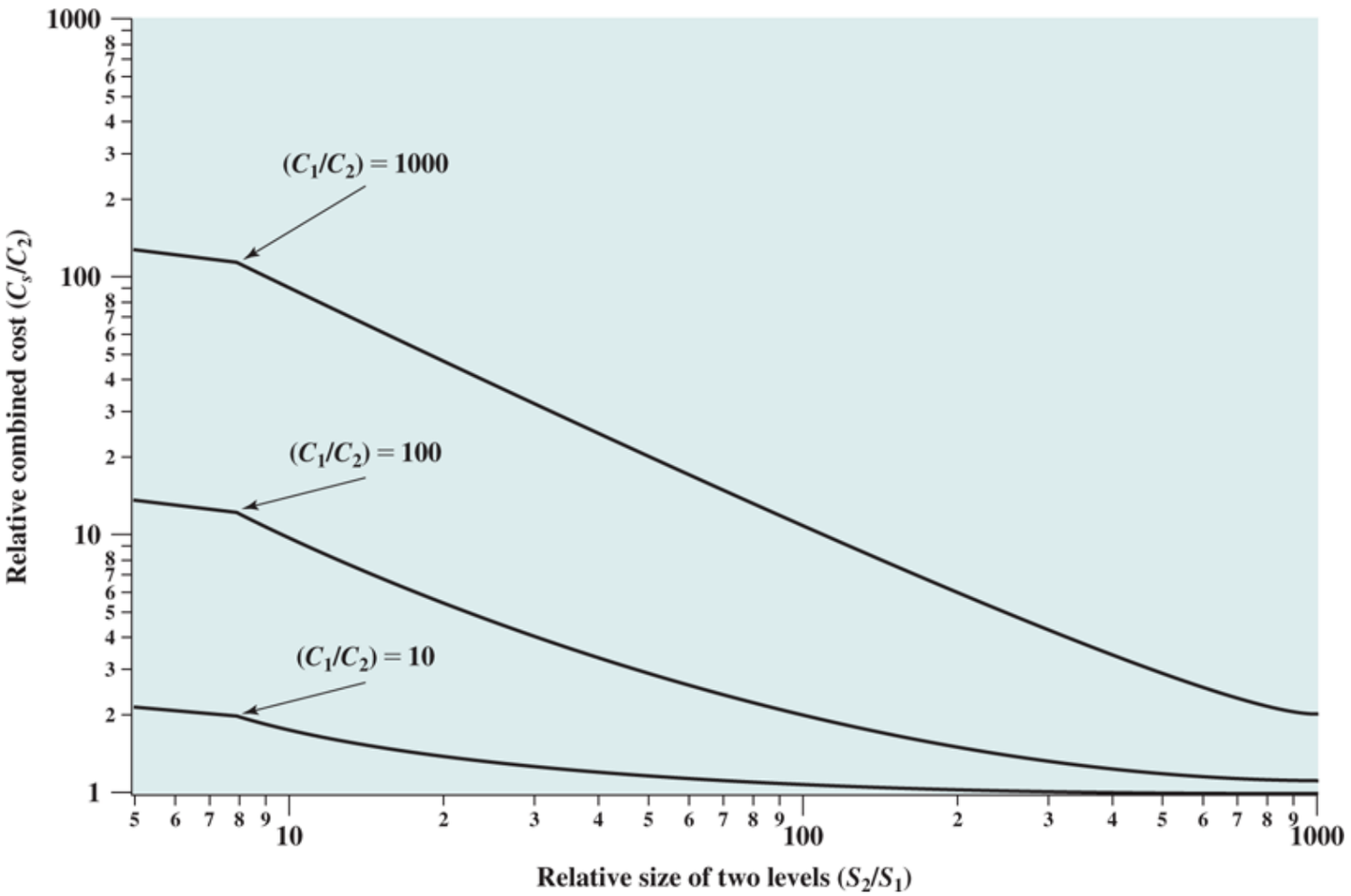


Figure 4.11 Relationship of Average Memory Cost to Relative Memory Size for a Two-Level Memory

Next, consider access time. For a two-level memory to provide a significant performance improvement, we need to have T_s approximately equal to T_1 ($T_s \approx T_1$). Given that T_1 is much less than T_2 ($T_1 << T_2$), a hit ratio of close to 1 is needed.

So we would like M1 to be small to hold down cost, and large to improve the hit ratio and therefore the performance. Is there a size of M1 that satisfies both requirements to a reasonable extent? We can answer this question with a series of subquestions:

- What value of hit ratio is needed so that $T_s \approx T_1$?
- What size of M1 will assure the needed hit ratio?
- Does this size satisfy the cost requirement?

To get at this, consider the quantity T_1 / T_s, which is referred to as the *access efficiency.* It is a measure of how close average access time (T_s) is to M1 access time (T_1). From **Equation (4.2)**,

$$\frac{T_1}{T_s} = \frac{1}{1 + (1 - H)\frac{T_2}{T_1}} \tag{4.4}$$

Figure 4.12 plots T_1 / T_s as a function of the hit ratio *H*, with the quantity T_2 / T_1 as a parameter. Typically, on-chip cache access time is about 25 to 50 times faster than main memory access time (i.e., T_2 / T_1 is 25 to 50), off-chip cache access time is about 5 to 15 times faster than main memory access time (i.e., T_2 / T_1 is 5 to 15), and main memory access time is about 1000 times faster than disk access time $(T_2 / T_1 = 1000)$. Thus, a hit ratio in the range of near 0.9 would seem to be needed to satisfy the performance requirement.

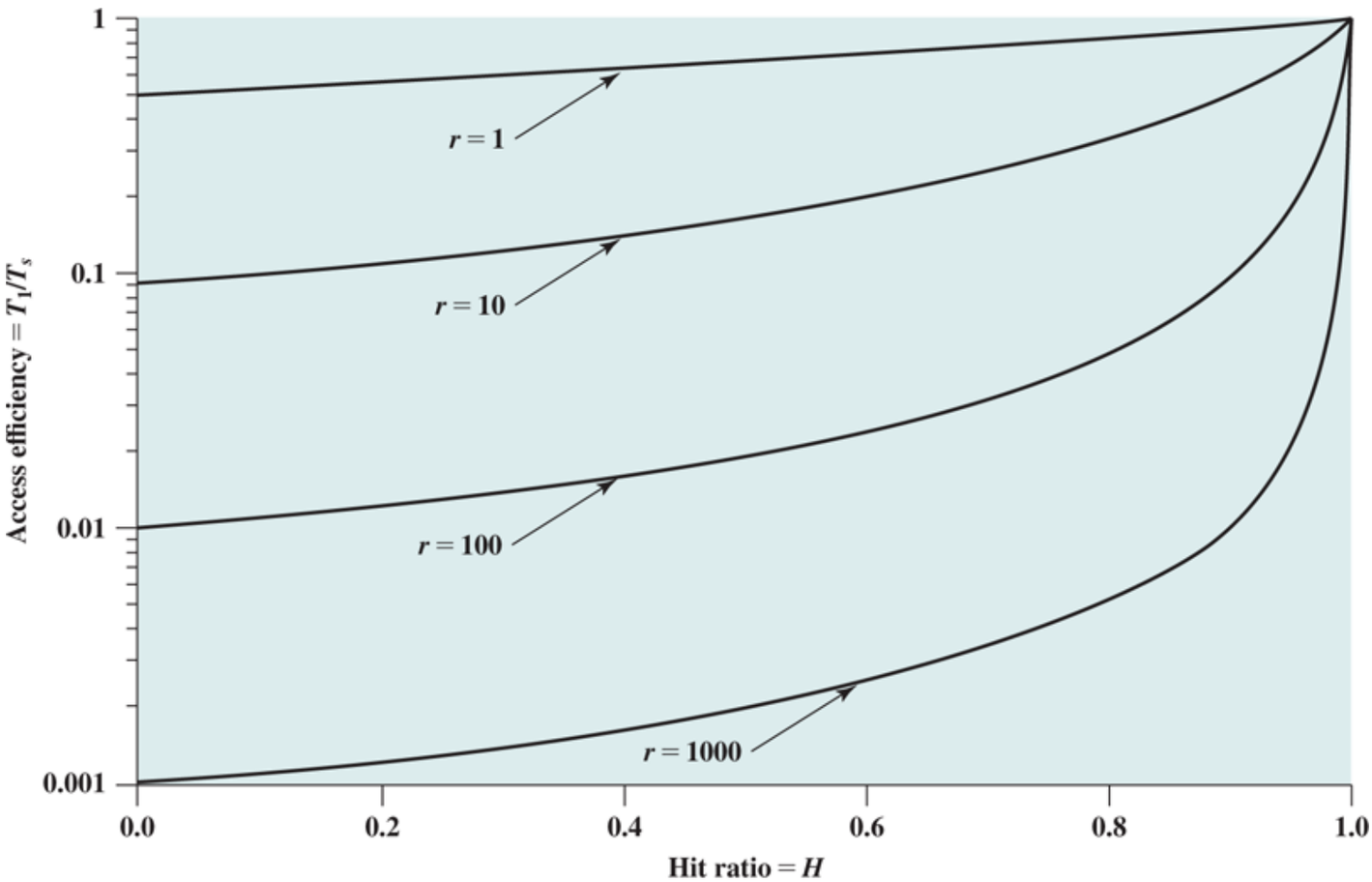


Figure 4.12 Access Efficiency as a Function of Hit Ratio $(r = T_2 / T_1)$

We can now phrase the question about relative memory size more exactly. Is a hit ratio of, say, 0.8 or better reasonable for $S_1 << S_2$? This will depend on a number of factors, including the nature of the software being executed and the details of the design of the two-level memory. The main determinant is, of course, the degree of locality. **Figure 4.13** suggests the effect that locality has on the hit ratio. Clearly, if M1 is the same size as M2, then the hit ratio will be 1.0: All of the items in M2 are always

also stored in M1. Now suppose that there is no locality; that is, references are completely random. In that case, the hit ratio should be a strictly linear function of the relative memory size. For example, if M1 is half the size of M2, then at any time half of the items from M2 are also in M1 and the hit ratio will be 0.5. In practice, however, there is some degree of locality in the references. The effects of moderate and strong locality are indicated in the figure. Note that **Figure 4.13** is not derived from any specific data or model; the figure suggests the type of performance that is seen with various degrees of locality.

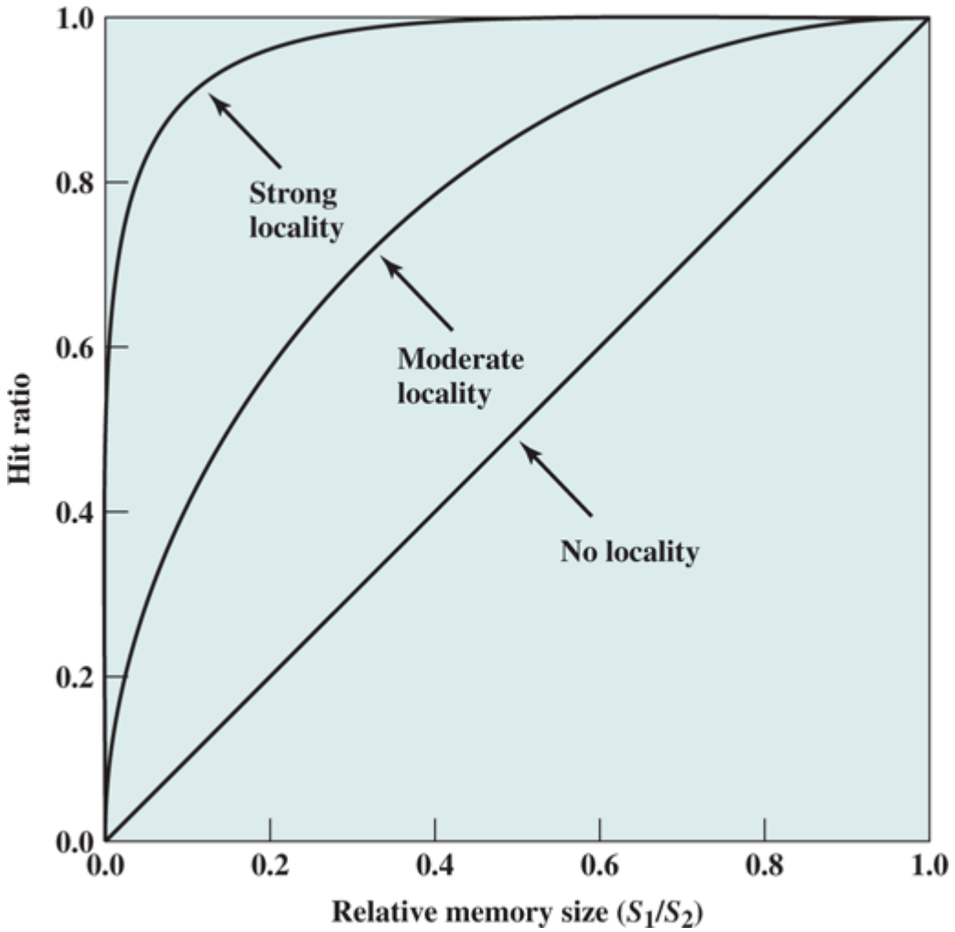


Figure 4.13 Hit Ratio as a Function of Relative Memory Size

So if there is strong locality, it is possible to achieve high values of hit ratio even with relatively small upper-level memory size. For example, numerous studies have shown that rather small cache sizes will yield a hit ratio above 0.75 *regardless of the size of main memory* (e.g., [AGAR89], [PRZY88], [STRE83], and [SMIT82]). A cache in the range of 1K to 128K words is generally adequate, whereas main memory is now typically in the gigabyte range. When we consider virtual memory and disk cache, we will cite other studies that confirm the same phenomenon, namely that a relatively small M1 yields a high value of hit ratio because of locality.

This brings us to the last question listed earlier: Does the relative size of the two memories satisfy the cost requirement? The answer is clearly yes. If we need only a relatively small upper-level memory to achieve good performance, then the average cost per bit of the two levels of memory will approach that of the cheaper lower-level memory.

Multilevel Memory Access[2]

[2]

I would like to thank Professor Roger Kieckhafer of Michigan Technological University for permission to use his lecture notes in developing this section.

This subsection develops a model for memory access performance in a memory hierarchy that has more than two levels. The following terminology is used:

M_i = Memory level (i) where $1 \le i \le n$, with n levels of memory.
S_i = Size, or capacity of level M_i (Bytes)
t_i = Total time needed to access data in level M_i
—Is the sum of all times in the path to a hit in level M_i
—May be an average (t_i)
h_i = Hit ratio of level M_i
= Conditional probability that the data for a memory access is resident in level M_i given that it is not resident in M_{i-1}
T_s = Mean time needed to access data

T_s is the performance metric that is of most interest. It measures the average time to access data, regardless of which level of the hierarchy needs to be accessed at the time of the access request. The higher the hit ratio at each level, the lower will be the value of T_s. Ideally, we would like h_1 to be very close to 1.0, in which case T_s will be very close to t_1.

Figure 4.14 is a flowchart that provides a simplified memory access model for a memory hierarchy, which we can use to develop a formula for the average access time. It can be described as follows:

1. The processor generates a memory address request. The first step is to determine if the cache block containing that address resides in the L1 cache (memory level M_1). The probability of that is h_1. If the desired word is present in the cache, that word is delivered to the processor. The average time required for this operation is t_1.
2. For subsequent levels *i*, $2 \le i < n$, if the addressed word is not found in M_{i-1}, then the memory management hardware checks to determine if it is in M_i, which occurs with a probability of h_i. If the desired word is present in M_i, the word is delivered from M_i to the processor, and the appropriate size block containing the word is copied to M_{i-1}. The average time for this operation is t_{i-1}.
3. In the typical memory hierarchy, M_n is a disk used in a virtual memory scheme. In this case, if the word is not found in any of the preceding levels and is found in M_n, then the word must be first moved as part of a page into main memory (M_{n-1}), from where it can be transferred to the processor. We designate the total time for this operation as t_n.

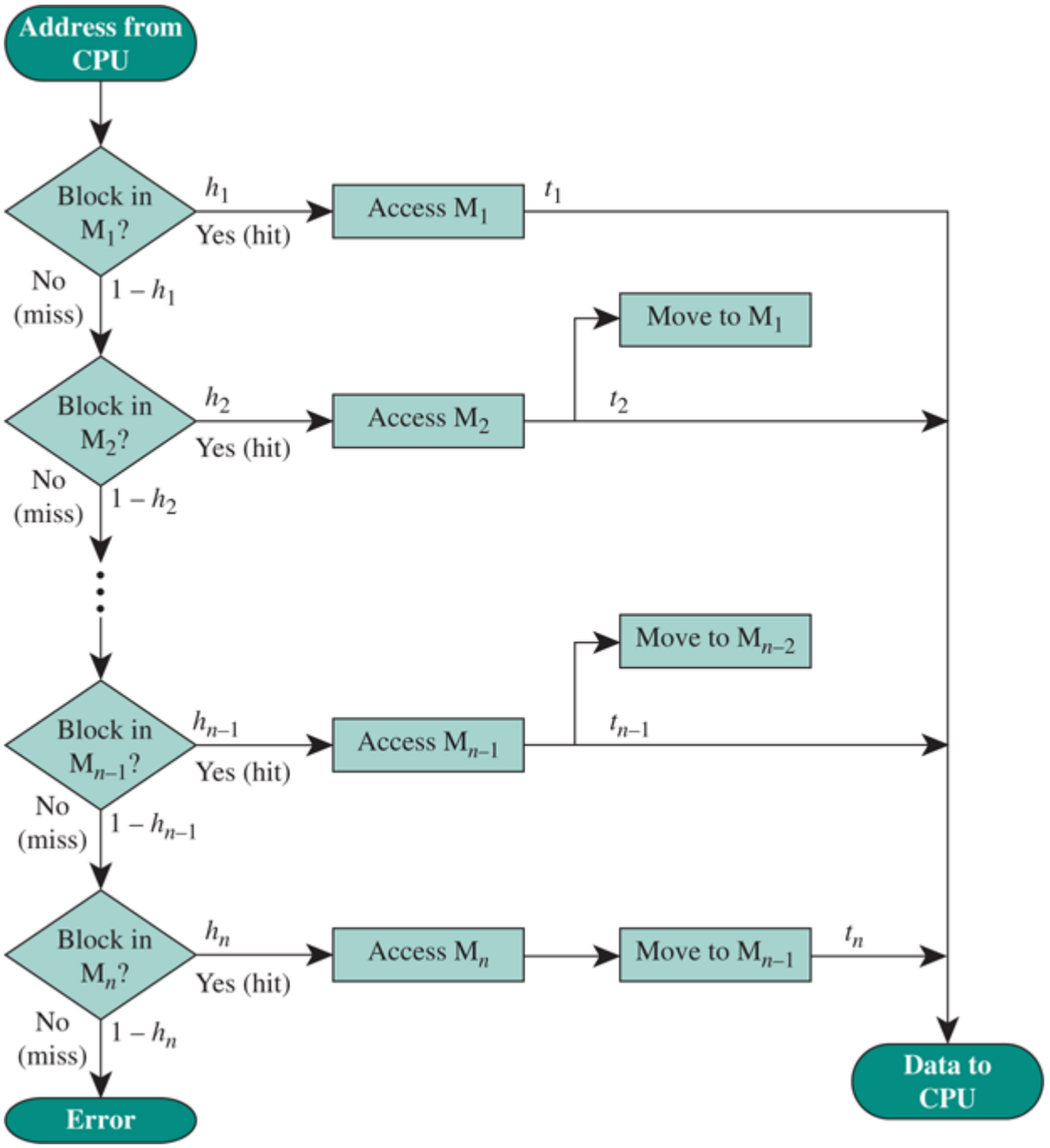


Figure 4.14 Multilevel Memory Access Performance Model

Each of the t_i consists of a number of components, including checking for the presence of the required word in level *i*, accessing the data if it is in level *i*, and transporting the data to the processor. The total value of t_i must also include the amount of time to check for a hit on all previous levels and experiencing a miss. If we designate the time expended in determining a miss at level *i* as t_{misi}, then t_i must include $t_{mis1} + t_{mis2} + \ldots + t_{misi-1}$. In addition, **Figure 4.14** indicates that the process of accessing memory and delivering a word to the processor is performed parallel to the process of copying the appropriate block of data to the preceding level in the hierarchy. If the two operations are performed in sequence, then the extra time involved is added to t_i.

Looking at **Figure 4.14**, there are a number of different paths from start to finish. The average time T_s can be expressed as the weighted average of the time of each path:

$$T_s = \sum_{\text{all paths}} [\text{Probability of taking a path} \times \text{Duration of that a path}] \quad (4.5)$$

$$= \sum_{\text{all paths}} \prod (\text{All probabilities in the path}) \times \sum (\text{All times in that path})$$

$$= \sum_{i=1}^{n} \prod_{j=0}^{i-1} (1 - h_j)\, h_I \times t_I$$

where h_0 is assigned the value 0.

For example, consider a simple system consisting of one level of cache (M_1), main memory (M_2), and secondary memory (M_3). Then,

$$\begin{aligned} T_s = \quad & h_1 \times t_1 \\ + \quad & (1 - h_1)\, h_2 \times t_2 \\ + \quad & (1 - h_1)\,(1 - h_2) \times t_3 \end{aligned}$$

Note that **Equation (4.5)** works whether the time delays for a given path are constants or variables. If the time delays are constant, then t_i is a constant equal to the sum of all the time delays (e.g. checking for presence, data access, and delivery to CPU). If one or more of the elements in the total time delay are variable, then t_i is the mean time delay calculated as the sum of the mean time delays of the component delays.

To use this model in designing a memory hierarchy, estimates are needed for the h_i and t_i. These can be developed either by simulation or by setting up a real system and varying the sizes of the various M_i.

4.5 Key Terms, Review Questions, and Problems

Key Terms

access time

addressable unit

associative memory

auxiliary memory

cache memory

coherence

data spatial locality

data temporal locality

direct access

dynamic instruction

hit ratio

horizontal coherence inclusion

instruction cache

instruction spatial locality

instruction temporal locality

L1 cache

L2 cache

L3 cache

L4 cache

locality

locality of reference

memory hierarchy

memory cycle time

multilevel cache

multilevel memory

random access

secondary memory

sequential access

spatial locality

static instruction

temporal locality

transfer rate

unit of transfer

vertical coherence

word

Review Questions

4.1 What are the differences among sequential access, direct access, and random access?
4.2 What is the general relationship among access time, memory cost, and capacity?
4.3 How does the principle of locality relate to the use of multiple memory levels?
4.4 What is the distinction between spatial locality and temporal locality?
4.5 In general, what are the strategies for exploiting spatial locality and temporal locality?
4.6 How do data locality and instruction locality relate to spatial locality and temporal locality?

Problems

4.1 Consider these terms: instruction spatial locality, instruction temporal locality, data spatial locality, data temporal locality. Match each of these terms to one of the following definitions:

a. Locality is quantified by computing the average distance (in terms of number of operand memory accesses) between two consecutive accesses to the same address, for every unique address in the program. The evaluation is performed in four distinct window sizes, analogous to cache block sizes.
b. Locality metric is quantified by computing the average distance (in terms of number of instructions) between two consecutive accesses to the same static instruction, for every unique static instruction in the program that is executed at least twice.
c. Locality for operand memory accesses is characterized by the ratio of the locality metric for window sizes mentioned in (a).
d. Locality is characterized by the ratio of the locality metric for the window sizes mentioned in (b).

4.2 Consider these two programs:

`for ( i = 1; i < n; i++) {`
`  Z[i] = X[i] - Y[i]`
`  Z[i] = Z[i] * Z[i]`
`}`
	`for ( i = 1; i < n; i++) {`
	`  Z[i] = X[i] - Y[i]`
	`}`
	`for ( i = 1; i < n; i++) {`
	`  Z[i] = Z[i] * Z[i]`
	`}`

Program A	Program B

a. The two programs perform the same function. Describe it.
b. Which version performs better, and why?

4.3 Consider the following code:

```
for (i = 0; i < 20; i++)
   for (j = 0; j < 10; j++)
      a[i] = a[i] * j
```

a. Give one example of the spatial locality in the code.
b. Give one example of the temporal locality in the code.

4.4 Consider a memory system with the following parameters:

$T_c = 100\text{ns}$	$C_c = 10^{-4}\$ / \text{bit}$
$T_m = 1200\text{ns}$	$C_m = 10^{-5}\$ / \text{bit}$

a. What is the cost of 1 MB of main memory?
b. What is the cost of 1 MB of main memory using cache memory technology?
c. If the effective access time is 10% greater than the cache access time, what is the hit ratio *H*?

4.5
a. Consider an L1 cache with an access time of 1 ns and a hit ratio of $H = 0.95$. Suppose that we can change the cache design (size of cache, cache organization) such that we increase *H* to 0.97, but increase access time to 1.5 ns. What conditions must be met for this change to result in improved performance?
b. Explain why this result makes intuitive sense.

4.6 Consider a single-level cache with an access time of 2.5 ns, a block size of 64 bytes, and a hit ratio of $H = 0.95$. Main memory uses a block transfer capability that has a first-word (4 bytes) access time of 50 ns and an access time of 5 ns for each word thereafter.
a. What is the access time when there is a cache miss? Assume that the cache waits until the line has been fetched from main memory and then re-executes for a hit.
b. Suppose that increasing the block size to 128 bytes increases the H to 0.97. Does this reduce the average memory access time?

4.7 A computer has a cache, main memory, and a disk used for virtual memory. If a referenced word is in the cache, 20 ns are required to access it. If it is in main memory but not in the cache, 60 ns are needed to load it into the cache, and then the reference is started again. If the word is not in main memory, 12 ns are required to fetch the word from the disk, followed by 60 ns to copy it to the cache, and then the reference is started again. The cache hit ratio is 0.9 and the main memory hit ratio is 0.6. What is the average time in nanoseconds required to access a referenced word on this system?
4.8 On the Motorola 68020 microprocessor, a cache access takes two clock cycles. Data access from main memory over the bus to the processor takes three clock cycles in the case of no wait state insertion; the data are delivered to the processor in parallel with delivery to the cache.

a. Calculate the effective length of a memory cycle given a hit ratio of 0.9 and a clocking rate of 16.67 MHz.
b. Repeat the calculations assuming insertion of two wait states of one cycle each per memory cycle. What conclusion can you draw from the results?

4.9 Assume a processor having a memory cycle time of 300 ns and an instruction processing rate of 1 MIPS. On average, each instruction requires one bus memory cycle for instruction fetch and one for the operand it involves.
a. Calculate the utilization of the bus by the processor.
b. Suppose that the processor is equipped with an instruction cache and the associated hit ratio is 0.5. Determine the impact on bus utilization.

4.10 The performance of a single-level cache system for a read operation can be characterized by the following equation:

$$T_a = T_c + (1 - H)\,T_m$$

where T_a is the average access time, T_c is the cache access time, T_m is the memory access time (memory to processor register), and *H* is the hit ratio. For simplicity, we assume that the word in question is loaded into the cache in parallel with the load to processor register. This is the same form as **Equation (4.2)** .
a. Define $T_b = \text{time}$ to transfer a block between cache and main memory, and $W = \text{fraction}$ of write references. Revise the preceding equation to account for writes as well as reads, using a write-through policy.
b. Define W_b as the probability that a block in the cache has been altered. Provide an equation for T_a for the write-back policy.

4.11 For a system with two levels of cache, define $T_{c1} = \text{first-level}$ cache access time; $T_{c2} = \text{second-level}$ cache access time; $T_m = \text{memory}$ access time; $H_1 = \text{first-level}$ cache hit ratio; $H_2 = \text{combined}$ first/second level cache hit ratio. Provide an equation for T_a for a read operation.

4.12 Assume the following performance characteristics on a cache read miss: one clock cycle to send an address to main memory and four clock cycles to access a 32-bit word from main memory and transfer it to the processor and cache.
a. If the cache block size is one word, what is the miss penalty (i.e., additional time required for a read in the event of a read miss)?
b. What is the miss penalty if the cache block size is four words and a multiple, nonburst transfer is executed?
c. What is the miss penalty if the cache block size is four words and a transfer is executed, with one clock cycle per word transfer?

4.13 For the cache design of the preceding problem, suppose that increasing the line size from one word to four words results in a decrease of the read miss rate from 3.2% to 1.1%. For both the nonburst transfer and the burst transfer case, what is the average miss penalty, averaged over all reads, for the two different line sizes?

4.14 Consider a two-level system with L1 instruction and data caches. For a given application, assume the following: instruction cache miss $\text{ratio} = 0.02$, data cache miss $\text{ratio} = 0.04$, and the fraction of instructions that are $\text{load / store} = 0.36$. The ideal value of CPI (cycles per instruction) without cache misses is 2.0. The penalty for a cache miss is 40 cycles. Calculate the CPI, taking misses into account.

4.15 Define $H_i = \text{probability}$ that the data for a memory access is resident in level M_i.

a. **Equation 4.5** uses the conditional probabilities h_i. Explain why in this form the equation is correct with the conditional probabilities rather than the unconditional probabilities H_i. That is, show that the following expression does not equal T_s.

$$\sum_{i-1}^{i} \prod_{j=1}^{1} (1 - h_j)\, h_I \times t_I$$

b. Rewrite **Equation 4.5** using H_i instead of h_i.

4.16 Define the access frequency f_i as the probability of successfully accessing (hit) M_i when there are misses at the preceding $i - 1$ levels.

a. Derive an expression for f_i.
b. Rewrite **Equation 4.5** using f_i instead of h_i.

Chapter 5 Cache Memory

Learning Objectives

After studying this chapter, you should be able to:

- Discuss the key elements of cache design.
- Distinguish among direct mapping, associative mapping, and set-associative mapping.
- Understand the principles of content-addressable memory.
- Explain the reasons for using multiple levels of cache.
- Understand the performance implications of cache design decisions.

With the exception of smaller embedded systems, all modern computer systems employ one or more layers of cache memory. Cache memory is vital to achieving high performance. This chapter begins with an overview of the basic principles of cache memory, then looks in detail at the key elements of cache design. This is followed by a discussion of the cache structures used in the Intel x86 family and the IBM z13 mainframe system. Finally, the chapter introduces some straightforward performance models that provide insight into cache design.

5.1 Cache Memory Principles

Cache memory is designed to combine the memory access time of expensive, high-speed memory combined with the large memory size of less expensive, lower-speed memory. The concept is illustrated in **Figure 5.1a**. There is a relatively large and slow main memory together with a smaller, faster cache memory. The cache contains a copy of portions of the main memory. When the processor attempts to read a word of memory, a check is made to determine if the word is in the cache. If so, the word is delivered to the processor. If not, a block of main memory, consisting of some fixed number of words, is read into the cache and then the word is delivered to the processor. Because of the phenomenon of locality of reference, when a block of data is fetched into the cache to satisfy a single memory reference, it is likely that there will be future references to that same memory location or to other words in the block.

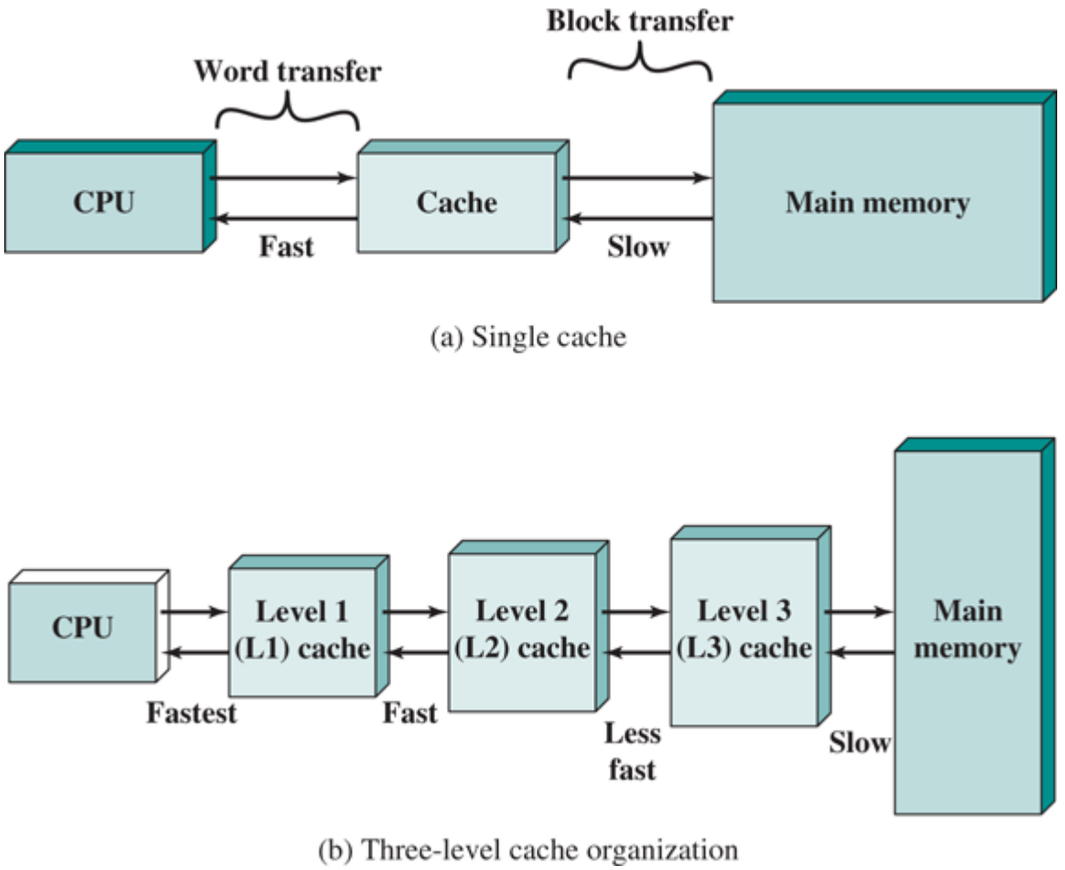


Figure 5.1 Cache and Main Memory

Figure 5.1b depicts the use of multiple levels of cache. The **L2 cache** is slower and typically larger than the **L1 cache**, and the **L3 cache** is slower and typically larger than the L2 cache.

Figure 5.2 depicts the structure of a cache/main-memory system. Several terms are introduced:

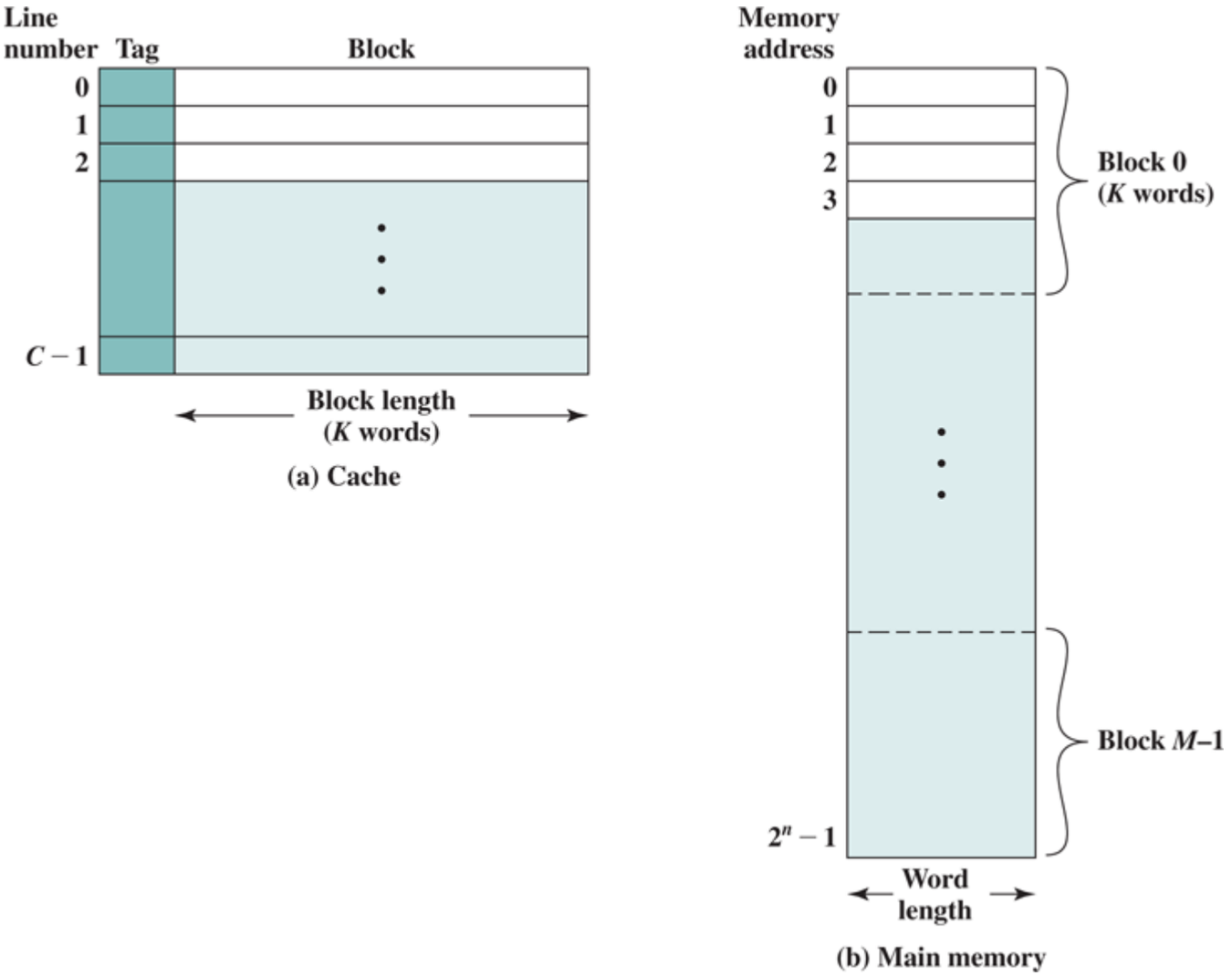


Figure 5.2 Cache/Main Memory Structure

- **Block:** The minimum unit of transfer between cache and main memory. In most of the literature, the term block refers both to the unit of data transferred and to the physical location in main memory or cache.
- **Frame:** To distinguish between the data transferred and the chunk of physical memory, the term frame, or *block frame*, is sometimes used with reference to caches. Some texts and some literature use the term with reference to the cache and some with reference to main memory. It use is not necessary for purposes of this text.
- **Line:** A portion of cache memory capable of holding one block, so-called because it is usually drawn as a horizontal object (i.e., all bytes of the line are typically drawn in one row). A line also includes control information.
- **Tag:** A portion of a cache line that is used for addressing purposes, as explained subsequently. A cache line may also include other control bits, as will be shown.

Main memory consists of up to 2^n addressable words, with each word having a unique *n*-bit address. For mapping purposes, this memory is considered to consist of a number of fixed-length blocks of *K* words each. That is, there are $M = 2^n / K$ blocks in main memory. The cache consists of *m* lines. Each line contains *K* words, plus a tag. Each line also includes control bits (not shown), such as a bit to indicate whether the line has been modified since being loaded into the cache. The length of a line, not including tag and control bits, is the **line size**. That is, the term line size refers to the number of data bytes, or block size, contained in a line. They may be as small as 32 bits, with each “word” being

a single byte; in this case the line size is 4 bytes. The number of lines is considerably less than the number of main memory blocks $(m \ll M)$. At any time, some subset of the blocks of memory resides in lines in the cache. If a word in a block of memory is read, that block is transferred to one of the lines of the cache. Because there are more blocks than lines, an individual line cannot be uniquely and permanently dedicated to a particular block. Thus, each line includes a tag that identifies which particular block is currently being stored. The tag is usually a portion of the main memory address, as described later in this section.

Figure 5.3 illustrates the read operation. The processor generates the read address (RA) of a word to be read. If the word is contained in the cache (cache hit), it is delivered to the processor.

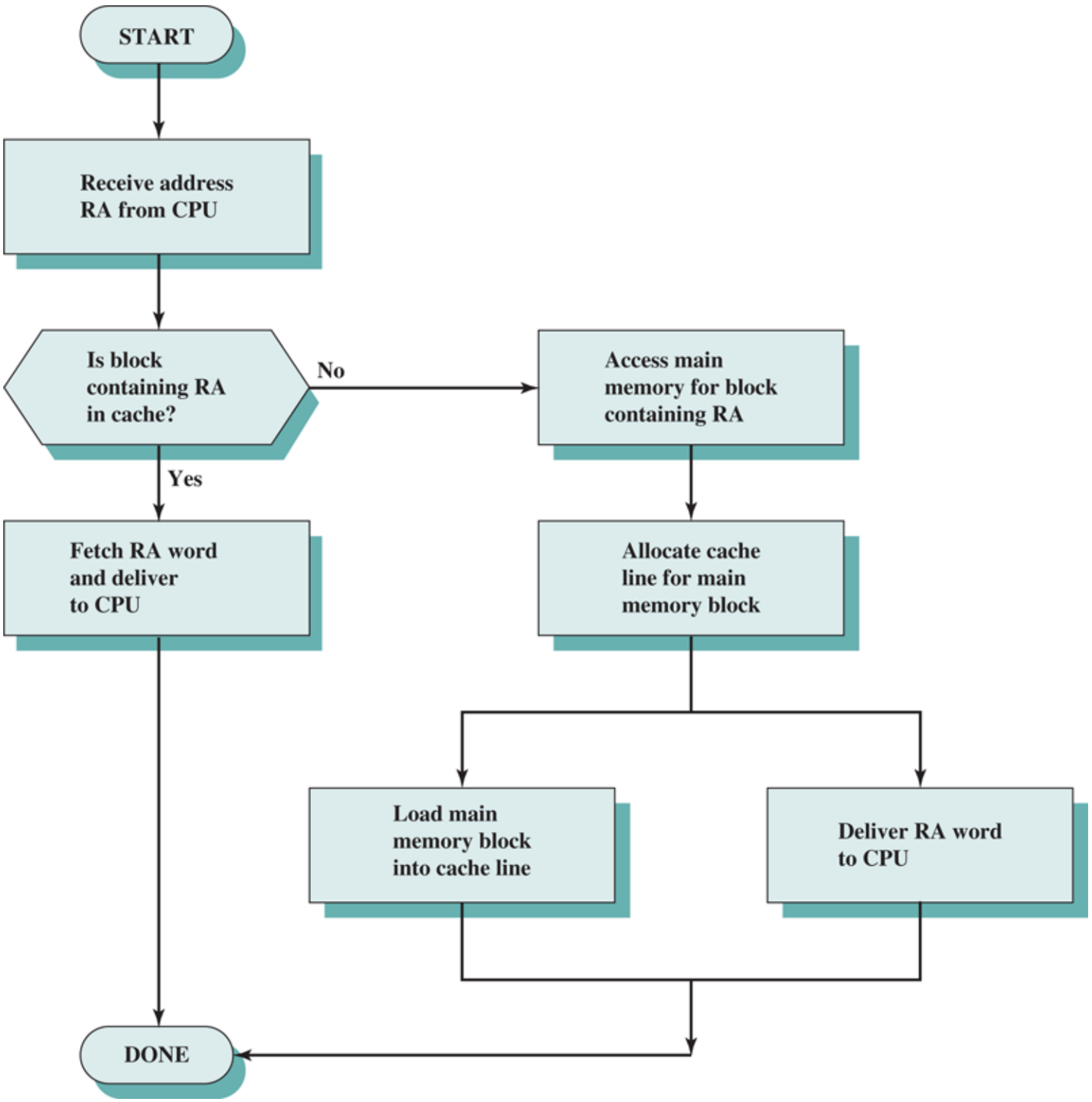


Figure 5.3 Cache Read Operation

If a cache miss occurs, two things must be accomplished: the block containing the word must be loaded in to the cache, and the word must be delivered to the processor. When a block is brought into a cache in the event of a miss, the block is generally not transferred in a single event. Typically, the transfer size between cache and main memory is less than the line size, with 128 bytes being a typical line size and a cache-main memory transfer size of 64 bits (2 bytes). To improve performance, the **critical word first** technique is commonly used. When there is a cache miss, the hardware requests the missed word first from memory and sends it to the processor as soon as it arrives. This enables the processor to continue execution while filling the rest of the words in the block. **Figure 5.3** shows these last two operations occurring in parallel and reflects the organization shown in **Figure 5.4**, which is typical of contemporary cache organizations. In this organization, the cache connects to the processor via data, control, and address lines. The data and address lines also attach to data and address buffers, which attach to a system bus from which main memory is reached. When a cache hit occurs, the data and address buffers are disabled and communication is only between processor and cache, with no system bus traffic. When a cache miss occurs, the desired address is loaded onto the system bus and the data are returned through the data buffer to both the cache and the processor.

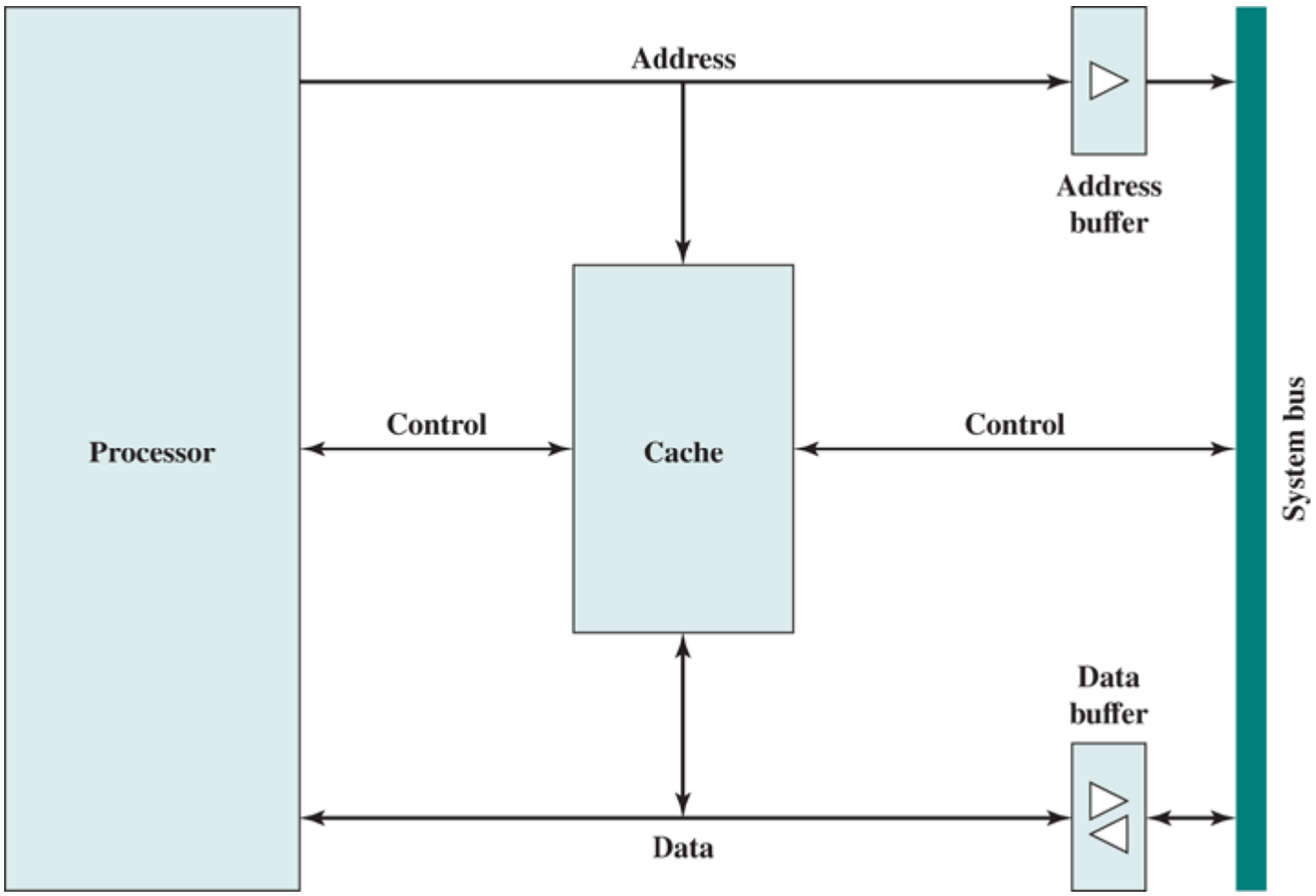


Figure 5.4 Typical Cache Organization

5.2 Elements of Cache Design

This section provides an overview of cache design parameters and reports some typical results. We occasionally refer to the use of caches in **high-performance computing (HPC)** . HPC deals with supercomputers and their software, especially for scientific applications that involve large amounts of data, vector and matrix computation, and the use of parallel algorithms. Cache design for HPC is quite different than for other hardware platforms and applications. Indeed, many researchers have found that HPC applications perform poorly on computer architectures that employ caches [BAIL93]. Other researchers have since shown that a cache hierarchy can be useful in improving performance if the application software is tuned to exploit the cache [WANG99, PRES01].[4]

[4] For a general discussion of HPC, see [DOWD98].

Although there are a large number of cache implementations, there are a few basic design elements that serve to classify and differentiate cache architectures. **Table 5.1** lists key elements.

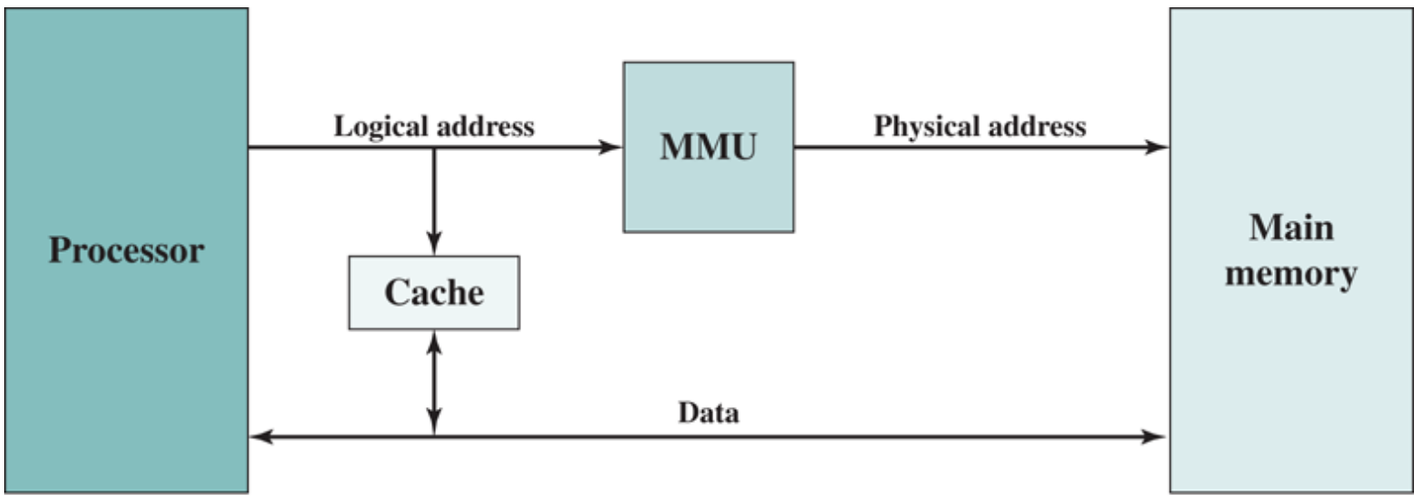
Table 5.1 Elements of Cache Design

Cache Addresses	**Write Policy**
Logical	Write through
Physical	Write back
Cache Size	**Line Size**
Mapping Function	**Number of Caches**
Direct	Single or two level
Associative	Unified or split
Set associative	
Replacement Algorithm	
Least recently used (LRU)	
First in first out (FIFO)	
Least frequently used (LFU)	
Random	

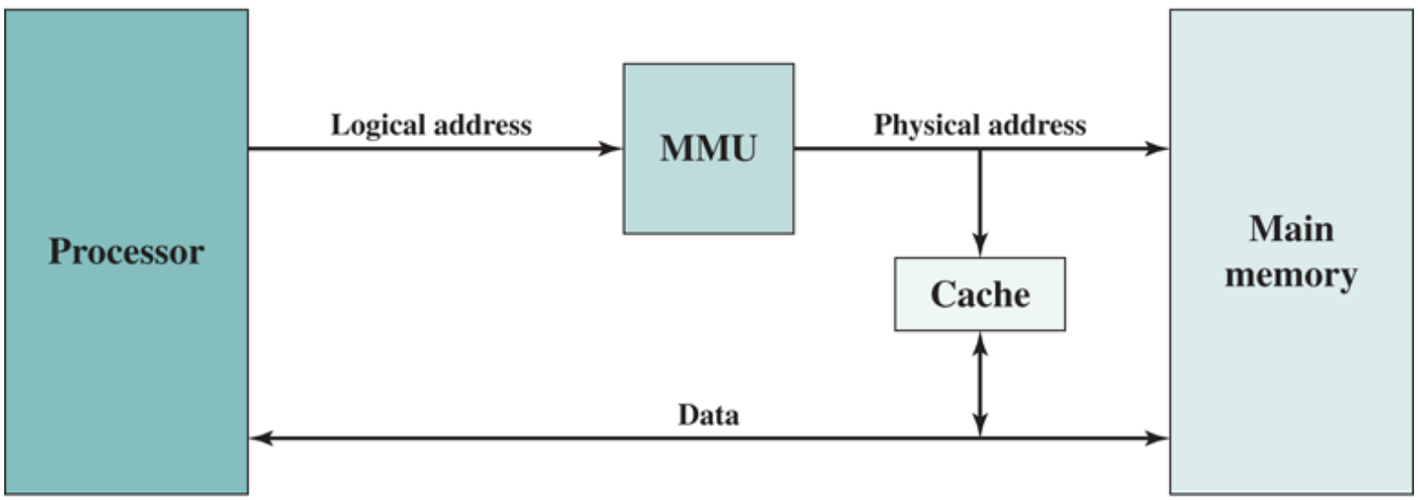
Cache Addresses

Almost all nonembedded processors, and many embedded processors, support virtual memory, a concept discussed in **Chapter 9**. In essence, virtual memory is a facility that allows programs to address memory from a logical point of view, without regard to the amount of main memory physically available. When virtual memory is used, the address fields of machine instructions contain virtual addresses. For reads to and writes from main memory, a hardware memory management unit (MMU) translates each virtual address into a physical address in main memory.

When virtual addresses are used, the system designer may choose to place the cache between the processor and the MMU or between the MMU and main memory (**Figure 5.5**). A **logical cache**, also known as a **virtual cache**, stores data using **virtual addresses**. The processor accesses the cache directly, without going through the MMU. A **physical cache** stores data using main memory **physical addresses**.



(a) Logical cache



(b) Physical cache

Figure 5.5 Logical and Physical Caches

One obvious advantage of the logical cache is that cache access speed is faster than for a physical cache, because the cache can respond before the MMU performs an address translation. The disadvantage has to do with the fact that most virtual memory systems supply each application with the same virtual memory address space. That is, each application sees a virtual memory that starts at

address 0. Thus, the same virtual address in two different applications refers to two different physical addresses. The cache memory must therefore be completely flushed with each application context switch, or extra bits must be added to each line of the cache to identify which virtual address space this address refers to.

The subject of logical versus physical cache is a complex one, and beyond the scope of this text. For a more in-depth discussion, see [CEKL97] and [JACO08].

Cache Size

The second item in **Table 5.1**, cache size, has already been discussed. We would like the size of the cache to be small enough so that the overall average cost per bit is close to that of main memory alone and large enough so that the overall average access time is close to that of the cache alone. There are several other motivations for minimizing cache size. The larger the cache, the larger the number of gates involved in addressing the cache. The result is that large caches tend to be slightly slower than small ones—even when built with the same integrated circuit technology and put in the same place on a chip and circuit board. The available chip and board area also limits cache size. Because the performance of the cache is very sensitive to the nature of the workload, it is impossible to arrive at a single "optimum" cache size. **Table 5.2** lists the cache sizes of some current and past processors.

Table 5.2 Cache Sizes of Some Processors

Processor	Type	Year of Introduction	L1 Cache**a**	L2 cache	L3 Cache
IBM 360/85	Mainframe	1968	16 to 32 kB	—	—
PDP-11/70	Minicomputer	1975	1 kB	—	—
IBM 3033	Mainframe	1978	64 kB	—	—
IBM 3090	Mainframe	1985	128 to 256 kB	—	—
Intel 80486	PC	1989	8 kB	—	—
Pentium	PC	1993	8 kB/8 kB	256 to 512 kB	—
PowerPC 620	PC	1996	32 kB/32 kB	—	—
IBM S/390 G6	Mainframe	1999	256 kB	8 MB	—
Pentium 4	PC/server	2000	8 kB/8 kB	256 kB	—
Itanium	PC/server	2001	16 kB/16 kB	96 kB	4 MB
Itanium 2	PC/server	2002	32 kB	256 kB	6 MB
IBM POWER5	High-end server	2003	64 kB	1.9 MB	36 MB

CRAY XD-1	Supercomputer	2004	64 kB/64 kB	1MB	—
IBM POWER6	PC/server	2007	64 kB/64 kB	4 MB	32 MB
IBM z10	Mainframe	2008	64 kB/128 kB	3 MB	24-48 MB
Intel Core i7 EE 990	Workstaton/Server	2011	$6 \times 32\text{kB} / 32\text{kB}$	$6 \times 1.5\text{MB}$	12 MB
IBM zEnterprise 196	Mainframe/Server	2011	$24 \times 64\text{kB} / 128\text{kB}$	$24 \times 1.5\text{MB}$	24 MB L3 192 MB L4
IBM z13	Mainframe/server	2015	$24 \times 96\text{kB} / 128\text{kB}$	$24 \times 2\text{MB} / 2\text{MB}$	64 MB L3 480 MB L4
Intel Core i0-7900X	Workstation/server	2017	$8 \times 32\text{kB} / 32\text{kB}$	$8 \times 1\text{MB}$	14 MB

[a] *Two values separated by a slash refer to instruction and data caches.*

Logical Cache Organization

Because there are fewer cache lines than main memory blocks, an algorithm is needed for mapping main memory blocks into cache lines. Further, a means is needed for determining which main memory block currently occupies a **cache line** . The choice of the mapping function dictates how the cache is logically organized. Three techniques can be used: direct, associative, and set-associative. We examine each of these in turn. In each case, we look at the general structure and then a specific example. **Table 5.3** provides a summary of key characteristics of the three approaches.

Table 5.3 Cache Access Methods

Method	Organization	Mapping of Main Memory Blocks to Cache	Access using Main Memory Address
Direct Mapped	Sequence of *m* lines	Each block of main memory maps to	*Line* portion of address used to access cache line; *Tag* portion used to check

		one unique line of cache.	for hit on that line.
Fully Associative	Sequence of *m* lines	Each block of main memory can map to any line of cache.	*Tag* portion of address used to check every line for hit on that line.
Set Associative	Sequence of *m* lines organized as *v* sets of *k* lines each $(m = v \times k)$	Each block of main memory maps to one unique cache set.	*Line* portion of address used to access cache set; *Tag* portion used to check every line in that set for hit on that line.

Example 5.1

For all three cases, the example includes the following elements:

- The cache can hold 64 kB.
- Data are transferred between main memory and the cache in blocks of 4 bytes each. This means that the cache is organized as $16K = 2^{14}$ lines of 4 bytes each.
- The main memory consists of 16 MB, with each byte directly addressable by a 24-bit address $(2^{24} = 16\text{M})$. Thus, for mapping purposes, we can consider main memory to consist of 4M blocks of 4 bytes each.

Direct Mapping

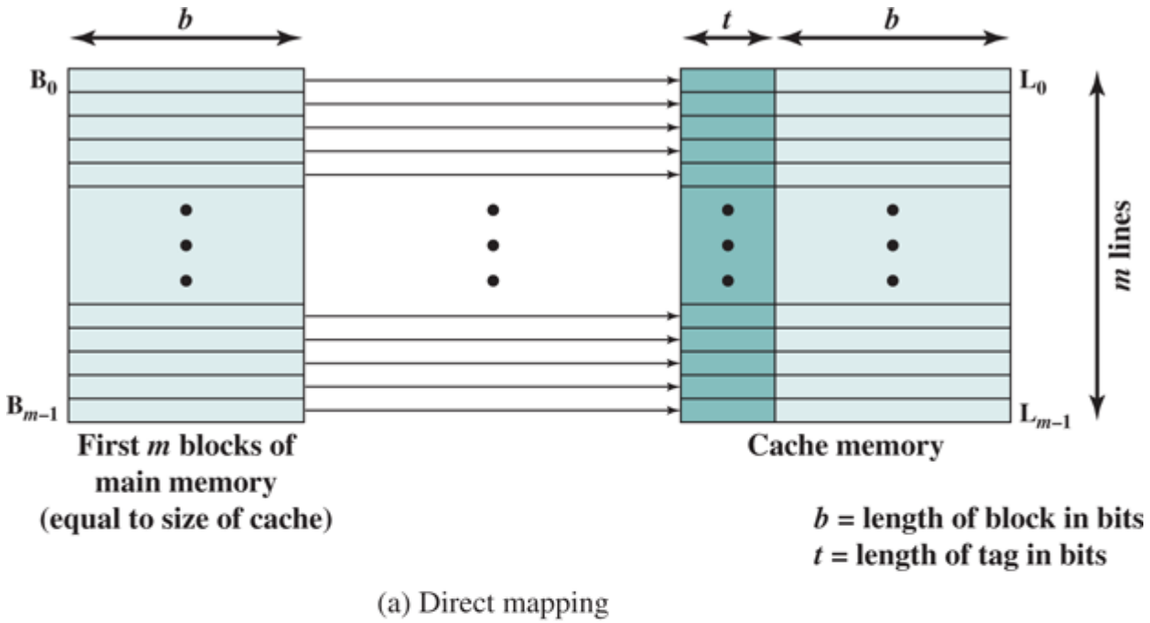
The simplest technique, known as direct mapping, maps each block of main memory into only one possible cache line. The mapping is expressed as

$$i = j \text{ modulo } m$$

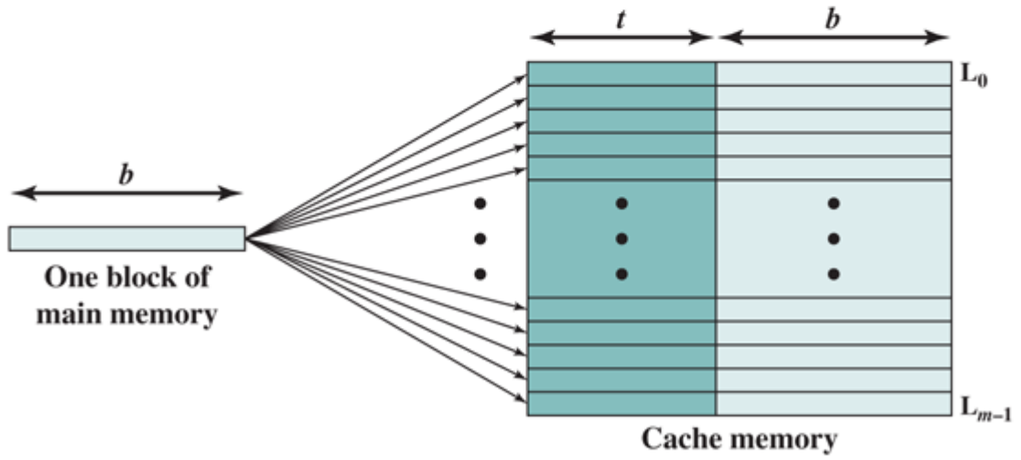
where

i = cache line number
j = main memory block number
m = number of lines in the cache

Figure 5.6a shows the mapping for the first *m* blocks of main memory. Each block of main memory maps into one unique line of the cache. The next *m* blocks of main memory map into the cache in the same fashion; that is, block B_m of main memory maps into line L_0 of cache, block B_{m+1} maps into line L_1, and so on.



(a) Direct mapping



(b) Associative mapping

Figure 5.6 Mapping from Main Memory to Cache: Direct and Associative

The mapping function is easily implemented using the main memory address. **Figure 5.7** illustrates the general mechanism. For purposes of cache access, each main memory address can be viewed as consisting of three fields. The least significant w bits identify a unique word or byte within a block of main memory; in most contemporary machines, the address is at the byte level. The remaining *s* bits specify one of the 2^s blocks of main memory. The cache logic interprets these *s* bits as a tag of $s - r$ bits (most significant portion) and a line field of *r* bits. This latter field identifies one of the $m = 2^r$ lines of the cache. To summarize,

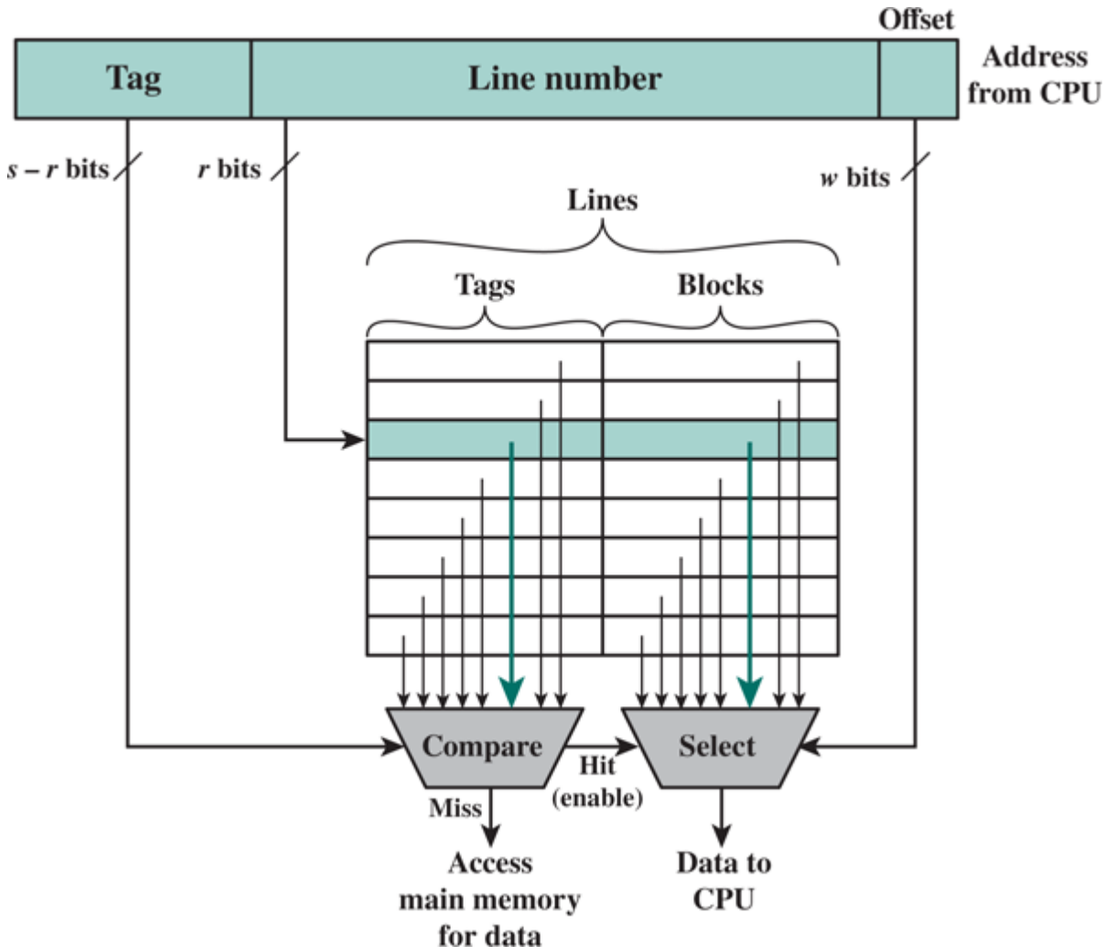


Figure 5.7 Direct-Mapping Cache Organization

- Address length = $(s + w)$ bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^{w} words or bytes
- Number of blocks in main memory = $\frac{2^{s+w}}{2^{w}} = 2^{s}$
- Number of lines in cache = $m = 2^{r}$
- Size of cache = 2^{r+w} words or bytes
- Size of tag = $(s - r)$ bits

The effect of this mapping is that blocks of main memory are assigned to lines of the cache as follows:

Cache line	Main memory blocks assigned
0	$0, m, 2m, \ldots, 2^{s} - m$
1	$1, m + 1, 2m + 1, \ldots, 2^{s} - m + 1$
⋮	⋮
$((m - 1))$	$m - 1, 2m - 1, 3m - 1, \ldots, 2^{s} - 1$

Thus, the use of a portion of the address as a line number provides a unique mapping of each block of main memory into the cache. When a block is actually read into its assigned line, it is necessary to tag the data to distinguish it from other blocks that can fit into that line. The most significant $s - r$ bits serve this purpose.

Figure 5.7 indicates the logical structure of the cache hardware access mechanism. When the cache hardware is presented with an address from the processor, the Line Number portion of the address is used to index into the cache. A compare function compares the tag of that line with the Tag field of the address. If there is a match (hit), an enable signal is sent to a select function, which uses the Offset field of the address and the Line Number field of the address to read the desired word or byte from the cache. If there is no match (miss) then the select function is not enabled and data is accessed from main memory, or the next level of cache. **Figure 5.7** illustrates the case in which the line number refers to the third line in the cache and there is a match, as indicated by the heavier arrowed lines.

Example 5.1a

Figure 5.8 shows our example system using direct mapping.[5] In the example, $m = 16K = 2^{14}$ and $i = j \text{ modulo } 2^{14}$. The mapping becomes

[5] In this and subsequent figures, memory values are represented in hexadecimal notation. See **Chapter 9** for a basic refresher on number systems (decimal, binary, hexadecimal).

Cache Line	Starting Memory Address of Block
0	000000, 010000, …, FF0000
1	000004, 010004, …, FF0004
⋮	⋮
$2^{14} - 1$	00FFFC, 01FFFC, …, FFFFFC

Note that no two blocks that map into the same line number have the same tag number. Thus, blocks with starting addresses 000000, 010000, …, FF0000 have tag numbers 00, 01, …, FF, respectively.

Referring back to **Figure 5.3**, a read operation works as follows. The cache system is presented with a 24-bit address. The 14-bit line number is used as an index into the cache to access a particular line. If the 8-bit tag number matches the tag number currently stored in that line, then the 2-bit word number is used to select one of the 4 bytes in that line. Otherwise, the 22-bit tag-plus-line field is used to fetch a block from main memory. The actual address that is used for the fetch is the 22-bit tag-plus-line concatenated with two 0 bits, so that 4 bytes are fetched starting on a block boundary.

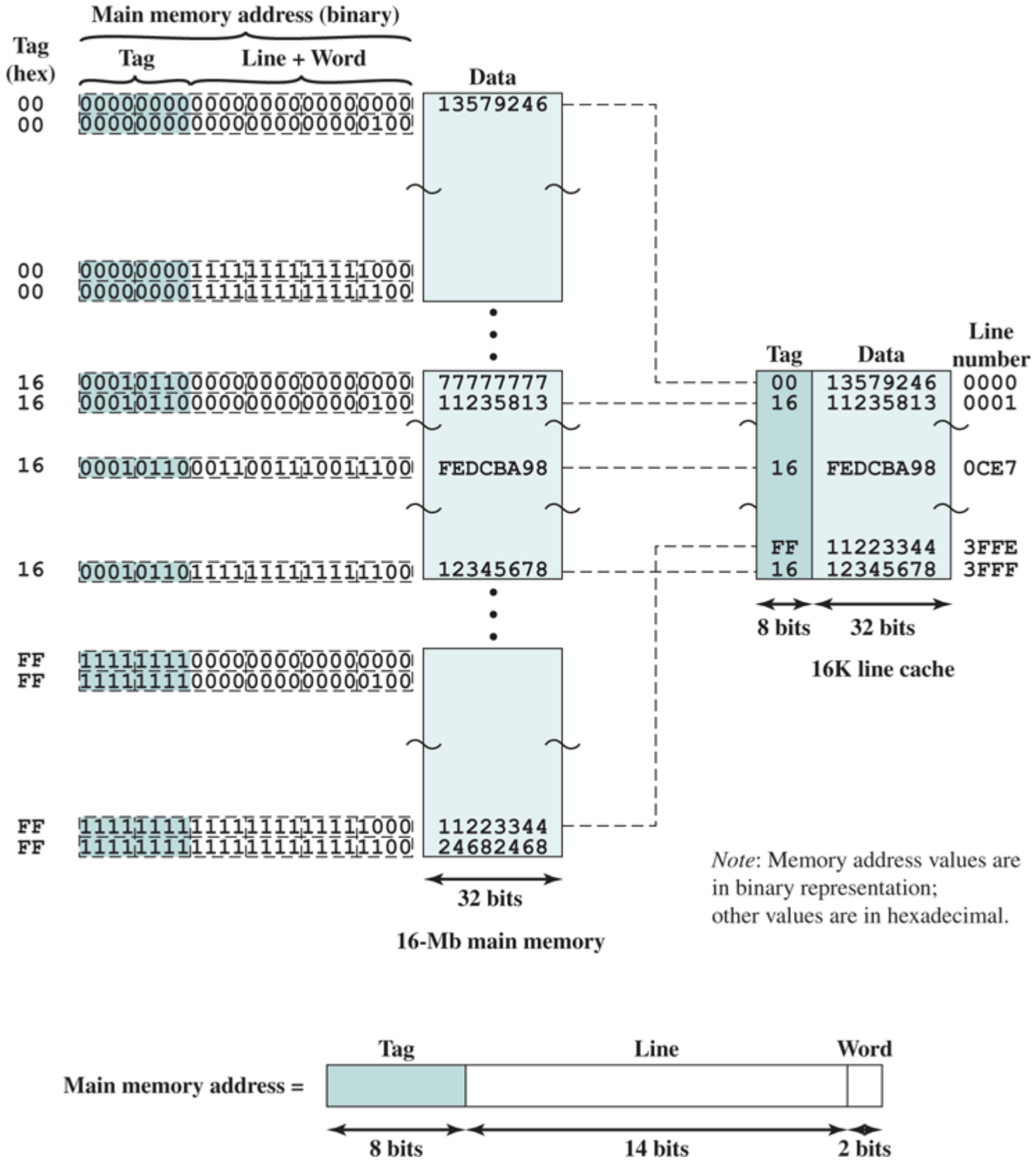


Figure 5.8 Direct Mapping Example

The direct mapping technique is simple and inexpensive to implement. Its main disadvantage is that there is a fixed cache location for any given block. Thus, if a program happens to reference words repeatedly from two different blocks that map into the same line, then the blocks will be continually swapped in the cache, and the hit ratio will be low (a phenomenon known as *thrashing*).

Aleksandr Lukin/123RF

Selective Victim Cache Simulator

One approach to lower the **miss** penalty is to remember what was discarded in case it is needed again. Since the discarded data has already been fetched, it can be used again at a small cost. Such recycling is possible using a **victim cache**. Victim cache was originally proposed as an approach to reduce the conflict misses of direct mapped caches without affecting its fast access time. Victim cache is a fully associative cache, whose size is typically 4 to 16 cache lines, residing between a direct mapped L1 cache and the next level of memory. This concept is explored in **Appendix B**.

CONTENT-ADDRESSABLE MEMORY

Before discussing associative cache organization, we need to introduce the concept of content-addressable memory (CAM), also known as associative storage [PAGI06]. Content-addressable memory (CAM) is constructed of static RAM (SRAM) cells (see static RAM) but is considerably more expensive and holds much less data than regular SRAM chips. Put another way, a CAM with the same data capacity as a regular SRAM is about 60% larger [SHAR03].

A CAM is designed such that when a bit string is supplied, the CAM searches its entire memory in parallel for a match. If the content is found, the CAM returns the address where the match is found and, in some architectures, also returns the associated data word. This process takes only one clock cycle.

Figure 5.9a is a simplified illustration of the search function of a small CAM with four horizontal words, each word containing five bits, or cells. CAM cells contain both storage and comparison circuitry. The is a match line corresponding to each word, feeding into match line sense amplifiers, and there is a differential search line pair corresponding to each bit of the search word. The encoder maps the match line of the matching location to its encoded address.

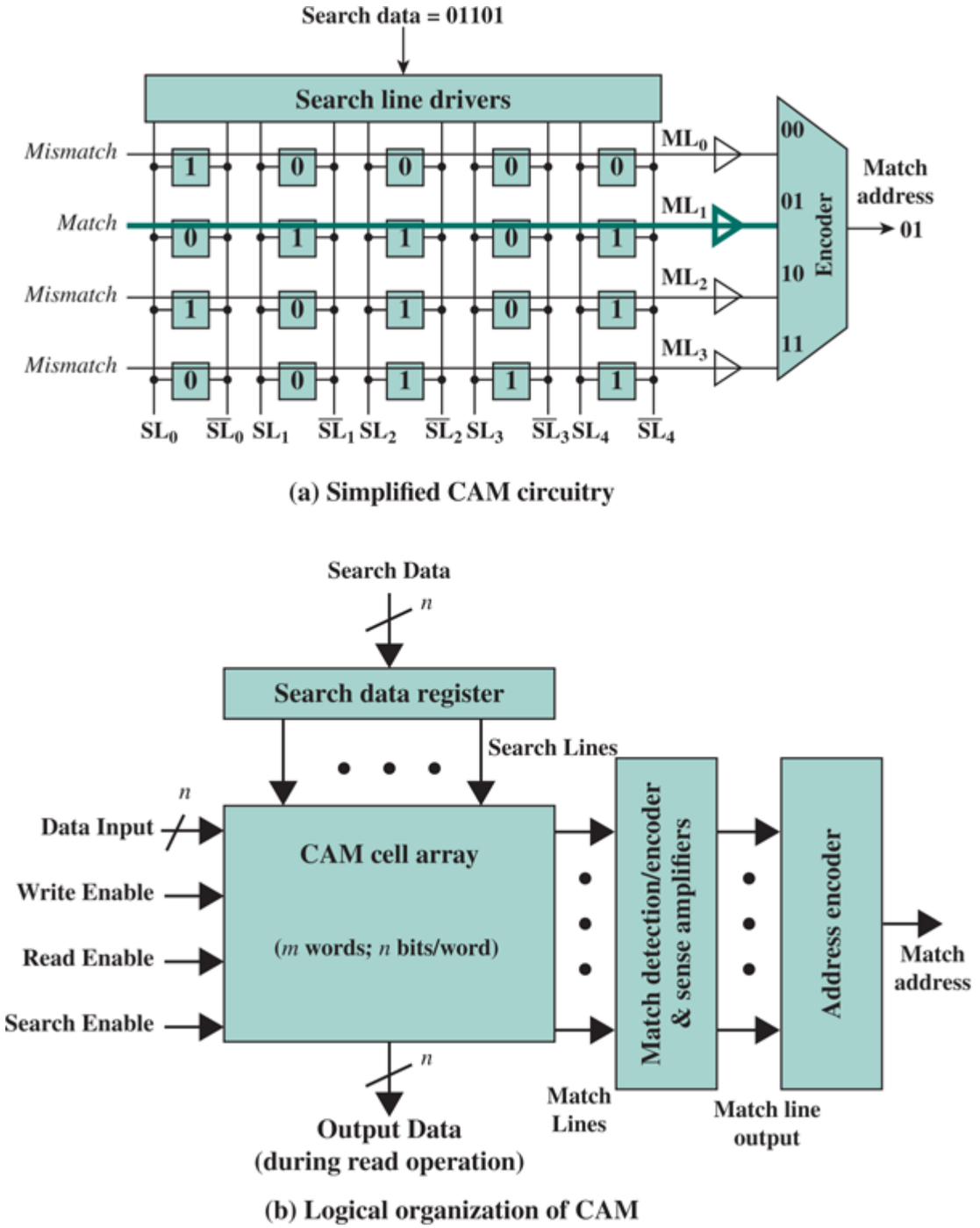


Figure 5.9 Content-Addressable Memory

Figure 5.9b shows a logical block diagram of a CAM cell array, consisting of *m* words of *n* bits each. Search, read, and write enable pins are used to enable one of the three operating modes of the CAM. For a search operation, the data to be searched is loaded in an *n*-bit search register that sets/resets the logic states of the search lines. The logic within and between cells of a row is such that a match lines is asserted if and only if all the cells in a row match the search line values. A simple read operation, as opposed to a search, is performed to read the data stored in the storage nodes of CAM cells using Read Enable control signal. The data words to be stored in CAM cell array are provided during a write operation through data input port.

ASSOCIATIVE MAPPING

Associative mapping overcomes the disadvantage of direct mapping by permitting each main memory block to be loaded into any line of the cache (**Figure 5.6b**). In this case, the cache control logic interprets a memory address simply as a Tag and a Word field. The Tag field uniquely identifies a block of main memory. To determine whether a block is in the cache, the cache control logic must simultaneously examine every line's tag for a match. **Figure 5.10** illustrates the logic.

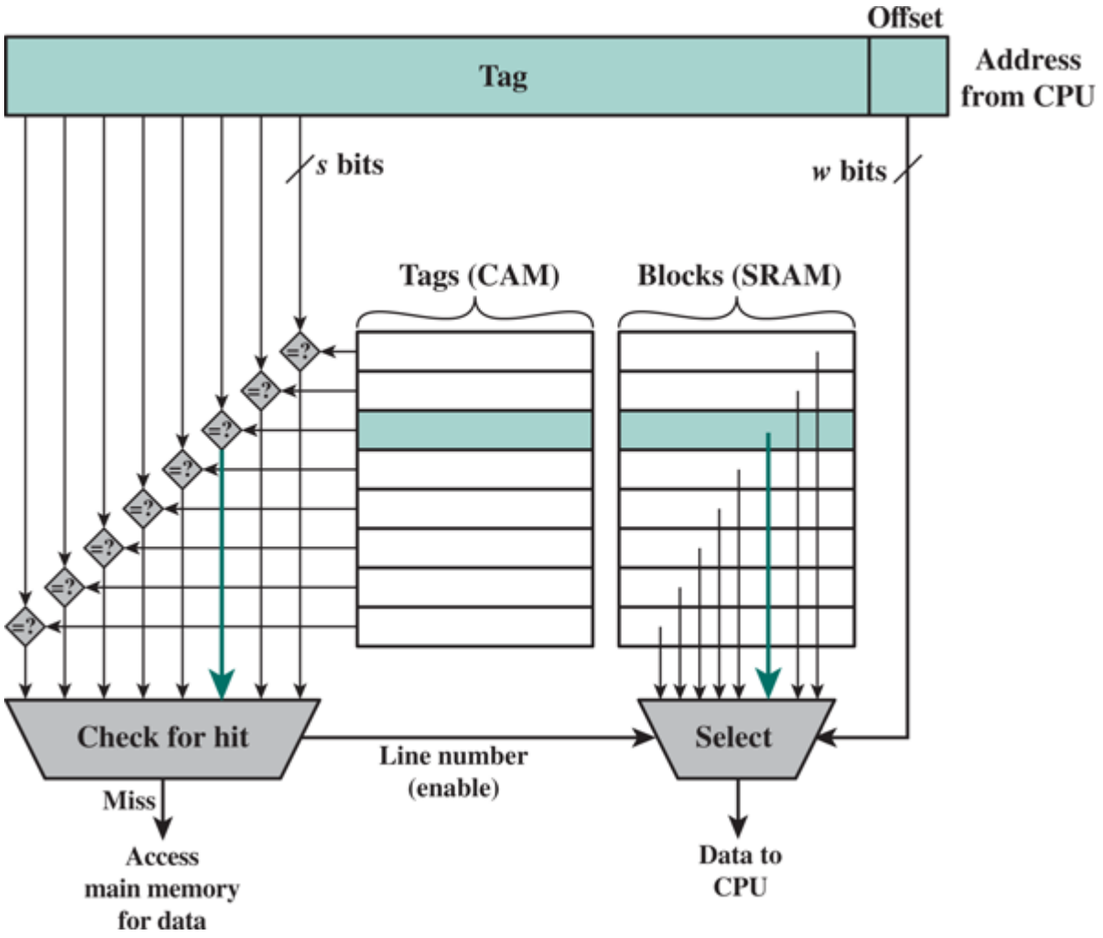


Figure 5.10 Fully Associative Cache Organization

Note that no field in the address corresponds to the line number, so that the number of lines in the cache is not determined by the address format. Instead, if there is a hit, the line number of the hit is sent to the select function by the cache hardware, as shown in **Figure 5.9**. To summarize,

- Address length = $(s + w)$ bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^{w} words or bytes
- Number of blocks in main memory = $\frac{2^{s+w}}{2^{w}} = 2^{s}$
- Number of lines in cache = undetermined
- Size of tag = s bits

Example 5.1b

Figure 5.11 shows our example using associative mapping. A main memory address consists of a 22-bit tag and a 2-bit byte number. The 22-bit tag must be stored with the 32-bit block of data for

each line in the cache. Note that it is the leftmost (most significant) 22 bits of the address that form the tag. Thus, the 24-bit hexadecimal address 16339C has the 22-bit tag 058CE7. This is easily seen in binary notation:

Memory address	0001	0110	0011	0011	1001	1100	(binary)
	1	6	3	3	9	C	(hex)
Tag (leftmost 22 bits)	00	0101	1000	1100	1110	0111	(binary)
	0	5	8	C	E	7	(hex)

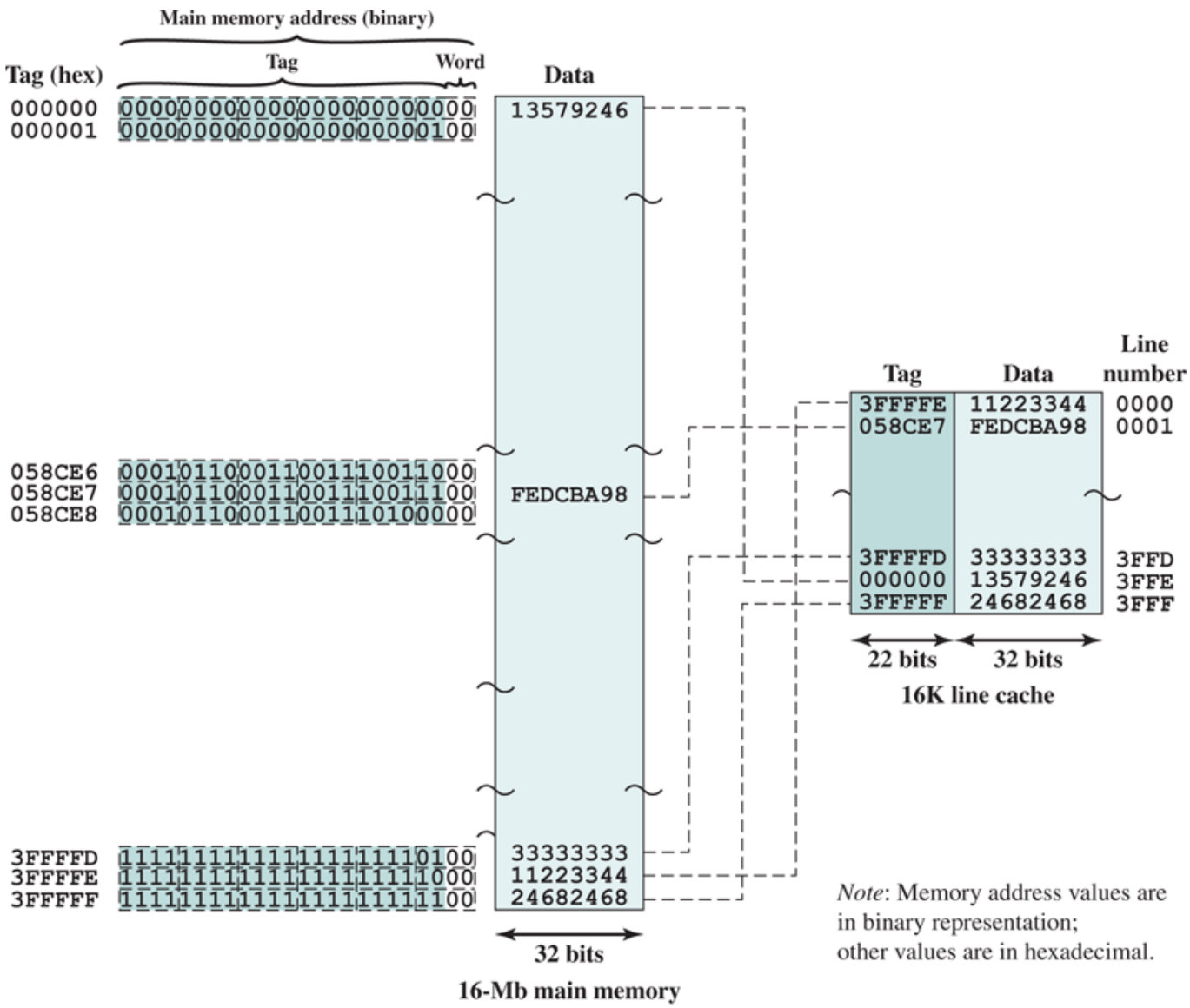


Figure 5.11 Associative Mapping Example

With associative mapping, there is flexibility as to which block to replace when a new block is read into the cache. Replacement algorithms, discussed later in this section, are designed to maximize the hit ratio. The principal disadvantage of associative mapping is the complex circuitry required to examine the tags of all cache lines in parallel.

Aleksandr Lukin/123RF

Cache Time Analysis Simulator

SET-ASSOCIATIVE MAPPING

Set-associative mapping is a compromise that exhibits the strengths of both the direct and associative approaches while reducing their disadvantages.

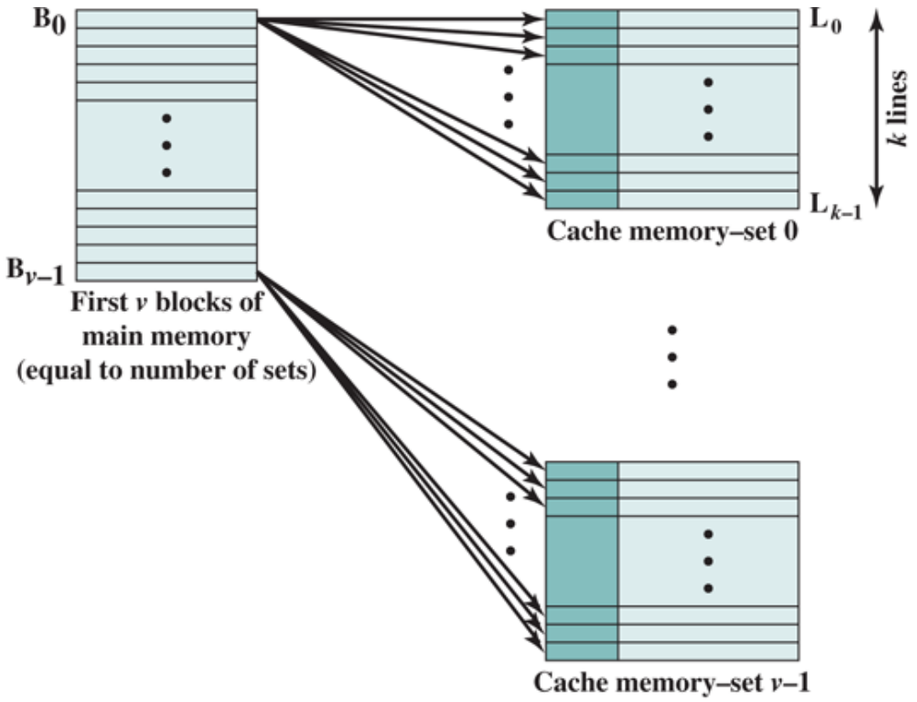
In this case, the cache consists of number sets, each of which consists of a number of lines. The relationships are

$$\begin{aligned} m &= v \times k \\ i &= j \text{ modulo } v \end{aligned}$$

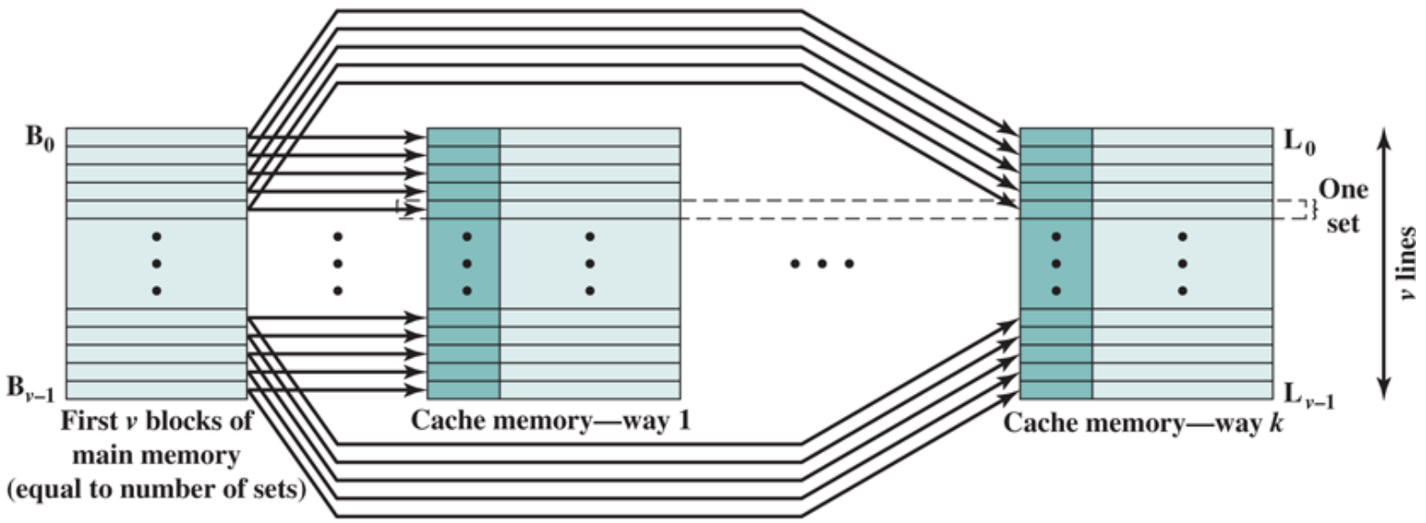
where

$$\begin{aligned} i &= \text{cache set number} \\ j &= \text{main memory block number} \\ m &= \text{number of lines in the cache} \\ v &= \text{number of sets} \\ k &= \text{number of lines in each set} \end{aligned}$$

This is referred to as *k*-way set-associative mapping. With set-associative mapping, block B_j can be mapped into any of the lines of set *j*. **Figure 5.12a** illustrates this mapping for the first *v* blocks of main memory. As with associative mapping, each word maps into multiple cache lines. For set-associative mapping, each word maps into all the cache lines in a specific set, so that main memory block B_0 maps into set 0, and so on. Thus, the set-associative cache can be physically implemented as *v* associative caches, typically implemented as *v* CAM memories. It is also possible to implement the set-associative cache as *k* direct mapping caches, as shown in **Figure 5.12b**. Each direct-mapped cache is referred to as a *way,* consisting of *v* lines. The first *v* lines of main memory are direct mapped into the *v* lines of each way; the next group of *v* lines of main memory are similarly mapped, and so on. The direct-mapped implementation is typically used for small degrees of associativity (small values of *k*) while the associative-mapped implementation is typically used for higher degrees of associativity [JACO08].



(a) *v* associative–mapped caches



(b) *k* direct–mapped caches

Figure 5.12 Mapping from Main Memory to Cache: *k*-Way Set Associative

For set-associative mapping, the cache control logic interprets a memory address as three fields: Tag, Set, and Word. The *d* set bits specify one of $v = 2^d$ sets. The *s* bits of the Tag and Set fields specify one of the 2^s blocks of main memory. **Figure 5.13** illustrates the cache control logic. With fully associative mapping, the tag in a memory address is quite large and must be compared to the tag of every line in the cache. With *k*-way set-associative mapping, the tag in a memory address is much smaller and is only compared to the *k* tags within a single set. As shown in **Figure 5.12**, if there is match of tags on any of the lines in the set, the corresponding select function is enabled and retrieves the desired work. If all comparisons report a miss, then the desired word is retrieved from main memory.

To summarize,

- Address length $= (s + w)$ bits
- Number of addressable units $= 2^{s+w}$ words or bytes
- Block size $=$ line size $= 2^{w}$ words or bytes
- Number of blocks in main memory $= \frac{2^{s+w}}{2^{w}} = 2^{s}$
- Number of lines in set $= k$
- Number of sets $= v = 2^{d}$
- Number of lines in cache $= m = kv = k \times 2^{d}$
- Size of cache $= k \times 2^{d+w}$ words or bytes
- Size of tag $= (s - d)$ bits

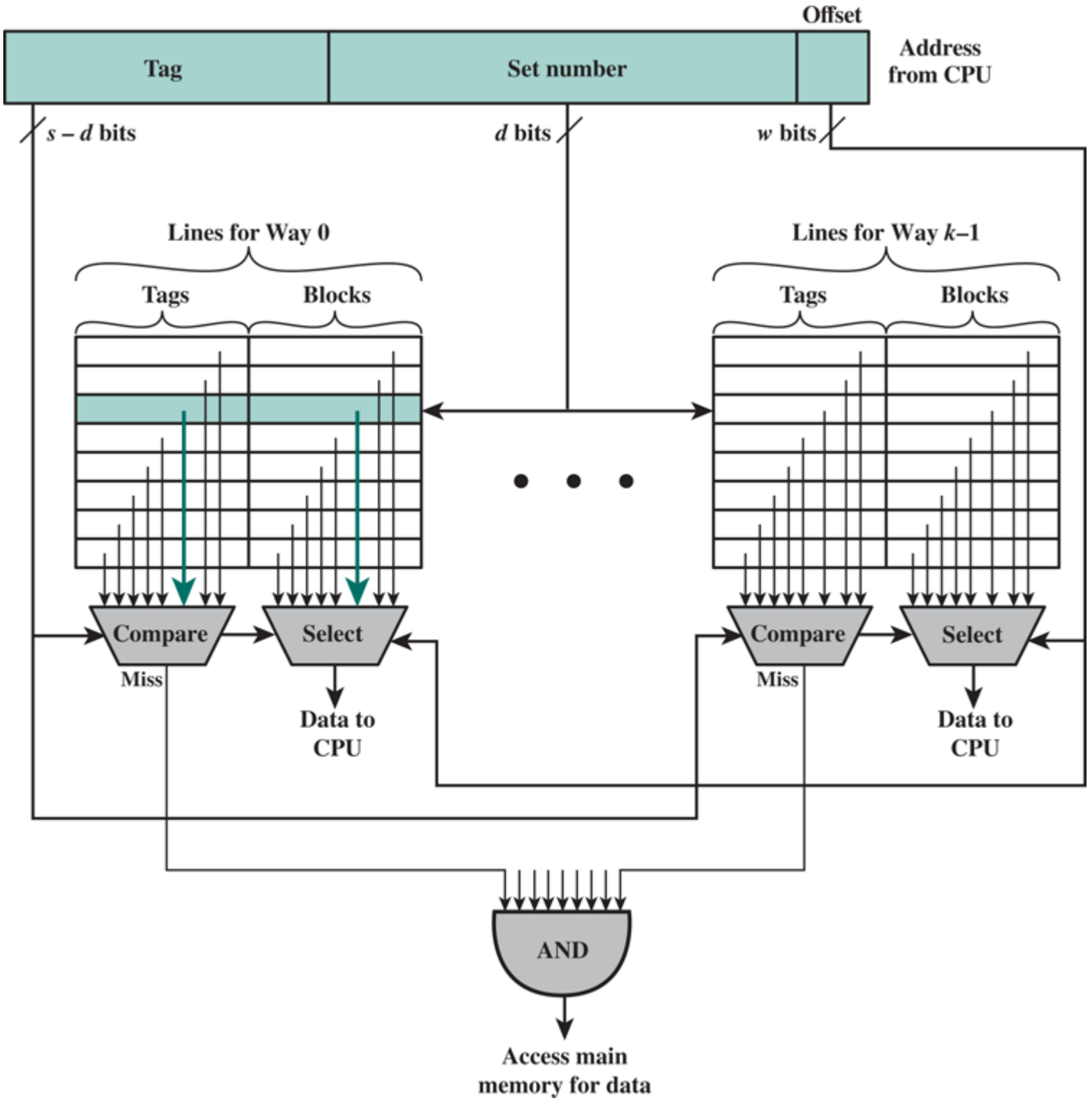


Figure 5.13 *k*-Way Set Associative Cache Organization

Example 5.1c

Figure 5.14 shows our example using two-way set-associative mapping with two lines in each set. The 13-bit set number identifies a unique set of two lines within the cache. It also gives the number of the block in main memory, modulo 2^{13}. This determines the mapping of blocks into lines. Thus, blocks 000000, 008000, …, FF8000 of main memory map into **cache set** 0. Any of those blocks can be loaded into either of the two lines in the set. Note that no two blocks that map into the same cache set have the same tag number. For a read operation, the 13-bit set number is used to determine which set of two lines is to be examined. Both lines in the set are examined for a match with the tag number of the address to be accessed.

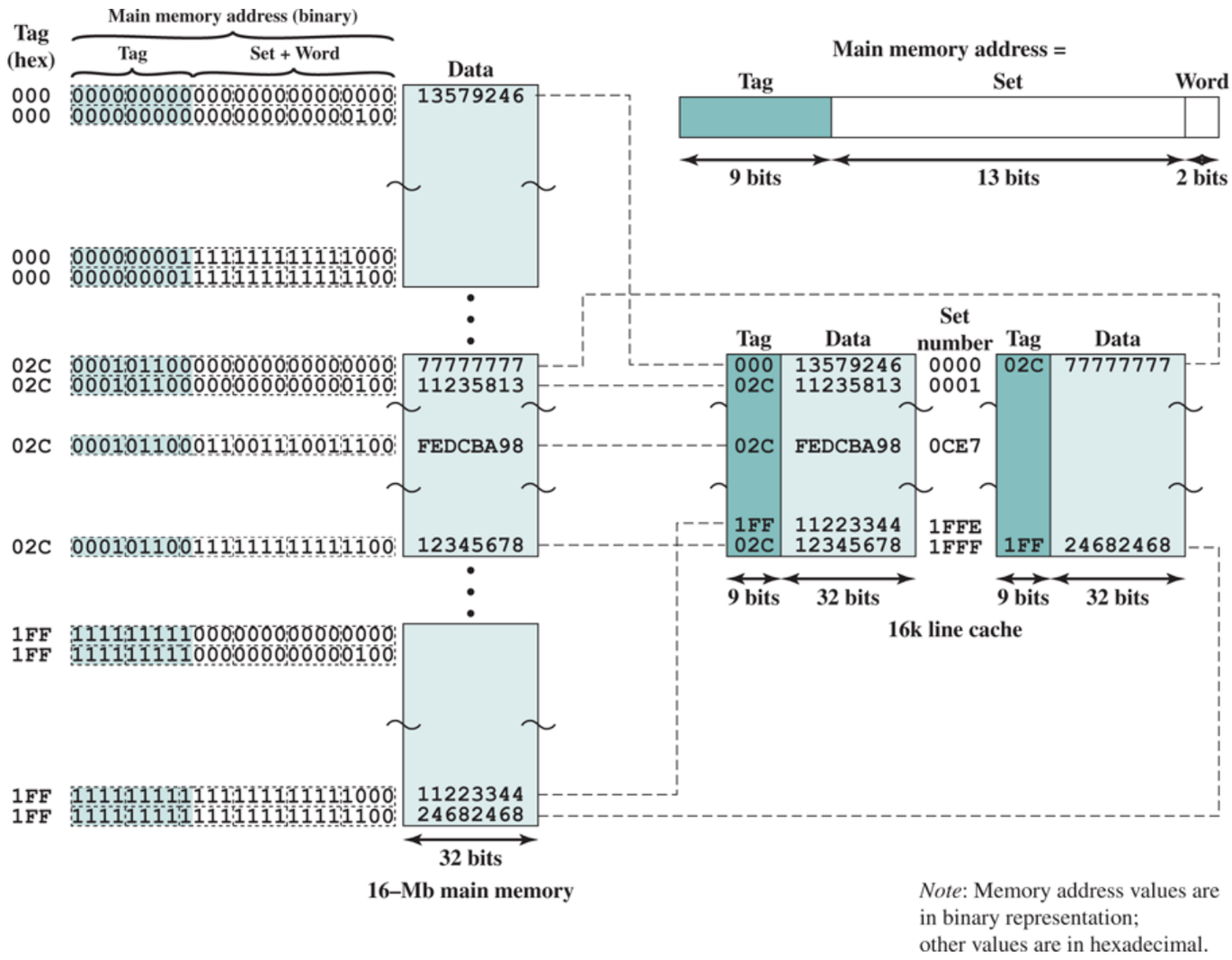


Figure 5.14 Two-Way Set-Associative Mapping Example

In the extreme case of $v = m$, $k = 1$, the set-associative technique reduces to direct mapping, and for $v = 1$, $k = m$, it reduces to associative mapping. The use of two lines per set $(v = m/2, k = 2)$ is the most common set-associative organization. It significantly improves the hit ratio over direct mapping. Four-way set associative $(v = m/4, k = 4)$ makes a modest additional improvement for a relatively small additional cost [MAYB84, HILL89]. Further increases in the number of lines per set have little effect.

Figure 5.15 shows the results of one simulation study of set-associative cache performance as a function of cache size [GENU04]. The difference in performance between direct and two-way set associative is significant up to at least a cache size of 64 kB. Note also that the difference between two-way and four-way at 4 kB is much less than the difference in going from for 4 kB to 8 kB in cache size. The complexity of the cache increases in proportion to the associativity, and in this case would not be justifiable against increasing cache size to 8 or even 16 kB. A final point to note is that beyond about 32 kB, increase in cache size brings no significant increase in performance.

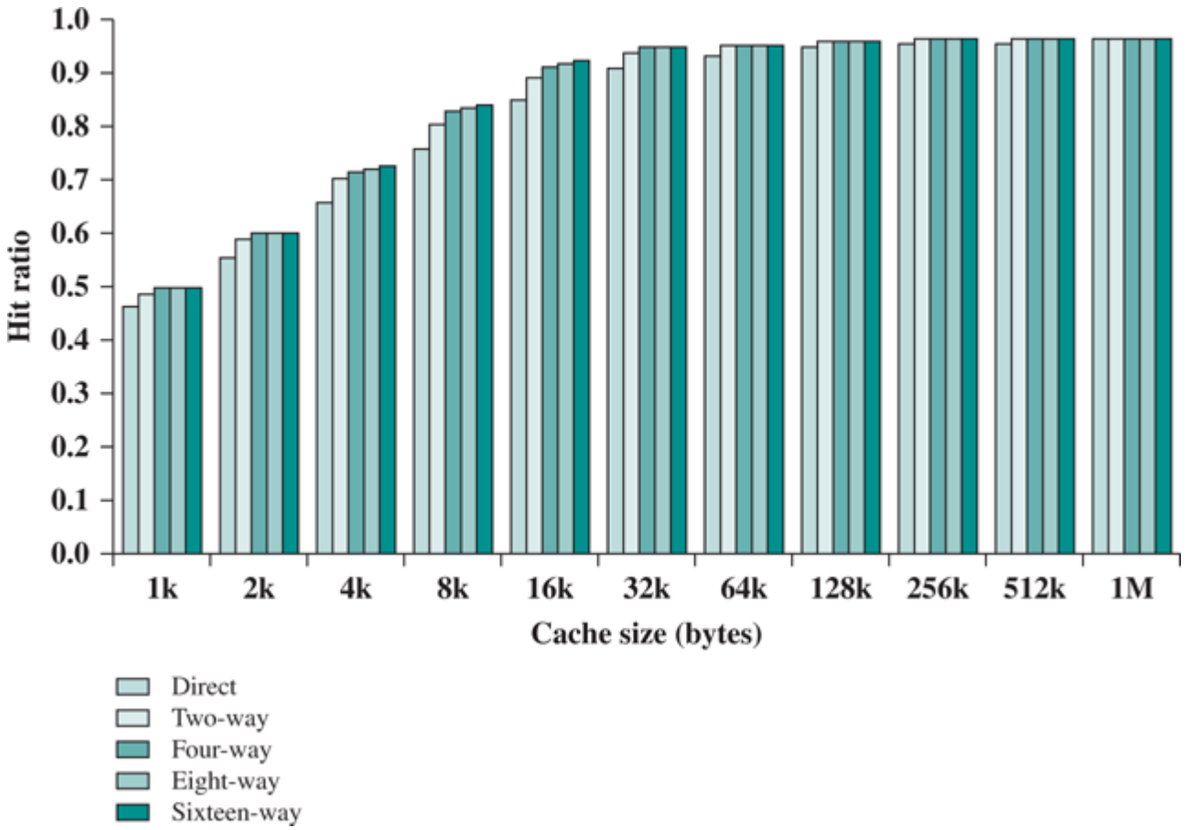


Figure 5.15 Varying Associativity over Cache Size

The results of **Figure 5.15** are based on simulating the execution of a GCC compiler. Different applications may yield different results. For example, [CANT01] reports on the results for cache performance using many of the CPU2000 SPEC benchmarks. The results of [CANT01] in comparing hit ratio to cache size follow the same pattern as **Figure 5.15**, but the specific values are somewhat different.

For both associative and set-associative caches, there is an additional time element for comparing tag fields. One way to reduce this time penalty in a set-associative cache is with way prediction. Way prediction allows the data array and tag array to be accessed in parallel. If the predicted way was correct (determined by a tag match), no penalty occurs. If the prediction was incorrect, additional cycles are needed to find the data. The way predictor is a table that guesses which “way” a given address should access, based on recent history. An implementation reported in [POWE01], using a number of SPEC CPU benchmark programs, found prediction rates that ranged from 50% to over 90% for a 4-way set associative cache. Another study [TSEN09] on a 4-way set associative cache using SPEC CPU programs resulted in prediction rates ranging from 85% to 95%.

Aleksandr Lukin/123RF

Cache Simulator

Multitask Cache Simulator

Replacement Algorithms

Once the cache has been filled, when a new block is brought into the cache, one of the existing blocks must be replaced. For direct mapping, there is only one possible line for any particular block, and no choice is possible. For the associative and set-associative techniques, a replacement algorithm is needed. To achieve high speed, such an algorithm must be implemented in hardware. A number of algorithms have been tried. We mention four of the most common. Probably the most effective is **least recently used (LRU):** Replace that block in the set that has been in the cache longest with no reference to it. For two-way set associative, this is easily implemented. Each line includes a USE bit. When a line is referenced, its USE bit is set to 1 and the USE bit of the other line in that set is set to 0. When a block is to be read into the set, the line whose USE bit is 0 is used. Because we are assuming that more recently used memory locations are more likely to be referenced, LRU should give the best hit ratio. LRU is also relatively easy to implement for a fully associative cache. The cache mechanism maintains a separate list of indexes to all the lines in the cache. When a line is referenced, it moves to the front of the list. For replacement, the line at the back of the list is used. Because of its simplicity of implementation, LRU is the most popular replacement algorithm.

Another possibility is first-in-first-out (FIFO): Replace that block in the set that has been in the cache longest. FIFO is easily implemented as a round-robin or circular buffer technique. Still another possibility is least frequently used (LFU): Replace that block in the set that has experienced the fewest references. LFU could be implemented by associating a counter with each line. A technique not based on usage (i.e., not LRU, LFU, FIFO, or some variant) is to pick a line at random from among the candidate lines. Simulation studies have shown that random replacement provides only slightly inferior performance to an algorithm based on usage [SMIT82].

Write Policy

When a block that is resident in the cache is to be replaced, there are two cases to consider. If the old block in the cache has not been altered, then it may be overwritten with a new block without first writing out the old block. If at least one write operation has been performed on a word in that line of the cache, then main memory must be updated by writing the line of cache out to the block of memory before bringing in the new block. A variety of write policies, with performance and economic trade-offs, is possible. There are two problems to contend with. First, more than one device may have access to main memory. For example, an I/O module may be able to read-write directly to memory. If a word has been altered only in the cache, then the corresponding memory word is invalid. Further, if the I/O device has altered main memory, then the cache word is invalid. A more complex problem occurs when multiple processors are attached to the same bus and each processor has its own local cache. Then, if a word is altered in one cache, it could conceivably invalidate a word in other caches.

The simplest technique is called **write through**. Using this technique, all write operations are made to main memory as well as to the cache, ensuring that main memory is always valid. Any other processor–cache module can monitor traffic to main memory to maintain consistency within its own cache. The main disadvantage of this technique is that it generates substantial memory traffic and may create a bottleneck. An alternative technique, known as **write back**, minimizes memory writes. With write back, updates are made only in the cache. When an update occurs, a **dirty bit**, or **use bit**, associated with the line is set. Then, when a block is replaced, it is written back to main memory if and only if the dirty bit is set. The problem with write back is that portions of main memory are invalid, and hence accesses by I/O modules can be allowed only through the cache. This makes for complex circuitry and a potential bottleneck. Experience has shown that the percentage of memory references that are writes is on the order of 15% [SMIT82]. However, for HPC applications, this number may approach 33% (vector-vector multiplication) and can go as high as 50% (matrix transposition).

Example 5.2

Consider a cache with a line size of 32 bytes and a main memory that requires 30 ns to transfer a 4-byte word. For any line that is written at least once before being swapped out of the cache, what is the average number of times that the line must be written before being swapped out for a write-back cache to be more efficient than a write-through cache?

For the write-back case, each dirty line is written back once, at swap-out time, taking $8 \times 30 = 240\text{ns}$. For the write-through case, each update of the line requires that one word be written out to main memory, taking 30 ns. Therefore, if the average line that gets written at least once gets written more than 8 times before swap out, then write back is more efficient.

There is another dimension to the write policy when a miss occurs at a cache level. There are two alternatives in the event of a write miss:

- **Write Allocate:** The block containing the word to be written is fetched from main memory (or next level cache) into the cache and the processor proceeds with the write cycle.
- **No Write Allocate:** The block containing the word to be written is modified in the main memory and not loaded into the cache.

Either of these policies can be used with either write through or write back. Most commonly, no write allocate is used with write through. The reasoning is that even if locality holds and a write will be made to the same block in the near future, the write-through policy will generate a write to main memory anyway, so bringing the block into the cache does not seem efficient. For example, the ARM Cortex processors can be configured to use write allocate or no write allocate with write back, but only no write allocate with write through.

With write back, write allocate is most commonly used, although some systems, such as the ARM Cortex, also allow no write allocate. The reasoning for using write allocate is that subsequent writes to the same block, if the block originally caused a miss, will hit in the cache next time, setting the dirty bit for the block. That will eliminate extra memory accesses and result in efficient execution. The write back, no write allocate option eliminates the time spent in bringing a block into the cache. Depending on locality patterns for reads and writes, there may be some advantage to this technique.

In a bus organization in which more than one device (typically a processor) has a cache and main memory is shared, a new problem is introduced. If data in one cache are altered, this invalidates not only the corresponding word in main memory, but also that same word in other caches (if any other cache happens to have that same word). Even if a write-through policy is used, the other caches may contain invalid data. A system that prevents this problem is said to maintain cache coherency. Possible approaches to cache coherency include the following:

- **Bus watching with write through:** Each cache controller monitors the address lines to detect write operations to memory by other bus masters. If another master writes to a location in shared memory that also resides in the cache memory, the cache controller invalidates that cache entry. This strategy depends on the use of a write-through policy by all cache controllers.
- **Hardware transparency:** Additional hardware is used to ensure that all updates to main memory via cache are reflected in all caches. Thus, if one processor modifies a word in its cache, this update is written to main memory. In addition, any matching words in other caches are similarly updated.
- **Noncacheable memory:** Only a portion of main memory is shared by more than one processor, and this is designated as noncacheable. In such a system, all accesses to shared memory are cache misses, because the shared memory is never copied into the cache. The noncacheable memory can be identified using chip-select logic or high-address bits.

Cache coherency is an active field of research. This topic is explored further in Part Five.

Line Size

Another design element is the line size. When a block of data is retrieved and placed in the cache, not only the desired word but also some number of adjacent words are retrieved. As the block size increases from very small to larger sizes, the hit ratio will at first increase because of the principle of **locality** , which states that data in the vicinity of a referenced word are likely to be referenced in the near future. As the block size increases, more useful data are brought into the cache. The hit ratio will begin to decrease, however, as the block becomes even bigger and the probability of using the newly fetched information becomes less than the probability of reusing the information that has to be replaced. Two specific effects come into play:

- Larger blocks reduce the number of blocks that fit into a cache. Because each block fetch overwrites older cache contents, a small number of blocks results in data being overwritten shortly after they are fetched.
- As a block becomes larger, each additional word is farther from the requested word and therefore less likely to be needed in the near future.

The relationship between block size and hit ratio is complex, depending on the locality characteristics of a particular program, and no definitive optimum value has been found. A size of from 8 to 64 bytes seems reasonably close to optimum [SMIT87, PRZY88, PRZY90, HAND98]. For HPC systems, 64- and 128-byte cache line sizes are most frequently used.

Number of Caches

When caches were originally introduced, the typical system had a single cache. More recently, the use of multiple caches has become the norm. Two aspects of this design issue concern the number of levels of caches and the use of unified versus split caches.

MULTILEVEL CACHES

As logic density has increased, it has become possible to have a cache on the same chip as the processor: the on-chip cache. Compared with a cache reachable via an external bus, the on-chip cache reduces the processor’s external bus activity and therefore speeds up execution times and increases overall system performance. When the requested instruction or data is found in the on-chip cache, the bus access is eliminated. Because of the short data paths internal to the processor, compared with bus lengths, on-chip cache accesses will complete appreciably faster than would even zero-wait state bus cycles. Furthermore, during this period the bus is free to support other transfers.

The inclusion of an on-chip cache leaves open the question of whether an off-chip, or external, cache is still desirable. Typically, the answer is yes, and most contemporary designs include both on-chip and external caches. The simplest such organization is known as a two-level cache, with the internal level 1 (L1) and the external cache designated as level 2 (L2). The reason for including an L2 cache is the following: If there is no L2 cache and the processor makes an access request for a memory location not in the L1 cache, then the processor must access DRAM or ROM memory across the bus. Due to the typically slow bus speed and slow memory access time, this results in poor performance. On the other hand, if an L2 SRAM (static RAM) cache is used, then frequently the missing information can be quickly retrieved. If the SRAM is fast enough to match the bus speed, then the data can be accessed using a zero-wait state transaction, the fastest type of bus transfer.

Two features of contemporary cache design for multilevel caches are noteworthy. First, for an off-chip L2 cache, many designs do not use the system bus as the path for transfer between the L2 cache and the processor, but use a separate data path, so as to reduce the burden on the system bus. Second,

with the continued shrinkage of processor components, a number of processors now incorporate the L2 cache on the processor chip, improving performance.

The potential savings due to the use of an L2 cache depends on the hit rates in both the L1 and L2 caches. Several studies have shown that, in general, the use of a second-level cache does improve performance (e.g., see [AZIM92], [NOVI93], [HAND98]). However, the use of multilevel caches does complicate all of the design issues related to caches, including size, replacement algorithm, and write policy; see [HAND98] and [PEIR99] for discussions.

Figure 5.16 shows the results of one simulation study of two-level cache performance as a function of cache size [GENU04]. The figure assumes that both caches have the same line size and shows the total hit ratio. That is, a **hit** is counted if the desired data appears in either the L1 or the L2 cache. The figure shows the impact of L2 on total hits with respect to L1 size. L2 has little effect on the total number of cache hits until it is at least double the L1 cache size. Note that the steepest part of the slope for an L1 cache of 8 kB is for an L2 cache of 16 kB. Again for an L1 cache of 16 kB, the steepest part of the curve is for an L2 cache size of 32 kB. Prior to that point, the L2 cache has little, if any, impact on total cache performance. The need for the L2 cache to be larger than the L1 cache to affect performance makes sense. If the L2 cache has the same line size and capacity as the L1 cache, its contents will more or less mirror those of the L1 cache.

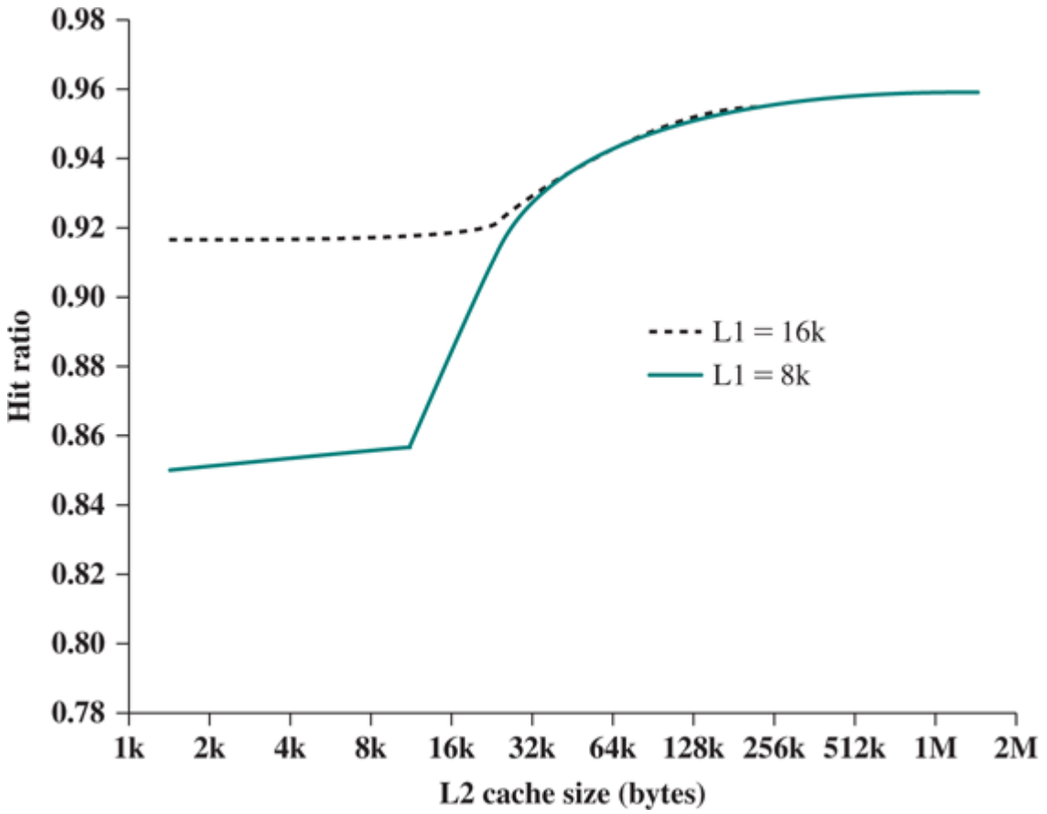


Figure 5.16 Total Hit Ratio (L1 and L2) for 8-kB and 16-kB L1

With the increasing availability of on-chip area available for cache, most contemporary microprocessors have moved the L2 cache onto the processor chip and added an L3 cache. Originally, the L3 cache was accessible over the external bus. More recently, most microprocessors have incorporated an on-chip L3 cache. In either case, there appears to be a performance advantage to adding the third level (e.g., see [GHAI98]). Further, large systems, such as the IBM mainframe zEnterprise systems, incorporate 3 on-chip cache levels and a fourth level of cache shared across multiple chips [BART15].

UNIFIED VERSUS SPLIT CACHES

When the on-chip cache first made an appearance, many of the designs consisted of a single cache used to store references to both data and instructions. More recently, it has become common to split the cache into two: one dedicated to instructions and one dedicated to data. These two caches both exist at the same level, typically as two L1 caches. When the processor attempts to fetch an instruction from main memory, it first consults the instruction L1 cache, and when the processor attempts to fetch data from main memory, it first consults the data L1 cache.

There are two potential advantages of a **unified cache**:

- For a given cache size, a unified cache has a higher hit rate than **split caches** because it balances the load between instruction and data fetches automatically. That is, if an execution pattern involves many more instruction fetches than data fetches, then the cache will tend to fill up with instructions, and if an execution pattern involves relatively more data fetches, the opposite will occur.
- Only one cache needs to be designed and implemented.

The trend is toward split caches at the L1 and unified caches for higher levels, particularly for superscalar machines, which emphasize parallel instruction execution and the prefetching of predicted future instructions. The key advantage of the split cache design is that it eliminates contention for the cache between the instruction fetch/decode unit and the execution unit. This is important in any design that relies on the pipelining of instructions. Typically, the processor will fetch instructions ahead of time and fill a buffer, or pipeline, with instructions to be executed. Suppose now that we have a unified instruction/**data cache**. When the execution unit performs a memory access to load and store data, the request is submitted to the unified cache. If, at the same time, the instruction prefetcher issues a read request to the cache for an instruction, that request will be temporarily blocked so that the cache can service the execution unit first, enabling it to complete the currently executing instruction. This cache contention can degrade performance by interfering with efficient use of the instruction pipeline. The split cache structure overcomes this difficulty.

Inclusion Policy

Recall from Chapter 4 that we defined the inclusion principle for memory hierarchies as follows: All information items are originally stored in level *Mn*, where *n* is the level most remote from the processor (lowest level). During the processing, subsets of *Mn* are copied into $Mn-1$. Similarity, subsets of $Mn-1$ are copied into $Mn-2$, and so on. This is expressed concisely as *Mi* $Mi+1$. Thus, if a word is found in *Mi*, then copies of the same word also exist in all lower layers $Mi+1, Mi+2, \ldots, Mn$. In a multilevel cache environment, in which there may be multiple caches at one level that share the same cache at the next lower level, inclusion between these two levels may not always be desirable. Three inclusion policies are found in contemporary cache systems:

The **inclusive policy** dictates that a piece of data in one cache is guaranteed to be also found in all lower levels of caches. The advantage of the inclusive policy is that it simplifies searching for data when there are multiple processors in the computing system. For example, if one processor wants to know whether another processor has the data it needs, it does not need to search all levels of caches of that other processor but only the lowest-level cache. This property is useful in enforcing cache coherence, which is discussed in **Chapter 20**.

The **exclusive policy** dictates that a piece of data in one cache is guaranteed *not* to be found in all lower levels of caches. The advantage of the exclusive policy is that it does not waste cache capacity since it does not store multiple copies of the same data in all of the caches. The disadvantage is the

need to search multiple cache levels when invalidating or updating a block. To minimize the search time, the higher-level tag sets are typically duplicated at the lowest cache level to centralize searching.

With the **noninclusive policy**, a piece of data in one cache may or may not be found in lower levels of caches. This can be contrasted with the other two policies with the following examples. Suppose that the L2 line size is a multiple of the L1 line size. For the inclusive policy, if a block is evicted from the L2 cache, the corresponding multiple blocks will be evicted from the L1 cache. In contrast, with a noninclusive policy, the L1 cache my retain portions of a block recently evicted from the L2 cache. For the same difference in block size, if a portion of a block is promoted from the L2 cache to the L1 cache, the exclusive policy requires the entire L2 block be evicted. In contrast, the noninclusive policy does not require this eviction. As with the exclusive policy, a noninclusive policy will generally maintain all higher-level cache sets at the lowest cache level.

5.3 Intel x86 Cache Organization

The evolution of cache organization is seen clearly in the evolution of Intel microprocessors (**Table 5.4**). The 80386 does not include an on-chip cache. The 80486 includes a single on-chip cache of 8 kB, using a line size of 16 bytes and a four-way set-associative organization. All of the Pentium processors include two on-chip L1 caches, one for data and one for instructions. For the Pentium 4, the L1 data cache is 16 kB, using a line size of 64 bytes and a four-way set-associative organization. The Pentium 4 **instruction cache** is described subsequently. The Pentium II also includes an L2 cache that feeds both of the L1 caches. The L2 cache is eight-way set associative with a size of 512 kB and a line size of 128 bytes. An L3 cache was added for the Pentium III and became on-chip with high-end versions of the Pentium 4.

Table 5.4 Intel Cache Evolution

Problem	Solution	Processor on Which Feature First Appears
External memory slower than the system bus.	Add external cache using faster memory technology.	386
Increased processor speed results in external bus becoming a bottleneck for cache access.	Move external cache on-chip, operating at the same speed as the processor.	486
Internal cache is rather small, due to limited space on chip.	Add external L2 cache using faster technology than main memory.	486
Contention occurs when both the Instruction Prefetcher and the Execution Unit simultaneously require access to the cache. In that case, the Prefetcher is stalled while the Execution Unit’s data access takes place.	Create separate data and instruction caches.	Pentium
Increased processor speed results in external bus becoming a bottleneck for L2 cache access.	Create separate back-side bus that runs at higher speed than the main (front-side) external bus. The BSB is dedicated to the L2 cache.	Pentium Pro
	Move L2 cache on to the processor chip.	Pentium II

Some applications deal with massive databases and must have rapid access to large amounts of data. The on-chip caches are too small.	Add external L3 cache.	Pentium III
	Move L3 cache on-chip.	Pentium 4

Figure 5.17 provides a simplified view of the Pentium 4 organization, highlighting the placement of the three caches. This cache architecture is similar to those of more modern x86 systems. The processor core consists of four major components:

- **Fetch/decode unit:** Fetches program instructions in order from the L2 cache, decodes these into a series of micro-operations, and stores the results in the L1 instruction cache.
- **Out-of-order execution logic:** Schedules execution of the micro-operations subject to data dependencies and resource availability; thus, micro-operations may be scheduled for execution in a different order than they were fetched from the instruction stream. As time permits, this unit schedules speculative execution of micro-operations that may be required in the future.
- **Execution units:** These units execute micro-operations, fetching the required data from the L1 data cache and temporarily storing results in registers.
- **Memory subsystem:** This unit includes the L2 and L3 caches and the system bus, which is used to access main memory when the L1 and L2 caches have a cache miss and to access the system I/O resources.

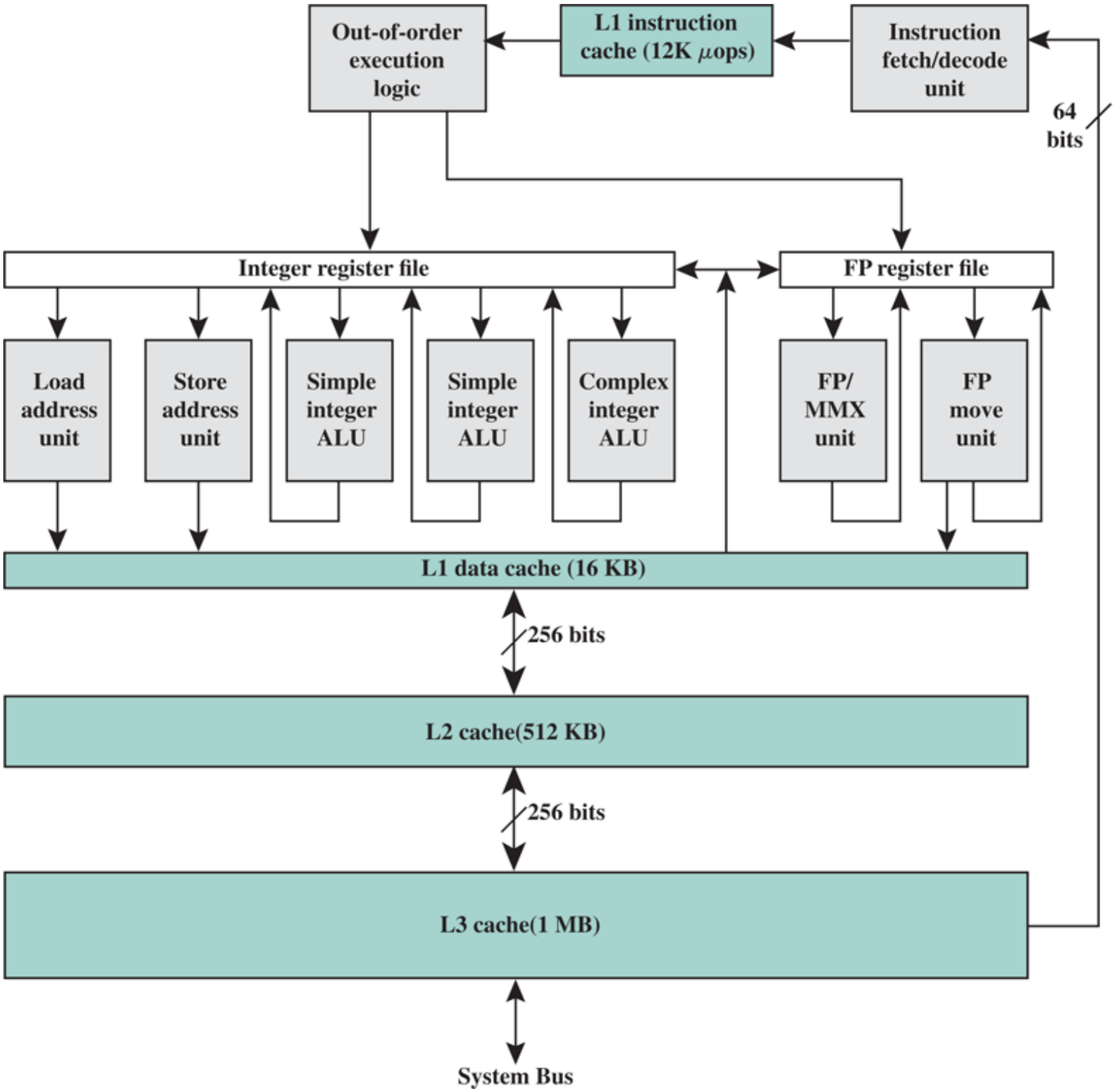


Figure 5.17 Pentium 4 Block Diagram

Unlike the organization used in all previous Pentium models, and in most other processors, the Pentium 4 instruction cache sits between the instruction decode logic and the execution core. The reasoning behind this design decision is as follows: As discussed more fully in Chapter 18, the Pentium processor decodes, or translates, Pentium machine instructions into simple RISC-like instructions called micro-operations. The use of simple, fixed-length micro-operations enables the use of superscalar pipelining and scheduling techniques that enhance performance. However, the Pentium machine instructions are cumbersome to decode; they have a variable number of bytes and many different options. It turns out that performance is enhanced if this decoding is done independently of the scheduling and pipelining logic. We return to this topic in **Chapter 18**.

The data cache employs a write-back policy: Data are written to main memory only when they are

removed from the cache and there has been an update. The Pentium 4 processor can be dynamically configured to support write-through caching.

The L1 data cache is controlled by two bits in one of the control registers, labeled the CD (cache disable) and NW (not write-through) bits (**Table 5.5**). There are also two Pentium 4 instructions that can be used to control the data cache: INVD invalidates (flushes) the internal cache memory and signals the external cache (if any) to invalidate. WBINVD writes back and invalidates internal cache and then writes back and invalidates external cache.

Table 5.5 Pentium 4 Cache Operating Modes
Note: $\mathrm{CD} = 0\,;\,\mathrm{NW} = 1$ is an invalid combination.

Control Bits		Operating Mode		
CD	NW	Cache Fills	Write Throughs	Invalidates
0	0	Enabled	Enabled	Enabled
1	0	Disabled	Enabled	Enabled
1	1	Disabled	Disabled	Disabled

Both the L2 and L3 caches are eight-way set-associative with a line size of 128 bytes.

5.4 The IBM z13 Cache Organization

The IBM z13 cache organization was introduced in **Chapter 5**. This section provides more detail. **Figure 5.18** illustrates the logical interconnections of the z13 cache system, showing the structure of a single processor drawer. A maximum system, called a central processing complex (CPC) consists of four drawers. Each drawer consists of two processor nodes, with each node containing 3 processor unit (PU) chips and one storage control (SC) chip. Each PU includes up to 8 cores. The L1, L2, and L3 caches are contained on each PU chip, and a separate SC chip holds the L4 cache for a processor node. Thus a maximum configuration contains 192 cores. Some key characteristics of each level are as follows:

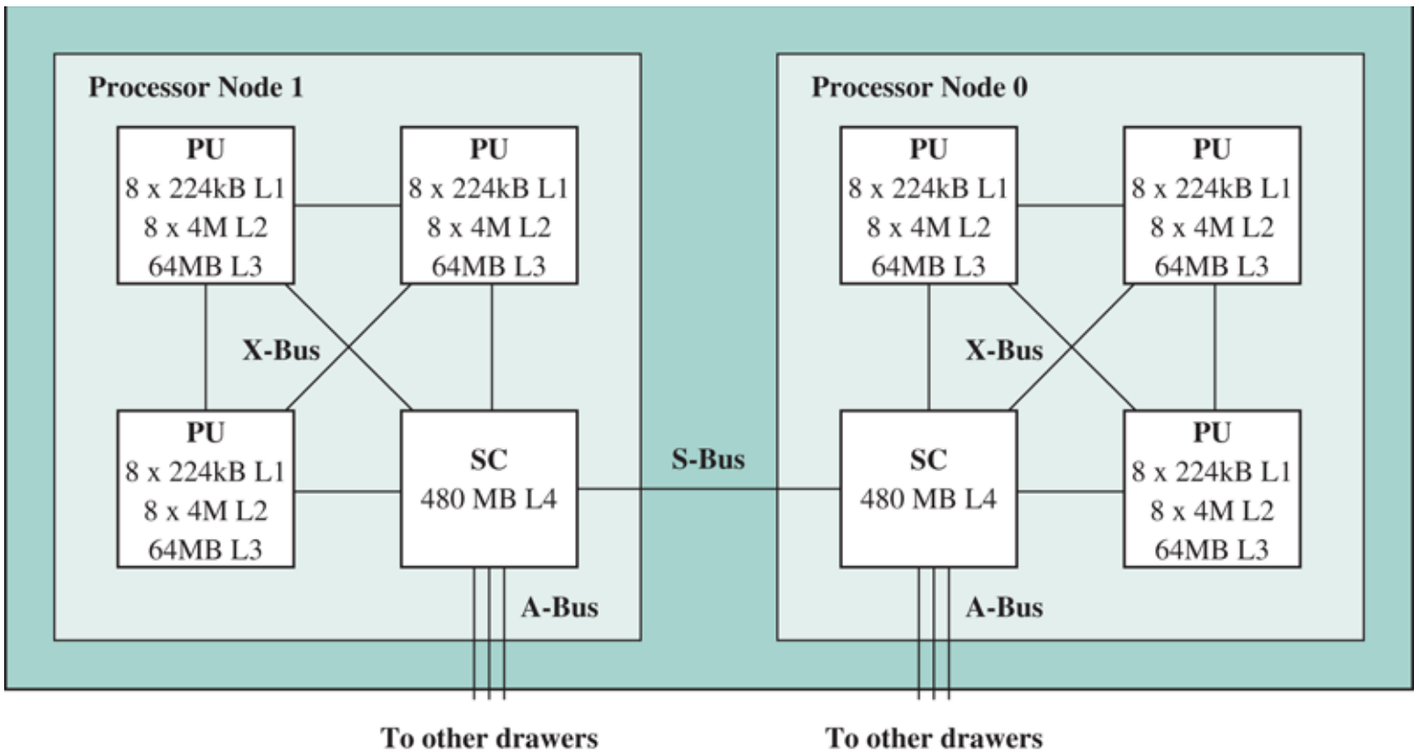


Figure 5.18 IBM z13 CPC Drawer Logical Structure

- **L1 cache:** Each core contains a 96-kB L1 I-cache and a 128-kB D-cache, for a maximum total of 18 MB of L1 I-cache and 24 MB of L1 D-cache. The L1 caches are designed as write-through caches.
- **L2 cache:** Each core contains a 2-MB L2 I-cache and a 2-MB L2 D-cache, for a maximum total of 384 MB of L2 I-cache and 384 MB of L2 D-cache. The L2 caches are designed as write-through caches.
- **L3 cache:** Each PU chip contains a 64-MB L3 cache, for a maximum total of 1.5 GB of L3 cache. The L3 cache is 16-way set associative and uses a line size of 256 bytes. The L3 cache uses the write-back policy.
- **L4 cache:** Each processor node contains a 480-MB L4 cache for a maximum total of 3.75 GB of L4 cache. The L4 cache is organized as a 30-way set-associative cache. The L4 cache uses the write-back policy.

The use of an L3 cache that is shared by 8 cores on a chip facilitates low-latency cross-processor cache line sharing and provides cache efficiency effects by elimination of redundant lines (single copy, multiple users), which is not possible with private caches. Thus, there are efficiency gains by devoting a substantial portion of each PU chip to a shared L3 cache as opposed to either increasing the size of

the L2 caches or providing private L3 caches for each core.

There are also efficiency benefits from providing an L4 cache chip on the same processor node, or motherboard, as the PU chips. The L4 cache enables smooth scaling from a single processor chip to a maximum system configuration by providing a significant buffer before main memory.

The interconnection design contributes to the overall efficiency of this arrangement. Within each PU chip, 160-GB/s bus bandwidth is used between L1/L2 and L2/L3 cache boundaries. The 80-GB/s XBus provides tightly coupled interconnection within a node at the L3/L4 cache boundary. The high-speed S-Bus connects via L4 between the nodes of a drawer, and A-Bus connections are provide to other drawers.

The cache write policy is tailored to the configuration. The L1 and L2 caches are write-through, taking advantage of the high-speed on-chip connection to the next cache level. Further, the L3 cache is most efficiently used if it always maintains the most recent version of any L2 cache line. Going from L3 to L4 is a lower speed, off-chip transmission and here a write-back policy is preferred to minimize traffic. Similarly, write-back is preferred going from L4 to main memory.

5.5 Cache Performance Models[6]

[6] Used with permission from Professor Roger Kieckhafer of Michigan Technological University.

This section looks first at the cache timing of the different cache access organizations, then at a model of design options for improving performance.

Cache Timing Model

We can derive some insight into the timing differences between the different cache access models by developing equations that show the different time delays. The following parameters are needed:

t_{ct} = time needed to compare the tag field of an address with the tag value in a cache line.

t_{rl} = time needed to read a line from the cache to retrieve the data block in the cache.

t_{xb} = time needed to transmit byte or word to the processor; this includes extracting the desired bytes from the fetched line and gating these bytes onto the bus to the processor.

t_{hit} = time expended at this cache level in the event of a hit.

t_{miss} = time expended at this cache level in the event of a miss.

First consider direct-mapped cache access. The first operation is checking the Tag field of an address against the tag value in the line designated by the Line field. If there is not a match (miss), then the operation is complete. If there is a match (hit), then the cache hardware reads the data block from the line in the cache and then fetches the byte or word indicated by the Offset field of the address. The timing equations therefore are:

$$t_{\text{hit}} = t_{\text{rl}} + t_{\text{xb}} + t_{\text{ct}} \quad t_{\text{miss}} = t_{\text{rl}} + t_{\text{ct}} \tag{5.1}$$

One of the advantages of a direct-mapped cache is that it allows simple and fast speculation. Once the address has been computed, the one cache line that might have a copy of that location in memory is known. That cache entry can be read, and the processor can continue to work with that data before it finishes checking that the tag actually matches the requested address. Thus checking and fetching are performed in parallel. Assuming the fetch time is larger, the timing equations become:

$$t_{\text{hit}} = t_{\text{rl}} + t_{\text{xb}} \quad t_{\text{miss}} = t_{\text{rl}} + t_{\text{ct}} \tag{5.2}$$

Next, consider a fully associative cache. In this case, the line number is not known until the tag comparison is completed. So the hit time is the same as for direct-mapped. Because this is a content-addressable memory, the miss time is simply the tag comparison time. That is, the tag comparison is made without the need to read a line of data from the cache, but is made in parallel to all of the lines of the cache internally to the cache. The equations in this case:

$$t_{\text{hit}} = t_{\text{rl}} + t_{\text{xb}} + t_{\text{ct}} \quad t_{\text{miss}} = t_{\text{ct}} \tag{5.3}$$

With set associative, it is not possible to transmit bytes and compare tags in parallel as can be done with direct-mapped with speculative access. However, the circuitry can be designed so that the data block from each line in a set can be loaded and then transmitted once the tag check is made. This yields the equation pair of **Equation 5.1**.

If set associative is augmented with way prediction, then the following equations hold:

$$t_{\text{hit}} = t_{\text{rl}} + t_{\text{xb}} + (1 - F_p)\ t_{\text{ct}}\ \ t_{\text{miss}} = t_{\text{rl}} + t_{\text{ct}} \tag{5.4}$$

where F_{p} is the fraction of time that the way prediction succeeds. Note that for $F_{\text{p}} = 1$, set associative with way prediction reduces to the same equations as direct-mapped with speculative access, which is the best case. For $F_{\text{p}} = 0$, the results are the same as without way prediction, which is the worst case. The prediction scheme typically used is to predict that the requested data is contained in the last block used from this set. If there is a high degree of spatial locality, then F_p will be close to 1.

Table 5.6 summarizes the cache timing equations.

Table 5.6 Cache Timing Equations

	Time for hit	Time for miss
Direct-Mapped	$t_{\text{hit}} = t_{\text{rl}} + t_{\text{xb}} + t_{\text{ct}}$	$t_{\text{miss}} = t_{\text{rl}} + t_{\text{ct}}$
Direct-Mapped with Speculation	$t_{\text{hit}} = t_{\text{rl}} + t_{\text{xb}}$	$t_{\text{miss}} = t_{\text{rl}} + t_{\text{ct}}$
Fully Associative	$t_{\text{hit}} = t_{\text{rl}} + t_{\text{xb}} + t_{\text{ct}}$	$t_{\text{miss}} = t_{\text{ct}}$
Set-Associative	$t_{\text{hit}} = t_{\text{rl}} + t_{\text{xb}} + t_{\text{ct}}$	$t_{\text{miss}} = t_{\text{rl}} + t_{\text{ct}}$
Set-Associative with Way Prediction	$t_{\text{hit}} = t_{\text{rl}} + t_{\text{xb}} + (1 - F_p)t_{\text{ct}}$	$t_{\text{miss}} = t_{\text{rl}} + t_{\text{ct}}$

Design Option for Improving Performance

Equation 4.5 expressed the mean time to access data in a memory hierarchy as follows:

$$T_s = \sum_{\text{all paths}} [\text{Probability of taking a path} \times \text{Duration of that a path}]$$

$$= \sum_{\text{all paths}} \prod (\text{All probabilities in the path}) \times \sum (\text{All times in that path})$$

$$= \sum_{i=1}^{n} \prod_{j=0}^{i-1} (1 - h_j)\, h_I \times t_I$$

where

n = Number of levels of memory.
t_i = Total time needed to access data in level M_i
= The sum of all times in the path to a hit in level M_i
h_i = Hit ratio of level M_i
= Conditional probability that the data for a memory access is resident in level M_i given that it is not resident in M_{i-1}
T_{S} = Mean time needed to access data

This can be rearranged to show the contribution of level M_1 explicitly:

$$\begin{aligned} T_s = & \\ & h_1 \times t_1 + (1 - h_1) \sum_{i=2}^{i} \prod_{j=2}^{1} (1 - h_j) h_1 \times t_i \\ = & \quad h_1 \times t_1 + (1 - h_1) t_{\text{penalty}} \end{aligned} \tag{5.3}$$

where t_{penalty} is the mean time to access data if there is a miss at level M_1. Note that t_1 is the same quantity as t_{hit} defined at the beginning of this section, because we are referring to level M_1.

Equation 5.3 provides insight into the approaches that can be taken to improve performance by showing three distinct parameters that can be altered. The value of T_s can be reduced by one of the following methods: reduce the hit time t_1, reduce the miss rate $(1 - h_1)$, and reduce the miss penalty t_{penalty}. The following is a list of widely used techniques that can be used to reduce one of these parameters (**Table 5.7**):

Table 5.7 Cache Performance Improvement Techniques

Technique	Reduce t_1	Reduce $(1 - h_1)$	Reduce t_{penalty}
Way Prediction			
Cache Capacity	Small	Large	
Line Size	Small	Large	
Degree of Associativity	Decrease	Increase	
More Flexible Replacement Policies			
Cache Unity	Split I-cache and D-cache	Unified cache	
Prefetching			
Write Through		Write allocate	No write allocate
Critical Word First			
Victim Cache			
Wider Busses			

- For a set-associative cache, the use of way prediction reduces t_{hit} (**Table 5.6**).
- The access time for a smaller, more compact cache is less than for a larger cache, reducing t_{hit}. On the other hand, in general, the larger the cache, the smaller the miss rate.
- Increasing the line size can decrease the miss rate because of spatial locality. However, a larger line size means that more time is spent bringing in a line on a miss. But the chance that the additional data brought into the cache by the larger line size will be used goes does down with the increased distance between addresses in the line. At some point, the amount of time spent fetching data that is not used into the cache becomes greater than the time saved through increasing the hit rate.
- The direct-mapped cache with speculation has the smallest value of t_{hit}, and the fully associative cache has the largest (**Table 5.6**). On the other hand, increasing the associativity of a cache can reduce its miss rate by reducing the number of conflict misses—misses that occur because more lines compete for a set in the cache than can fit in the set.
- If a cache is split between I-cache and D-cache, each cache is smaller, therefore reducing t_{hit}. But for the overall miss rate including instructions and data, a unified cache is likely to provide a reduced miss rate.
- The prefetching of blocks whose access is predicted for the near future can reduce t_{hit}.
- If write through is used with write allocate, the miss rate should be lower than write through, no write allocate. This is because the block that caused the cache miss is now in the cache and it is likely that future writes or perhaps reads will be to the same block. But, if no write allocate is used, then the time to complete the operation is less, reducing t_{penalty}.
- The use of the critical word first policy reduces the miss penalty by getting the request word to the processor as quickly as possible, not waiting for a cache line to be filled.
- As discussed in **Section 5.2**, a victim cache can be used to reduce the miss penalty.

A wider memory bus enables the transmission of more words in parallel between main memory and the cache, reducing the number of transfers required to load an entire cache block. This reduces the miss penalty time.

5.6 Key Terms, Review Questions, and Problems

Key Terms

associative mapping

cache block

cache hit

cache line

cache memory

cache miss

cache set

content-addressable memory

critical word first

data cache

direct mapping

dirty bit

frame

instruction cache

line

line size

logical cache

multilevel cache

no write allocate

physical cache

replacement algorithm

set-associative mapping

split cache

tag

unified cache

use bit

victim cache

virtual cache

write allocate

write back

write through

Review Questions

5.1 What are the differences among direct mapping, associative mapping, and set-associative mapping?
5.2 What is the difference between associative cache memory and content-addressable memory?
5.3 For a direct-mapped cache, a main memory address is viewed as consisting of three fields. List and define the three fields.
5.4 For an associative cache, a main memory address is viewed as consisting of two fields. List and define the two fields.
5.5 For a set-associative cache, a main memory address is viewed as consisting of three fields. List and define the three fields.
5.6 What is the distinction between spatial locality and temporal locality?
5.7 In general, what are the strategies for exploiting spatial locality and temporal locality?

Problems

5.1 A cache has a line size of 64 bytes. To determine which byte within a cache line an address points to, how many bits are in the Offset field?
5.2 A set-associative cache consists of 64 lines, or slots, divided into four-line sets. Main memory contains 4K blocks of 128 words each. Show the format of main memory addresses.
5.3 A two-way set-associative cache has lines of 16 bytes and a total size of 8 kB. The 64-MB main memory is byte addressable. Show the format of main memory addresses.
5.4 For the hexadecimal main memory addresses 111111, 666666, BBBBBB, show the following information, in hexadecimal format:

a. Tag, Line, and Word values for a direct-mapped cache, using the format of **Figure 5.7**
b. Tag and Word values for an associative cache, using the format of **Figure 5.10**
c. Tag, Set, and Word values for a two-way set-associative cache, using the format of **Figure 5.13**

5.5 List the following values:

a. For the direct cache example of **Figure 5.7** : address length, number of addressable units, block size, number of blocks in main memory, number of lines in cache, size of tag
b. For the associative cache example of **Figure 5.10** : address length, number of addressable units, block size, number of blocks in main memory, number of lines in cache, size of tag
c. For the two-way set-associative cache example of **Figure 5.13** : address length, number of addressable units, block size, number of blocks in main memory, number of lines in set, number of sets, number of lines in cache, size of tag

5.6 Consider a 32-bit microprocessor that has an on-chip 16-kB four-way set-associative cache. Assume that the cache has a line size of four 32-bit words. Draw a block diagram of this cache showing its organization and how the different address fields are used to determine a cache hit/miss. Where in the cache is the word from memory location ABCDE8F8 mapped?
5.7 Given the following specifications for an external cache memory: four-way set associative; line size of two 16-bit words; able to accommodate a total of 4K 32-bit words from main memory; used with a 16-bit processor that issues 24-bit addresses. Design the cache structure

with all pertinent information and show how it interprets the processor's addresses.

5.8 The Intel 80486 has an on-chip, unified cache. It contains 8 kB and has a four-way set-associative organization and a block length of four 32-bit words. The cache is organized into 128 sets. There is a single "line valid bit" and three bits, B0, B1, and B2 (the "LRU" bits), per line. On a cache miss, the 80486 reads a 16-byte line from main memory in a bus memory read burst. Draw a simplified diagram of the cache and show how the different fields of the address are interpreted.

5.9 Consider a machine with a byte addressable main memory of 2^{16} bytes and block size of 8 bytes. Assume that a direct mapped cache consisting of 32 lines is used with this machine.

a. How is a 16-bit memory address divided into tag, line number, and byte number?
b. Into what line would bytes with each of the following addresses be stored?

0001	0001	0001	1011
1100	0011	0011	0100
1101	0000	0001	1101
1010	1010	1010	1010

c. Suppose the byte with address 0001 1010 0001 1010 is stored in the cache. What are the addresses of the other bytes stored along with it?
d. How many total bytes of memory can be stored in the cache?
e. Why is the tag also stored in the cache?

5.10 For its on-chip cache, the Intel 80486 uses a replacement algorithm referred to as **pseudo least recently used**. Associated with each of the 128 sets of four lines (labeled L0, L1, L2, L3) are three bits B0, B1, and B2. The replacement algorithm works as follows: When a line must be replaced, the cache will first determine whether the most recent use was from L0 and L1 or L2 and L3. Then the cache will determine which of the pair of blocks was least recently used and mark it for replacement. **Figure 5.19** illustrates the logic.

a. Specify how the bits B0, B1, and B2 are set and then describe in words how they are used in the replacement algorithm depicted in **Figure 5.19** .
b. Show that the 80486 algorithm approximates a true LRU algorithm. *Hint:* Consider the case in which the most recent order of usage is L0, L2, L3, L1.
c. Demonstrate that a true LRU algorithm would require 6 bits per set.

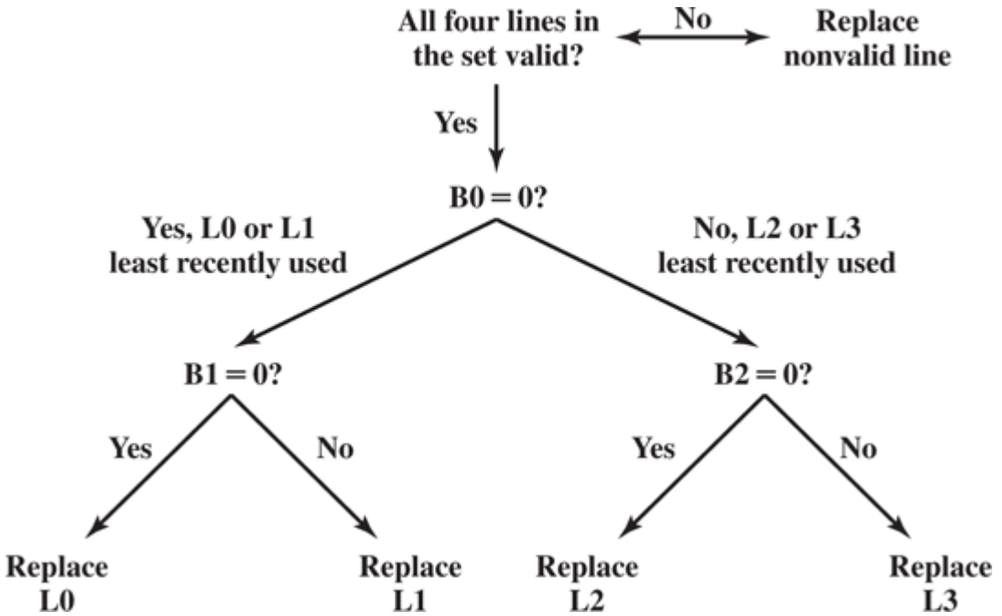


Figure 5.19 Intel 80486 On-Chip Cache Replacement Strategy

5.11 A set-associative cache has a block size of four 16-bit words and a set size of 2. The cache can accommodate a total of 4096 words. The main memory size that is cacheable is $64\text{K} \times 32$ bits. Design the cache structure and show how the processor's addresses are interpreted.

5.12 Consider a memory system that uses a 32-bit address to address at the byte level, plus a cache that uses a 64-byte line size.

a. Assume a direct mapped cache with a tag field in the address of 20 bits. Show the address format and determine the following parameters: number of addressable units, number of blocks in main memory, number of lines in cache, size of tag.
b. Assume an associative cache. Show the address format and determine the following parameters: number of addressable units, number of blocks in main memory, number of lines in cache, size of tag.
c. Assume a four-way set-associative cache with a tag field in the address of 9 bits. Show the address format and determine the following parameters: number of addressable units, number of blocks in main memory, number of lines in set, number of sets in cache, number of lines in cache, size of tag.

5.13 Consider a computer with the following characteristics: total of 1 MB of main memory; word size of 1 byte; block size of 16 bytes; and cache size of 64 kB.

a. For the main memory addresses of F0010, 01234, and CABBE, give the corresponding tag, cache line address, and word offsets for a direct-mapped cache.
b. Give any two main memory addresses with different tags that map to the same cache slot for a direct-mapped cache.
c. For the main memory addresses of F0010 and CABBE, give the corresponding tag and offset values for a fully-associative cache.
d. For the main memory addresses of F0010 and CABBE, give the corresponding tag, cache set, and offset values for a two-way set-associative cache.

5.14 Describe a simple technique for implementing an LRU replacement algorithm in a four-way set-associative cache.

5.15 Consider again **Example 5.2** . How does the answer change if the main memory uses a block transfer capability that has a first-word access time of 30 ns and an access time of 5 ns for each word thereafter?
A computer system contains a main memory of 32K 16-bit words. It also has a 4K word cache divided into four-line sets with 64 words per line. Assume that the cache is initially empty. The processor fetches words from locations 0, 1, 2, . . . , 4351 in that order. It then repeats this fetch sequence nine more times. The cache is 10 times faster than main memory. Estimate the improvement resulting from the use of the cache. Assume an LRU policy for block replacement.
5.16 Consider a cache of 4 lines of 16 bytes each. Main memory is divided into blocks of 16 bytes each. That is, block 0 has bytes with addresses 0 through 15, and so on. Now consider a program that accesses memory in the following sequence of addresses:
Once: 63 through 70.
Loop ten times: 15 through 32; 80 through 95.

a. Suppose the cache is organized as direct mapped. Memory blocks 0, 4, and so on are assigned to line 1; blocks 1, 5, and so on to line 2; and so on. Compute the hit ratio.
b. Suppose the cache is organized as two-way set associative, with two sets of two lines each. Even-numbered blocks are assigned to set 0 and odd-numbered blocks are assigned to set 1. Compute the hit ratio for the two-way set-associative cache using the least recently used replacement scheme.

5.17 Consider a cache with a line size of 64 bytes. Assume that on average 30% of the lines in the cache are dirty. A word consists of 8 bytes.

a. Assume there is a 3% miss rate (0.97 hit ratio). Compute the amount of main memory traffic, in terms of bytes per instruction for both write-through and write-back policies. Memory is read into cache one line at a time. However, for write back, a single word can be written from cache to main memory.
b. Repeat part a for a 5% rate.
c. Repeat part a for a 7% rate.
d. What conclusion can you draw from these results?

5.18 The level below a cache in the memory hierarchy requires 60 ns to read or write a word of data. If the cache line size is 8 words, how many times does the average line have to be written (counting only lines that are written at least once) before a write-back cache is more efficient than a write-through cache?

Chapter 6 Internal Memory

Learning Objectives

After studying this chapter, you should be able to:

- Present an overview of the principle types of semiconductor main memory.
- Understand the operation of a basic code that can detect and correct single-bit errors in 8-bit words.
- Summarize the properties of contemporary **DDR DRAM** organizations.
- Understand the difference between **NOR** and **NAND** flash memory.
- Present an overview of the newer nonvolatile solid-state memory technologies.

We begin this chapter with a survey of semiconductor main memory subsystems,

including ROM, DRAM, and SRAM memories. Then we look at error control techniques used to enhance memory reliability. Following this, we look at more advanced DRAM architectures.

6.1 Semiconductor Main Memory

In earlier computers, the most common form of random-access storage for computer main memory employed an array of doughnut-shaped ferromagnetic loops referred to as *cores*. Hence, main memory was often referred to as *core*, a term that persists to this day. The advent of, and advantages of, microelectronics has long since vanquished the magnetic core memory. Today, the use of semiconductor chips for main memory is almost universal. Key aspects of this technology are explored in this section.

Organization

The basic element of a **semiconductor memory** is the memory cell. Although a variety of electronic technologies are used, all semiconductor memory cells share certain properties:

- They exhibit two stable (or semistable) states, which can be used to represent binary 1 and 0.
- They are capable of being written into (at least once), to set the state.
- They are capable of being read to sense the state.

Figure 6.1 depicts the operation of a memory cell. Most commonly, the cell has three functional terminals capable of carrying an electrical signal. The select terminal, as the name suggests, selects a memory cell for a read or write operation. The control terminal indicates read or write. For writing, the other terminal provides an electrical signal that sets the state of the cell to 1 or 0. For reading, that terminal is used for output of the cell's state. The details of the internal organization, functioning, and timing of the memory cell depend on the specific integrated circuit technology used and are beyond the scope of this book, except for a brief summary. For our purposes, we will take it as given that individual cells can be selected for reading and writing operations.

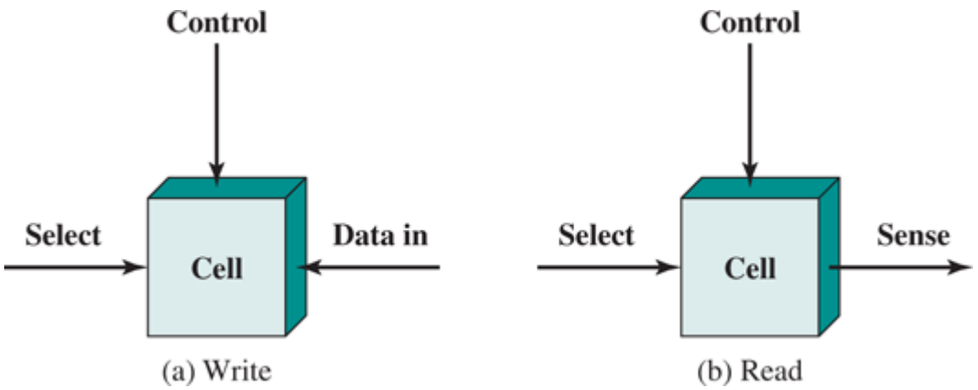


Figure 6.1 Memory Cell Operation

DRAM and SRAM

All of the memory types that we will explore in this chapter are random access. That is, individual words of memory are directly accessed through wired-in addressing logic.

Table 6.1 lists the major types of semiconductor memory. The most common is referred to as **random-access memory (RAM)** . This is, in fact, a misuse of the term, because all of the types listed in the table are random access. One distinguishing characteristic of memory that is designated as RAM is that it is possible both to read data from the memory and to write new data into the memory easily and rapidly. Both the reading and writing are accomplished through the use of electrical signals.

Table 6.1 Semiconductor Memory Types

Memory Type	Category	Erasure	Write Mechanism	Volatility
Random-access memory (RAM)	Read-write memory	Electrically, byte-level	Electrically	Volatile
Read-only memory (ROM)	Read-only memory	Not possible	Masks	Nonvolatile
Programmable ROM (PROM)			Electrically	
Erasable PROM (EPROM)		UV light, chip-level		
Electrically Erasable PROM (EEPROM)	Read-mostly memory	Electrically, byte-level		
Flash memory		Electrically, block-level		

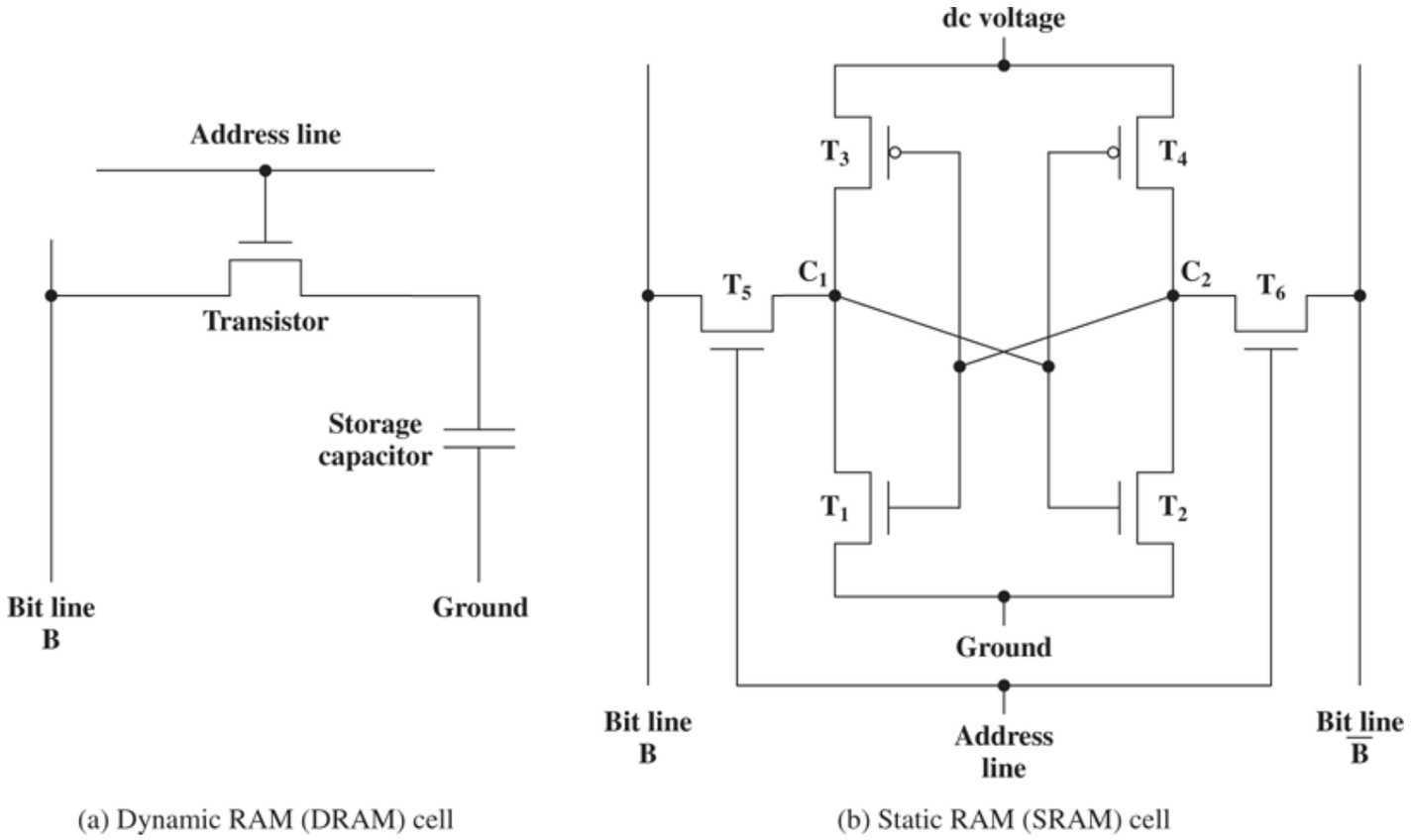
The other distinguishing characteristic of traditional RAM is that it is volatile. A RAM must be provided with a constant power supply. If the power is interrupted, then the data are lost. Thus, RAM can be used only as temporary storage. The two traditional forms of RAM used in computers are DRAM and SRAM. Newer forms of RAM, discussed in Section 6.5, are nonvolatile.

DYNAMIC RAM

RAM technology is divided into two technologies: dynamic and static. A **dynamic RAM (DRAM)** is made with cells that store data as charge on capacitors. The presence or absence of charge in a capacitor is interpreted as a binary 1 or 0. Because capacitors have a natural tendency to discharge, dynamic

RAMs require periodic charge refreshing to maintain data storage. The term *dynamic* refers to this tendency of the stored charge to leak away, even with power continuously applied.

Figure 6.2a is a typical DRAM structure for an individual cell that stores one bit. The address line is activated when the bit value from this cell is to be read or written. The transistor acts as a switch that is closed (allowing current to flow) if a voltage is applied to the address line and open (no current flows) if no voltage is present on the address line.



(a) Dynamic RAM (DRAM) cell

(b) Static RAM (SRAM) cell

Figure 6.2 Typical Memory Cell Structures

For the write operation, a voltage signal is applied to the bit line; a high voltage represents 1, and a low voltage represents 0. A signal is then applied to the address line, allowing a charge to be transferred to the capacitor.

For the read operation, when the address line is selected, the transistor turns on and the charge stored on the capacitor is fed out onto a bit line and to a sense amplifier. The sense amplifier compares the capacitor voltage to a reference value and determines if the cell contains a logic 1 or a logic 0. The readout from the cell discharges the capacitor, which must be restored to complete the operation.

Although the DRAM cell is used to store a single bit (0 or 1), it is essentially an analog device. The capacitor can store any charge value within a range; a threshold value determines whether the charge is interpreted as 1 or 0.

STATIC RAM

In contrast, a **static RAM (SRAM)** is a digital device that uses the same logic elements used in the processor. In a SRAM, binary values are stored using traditional flip-flop logic-gate configurations (see **Chapter 12** for a description of flip-flops). A static RAM will hold its data as long as power is supplied to it.

Figure 6.2b is a typical SRAM structure for an individual cell. Four transistors (T_1, T_2, T_3, T_4) are cross connected in an arrangement that produces a stable logic state. In logic state 1, point C_1 is high and point C_2 is low; in this state, T_1 and T_4 are off and T_2 and T_3 are on.[1] In logic state 0, point C_1 is low and point C_2 is high; in this state, T_1 and T_4 are on and T_2 and T_3 are off. Both states are stable

as long as the direct current (dc) voltage is applied. Unlike the DRAM, no refresh is needed to retain data.

[1] The circles associated with T_3 and T_4 in **Figure 6.2b** indicate signal negation.

As in the DRAM, the SRAM address line is used to open or close a switch. The address line controls two transistors (T_5 and T_6). When a signal is applied to this line, the two transistors are switched on, allowing a read or write operation. For a write operation, the desired bit value is applied to line B, while its complement is applied to line B. This forces the four transistors (T_1, T_2, T_3, T_4) into the proper state. For a read operation, the bit value is read from line B.

SRAM VERSUS DRAM

Both static and dynamic RAMs are volatile; that is, power must be continuously supplied to the memory to preserve the bit values. A dynamic memory cell is simpler and smaller than a static memory cell. Thus, a DRAM is more dense $(\text{smaller cells} = \text{more cells per unit area})$ and less expensive than a corresponding SRAM. On the other hand, a DRAM requires the supporting refresh circuitry. For larger memories, the fixed cost of the refresh circuitry is more than compensated for by the smaller variable cost of DRAM cells. Thus, DRAMs tend to be favored for large memory requirements. A final point is that SRAMs are somewhat faster than DRAMs. Because of these relative characteristics, SRAM is used for cache memory (both on and off chip), and DRAM is used for main memory.

Types of ROM

As the name suggests, a **read-only memory (ROM)** contains a permanent pattern of data that cannot be changed. A ROM is nonvolatile; that is, no power source is required to maintain the bit values in memory. While it is possible to read a ROM, it is not possible to write new data into it. An important application of ROMs is microprogramming, discussed in Part Four. Other potential applications include

- Library subroutines for frequently wanted functions
- System programs
- Function tables

For a modest-sized requirement, the advantage of ROM is that the data or program is permanently in main memory and need never be loaded from a secondary storage device.

A ROM is created like any other integrated circuit chip, with the data actually wired into the chip as part of the fabrication process. This presents two problems:

- The data insertion step includes a relatively large fixed cost, whether one or thousands of copies of a particular ROM are fabricated.
- There is no room for error. If one bit is wrong, the whole batch of ROMs must be thrown out.

When only a small number of ROMs with a particular memory content is needed, a less expensive alternative is the **programmable ROM (PROM)** . Like the ROM, the PROM is **nonvolatile** and may be written into only once. For the PROM, the writing process is performed electrically and may be performed by a supplier or customer at a time later than the original chip fabrication. Special equipment is required for the writing or “programming” process. PROMs provide flexibility and convenience. The ROM remains attractive for high-volume production runs.

Another variation on read-only memory is the **read-mostly memory**, which is useful for applications in

which read operations are far more frequent than write operations but for which nonvolatile storage is required. There are three common forms of read-mostly memory: EPROM, EEPROM, and flash memory.

The optically **erasable programmable read-only memory (EPROM)** is read and written electrically, as with PROM. However, before a write operation, all the storage cells must be erased to the same initial state by exposure of the packaged chip to ultraviolet radiation. Erasure is performed by shining an intense ultraviolet light through a window that is designed into the memory chip. This erasure process can be performed repeatedly; each erasure can take as much as 20 minutes to perform. Thus, the EPROM can be altered multiple times and, like the ROM and PROM, holds its data virtually indefinitely. For comparable amounts of storage, the EPROM is more expensive than PROM, but it has the advantage of the multiple update capability.

A more attractive form of read-mostly memory is **electrically erasable programmable read-only memory (EEPROM)**. This is a read-mostly memory that can be written into at any time without erasing prior contents; only the byte or bytes addressed are updated. The write operation takes considerably longer than the read operation, on the order of several hundred microseconds per byte. The EEPROM combines the advantage of nonvolatility with the flexibility of being updatable in place, using ordinary bus control, address, and data lines. EEPROM is more expensive than EPROM and also is less dense, supporting fewer bits per chip.

Another form of semiconductor memory is **flash memory** (so named because of the speed with which it can be reprogrammed). First introduced in the mid-1980s, flash memory is intermediate between EPROM and EEPROM in both cost and functionality. Like EEPROM, flash memory uses an electrical erasing technology. An entire flash memory can be erased in one or a few seconds, which is much faster than EPROM. In addition, it is possible to erase just blocks of memory rather than an entire chip. Flash memory gets its name because the microchip is organized so that a section of memory cells are erased in a single action or “flash.” However, flash memory does not provide byte-level erasure. Like EPROM, flash memory uses only one transistor per bit, and so achieves the high density (compared with EEPROM) of EPROM.

Chip Logic

As with other integrated circuit products, semiconductor memory comes in packaged chips (**Figure 1.10**). Each chip contains an array of memory cells.

In the memory hierarchy as a whole, we saw that there are trade-offs among speed, density, and cost. These trade-offs also exist when we consider the organization of memory cells and functional logic on a chip. For semiconductor memories, one of the key design issues is the number of bits of data that may be read/written at a time. At one extreme is an organization in which the physical arrangement of cells in the array is the same as the logical arrangement (as perceived by the processor) of words in memory. The array is organized into W words of B bits each. For example, a 16-Mbit chip could be organized as 1M 16-bit words. At the other extreme is the so-called 1-bit-per-chip organization, in which data are read/written one bit at a time. We will illustrate memory chip organization with a DRAM; ROM organization is similar, though simpler.

Figure 6.3 shows a typical organization of a 16-Mbit DRAM. In this case, 4 bits are read or written at a time. Logically, the memory array is organized as four square arrays of 2048 by 2048 elements. Various physical arrangements are possible. In any case, the elements of the array are connected by both horizontal (row) and vertical (column) lines. Each horizontal line connects to the Select terminal of each cell in its row; each vertical line connects to the Data-In/Sense terminal of each cell in its column.

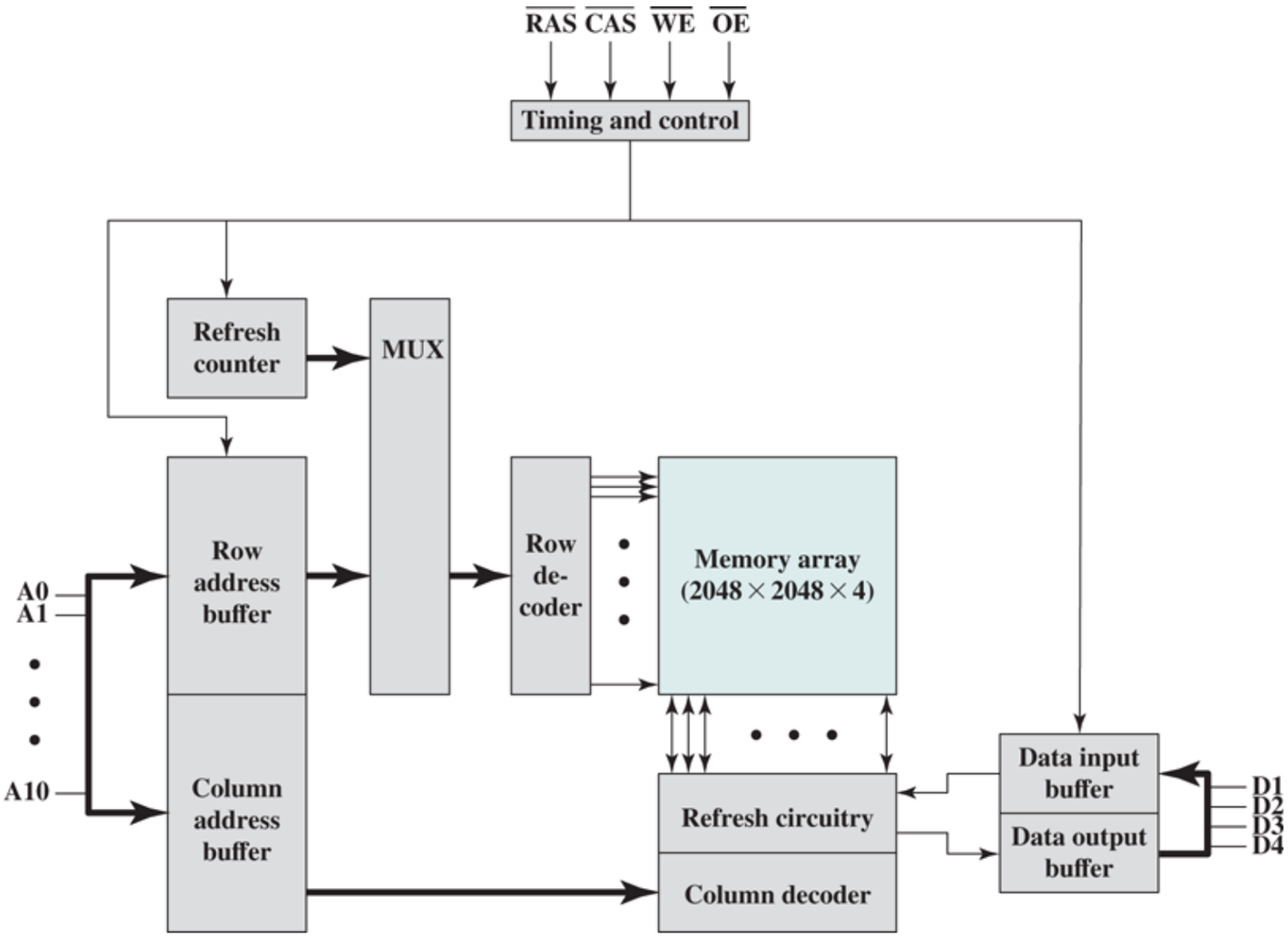


Figure 6.3 Typical 16-Mbit DRAM $(4M \times 4)$

Address lines supply the address of the word to be selected. A total of $log_2 W$ lines are needed. In our example, 11 address lines are needed to select one of 2048 rows. These 11 lines are fed into a row decoder, which has 11 lines of input and 2048 lines for output. The logic of the decoder activates a single one of the 2048 outputs depending on the bit pattern on the 11 input lines $(2^{11} = 2048)$.

An additional 11 address lines select one of 2048 columns of 4 bits per column. Four data lines are used for the input and output of 4 bits to and from a data buffer. On input (write), the bit driver of each bit line is activated for a 1 or 0 according to the value of the corresponding data line. On output (read), the value of each bit line is passed through a sense amplifier and presented to the data lines. The row line selects which row of cells is used for reading or writing.

Because only 4 bits are read/written to this DRAM, there must be multiple DRAMs connected to the memory controller to read/write a word of data to the bus.

Note that there are only 11 address lines (A0–A10), half the number you would expect for a 2048×2048 array. This is done to save on the number of pins. The 22 required address lines are passed through select logic external to the chip and multiplexed onto the 11 address lines. First, 11 address signals are passed to the chip to define the row address of the array, and then the other 11 address signals are presented for the column address. These signals are accompanied by row address select $(\overline{\text{RAS}})$ and column address select $(\overline{\text{CAS}})$ signals to provide timing to the chip.

The write enable (WE) and output enable $(\overline{\mathrm{OE}})$ pins determine whether a write or read operation is performed. Two other pins, not shown in **Figure 6.3**, are ground (Vss) and a voltage source (Vcc).

As an aside, multiplexed addressing plus the use of square arrays result in a quadrupling of memory size with each new generation of memory chips. One more pin devoted to addressing doubles the number of rows and columns, and so the size of the chip memory grows by a factor of 4.

Figure 6.3 also indicates the inclusion of refresh circuitry. All DRAMs require a refresh operation. A simple technique for refreshing is, in effect, to disable the DRAM chip while all data cells are refreshed. The refresh counter steps through all of the row values. For each row, the output lines from the refresh counter are supplied to the row decoder and the RAS line is activated. The data are read out and written back into the same location. This causes each cell in the row to be refreshed.

Chip Packaging

As was mentioned in **Chapter 2** , an An integrated circuit is mounted on a package that contains pins for connection to the outside world.

Figure 6.4a shows an example EPROM package, which is an 8-Mbit chip organized as $1\mathrm{M} \times 8$. In this case, the organization is treated as a one-word-per-chip package. The package includes 32 pins, which is one of the standard chip package sizes. The pins support the following signal lines:

- The address of the word being accessed. For 1M words, a total of $20 (2^{20} = 1\mathrm{M})$ pins are needed (A0–A19).
- The data to be read out, consisting of 8 lines (D0–D7).
- The power supply to the chip $(\mathrm{V_{cc}})$.
- A ground pin $(\mathrm{V_{ss}})$.
- A chip enable (CE) pin. Because there may be more than one memory chip, each of which is connected to the same address bus, the CE pin is used to indicate whether or not the address is valid for this chip. The CE pin is activated by logic connected to the higher-order bits of the address bus (i.e., address bits above A19). The use of this signal is illustrated presently.
- A program voltage $(\mathrm{V_{pp}})$ that is supplied during programming (write operations).

A typical DRAM pin configuration is shown in **Figure 6.4b** , for a 16-Mbit chip organized as $4M \times 4$.

There are several differences from a ROM chip. Because a RAM can be updated, the data pins are input/output. The write enable (WE) and output enable (OE) pins indicate whether this is a write or read operation. Because the DRAM is accessed by row and column, and the address is multiplexed, only 11 address pins are needed to specify the 4M row/column combinations $(2^{11} \times 2^{11} = 2^{22} = 4\mathrm{M})$.

The functions of the row address select (RAS) and column address select (CAS) pins were discussed previously. Finally, the no connect (NC) pin is provided so that there are an even number of pins.

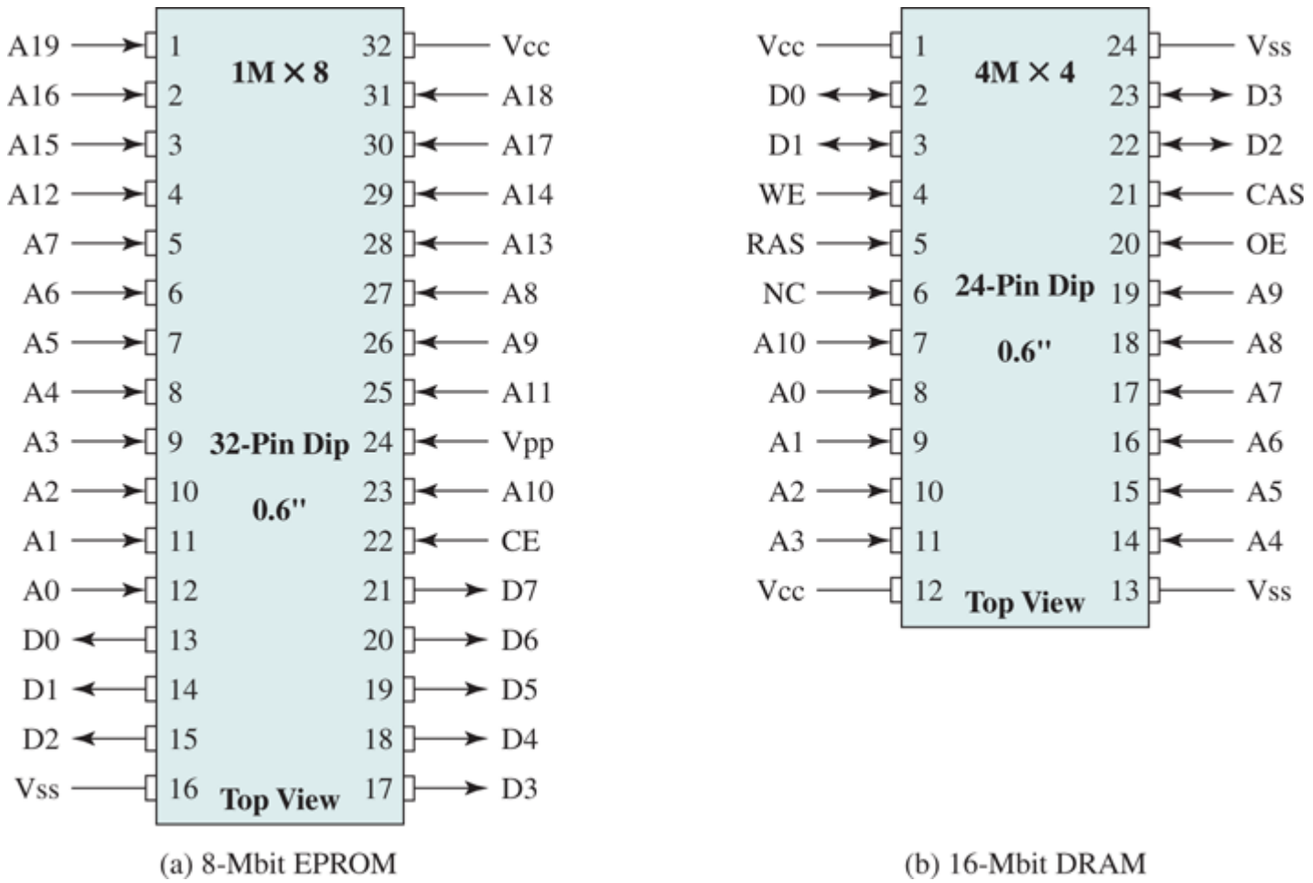


Figure 6.4 Typical Memory Package Pins and Signals

Module Organization

If a RAM chip contains only one bit per word, then clearly we will need at least a number of chips equal to the number of bits per word. As an example, **Figure 6.5** shows how a memory module consisting of 256K 8-bit words could be organized. For 256K words, an 18-bit address is needed and is supplied to the module from some external source (e.g., the address lines of a bus to which the module is attached). The address is presented to $8256\text{K} \times 1\text{-bit}$ chips, each of which provides the input/output of one bit.

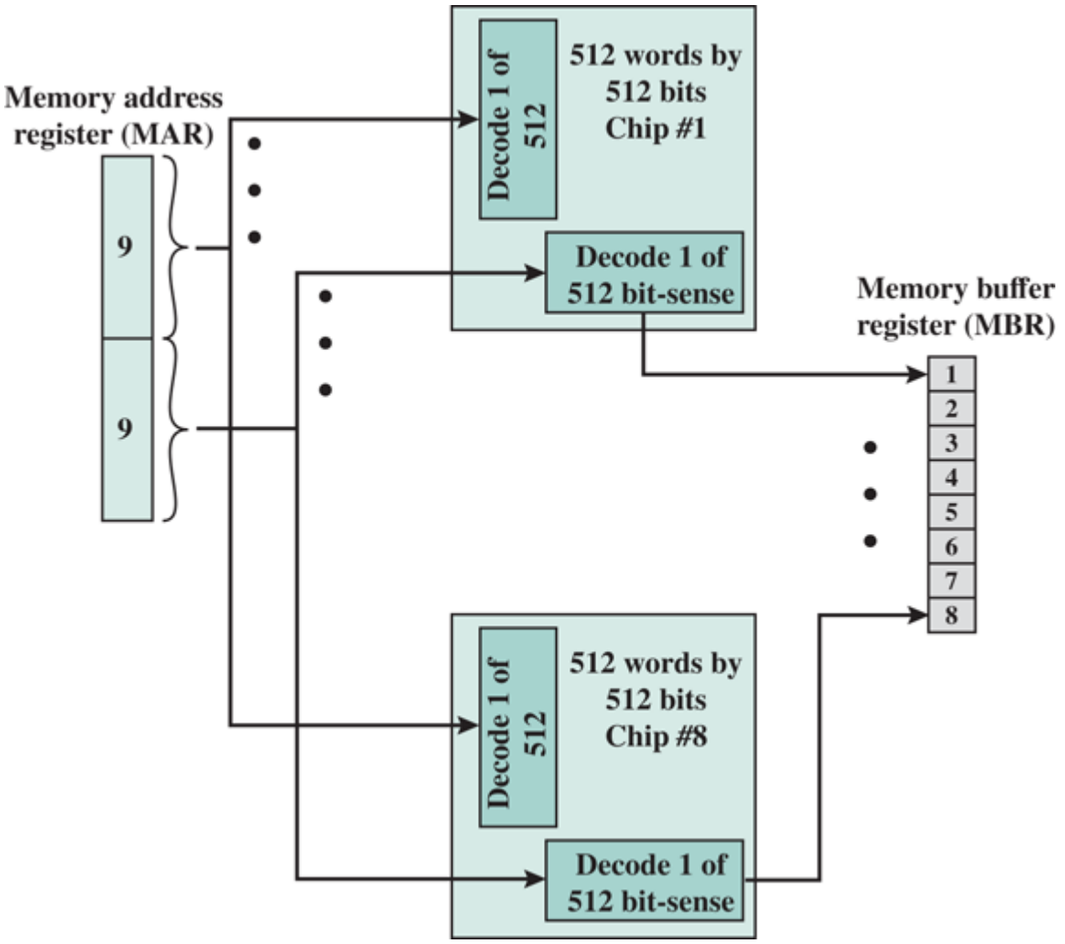


Figure 6.5 256-KByte Memory Organization

This organization works as long as the size of memory in words equals the number of bits per chip. In the case in which larger memory is required, an array of chips is needed. **Figure 6.6** shows the possible organization of a memory consisting of 1M word by 8 bits per word. In this case, we have four columns of chips, each column containing 256K words arranged as in **Figure 6.5** . For 1M word, 20 address lines are needed. The 18 least significant bits are routed to all 32 modules. The high-order 2 bits are input to a group select logic module that sends a chip enable signal to one of the four columns of modules.

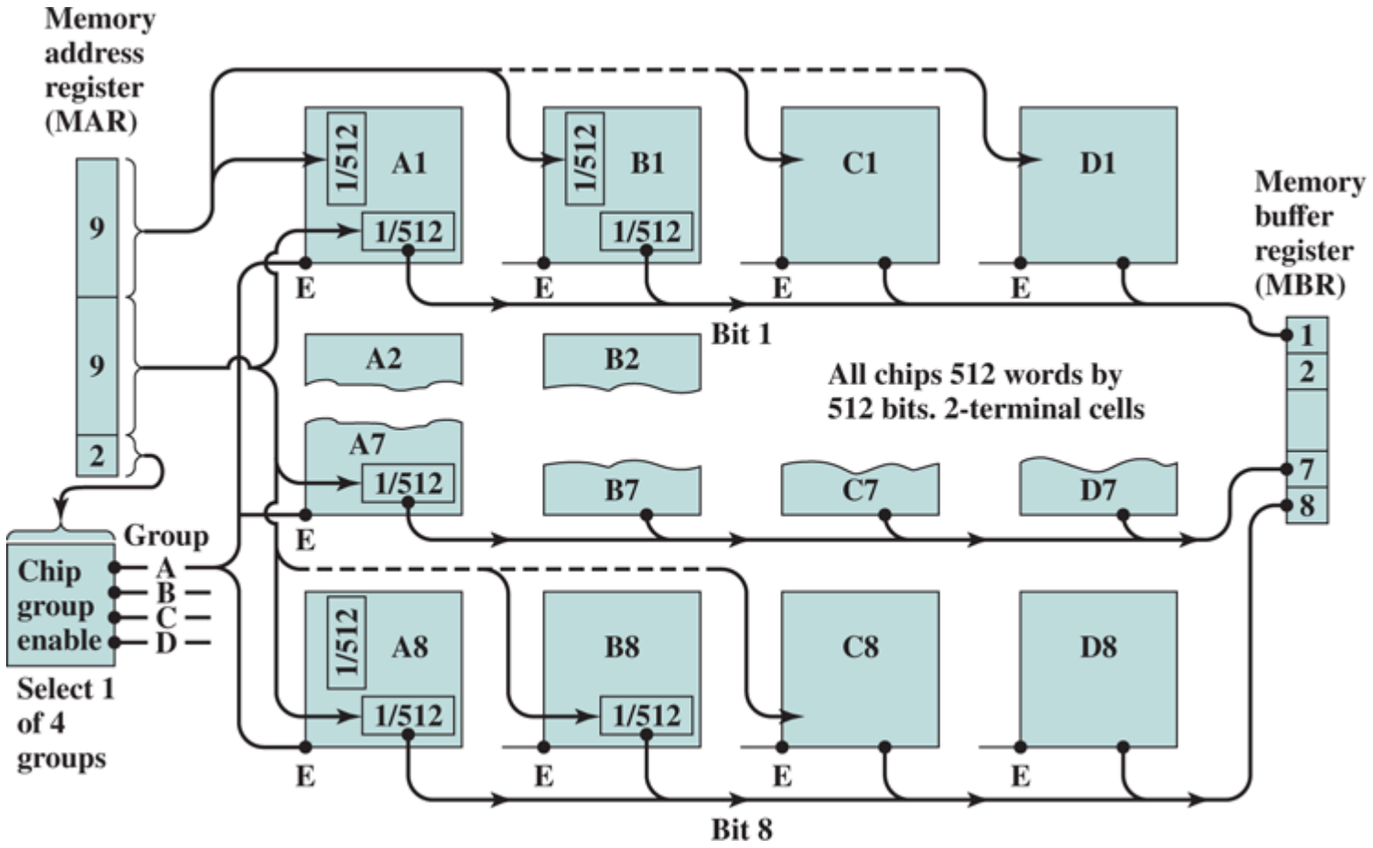


Figure 6.6 1-MB Memory Organization

Interleaved Memory

Main memory is composed of a collection of DRAM memory chips. A number of chips can be grouped together to form a *memory bank*. It is possible to organize the memory banks in a way known as interleaved memory. Each bank is independently able to service a memory read or write request, so that a system with *K* banks can service *K* requests simultaneously, increasing memory read or write rates by a factor of *K*. If consecutive words of memory are stored in different banks, then the transfer of a block of memory is speeded up. Appendix C explores the topic of interleaved memory.

Aleksandr Lukin/123RF

Interleaved Memory Simulator

6.2 Error Correction

A semiconductor memory system is subject to errors. These can be categorized as hard failures and soft errors. A **hard failure** is a permanent physical defect so that the memory cell or cells affected cannot reliably store data but become stuck at 0 or 1 or switch erratically between 0 and 1. Hard errors can be caused by harsh environmental abuse, manufacturing defects, and wear. A **soft error** is a random, nondestructive event that alters the contents of one or more memory cells without damaging the memory. Soft errors can be caused by power supply problems or alpha particles. These particles result from radioactive decay and are distressingly common because radioactive nuclei are found in small quantities in nearly all materials. Both hard and soft errors are clearly undesirable, and most modern main memory systems include logic for both detecting and correcting errors.

Figure 6.7 illustrates in general terms how the process is carried out. When data are to be written into memory, a calculation, depicted as a function *f*, is performed on the data to produce a code. Both the code and the data are stored. Thus, if an *M*-bit word of data is to be stored and the code is of length *K* bits, then the actual size of the stored word is $M + K$ bits.

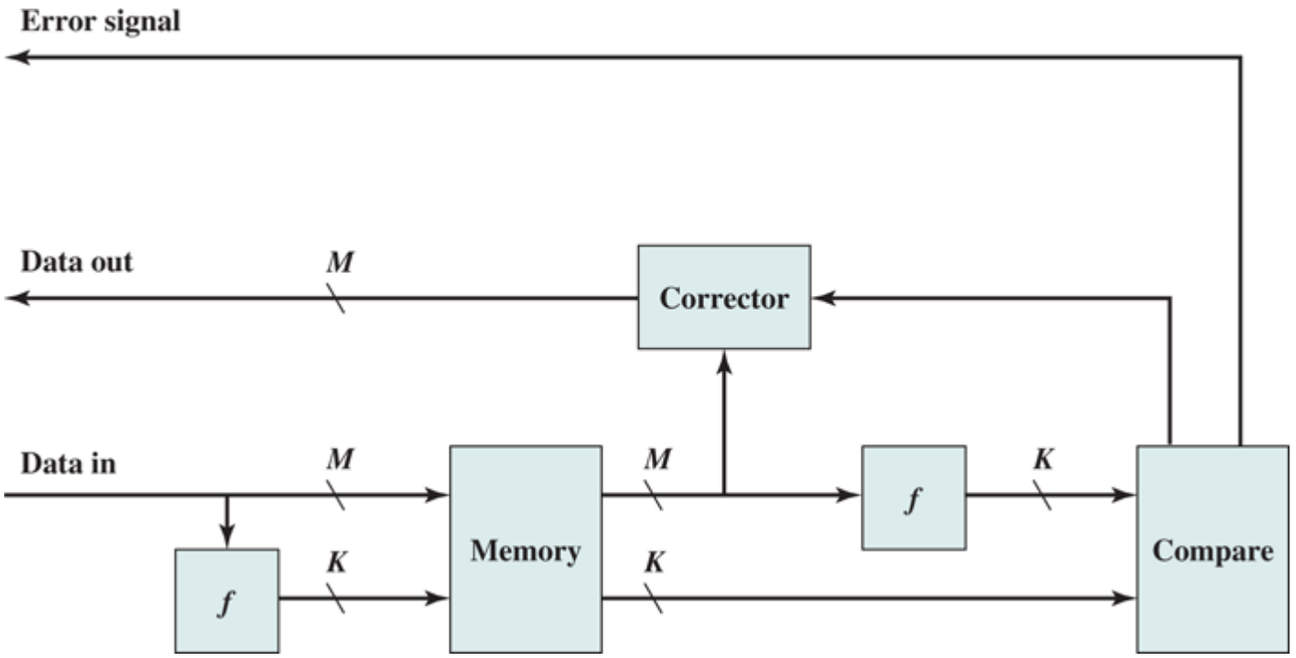


Figure 6.7 Error-Correcting Code Function

When the previously stored word is read out, the code is used to detect and possibly correct errors. A new set of *K* code bits is generated from the *M* data bits and compared with the fetched code bits. The comparison yields one of three results:

- No errors are detected. The fetched data bits are sent out.
- An error is detected, and it is possible to correct the error. The data bits plus **error correction** bits are fed into a corrector, which produces a corrected set of *M* bits to be sent out.
- An error is detected, but it is not possible to correct it. This condition is reported.

Codes that operate in this fashion are referred to as **error-correcting codes** . A code is characterized by the number of bit errors in a word that it can correct and detect.

The simplest of the error-correcting codes is the **Hamming code** devised by Richard Hamming at Bell Laboratories. **Figure 6.8** uses Venn diagrams to illustrate the use of this code on 4-bit words $(M = 4)$.

With three intersecting circles, there are seven compartments. We assign the 4 data bits to the inner compartments (**Figure 6.8a**). The remaining compartments are filled with what are called *parity bits.* Each parity bit is chosen so that the total number of 1s in its circle is even (**Figure 6.8b**). Thus, because circle A includes three data 1s, the parity bit in that circle is set to 1. Now, if an error changes one of the data bits (**Figure 6.8c**), it is easily found. By checking the parity bits, discrepancies are found in circle A and circle C but not in circle B. Only one of the seven compartments is in A and C but not B (Figure 6.8d). The error can therefore be corrected by changing that bit.

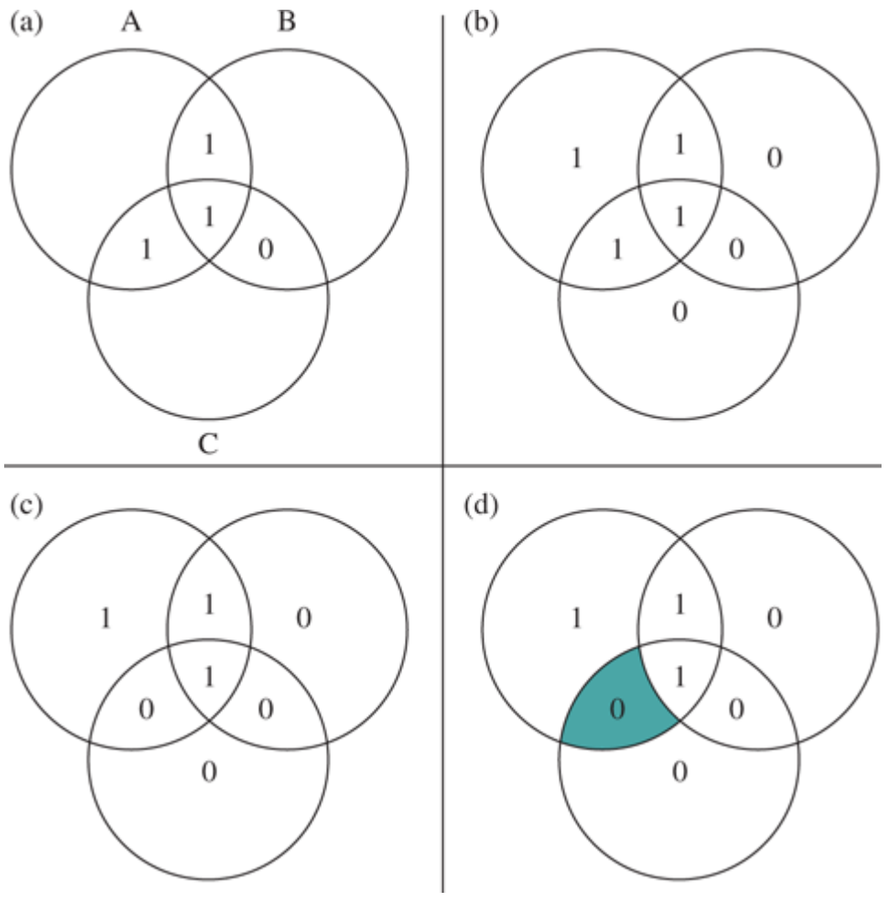


Figure 6.8 Hamming Error-Correcting Code

To clarify the concepts involved, we will develop a code that can detect and correct single-bit errors in 8-bit words.

To start, let us determine how long the code must be. Referring to **Figure 6.7**, the comparison logic receives as input two *K*-bit values. A bit-by-bit comparison is done by taking the exclusive-OR of the two inputs. The result is called the *syndrome word.* Thus, each bit of the **syndrome** is 0 or 1 according to if there is or is not a match in that bit position for the two inputs.

The syndrome word is therefore *K* bits wide and has a range between 0 and $2^K - 1$. The value 0 indicates that no error was detected, leaving $2^K - 1$ values to indicate, if there is an error, which bit was in error. Now, because an error could occur on any of the *M* data bits or *K* check bits, we must have

$$2^K - 1 \geq M + K$$

This inequality gives the number of bits needed to correct a single bit error in a word containing *M*

data bits. For example, for a word of 8 data bits $(M = 8)$, we have

- $K = 3 : 2^3 - 1 < 8 + 3$
- $K = 4 : 2^4 - 1 > 8 + 4$

Thus, eight data bits require four check bits. The first three columns of **Table 6.2** lists the number of check bits required for various data word lengths.

Table 6.2 Increase in Word Length with Error Correction

	Single-Error Correction		Single-Error Correction/ Double-Error Detection	
Data Bits	Check Bits	% Increase	Check Bits	% Increase
8	4	50.0	5	62.5
16	5	31.25	6	37.5
32	6	18.75	7	21.875
64	7	10.94	8	12.5
128	8	6.25	9	7.03
256	9	3.52	10	3.91

For convenience, we would like to generate a 4-bit syndrome for an 8-bit data word with the following characteristics:

- If the syndrome contains all 0s, no error has been detected.
- If the syndrome contains one and only one bit set to 1, then an error has occurred in one of the 4 check bits. No correction is needed.
- If the syndrome contains more than one bit set to 1, then the numerical value of the syndrome indicates the position of the data bit in error. This data bit is inverted for correction.

To achieve these characteristics, the data and check bits are arranged into a 12-bit word as depicted in **Figure 6.9**. The bit positions are numbered from 1 to 12. Those bit positions whose position numbers are powers of 2 are designated as check bits. The check bits are calculated as follows, where the symbol $\oplus$ designates the exclusive-OR operation:

$$
\begin{aligned}
C1 &= D1 \oplus D2 \oplus D4 \oplus D5 \oplus D7 \\
C2 &= D1 \oplus D3 \oplus D4 \oplus D6 \oplus D7 \\
C4 &= D2 \oplus D3 \oplus D4 \oplus D8 \\
C8 &= \oplus D5 \oplus D6 \oplus D7 \oplus D8
\end{aligned}
$$

Each check bit operates on every data bit whose position number contains a 1 in the same bit position as the position number of that check bit. Thus, data bit positions 3, 5, 7, 9, and 11 (D1, D2, D4, D5, D7) all contain a 1 in the least significant bit of their position number as does C1; bit positions 3, 6, 7, 10, and 11 all contain a 1 in the second bit position, as does C2; and so on. Looked at another way, bit position *n* is checked by those bits C such that $\sum = n$. For example, position 7 is checked by bits

in position 4, 2, and 1; and $7 = 4 + 2 + 1$.

Bit position	12	11	10	9	8	7	6	5	4	3	2	1
Position number	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Data bit	D8	D7	D6	D5		D4	D3	D2		D1		
Check bit					C8				C4		C2	C1

Figure 6.9 Layout of Data Bits and Check Bits

Let us verify that this scheme works with an example. Assume that the 8-bit input word is 00111001, with data bit D1 in the rightmost position. The calculations are as follows:

$$\begin{aligned} C1 &= 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1 \\ C2 &= 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1 \\ C4 &= 0 \oplus 0 \oplus 1 \oplus 0 = 1 \\ C8 &= 1 \oplus 1 \oplus 0 \oplus 0 = 0 \end{aligned}$$

Suppose now that data bit 3 sustains an error and is changed from 0 to 1. When the check bits are recalculated, we have

$$\begin{aligned} C1 &= 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1 \\ C2 &= 1 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 0 \\ C4 &= 0 \oplus 1 \oplus 1 \oplus 0 = 0 \\ C8 &= 1 \oplus 1 \oplus 0 \oplus 0 = 0 \end{aligned}$$

When the new check bits are compared with the old check bits, the syndrome word is formed:

$$\begin{array}{cccc} C8 & C4 & C2 & C1 \\ 0 & 1 & 1 & 1 \\ \oplus\ 0 & 0 & 0 & 1 \\ \hline 0 & 1 & 1 & 0 \end{array}$$

The result is 0110, indicating that bit position 6, which contains data bit 3, is in error.

Figure 6.10 illustrates the preceding calculation. The data and check bits are positioned properly in the 12-bit word. Four of the data bits have a value 1 (shaded in the table), and their bit position values are XORed to produce the Hamming code 0111, which forms the four check digits. The entire block that is stored is 001101001111. Suppose now that data bit 3, in bit position 6, sustains an error and is changed from 0 to 1. The resulting block is 001101101111, with a Hamming code of 0001. An XOR of the Hamming code and all of the bit position values for nonzero data bits results in 0110. The nonzero result detects an error and indicates that the error is in bit position 6.

Bit position	12	11	10	9	8	7	6	5	4	3	2	1
Position number	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Data bit	D8	D7	D6	D5		D4	D3	D2		D1		
Check bit					C8				C4		C2	C1
Word stored as	0	0	1	1	0	1	0	0	1	1	1	1
Word fetched as	0	0	1	1	0	1	1	0	1	1	1	1
Position number	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Check bit					0				0		0	1

Figure 6.10 Check Bit Calculation

The code just described is known as a **single-error-correcting (SEC) code**. More commonly, semiconductor memory is equipped with a **single-error-correcting, double-error-detecting (SEC-DED) code**. As **Table 6.2** shows, such codes require one additional bit compared with SEC codes.

Figure 6.11 illustrates how such a code works, again with a 4-bit data word. The sequence shows that if two errors occur (**Figure 6.11c**), the checking procedure goes astray (d) and worsens the problem by creating a third error (e). To overcome the problem, an eighth bit is added that is set so that the total number of 1s in the diagram is even. The extra parity bit catches the error (f).

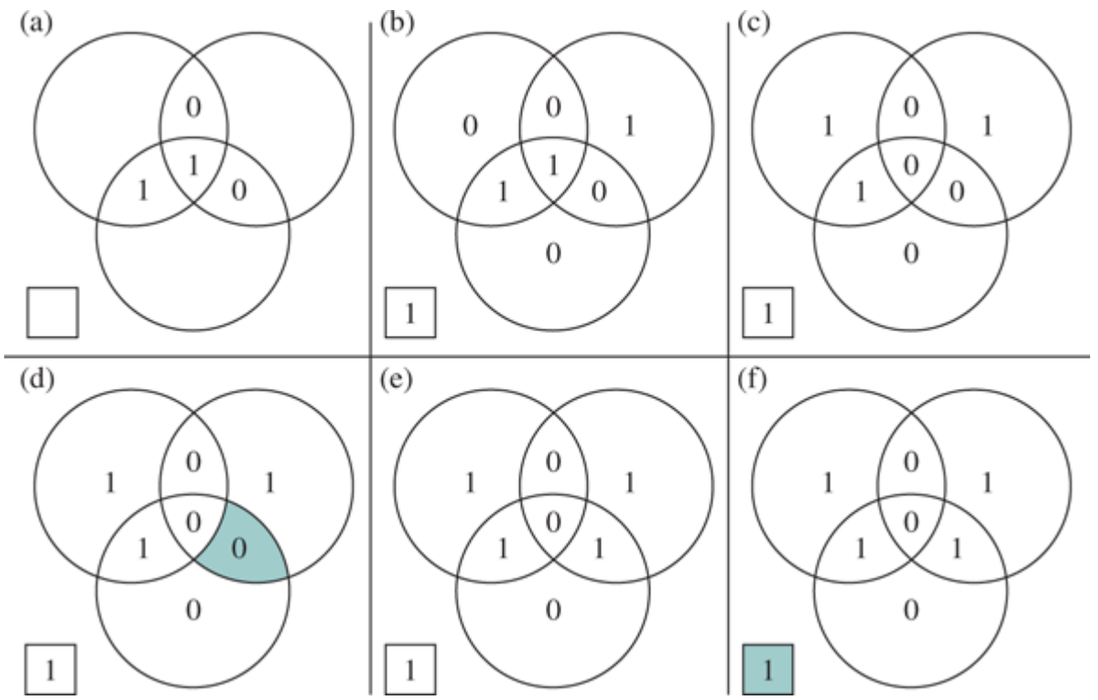


Figure 6.11 Hamming SEC-DEC Code

An error-correcting code enhances the reliability of the memory at the cost of added complexity. With a 1-bit-per-chip organization, an SEC-DED code is generally considered adequate. For example, the IBM 30xx implementations used an 8-bit SEC-DED code for each 64 bits of data in main memory. Thus, the size of main memory is actually about 12% larger than is apparent to the user. The VAX computers used a 7-bit SEC-DED for each 32 bits of memory, for a 22% overhead. Contemporary

DRAM systems may have anywhere from 7% to 20% overhead [SHAR03].

6.3 DDR DRAM

As discussed in **Chapter 1**, One of the most critical system bottlenecks when using high-performance processors is the interface to internal main memory. This interface is the most important pathway in the entire computer system. The basic building block of main memory remains the DRAM chip, as it has for decades; until recently, there had been no significant changes in DRAM architecture since the early 1970s. The traditional DRAM chip is constrained both by its internal architecture and by its interface to the processor's memory bus.

We have seen that one attack on the performance problem of DRAM main memory has been to insert one or more levels of high-speed SRAM cache between the DRAM main memory and the processor. But SRAM is much costlier than DRAM, and expanding cache size beyond a certain point yields diminishing returns.

In recent years, a number of enhancements to the basic DRAM architecture have been explored. The schemes that currently dominate the market are SDRAM and DDR-DRAM. We examine each of these in turn.

Synchronous DRAM

One of the most widely used forms of DRAM is the **synchronous DRAM (SDRAM)**. Unlike the traditional DRAM, which is asynchronous, the SDRAM exchanges data with the processor synchronized to an external clock signal and running at the full speed of the processor/memory bus without imposing wait states.

In a typical DRAM, the processor presents addresses and control levels to the memory, indicating that a set of data at a particular location in memory should be either read from or written into the DRAM. After a delay, the access time, the DRAM either writes or reads the data. During the access-time delay, the DRAM performs various internal functions, such as activating the high capacitance of the row and column lines, sensing the data, and routing the data out through the output buffers. The processor must simply wait through this delay, slowing system performance.

With synchronous access, the DRAM moves data in and out under control of the system clock. The processor or other master issues the instruction and address information, which is latched by the DRAM. The DRAM then responds after a set number of clock cycles. Meanwhile, the master can safely do other tasks while the SDRAM is processing the request.

Figure 6.12 shows the internal logic of a typical 256-Mb SDRAM typical of SDRAM organization, and **Table 6.3** defines the various pin assignments. The SDRAM employs a burst mode to eliminate the address setup time and row and column line precharge time after the first access. In burst mode, a series of data bits can be clocked out rapidly after the first bit has been accessed. This mode is useful when all the bits to be accessed are in sequence and in the same row of the array as the initial access. In addition, the SDRAM has a multiple-bank internal architecture that improves opportunities for on-chip parallelism.

Table 6.3 SDRAM Pin Assignments

A0 to A13	Address inputs
BA0, BA1	Bank address lines

CLK	Clock input
CKE	Clock enable
$\overline{CS}$	Chip select
$\overline{RAS}$	Row address strobe
$\overline{CAS}$	Column address strobe
$\overline{WE}$	Write enable
DQ0 to DQ7	Data input/output
DQM	Data mask

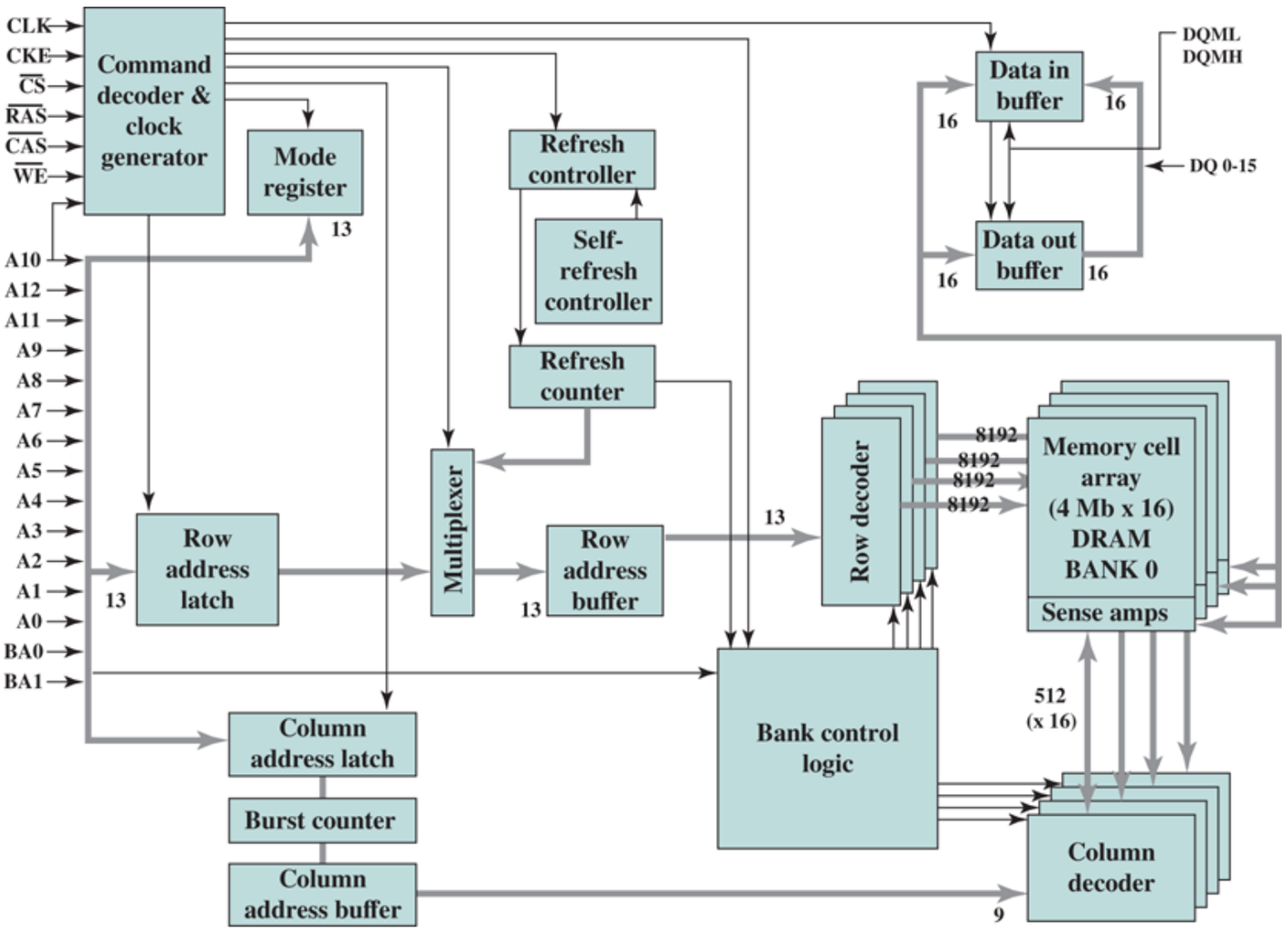


Figure 6.12 256-Mb Synchronous Dynamic RAM (SDRAM)

The mode register and associated control logic is another key feature differentiating SDRAMs from conventional DRAMs. It provides a mechanism to customize the SDRAM to suit specific system

needs. The mode register specifies the burst length, which is the number of separate units of data synchronously fed onto the bus. The register also allows the programmer to adjust the latency between receipt of a read request and the beginning of data transfer.

The SDRAM performs best when it is transferring large blocks of data sequentially, such as for applications like word processing, spreadsheets, and multimedia.

Figure 6.13 shows an example of SDRAM operation, using a **timing diagram**. A timing diagram shows the signal level on a line as a function of time. By convention, the binary 1 signal level is depicted as a higher level than that of binary 0. Usually, binary 0 is the default value. That is, if no data or other signal is being transmitted, then the level on a line is that which represents binary 0. A signal transition from 0 to 1 is frequently referred to as the signal's *leading edge*; a transition from 1 to 0 is referred to as a *trailing edge*. Such transitions are not instantaneous, but this transition time is usually small compared with the duration of a signal level. For clarity, the transition is usually depicted as an angled line that exaggerates the relative amount of time that the transition takes. Signals are sometimes represented in groups, shown as shaded areas in **Figure 6.13**. For example, if data are transferred a byte at a time, then eight lines are required. Generally, it is not important to know the exact value being transferred on such a group, but rather whether signals are present or not.

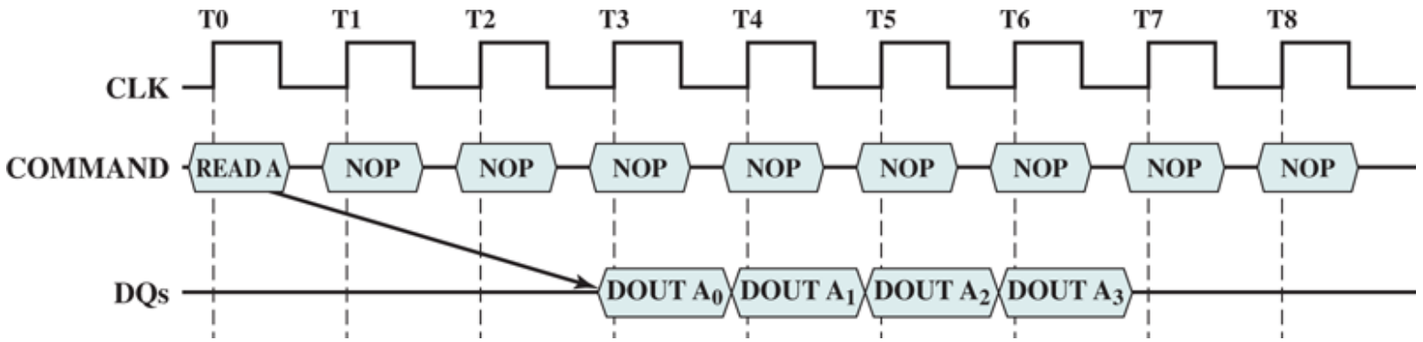


Figure 6.13 SDRAM Read Timing $(\text{burstlength} = 4, \text{CASlatency} = 2)$

For the SDRAM operation in **Figure 6.13**, the burst length is 4 and the latency is 2. The burst read command is initiated by having $\overline{\text{CS}}$ and $\overline{\text{CAS}}$ low while holding $\overline{\text{RAS}}$ and $\overline{\text{WE}}$ high at the rising edge of the clock. The address inputs determine the starting column address for the burst, and the mode register sets the type of burst (sequential or interleave) and the burst length (1, 2, 4, 8, full page). The delay from the start of the command to when the data from the first cell appears on the outputs is equal to the value of the $\overline{\text{CAS}}$ latency that is set in the mode register.

DDR SDRAM

Although SDRAM is a significant improvement on asynchronous RAM, it still has shortcomings that unnecessarily limit the I/O data rate that can be achieved. To address these shortcomings a newer version of SDRAM, referred to as double-data-rate DRAM (DDR DRAM) provides several features that dramatically increase the data rate. DDR DRAM was developed by the JEDEC Solid State Technology Association, the Electronic Industries Alliance's semiconductor-engineering-standardization body. Numerous companies make DDR chips, which are widely used in desktop computers and servers.

DDR achieves higher data rates in three ways. First, the data transfer is synchronized to both the rising and falling edge of the clock, rather than just the rising edge. This doubles the data rate; hence

the term *double data rate*. Second, DDR uses higher clock rate on the bus to increase the transfer rate. Third, a buffering scheme is used, as explained subsequently.

JEDEC has thus far defined four generations of the DDR technology (**Table 6.4**). The initial DDR version makes use of a 2-bit prefetch buffer. The prefetch buffer is a memory cache located on the SDRAM chip. It enables the SDRAM chip to preposition bits to be placed on the data bus as rapidly as possible. The DDR I/O bus uses the same clock rate as the memory chip, but because it can handle two bits per cycle, it achieves a data rate that is double the clock rate. The 2-bit prefetch buffer enables the SDRAM chip to keep up with the I/O bus.

Table 6.4 DDR Characteristics

	DDR1	DDR2	DDR3	DDR4
Prefetch buffer (bits)	2	4	8	8
Voltage level (V)	2.5	1.8	1.5	1.2
Front side bus data rates (Mbps)	200—400	400—1066	800—2133	2133—4266

To understand the operation of the prefetch buffer, we need to look at it from the point of view of a word transfer. The prefetch buffer size determines how many words of data are fetched (across multiple SDRAM chips) every time a column command is performed with DDR memories. Because the core of the DRAM is much slower than the interface, the difference is bridged by accessing information in parallel and then serializing it out the interface through a multiplexor (MUX). Thus, DDR prefetches two words, which means that every time a read or a write operation is performed, it is performed on two words of data, and bursts out of, or into, the SDRAM over one clock cycle on both clock edges for a total of two consecutive operations. As a result, the DDR I/O interface is twice as fast as the SDRAM core.

Although each new generation of SDRAM results is much greater capacity, the core speed of the SDRAM has not changed significantly from generation to generation. To achieve greater data rates than those afforded by the rather modest increases in SDRAM clock rate, JEDEC increased the buffer size. For DDR2, a 4-bit buffer is used, allowing for words to be transferred in parallel, increasing the effective data rate by a factor of 4. For DDR3, an 8-bit buffer is used and a factor of 8 speedup is achieved (**Figure 6.14**).

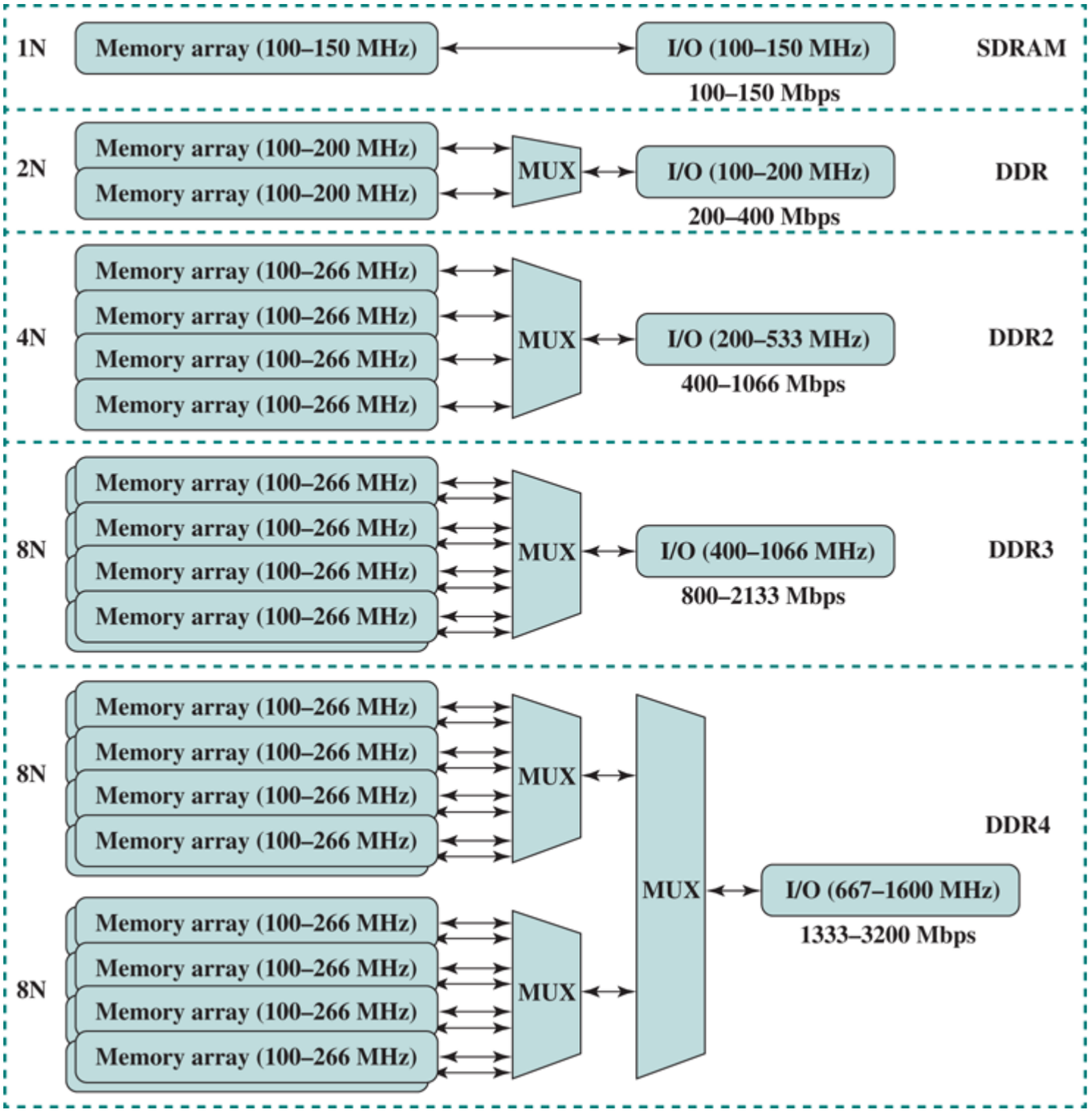


Figure 6.14 DDR Generations

The downside to the prefetch is that it effectively determines the minimum burst length for the SDRAMs. For example, it is very difficult to have an efficient burst length of four words with DDR3's prefetch of eight. Accordingly, the JEDEC designers chose not to increase the buffer size to 16 bits for DDR4, but rather to introduce the concept of a **bank group** [ALLA13]. Bank groups are separate entities such that they allow a column cycle to complete within a bank group, but that column cycle does not impact what is happening in another bank group. Thus, two prefetches of eight can be operating in parallel in the two bank groups. This arrangement keeps the prefetch buffer size the same as for DDR3, while increasing performance as if the prefetch is larger.

Figure 6.14 shows a configuration with two bank groups. With DDR4, up to 4 bank groups can be used.

6.4 Edram

An increasingly widespread technology used in the memory hierarchy is the embedded DRAM (eDRAM). eDRAM is a DRAM integrated on the same chip or MCM of an application-specific integrated circuit (ASIC) or microprocessor. For a number of metrics, eDRAM is intermediate between on-chip SRAM and off-chip DRAM:

- For the same surface area, eDRAM provides a larger size memory than SRAM but smaller than off-chip DRAM.
- eDRAM's cost-per-bit is higher when compared to equivalent stand-alone DRAM chips used as external memory, but it has a lower cost-per-bit than SRAM.
- Access time to eDRAM is greater than SRAM but, because of its proximity and the ability to use wider busses, eDRAM provides faster access than DRAM.

A variety of technologies are used in fabricating eDRAMs, but fundamentally they use the same designs and architectures as DRAM.

[JACO08] lists the following as trends that have led to increasing use of eDRAM:

- For larger systems and high-end applications, the spatial locality curves have become flatter and wider, meaning that the likely area of memory for upcoming references is larger. This makes DRAM-based caches attractive due to their bit density.
- On-chip or on-MCM eDRAM matches the performance of off-chip SRAM, so that greater cache size can be achieved by replacing some on-chip area that would otherwise be dedicated to SRAM with DRAM, avoiding or reducing the need for off-chip SRAM or DRAM.
- eDRAM generally dissipates less power than SRAM.

IBM z13 eDRAM Cache Structure

The IBM z13 system uses eDRAM at two levels of the cache hierarchy (see **Figure 4.10**). Each processor unit (PU) chip, with up to eight cores, has a shared 64-MB eDRAM L3 cache. This is an example of an eDRAM integrated on the same chip as the microprocessors. Three PU chips share a 480-MB eDRAM L4 cache (see **Figure 5.18**). The L4 cache is on a separate storage control (SC) chip. This is an example of an eDRAM integrated on the same chip with other memory-related logic. The L4 cache on each SC chip has 480 MB of noninclusive cache and a 224-MB Non-data Inclusive Coherent (NIC) directory. The NIC directory consists of tags that point to L3-owned lines that have not been included in L4 cache.

Figure 6.15 shows the physical layout of an SC chip. About 60% if the surface area of the SC chip is devoted to the L4 cache and the NIC directory. The remainder of the chip includes L4 cache controller logic and I/O logic.

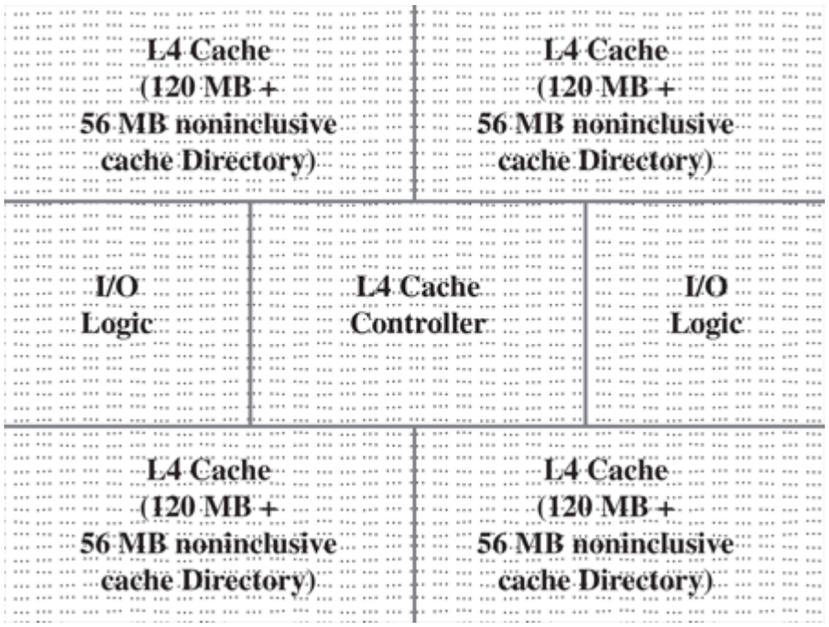
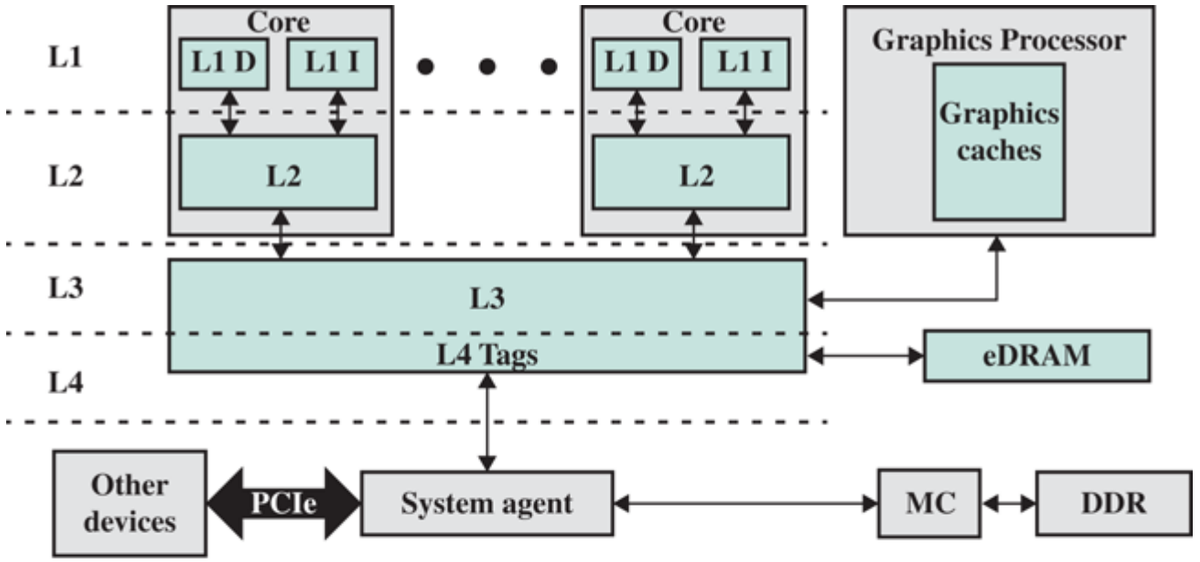


Figure 6.15 IBM z13 Storage Control (SC) Chip Layout

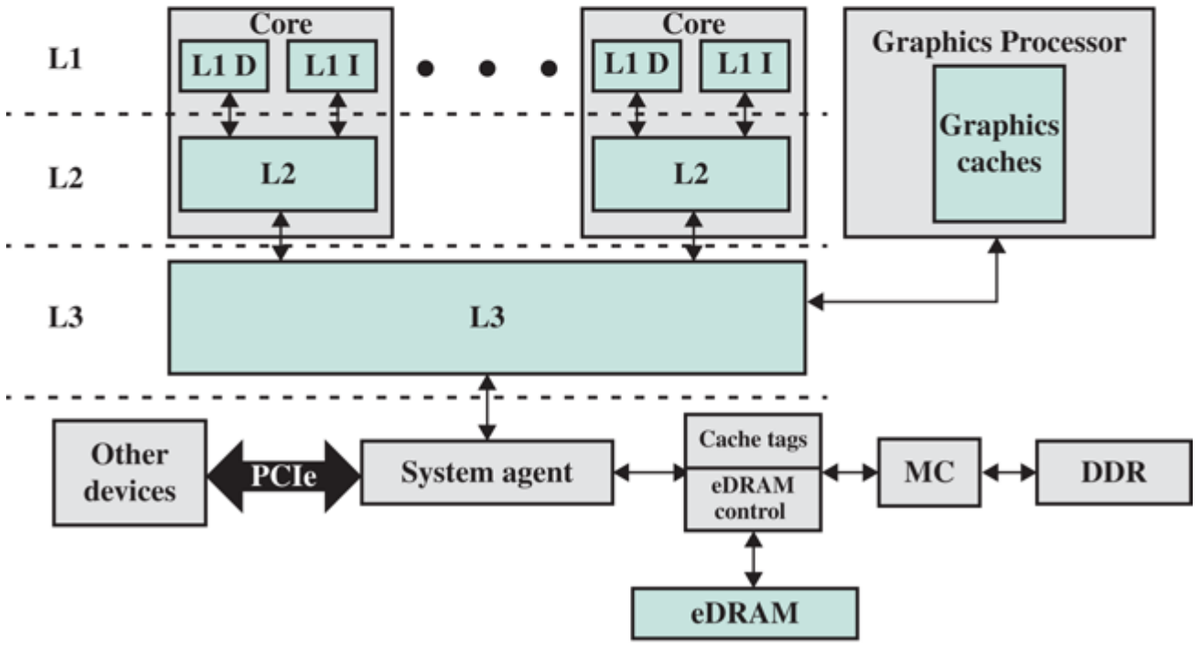
Intel Core System Cache Structure

Intel has shipped a number of products with an eDRAM positioned as an L4 cache. **Figure 6.16a** shows this arrangement. The eDRAM is accessed by a store of L4 tags contained within the L3 cache of each core, and as a result acts more as a victim cache to the L3 rather than as a DRAM implementation. Any instructions or hardware that requires data from the eDRAM has to go through the L3 and do the L4 tag conversion, limiting its potential.

In more recent products, Intel removed the eDRAM from its position as an L4 cache, as shown in **Figure 6.16b**. This removed an undesired dependency between the capacity of the eDRAM and the number of cores. In this new arrangement, the eDRAM is effectively no longer a true L4 cache but rather a memory side cache. This has a number of benefits such that each and every memory access that goes through the memory controller gets looked up in the eDRAM. On a satisfied hit, the value is obtained from there. On a miss, a value gets allocated and stored in the eDRAM. Thus, rather than acting as a pseudo-L4 cache, the eDRAM becomes a DRAM buffer and automatically transparent to any software (CPU or IGP) that requires DRAM access. As a result, other hardware that communicates through the system agent (such as PCIe devices or data from the chipset) and requires information in DRAM does not need to navigate through the L3 cache on the processor.



(a) Original use of eDRAM



(b) More recent use of eDRAM

MC = memory controller

Figure 6.16 Use of eDRAM in Intel Core Systems

6.5 Flash Memory

Another form of semiconductor memory is flash memory. Flash memory is used both for internal memory and external memory applications. Here, we provide a technical overview and look at its use for internal memory.

First introduced in the mid-1980s, flash memory is intermediate between EPROM and EEPROM in both cost and functionality. Like EEPROM, flash memory uses an electrical erasing technology. An entire flash memory can be erased in one or a few seconds, which is much faster than EPROM. In addition, it is possible to erase just blocks of memory, rather than an entire chip. Flash memory gets its name because the microchip is organized so that a section of memory cells are erased in a single action or “flash.” However, flash memory does not provide byte-level erasure. Like EPROM, flash memory uses only one transistor per bit, and so achieves the high density (compared with EEPROM) of EPROM.

Operation

Figure 6.17 illustrates the basic operation of a flash memory. For comparison, **Figure 6.17a** depicts the operation of a transistor. Transistors exploit the properties of semiconductors so that a small voltage applied to the gate can be used to control the flow of a large current between the source and the drain.

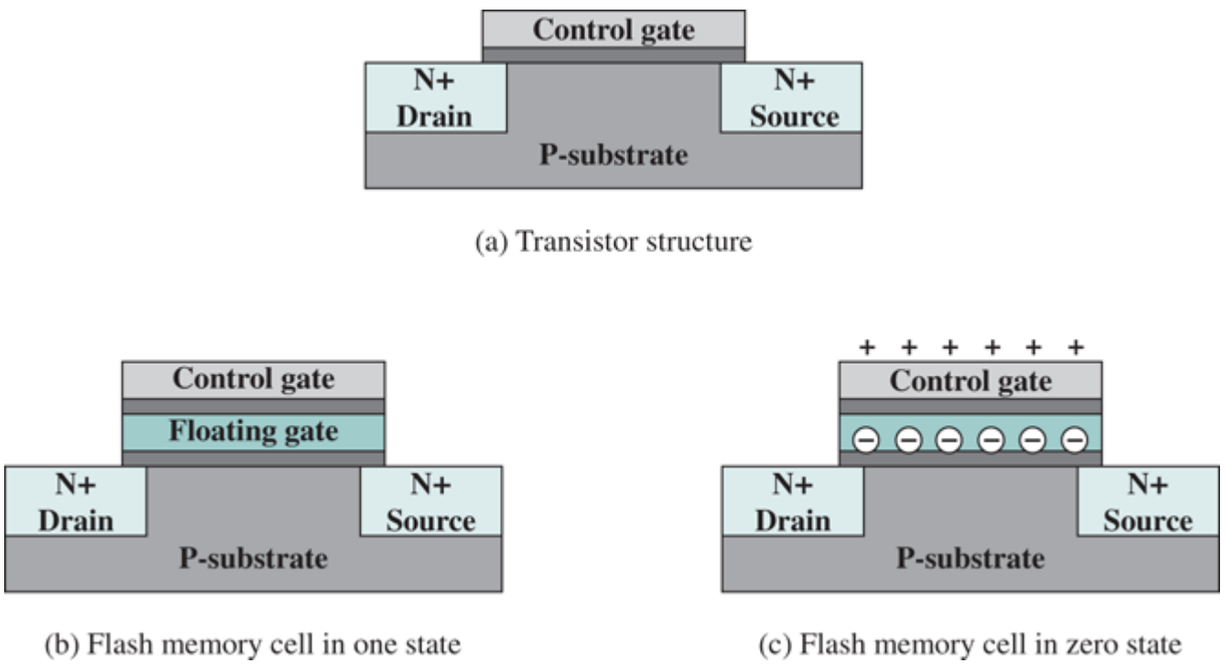


Figure 6.17 Flash Memory Operation

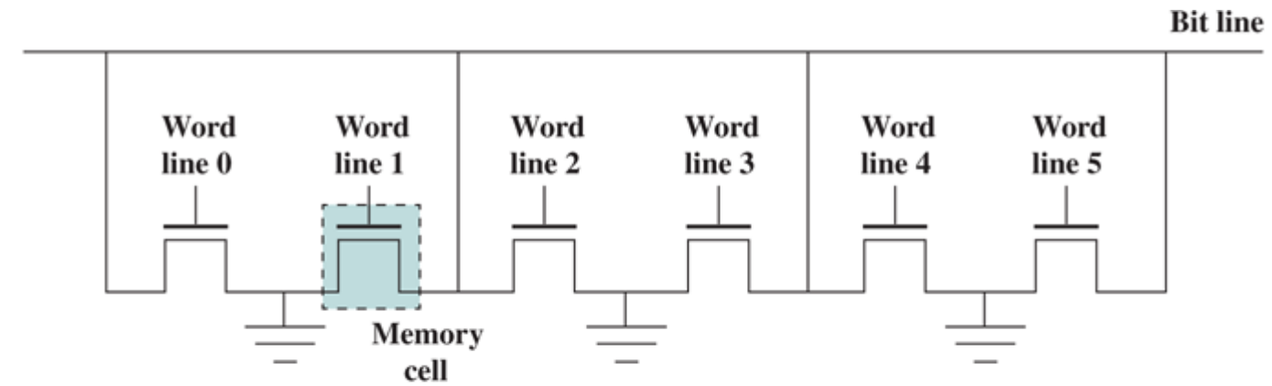
In a flash memory cell, a second gate—called a floating gate, because it is insulated by a thin oxide layer—is added to the transistor. Initially, the floating gate does not interfere with the operation of the transistor (**Figure 6.17b**). In this state, the cell is deemed to represent binary 1. Applying a large voltage across the oxide layer causes electrons to tunnel through it and become trapped on the floating gate, where they remain even if the power is disconnected (**Figure 6.17c**). In this state, the cell is deemed to represent binary 0. The state of the cell can be read by using external circuitry to test whether the transistor is working or not. Applying a large voltage in the opposite direction removes the electrons from the floating gate, returning to a state of binary 1.

An important characteristic of flash memory is that it is persistent memory, which means that it retains data when there is no power applied to the memory. Thus, it is useful for secondary (external) storage, and as an alternative to random access memory in computers.

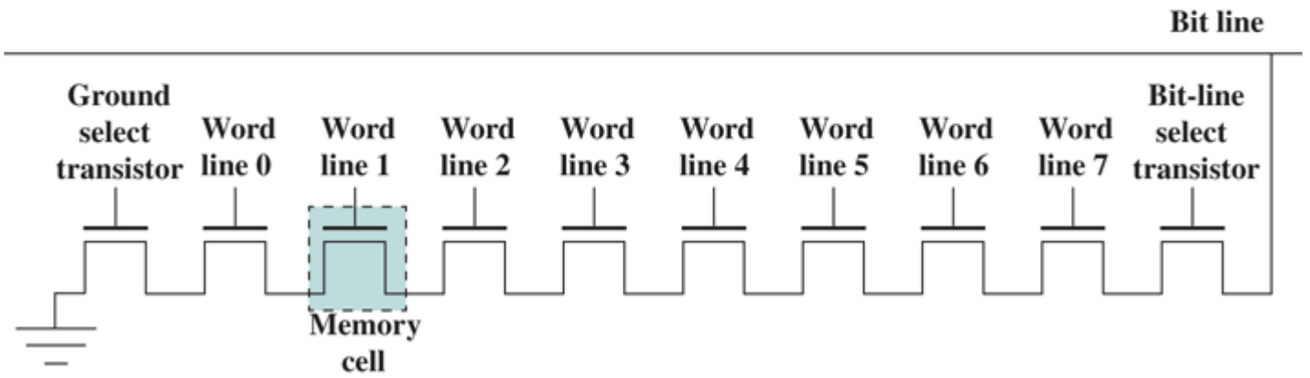
NOR and NAND Flash Memory

There are two distinctive types of flash memory, designated as NOR and NAND (**Figure 6.18**). In **NOR flash memory**, the basic unit of access is a bit, referred to as a *memory cell*. Cells in NOR flash are connected in parallel to the bit lines so that each cell can be read/write/erased individually. If any memory cell of the device is turned on by the corresponding word line, the bit line goes low. This is similar in function to a NOR logic gate.[2]

[2] See Chapter 12 for a discussion of NOR and NAND gates.



(a) NOR flash structure



(b) NAND flash structure

Figure 6.18 Flash Memory Structures

NAND flash memory is organized in transistor arrays with 16 or 32 transistors in series. The bit line goes low only if all the transistors in the corresponding word lines are turned on. This is similar in function to a NAND logic gate.

Although the specific quantitative values of various characteristics of NOR and NAND are changing year by year, the relative differences between the two types has remained stable. These differences are usefully illustrated by the Kiviat graphs[3] shown in **Figure 6.19**.

[3] A Kiviat graph provides a pictorial means of comparing systems along multiple variables [MORR74]. The variables are laid out at as lines of equal angular intervals within a circle, each line going from the center of the circle to the circumference. A given system is defined by one point on each line; the closer to the circumference, the better the value. The points are connected to yield a shape that is characteristic of that system. The more area enclosed in the shape, the "better" is the system.

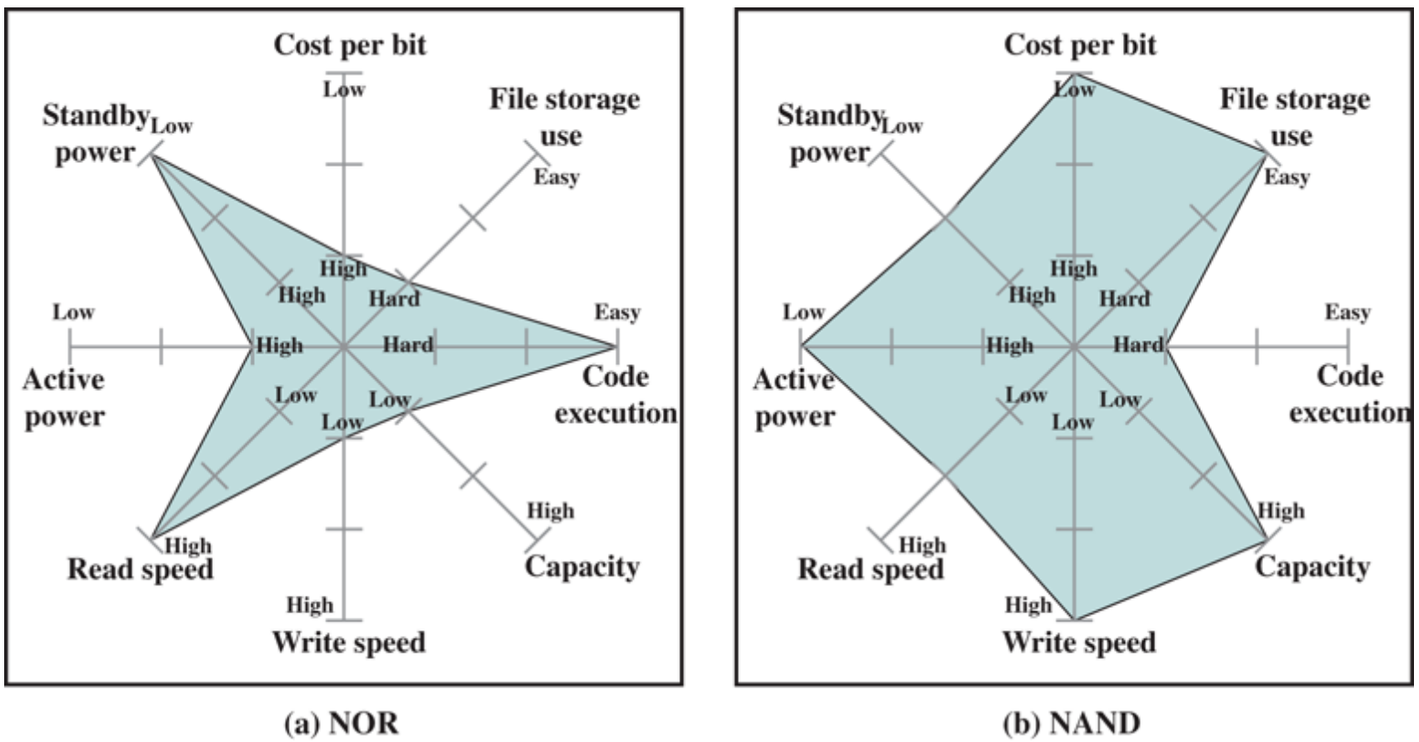


Figure 6.19 Kiviat Graphs for Flash Memory

NOR flash memory provides high-speed random access. It can read and write data to specific locations, and can reference and retrieve a single byte. NAND reads and writes in small blocks. NAND provides higher bit density than NOR and greater write speed. NAND flash does not provide a random-access external address bus, so the data must be read on a blockwise basis (also known as page access), where each block holds hundreds to thousands of bits.

For internal memory in embedded systems, NOR flash memory has traditionally been preferred. NAND memory has made some inroads, but NOR remains the dominant technology for internal memory. It is ideally suited for microcontrollers where the amount of program code is relatively small and a certain amount of application data does not vary. For example, the flash memory in **Figure 1.16** is NOR memory.

NAND memory is better suited for external memory, such as USB flash drives, memory cards (in digital cameras, MP3 players, etc.), and in what are known as solid-state disks (SSDs). We discuss SSDs in **Chapter 7**.

6.6 Newer Nonvolatile Solid-State Memory Technologies

The traditional memory hierarchy has consisted of three levels (Figure 5.20):

- **Static RAM (SRAM):** SRAM provides rapid access time, but is the most expensive and the least dense (bit density). SRAM is suitable for cache memory.
- **Dynamic RAM (DRAM):** Cheaper, denser, and slower than SRAM, DRAM has traditionally been the choice for off-chip main memory.
- **Hard disk:** A magnetic disk provides very high bit density and very low cost per bit, with relatively slow access times. It is the traditional choice for external storage as part of the memory hierarchy.

Into this mix, as we have seen, has been added flash memory. Flash memory has the advantage over traditional memory that it is nonvolatile. NOR flash is best suited to storing programs and static application data in embedded systems, while NAND flash has characteristics intermediate between DRAM and hard disks.

Over time, each of these technologies has seen improvements in scaling: higher bit density, higher speed, lower power consumption, and lower cost. However, for semiconductor memory, it is becoming increasingly difficult to continue the pace of improvement [ITRS14].

Recently, there have been breakthroughs in developing new forms of nonvolatile semiconductor memory that continue scaling beyond flash memory. The most promising technologies are spin-transfer torque RAM (STT-RAM), phase-change RAM (PCRAM), and resistive RAM (ReRAM) ([ITRS14], [GOER12]). All of these are in volume production. However, because NAND Flash and to some extent NOR Flash are still dominating the applications, these emerging memories have been used in specialty applications and have not yet fulfilled their original promise to become dominating mainstream high-density **nonvolatile memory.** This is likely to change in the next few years.

Figure 6.20 shows how these three technologies are likely to fit into the memory hierarchy.

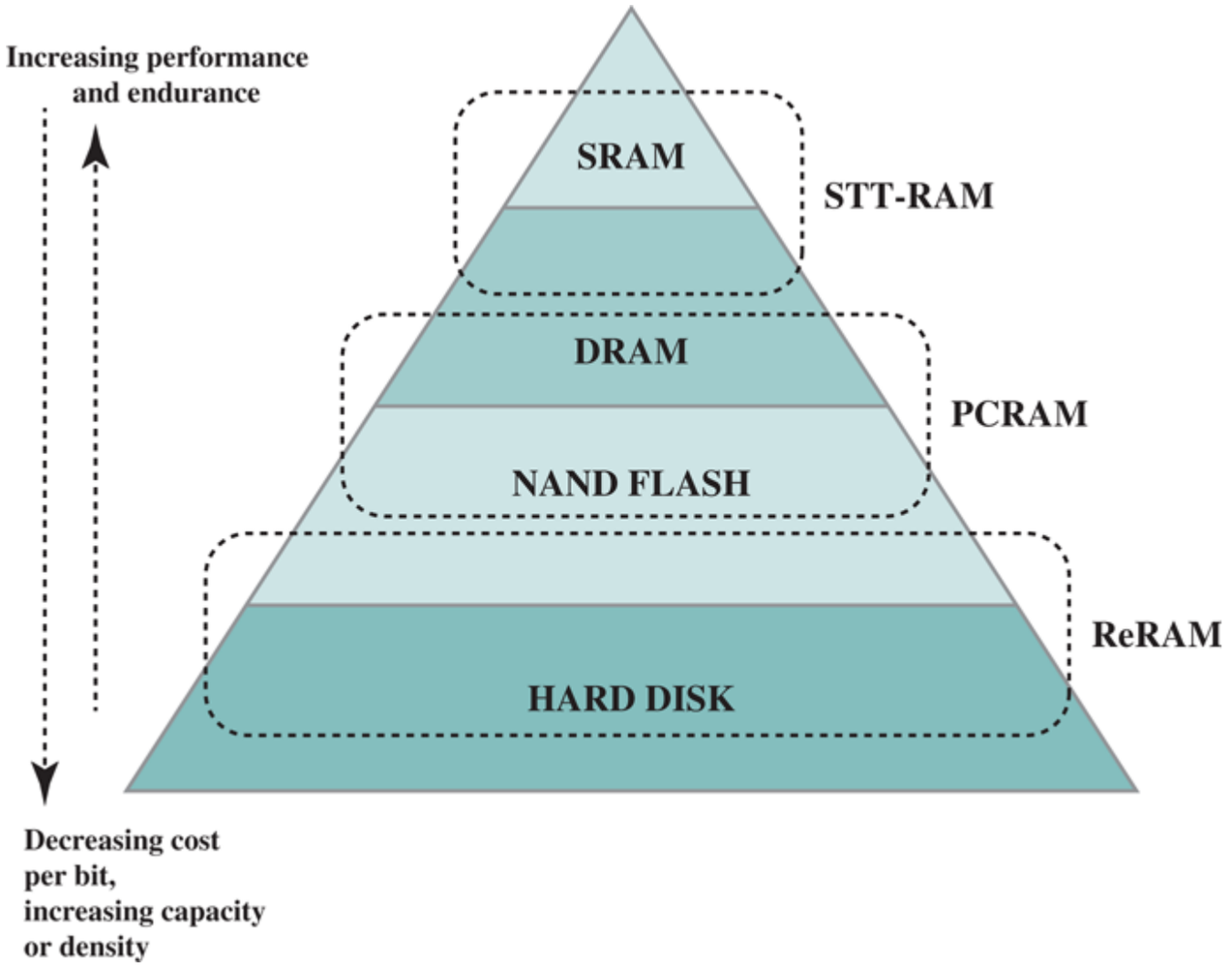


Figure 6.20 Nonvolatile RAM within the Memory Hierarchy

STT-RAM

STT-RAM is a new type of **magnetic RAM (MRAM)**, which features non-volatility, fast writing/reading speed $(<10\text{ns})$, high programming endurance $(>10^{15}\text{cycles})$ and zero standby power [KULT13]. The storage capability or programmability of MRAM arises from magnetic tunneling junction (MTJ), in which a thin tunneling dielectric is sandwiched between two ferromagnetic layers. One ferromagnetic layer (pinned or reference layer) is designed to have its magnetization pinned, while the magnetization of the other layer (free layer) can be flipped by a write event. An MTJ has a low (high) resistance if the magnetizations of the free layer and the pinned layer are parallel (anti-parallel). In first-generation MRAM design, the magnetization of the free layer is changed by the current-induced magnetic field. In STT-RAM, a new write mechanism, called *polarization-current-induced magnetization switching*, is introduced. For STT-RAM, the magnetization of the free layer is flipped by the electrical current directly. Because the current required to switch an MTJ resistance state is proportional to the MTJ cell area, STT-RAM is believed to have a better scaling property than the first-generation MRAM. **Figure 6.21a** illustrates the general configuration.

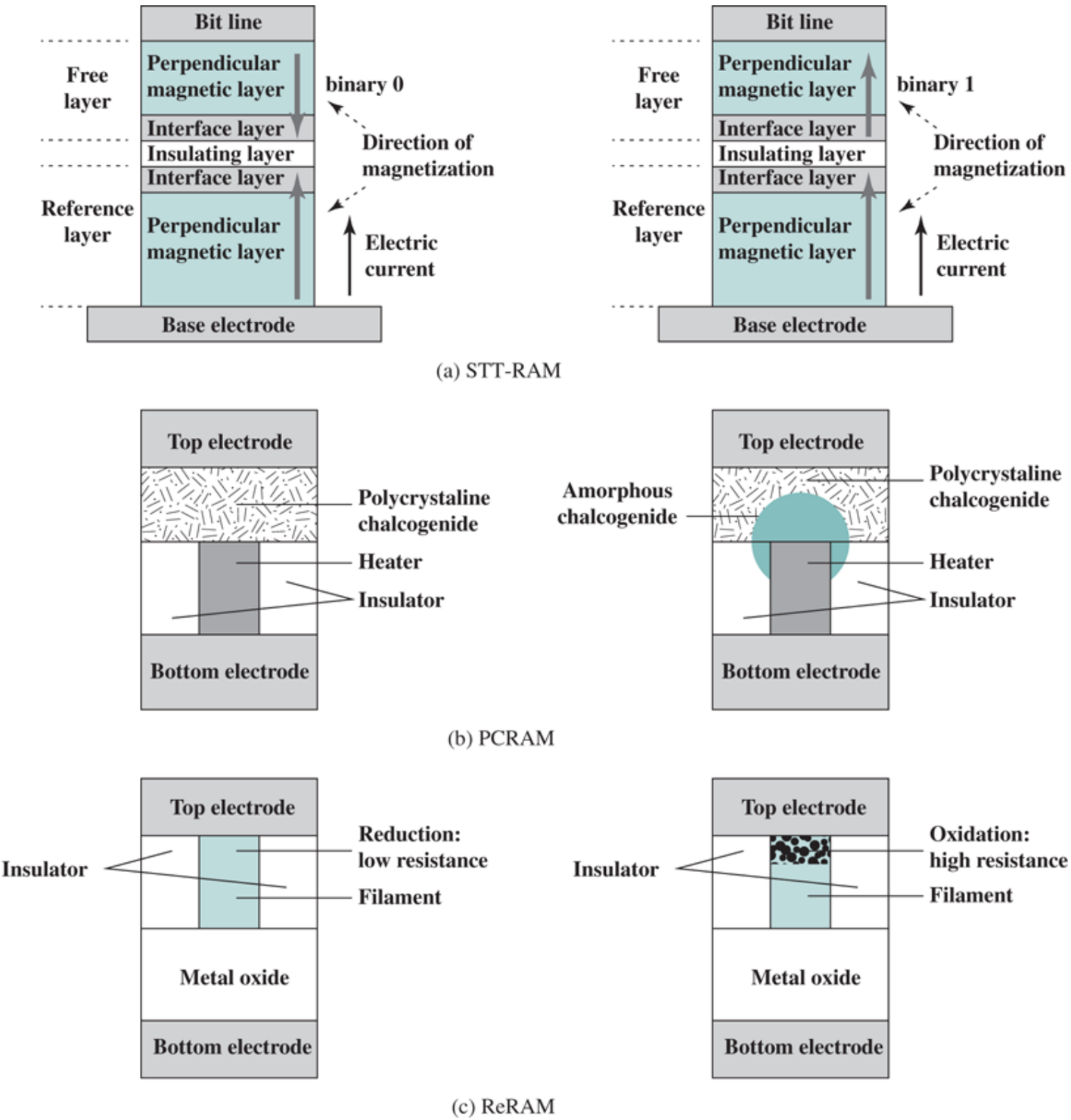


Figure 6.21 Nonvolatile RAM Technologies

STT-RAM is a good candidate for either cache or main memory.

PCRAM

Phase-change RAM (pcram) is the most mature of the new technologies, with an extensive technical literature ([RAOU09], [ZHOU09], [LEE10]).

PCRAM technology is based on a chalcogenide alloy material, which is similar to those commonly

used in optical storage media (compact discs and digital versatile discs). The data storage capability is achieved from the resistance differences between an amorphous (high-resistance) and a crystalline (low-resistance) phase of the chalcogenide-based material. In SET operation, the phase change material is crystallized by applying an electrical pulse that heats a significant portion of the cell above its crystallization temperature. In RESET operation, a larger electrical current is applied and then abruptly cut off in order to melt and then quench the material, leaving it in the amorphous state. **Figure 6.21b** illustrates the general configuration.

PCRAM is a good candidate to replace or supplement DRAM for main memory.

ReRAM

ReRAM (also known as RRAM) works by creating resistance rather than directly storing charge. An electric current is applied to a material, changing the resistance of that material. The resistance state can then be measured and a 1 or 0 is read as the result. Much of the work done on ReRAM to date has focused on finding appropriate materials and measuring the resistance state of the cells. ReRAM designs are low voltage, endurance is far superior to flash memory, and the cells are much smaller—at least in theory. **Figure 6.21c** shows one ReRam configuration.

ReRAM is a good candidate to replace or supplement both secondary storage and main memory.

6.7 Key Terms, Review Questions, and Problems

Key Terms

bank group

double data rate DRAM (DDR DRAM)

dynamic RAM (DRAM)

electrically erasable programmable ROM (EEPROM)

erasable programmable ROM (EPROM)

error correcting code (ECC)

error correction

flash memory

Hamming code

hard failure

magnetic RAM (MRAM)

NAND flash memory

nonvolatile memory

NOR flash memory

phase-change RAM (PCRAM)

programmable ROM (PROM)

random access memory (RAM)

read-mostly memory

read-only memory (ROM)

resistive RAM (ReRAM)

semiconductor memory

single-error-correcting (SEC) code

single-error-correcting, double-error-detecting (SEC-DED) code

soft error

spin-transfer torque RAM (STT-RAM)

static RAM (SRAM)

synchronous DRAM (SDRAM)

syndrome

timing diagram

volatile memory

Review Questions

6.1 What are the key properties of semiconductor memory?
6.2 What are two interpretations of the term *random-access memory*?
6.3 What is the difference between DRAM and SRAM in terms of application?
6.4 What is the difference between DRAM and SRAM in terms of characteristics such as speed, size, and cost?
6.5 Explain why one type of RAM is considered to be analog and the other digital.
6.6 What are some applications for ROM?
6.7 What are the differences among EPROM, EEPROM, and flash memory?
6.8 Explain the function of each pin in **Figure 5.4b** .
6.9 What is a parity bit?
6.10 How is the syndrome for the Hamming code interpreted?
6.11 How does SDRAM differ from ordinary DRAM?
6.12 What is DDR RAM?
6.13 What is the difference between NAND and NOR flash memory?
6.14 List and briefly define three newer nonvolatile solid-state memory technologies.

Problems

6.1 Suggest reasons why RAMs traditionally have been organized as only one bit per chip whereas ROMs are usually organized with multiple bits per chip.
6.2 Consider a dynamic RAM that must be given a refresh cycle 64 times per ms. Each refresh operation requires 150 ns; a memory cycle requires 250 ns. What percentage of the memory's total operating time must be given to refreshes?
6.3 **Figure 6.22** shows a simplified timing diagram for a DRAM read operation over a bus. The access time is considered to last from t_1 to t_2. Then there is a recharge time, lasting from t_2 to t_3, during which the DRAM chips will have to recharge before the processor can access them again.

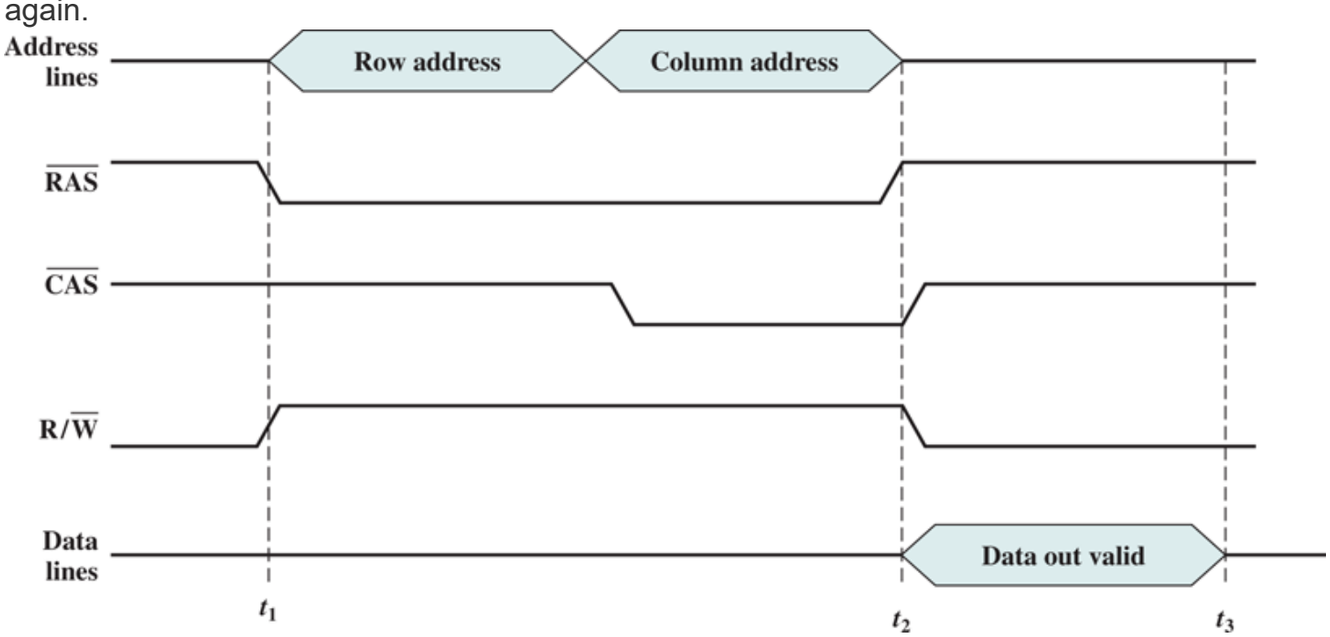


Figure 6.22 Simplified DRAM Read Timing

a. Assume that the access time is 60 ns and the recharge time is 40 ns. What is the memory cycle time? What is the maximum data rate this DRAM can sustain, assuming a 1-bit output?
b. Constructing a 32-bit wide memory system using these chips yields what data transfer rate?

6.4 **Figure 6.6** indicates how to construct a module of chips that can store 1 MB based on a group of four 256-Kbyte chips. Let's say this module of chips is packaged as a single 1-MB chip, where the word size is 1 byte. Give a high-level chip diagram of how to construct an 8-MB computer memory using eight 1-MB chips. Be sure to show the address lines in your diagram and what the address lines are used for.

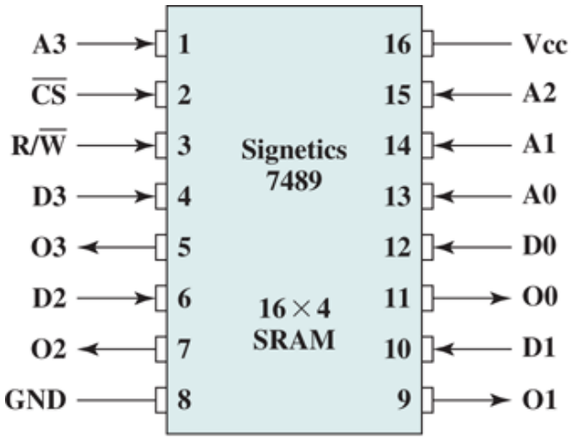
6.5 On a typical Intel 8086-based system, connected via system bus to DRAM memory, for a read operation, $\overline{\text{RAS}}$ is activated by the trailing edge of the Address Enable signal (**Figure A.1** in **Appendix A**). However, due to propagation and other delays, $\overline{\text{RAS}}$ does not go active until 50 ns after Address Enable returns to a low. Assume the latter occurs in the middle of the second half of state T_1 (somewhat earlier than in **Figure A.1**). Data are read by the processor at the end of T_3. For timely presentation to the processor, however, data must be provided 60 ns earlier by memory. This interval accounts for propagation delays along the data paths (from memory to processor) and processor data hold time requirements. Assume a clocking rate of 10 MHz.

a. How fast (access time) should the DRAMs be if no wait states are to be inserted?
b. How many wait states do we have to insert per memory read operation if the access time of the DRAMs is 150 ns?

6.6 The memory of a particular microcomputer is built from $64\text{K} \times 1\text{DRAMs}$. According to the data sheet, the cell array of the DRAM is organized into 256 rows. Each row must be refreshed at least once every 4 ms. Suppose we refresh the memory on a strictly periodic basis.

a. What is the time period between successive refresh requests?
b. How long a refresh address counter do we need?

6.7 **Figure 6.23** shows one of the early SRAMs, the 16×4 Signetics 7489 chip, which stores 16 4-bit words.

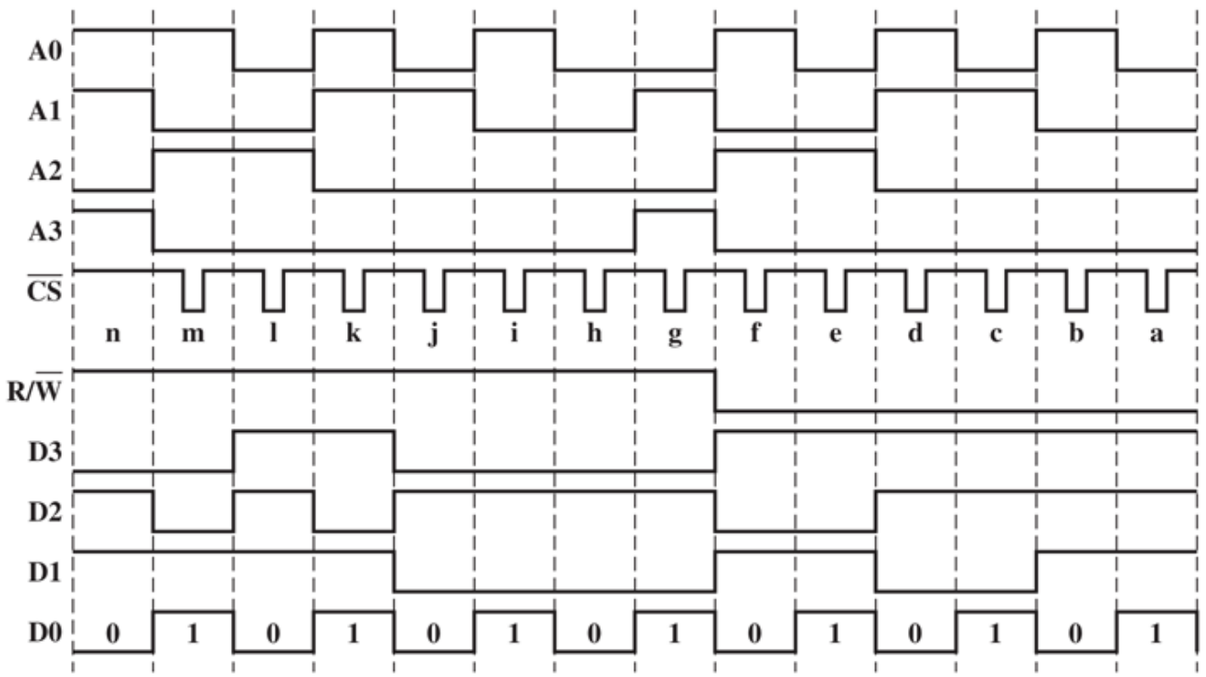


(a) Pin layout

Operating Mode	Inputs			Outputs
	$\overline{CS}$	R/$\overline{W}$	D*n*	O*n*
Write	L	L	L	L
	L	L	H	H
Read	L	H	X	Data
Inhibit writing	H	L	L	H
	H	L	H	L
Store - disable outputs	H	H	X	H

H = high voltage level
L = low voltage level
X = don't care

(b) Truth table



(c) Pulse train

Figure 6.23 The Signetics 7489 SRAM

a. List the mode of operation of the chip for each CS input pulse shown in **Figure 6.23c** .
b. List the memory contents of word locations 0 through 6 after pulse n.
c. What is the state of the output data leads for the input pulses h through m?

6.8 Design a 16-bit memory of total capacity 8192 bits using SRAM chips of size 64×1 bit. Give the array configuration of the chips on the memory board showing all required input and output signals for assigning this memory to the lowest address space. The design should allow for both byte and 16-bit word accesses.

6.9 A common unit of measure for failure rates of electronic components is the **Failure unIT** (FIT), expressed as a rate of failures per billion device hours. Another well known but less used measure is **mean time between failures (MTBF),** which is the average time of operation of a particular component until it fails. Consider a 1 MB memory of a 16-bit microprocessor with $256\text{K} \times 1$ DRAMs. Calculate its MTBF assuming 2000 FITS for each DRAM.

6.10 For the Hamming code shown in **Figure 6.10** , show what happens when a check bit rather than a data bit is in error?

6.11 Suppose an 8-bit data word stored in memory is 11000010. Using the Hamming algorithm, determine what check bits would be stored in memory with the data word. Show how you got your answer.

6.12 For the 8-bit word 00111001, the check bits stored with it would be 0111. Suppose when the word is read from memory, the check bits are calculated to be 1101. What is the data word that was read from memory?

6.13 How many check bits are needed if the Hamming error correction code is used to detect single bit errors in a 1024-bit data word?

6.14 Develop an SEC code for a 16-bit data word. Generate the code for the data word 0101000000111001. Show that the code will correctly identify an error in data bit 5.

Chapter 7 External Memory

Learning Objectives

After studying this chapter, you should be able to:

- Understand the key properties of magnetic disks.
- Understand the performance issues involved in **magnetic disk** access.
- Explain the concept of **RAID** and describe the various levels.
- Compare and contrast hard disk drives and solid disk drives.
- Describe in general terms the operation of **flash memory**.
- Understand the differences among the different optical disk storage media.
- Present an overview of **magnetic tape** storage technology.

This chapter examines a range of external memory devices and systems. We begin with the most important device, the magnetic disk. Magnetic disks are the foundation of external memory on virtually all computer systems. The next section examines the use of disk arrays to achieve greater performance, looking specifically at the family of systems known as RAID (Redundant Array of Independent Disks). An increasingly important component of many computer systems is the solid state disk, which is discussed next. Then, external ***optical memory*** *is examined. Finally, magnetic tape is described.*