Click Here

This book is intended primarily as a text for computer science and electrical engineering courses at the advanced undergraduate or beginning graduate levels. While its emphasis is on computer hardware and systems, the relevant aspects of software are also treated. Set of rules describing computer system This article's lead section may be too short to adequately summarize the key points. Please consider expanding the lead to provide an accessible overview of all important aspects of the article. (November 2023) Block diagram of a basic computer with uniprocessor CPU. Black lines indicate control flow, whereas red lines indicate data flow. Arrows indicate the direction of flow. In computer science and computer engineering, computer architecture is a description of the structure of a computer system made from component parts.[1] It can sometimes be a high-level description that ignores details of the implementation.[2] At a more detailed level, the description may include the instruction set architecture design, microarchitecture design, logic design, and implementation.[3] The first documented computer architecture was in the correspondence between Charles Babbage and Ada Lovelace, describing the analytical engine. While building the computer Z1 in 1936, Konrad Zuse described in two patent applications for his future projects that machine instructions could be stored in the same storage used for data, i.e., the stored-program concept.[4][5] Two other early and important examples are: John von Neumann's 1945 paper, First Draft of a Report on the EDVAC, which described an organization of logical elements;[6] and Alan Turing's more detailed Proposed Electronic Calculator for the Automatic Computing Engine, also 1945 and which cited John von Neumann's paper.[7] The term "architecture" in computer literature can be traced to the work of Lyle R. Johnson and Frederick P. Brooks, Jr., members of the Machine Organization department in IBM's main research center in 1959. Johnson had the opportunity to write a proprietary research communication about the Stretch, an IBM-developed supercomputer for Los Alamos National Laboratory (at the time known as Los Alamos Scientific Laboratory). To describe the level of detail for discussing the luxuriously embellished computer, he noted that his description of formats, instruction types, hardware parameters, and speed enhancements were at the level of "system architecture", a term that seemed more useful than "machine organization".[8] Subsequently, Brooks, a Stretch designer, opened Chapter 2 of a book called Planning a Computer System: Project Stretch by stating, "Computer architecture, like other architecture, is the art of determining the needs of the user of a structure and then designing to meet those needs as effectively as possible within economic and technological constraints."[9] Brooks went on to help develop the IBM System/360 line of computers, in which "architecture" became a noun defining "what the user needs to know".[10] The System/360 line was succeeded by several compatible lines of computers, including the current IBM Z line. Later, computer users came to use the term in many less explicit ways.[11] The earliest computer architectures were designed on paper and then directly built into the final hardware form.[12] Later, computer architecture prototypes were physically built in the form of a transistor–transistor logic (TTL) computer—such as the prototypes of the 6800 and the PA-RISC—tested, and tweaked, before committing to the final hardware form. As of the 1990s, new computer architectures are typically "built", tested, and tweaked—inside some other computer architecture in a computer architecture simulator; or inside a FPGA as a soft microprocessor; or both—before committing to the final hardware form.[13] The discipline of computer architecture has three main subcategories:[14] Instruction set architecture (ISA): defines the machine code that a processor reads and acts upon as well as the word size, memory address modes, processor registers, and data type. Microarchitecture: also known as "computer organization", this describes how a particular processor will implement the ISA.[15] The size of a computer's CPU cache for instance, is an issue that generally has nothing to do with the ISA. Systems design: includes all of the other hardware components within a computing system, such as data processing other than the CPU (e.g., direct memory access), virtualization, and multiprocessing. There are other technologies in computer architecture. The following technologies are used in bigger companies like Intel, and were estimated in 2002[14] to count for 1% of all of computer architecture: Macroarchitecture: architectural layers more abstract than microarchitecture Assembly instruction set architecture: A smart assembler may convert an abstract assembly language common to a group of machines into slightly different machine language for different implementations. Programmer-visible macroarchitecture: higher-level language tools such as compilers may define a consistent interface or contract to programmers using them, abstracting differences between underlying ISAs and microarchitectures. For example, the C, C++, or Java standards define different programmer-visible macroarchitectures. Microcode: microcode is software that translates instructions to run on a chip. It acts like a wrapper around the hardware, presenting a preferred version of the hardware's instruction set interface. This instruction translation facility gives chip designers flexible options: E.g. 1. A new improved version of the chip can use microcode to present the exact same instruction set as the old chip version, so all software targeting that instruction set will run on the new chip without needing changes. E.g. 2. Microcode can present a variety of instruction sets for the same underlying chip, allowing it to run a wider variety of software. Pin architecture: The hardware functions that a microprocessor should provide to a hardware platform, e.g., the x86 pins A20M, FERR/IGNNE or FLUSH. Also, messages that the processor should emit so that external caches can be invalidated (emptied). Pin architecture functions are more flexible than ISA functions because external hardware can adapt to new encodings, or change from a pin to a message. The term "architecture" fits, because the functions must be provided for compatible systems, even if the detailed method changes. Computer architecture is concerned with balancing the performance, efficiency, cost, and reliability of a computer system. The case of instruction set architecture can be used to illustrate the balance of these competing factors. More complex instruction sets enable programmers to write more space efficient programs, since a single instruction can encode some higher-level abstraction (such as the x86 Loop instruction).[16] However, longer and more complex instructions take longer for the processor to decode and can be more costly to implement effectively. The increased complexity from a large instruction set also creates more room for unreliability when instructions interact in unexpected ways. The implementation involves integrated circuit design, packaging, power, and cooling. Optimization of the design requires familiarity with topics from compilers and operating systems to logic design and packaging.[17] Main article: Instruction set architecture This section does not cite any sources. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. (March 2018) (Learn how and when to remove this message) An instruction set architecture (ISA) is the interface between the computer's software and hardware and also can be viewed as the programmer's view of the machine. Computers do not understand high-level programming languages such as Java, C++, or most programming languages used. A processor only understands instructions encoded in some numerical fashion, usually as binary numbers. Software tools, such as compilers, translate those high level languages into instructions that the processor can understand. Besides instructions, the ISA defines items in the computer that are available to a program—e.g., data types, registers, addressing modes, and memory. Instructions locate these available items with register indexes (or names) and memory addressing modes. The ISA of a computer is usually described in a small instruction manual, which describes how the instructions are encoded. Also, it may define short (vaguely) mnemonic names for the instructions. The names can be recognized by a software development tool called an assembler. An assembler is a computer program that translates a human-readable form of the ISA into a computer-readable form. Disassemblers are also widely available, usually in debuggers and software programs to isolate and correct malfunctions in binary computer programs. ISAs vary in quality and completeness. A good ISA compromises between programmer convenience (how easy the code is to understand), size of the code (how much code is required to do a specific action), cost of the computer to interpret the instructions (more complexity means more hardware needed to decode and execute the instructions), and speed of the computer (with more complex decoding hardware comes longer decode time). Memory organization defines how instructions interact with the memory, and how memory interacts with itself. During design emulation, emulators can run programs written in a proposed instruction set. Modern emulators can measure size, cost, and speed to determine whether a particular ISA is meeting its goals. Main article: Microarchitecture Computer organization helps optimize performance-based products. For example, software engineers need to know the processing power of processors. They may need to optimize software in order to gain the most performance for the lowest price. This can require quite a detailed analysis of the computer's organization. For example, in an SD card, the designers might need to arrange the card so that the most data can be processed in the fastest possible way. Computer organization also helps plan the selection of a processor for a particular project. Multimedia projects may need very rapid data access, while virtual machines may need fast interrupts. Sometimes certain tasks need additional components as well. For example, a computer capable of running a virtual machine needs virtual memory hardware so that the memory of different virtual computers can be kept separated. Computer organization and features also affect power consumption and processor cost. Once an instruction set and microarchitecture have been designed, a practical machine must be developed. This design process is called the implementation. Implementation is usually not considered architectural design, but rather hardware design engineering. Implementation can be further broken down into several steps: Logic implementation designs the circuits required at a logic-gate level. Circuit implementation does transistor-level designs of basic elements (e.g., gates, multiplexers, latches) as well as of some larger blocks (ALUs, caches etc.) that may be implemented at the logic-gate level, or even at the physical level if the design calls for it. Physical implementation draws physical circuits. The different circuit components are placed in a chip floor plan or on a board and the wires connecting them are created. Design validation tests the computer as a whole to see if it works in all situations and all timings. Once the design validation process starts, the design at the logic level are tested using logic emulators. However, this is usually too slow to run a realistic test. So, after making corrections based on the first test, prototypes are constructed using Field-Programmable Gate-Arrays (FPGAs). Most hobby projects stop at this stage. The final step is to test prototype integrated circuits, which may require several redesigns. For CPUs, the entire implementation process is organized differently and is often referred to as CPU design. The exact form of a computer system depends on the constraints and goals. Computer architectures usually trade off standards, power versus performance, cost, memory capacity, latency (latency is the amount of time that it takes for information from one node to travel to the source) and throughput. Sometimes other considerations, such as features, size, weight, reliability, and expandability are also factors. The most common scheme does an in-depth power analysis and figures out how to keep power consumption low while maintaining adequate performance. Modern computer performance is often described in instructions per cycle (IPC), which measures the efficiency of the architecture at any clock frequency; a faster IPC rate means the computer is faster. Older computers had IPC counts as low as 0.1 while modern processors easily reach nearly 1. Superscalar processors may reach three to five IPC by executing several instructions per clock cycle.[citation needed] Counting machine-language instructions would be misleading because they can do varying amounts of work in different ISAs. The "instruction" in the standard measurements is not a count of the ISA's machine-language instructions, but a unit of measurement, usually based on the speed of the VAX computer architecture. Many people used to measure a computer's speed by the clock rate (usually in MHz or GHz). This refers to the cycles per second of the main clock of the CPU. However, this metric is somewhat misleading, as a machine with a higher clock rate may not necessarily have greater performance. As a result, manufacturers have moved away from clock speed as a measure of performance. Other factors influence speed, such as the mix of functional units, bus speeds, available memory, and the type and order of instructions in the programs. There are two main types of speed: latency and throughput. Latency is the time between the start of a process and its completion. Throughput is the amount of work done per unit time. Interrupt latency is the guaranteed maximum response time of the system to an electronic event (like when the disk drive finishes moving some data). Performance is affected by a very wide range of design choices — for example, pipelining a processor usually makes latency worse, but makes throughput better. Computers that control machinery usually need low interrupt latencies. These computers operate in a real-time environment and fail if an operation is not completed in a specified amount of time. For example, computer-controlled anti-lock brakes must begin braking within a predictable and limited time period after the brake pedal is sensed or else failure of the brake will occur. Benchmarking takes all these factors into account by measuring the time a computer takes to run through a series of test programs. Although benchmarking shows strengths, it should not be how you choose a computer. Often the measured machines split on different measures. For example, one system might handle scientific applications quickly, while another might render video games more smoothly. Furthermore, designers may target and add special features to their products, through hardware or software, that permit a specific benchmark to execute quickly but do not offer similar advantages to general tasks. Main articles: Low-power electronics and Performance per watt Power efficiency is another important measurement in modern computers. Higher power efficiency can often be traded for lower speed or higher cost. The typical measurement when referring to power consumption in computer architecture is MIPS/W (millions of instructions per second per watt). Modern circuits have less power required per transistor as the number of transistors per chip grows.[18] This is because each transistor that is put in a new chip requires its own power supply and requires new pathways to be built to power it.[clarification needed] However, the number of transistors per chip is starting to increase at a slower rate. Therefore, power efficiency is starting to become as important, if not more important than fitting more and more transistors into a single chip. Recent processor designs have shown this emphasis as they put more focus on power efficiency rather than cramming as many transistors into a single chip as possible.[19] In the world of embedded computers, power efficiency has long been an important goal next to throughput and latency. Increases in clock frequency have grown more slowly over the past few years, compared to power reduction improvements. This has been driven by the end of Moore's Law and demand for longer battery life and reductions in size for mobile technology. This change in focus from higher clock rates to power consumption and miniaturization can be shown by the significant reductions in power consumption, as much as 50%, that were reported by Intel in their release of the Haswell microarchitecture; where they dropped their power consumption benchmark from 30–40 watts down to 10–20 watts.[20] Comparing this to the processing speed increase of 3 GHz to 4 GHz (2002 to 2006), it can be seen that the focus in research and development is shifting away from clock frequency and moving towards consuming less power and taking up less space.[21] Electronics portal Bit-serial architecture Comparison of CPU architectures Computer hardware CPU design Dataflow architecture Floating point Flynn's taxonomy Harvard architecture (Modified) Influence of the IBM PC on the personal computer market Orthogonal instruction set Reconfigurable computing Software architecture Transport triggered architecture Von Neumann architecture

^ Dragoni, Nicole (n.d.). "Introduction to peer to peer computing" (PDF). DTU Compute – Department of Applied Mathematics and Computer Science. Lyngby, Denmark. ^ Clements, Alan. Principles of Computer Hardware (Fourth ed.). p. 1. Architecture describes the internal organization of a computer in an abstract way; that is, it defines the capabilities of the computer and its programming model. You can have two computers that have been constructed in different ways with different technologies but with the same architecture. ^ Hennessy, John; Patterson, David. Computer Architecture: A Quantitative Approach (Fifth ed.). p. 11. This task has many aspects, including instruction set design, functional organization, logic design, and implementation. ^ Williams, F. C.; Kilburn, T. (25 September 1948), "Electronic Digital Computers", Nature, 162 (4117): 487, Bibcode:1948Natur.162..487W, doi:10.1038/162487a0, S2CID 4110351 ^ Susanne Faber, "Konrad Zuses Bemuehungen um die Patentanmeldung der Z3", 2000 ^ Neumann, John (1945). First Draft of a Report on the EDVAC. p. 9. ^ Reproduced in B. J. Copeland (Ed.), "Alan Turing's Automatic Computing Engine", Oxford University Press, 2005, pp. 369-454. ^ Johnson, Lyle (1960). "A Description of Stretch" (PDF). p. 1. Retrieved 7 October 2017. ^ Buchholz, Werner (1962). Planning a Computer System. p. 5. ^ "System 360, From Computers to Computer Systems". IBM100. 7 March 2012. Archived from the original on April 3, 2012. Retrieved 11 May 2017. ^ Hellige, Hans Dieter (2004). "Die Genese von Wissenschaftskonzeptionen der Computerarchitektur: Vom "system of organs" zum Schichtmodell des Designraums". Geschichten der Informatik: Visionen, Paradigmen, Leitmotive. pp. 411–472. ^ ACE underwent seven paper designs in one year, before a prototype was initiated in 1948. [B. J. Copeland (Ed.), "Alan Turing's Automatic Computing Engine", OUP, 2005, p. 57] ^ Schmalz, M.S. "Organization of Computer Systems". UF CISE. Retrieved 11 May 2017. ^ a b John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach (Third ed.). Morgan Kaufmann Publishers. ^ Laplante, Phillip A. (2001). Dictionary of Computer Science, Engineering, and Technology. CRC Press. pp. 94–95. ISBN 0-8493-2691-5. ^ Null, Linda (2019). The Essentials of Computer Organization and Architecture (5th ed.). Burlington, MA: Jones & Bartlett Learning. p. 280. ISBN 9781284123036. ^ Martin, Milo. "What is computer architecture?" (PDF). UPENN. Retrieved 11 May 2017. ^ "Integrated circuits and fabrication" (PDF). Retrieved 8 May 2017. ^ "Exynos 9 Series (8895)". Samsung. Retrieved 8 May 2017. ^ "Measuring Processor Power TDP vs ACP" (PDF). Intel. April 2011. Retrieved 5 May 2017. ^ "History of Processor Performance" (PDF). cs.columbia.edu. 24 April 2012. Retrieved 5 May 2017. John L. Hennessy and David Patterson (2006). Computer Architecture: A Quantitative Approach (Fourth ed.). Morgan Kaufmann. ISBN 978-0-12-370490-0. Barton, Robert S., "Functional Design of Computers", Communications of the ACM 4(9): 405 (1961). Barton, Robert S., "A New Approach to the Functional Design of a Digital Computer", Proceedings of the Western Joint Computer Conference, May 1961, pp. 393–396. About the design of the Burroughs B5000 computer. Bell, C. Gordon; and Newell, Allen (1971). "Computer Structures: Readings and Examples", McGraw-Hill. Blaauw, G.A., and Brooks, F.P., Jr., "The Structure of System/360, Part I-Outline of the Logical Structure", IBM Systems Journal, vol. 3, no. 2, pp. 119–135, 1964. Tanenbaum, Andrew S. (1979). Structured Computer Organization. Englewood Cliffs, New Jersey: Prentice-Hall. ISBN 0-13-148521-0. Wikimedia Commons has media related to Computer architecture. Carnegie Mellon Computer Architecture Lectures ISCA: Proceedings of the International Symposium on Computer Architecture Micro: IEEE/ACM International Symposium on Microarchitecture HPCA: International Symposium on High Performance Computer Architecture ASPLOS: International Conference on Architectural Support for Programming Languages and Operating Systems ACM Transactions on Architecture and Code Optimization IEEE Transactions on Computers The von Neumann Architecture of Computer Systems at the Wayback Machine (archived 2017-10-31) Retrieved from " Title Computer Architecture and Organization Author John Patrick Hayes Language English ISBN 0071159975 / 9780071159975 Pages 604 File Size 27.7 MB Total Downloads 2,950 Total Views 20,867 Pages In File 628 Identifier 0071159975,9780071159975 Org File Size 28,996,988 Extension pdf Download Computer Architecture and Organization PDF The third edition of Computer Architecture and Organization features a comprehensive updating of the material-especially case studies, worked examples, and problem sets-while retaining the book's time-proven emphasis on basic prinicples. Reflecting the dramatic changes in computer technology that have taken place over the last decade, the treatment of performance-related topics such as pipelines, caches, and RISC's has been expanded. Many examples and end-of-chapter problems have also been added.... Computer architecture and organisation 1. 2. 3. 4. Computer architecture and organisation Table of Contents Preface Index Computer architecture and organisation Hayes, John P. (John Patrick), 1944- This book was produced in EPUB format by the Internet Archive. The book pages were scanned and converted to EPUB format automatically. This process relies on optical character recognition, and is somewhat susceptible to errors. The book may not offer the correct reading sequence, and there may be weird characters, non-words, and incorrect guesses at structure. Some page numbers and headers or footers may remain from the scanned page. The process which identifies images might have found stray marks on the page which are not actually images from the book. The hidden page numbering which may be available to your ereader corresponds to the numbered pages in the print edition, but is not an exact match; page numbers will increment at the same rate as the corresponding print edition, but we may have started numbering before the print book's visible page numbers. The Internet Archive is working to improve the scanning process and resulting books, but in the meantime, we hope that this book will be useful to you. The Internet Archive was founded in 1996 to build an Internet library and to promote universal access to all knowledge. The Archive's purposes include offering permanent access for researchers, historians, scholars, people with disabilities, and the general public to historical collections that exist in digital format. The Internet Archive includes texts, audio, moving images, and software as well as archived web pages, and provides specialized services for information access for the blind and other persons with disabilities. Created with abbyy2epub (v.1.7.0) Computer Architecture andOrganization in McGRAW-HILL INTERNATIONAL EDIT! Computer Science Seri Computer Architecture and Organization McGraw-Hill Series in Computer Science SENIOR CONSULTING EDITOR C.L. Liu, University of Illinois at Urbana-Champaign CONSULTING EDITOR Allen B. Tucker, Bowdoin College Fundamentals of Computing and Programming Computer Organization and Architecture Computers in Society/Ethics Systems and Languages Theoretical Foundations Software Engineering and Database Artificial Intelligence Networks, Parallel and Distributed Computing Graphics and Visualization The MIT Electrical and Computer Science Series McGraw-Hill Series in Computer Organization and Architecture Bell and Newell: Computer Structures: Readings and Examples Cavanagh: Digital Computer Arithmetic: Design and Implementation Feldman and Retter: Computer Architecture and Logic Design Gear: Computer Organization and Programming: With an Emphasis on Personal Computers Hamacher, Vranesic, and Zaky: Computer Organization Hayes: Computer Architecture and Organization Hayes: Digital System Design and Microprocessors Horvath: Introduction to Microprocessors Using the MC6809 or the MC68000 Hwang: Scalable Parallel and Cluster Computing: Architecture and Programming Hwang and Briggs: Computer Architecture and Parallel Processing Lawrence and Mauch: Real-Time Microcomputer System Design Siweiorek, Bell and Newell: Computer Structures: Principles & Examples Stone: Introduction to Computer Organization and Data Structures Stone and Siewiorek: Introduction to Computer Organization and Data Structures:PDP-11 Edition Ward and Halstead: Computational Structures McGraw-Hill Series in Computer Engineering SENIOR CONSULTING EDITORS Stephen W. Director, University of Michigan, Ann Arbor C.L. Liu, University of Illinois, Urbana-Champaign Bartee: Computer Architecture and Logic Design Bose, Liang: Neural Network Fundamentals with Graphs, Algorithms, and Applications Chang and Sze: ULSI Technology De Micheli: Synthesis and Optimization of Digital Circuits Feldman and Retter: Computer Architecture: A Designer's Text Based on a Generic RISC Hamacher, Vranesic, and Zaky: Computer Organization Hayes: Computer Architecture and Organization Horvath: Introduction to Microprocessors Using the MC6809 or the MC68000 Hwang: Advanced Computer Architecture: Parallelism, Scalability, Programmability Hwang: Scalable Parallel and Cluster Computing: Architecture and Programming Kang and Leblebici: CMOS Digital Integrated Circuits: Analysis and Design Kohavi: Switching and Finite Automata Theory Krishna and Shin: Real-Time Systems Lawrence-Mauch: Real-Time Microcomputer System Design: An Introduction Levine: Vision in Man and Machine Navabi: VHDL: Analysis and Modeling of Digital Systems Peatman: Design with Microcontrollers Peatman: Digital Hardware Design Rosen: Discrete Mathematics and Its Applications Ross: Fuzzy Logic with Engineering Applications Sandige: Modern Digital Design Sarrafzadeh and Wong: An Introduction to VLSI Physical Design Schalkoff: Artificial Neural Networks Stadler: Analytical Robotics and Mechatronics Sze: VLSI Technology Taub: Digital Circuits and Microprocessors Wear, Pinkert, Wear, and Lane: Computers: An Introduction to Hardware and Software Design ABOUT THE AUTHOR JOHN P. HAYES is a professor in the electrical engineering and computer sciencedepartment at the University of Michigan, where he was the founding director of theAdvanced Computer Architecture Laboratory. He teaches and conducts research inthe areas of computer architecture; computer-aided design, verification, and testing;VLSI design; and fault-tolerant systems. Dr. Hayes is the author of two patents,more than 150 technical papers, and five books, including Layout Minimization forCMOS Cells (Kluwer, 1992, coauthored with R. L. Maziasz) and Introduction toDigital Logic Design (Addison-Wesley, 1993). He has served as editor of variousjournals, including the IEEE Transactions on Parallel and Distributed Systems andthe Journal of Electronic Testing, and was technical program chairman of the 1991International Computer Architecture Symposium, Toronto. Dr. Hayes received his undergraduate degree from the National University of Ire-land, Dublin, and his M.S. and Ph.D. degrees in electrical engineering from the Uni-versity of Illinois, Urbana-Champaign. Prior to joining the University of Michigan,he was a faculty member at the University of Southern California. Dr. Hayes hasalso held visiting positions at various academic and industrial organizations, includ-ing Stanford University, McGill University, Universite de Montreal, and Logic-Vision Inc. He is a fellow of the Institute of Electrical and Electronics Engineersand a member of the Association for Computing Machinery and Sigma Xi. To My FatherPatrick J. Hayes(1910-1968)In Memoriam CONTENTS PREFACE This book is about the design of computers; it covers both their overall design, orarchitecture, and their internal details, or organization. It aims to provide a comprehensive and self-contained view of computer design at an introductory level, pri-marily from a hardware viewpoint. The third edition of Computer Architecture andOrganization is intended as a text for computer science, computer engineering, andelectrical engineering courses at the undergraduate or beginning graduate levels; itshould also be useful for self-study. This text assumes little in the way of prerequi-sites beyond some familiarity with computer programming, binary numbers, anddigital logic. Like the previous editions, the book focuses on basic principles buthas been thoroughly updated and has substantially more coverage of performance-related issues. The book is divided into seven chapters. Chapter 1 discusses the nature and lim-itations of computation. This chapter surveys the historical evolution of computerdesign to introduce and motivate the key ideas encountered later. Chapter 2 dealswith computer design methodology and examines the two major computer designlevels, the register (or register transfer) and processor levels, in detail. It alsoreviews gate-level logic design and discusses computer-aided design (CAD) andperformance evaluation methods. Chapter 3 describes the central processing unit(CPU), or microprocessor that lies at the heart of every computer, focusing oninstruction set design and data representation. The next two chapters address CPUdesign issues: Chapter 4 covers the data-processing part, or datapath, of a proces-sor, while Chapter 5 deals with control-unit design. The principles of arithmetic-logic unit (ALU) design for both fixed-point and floating-point operations arecovered in Chapter 4. Both hardwired and microprogrammed control are examinedin Chapter 5, along with the design of pipelined and superscalar processors. Chap-ter 6 deals with a computer's memory subsystem; the chapter discusses the princi-pal memory technologies and their characteristics from a hierarchical viewpoint,with emphasis on cache memories. Finally, Chapter 7 addresses the overall organi-zation of a computer system, including inter- and intrasystem communication,input-output (10) systems, and parallel processing to achieve very high perfor-mance and reliability. Various representative computer systems, such as von Neu-mann's classic IAS computer, the ARM RISC microprocessor, the Intel Pentium,the Motorola PowerPC, the MIPS RXOOO, and the Tandem NonStop fault-tolerantmultiprocessor, appear as examples throughout the book. The book has been in use for many years at universities around the world. It con-tains more than sufficient material for a typical one-semester (15 week) course,allowing the instructor some leeway in choosing the topics to emphasize. Much ofthe background material in Chapter 1 and the first part of Chapter 2 can be left as areading assignment, or omitted if the students are suitably prepared. The moreadvanced material in Chapter 7 can be covered briefly or skipped if desired withoutloss of continuity. The Instructor's Manual contains some representative courseoutlines. This edition updates the contents of the previous edition and responds to thesuggestions of its users while retaining the book's time-proven emphasis.on basic Preface concepts. The third edition is somewhat shorter than its predecessors, and thematerial is more accessible to readers who are less familiar with computers. Everysection has been rewritten to reflect the dramatic changes that have occurred in thecomputer industry over the last decade. The main structural changes are the reor-ganization of the two old chapters on processor design and control design intothree chapters: the new Chapters 3, 4, and 5; and the consolidation of the two oldchapters on system organization and parallel processing in the new Chapter 7. Thetreatment of performance-related topics such as pipeline control, cache design, andsuperscalar architecture has been expanded. Topics that receive less space in thisedition include gate-level design, microprogramming, operating systems, and vec-tor processing. The third edition also includes many new examples (case studies)and end-of-chapter problems. There are now more than 300 problems, about 80percent of which are new to this edition. Course instructors can obtain an Instruc-tor's Manual, which contains solutions to all the problems, directly from the pub-lisher. The specific changes made in the third edition are as follows: The historicalmaterial in Chapter 1 has been streamlined and brought up to date. Gate-leveldesign has been de-emphasized in Chapter 2, while the discussion of performanceevaluation has been expanded. A new section on programmable logic devices(PLDs) has been added, and the role of computer-aided design (CAD) has beenstressed. The old third chapter (on processor design) has been split into Chapter 3,"Processor Basics," and Chapter 4, "Datapath Design." Chapter 3 contains anexpanded treatment of RISC and CISC CPUs and their instruction sets. It intro-duces the ARM and MIPS RX000 microprocessor series as major examples; theMotorola 680X0 series continues to be used as an example, however. The materialon computer arithmetic and ALU design now appears in Chapter 4. The old chapteron control design, which is now Chapter 5, has been completely revised with amore practical treatment of hardwired control and a briefer treatment of micropro-gramming. A new section on pipeline control includes some material from the oldChapter 7, as well as new material on superscalar processing. Chapter 6 presents anupdated treatment of the old fifth chapter on memory organization. Chapter 6 con-tinues to present a systematic, hierarchical view of computer memories but has agreatly expanded treatment of cache memories. Chapter 7, "System Organization,"merges material from the old sixth and seventh chapters. The sections on operatingsystems and parallel processing have been shortened and modernized. The material for this book has been developed primarily for courses on computerarchitecture and organization that I have taught over the years, initially at the Uni-versity of Southern California and later at the University of Michigan. I am gratefulto my colleagues and students at these and other schools for their many helpfulcomments and suggestions. As always, I owe a special thanks to my wife Terrie for proofreading assistance,as well as her never-failing support and love. John P. Hayes CHAPTER 1 Computing and Computers This chapter provides a broad overview of digital computers while introducingmany of the concepts that are covered in depth later. It first examines the natureand limitations of the computing process. Then it briefly traces the historical devel-opment of computing machines and ends with a discussion of contemporary VLSI-based computer systems. 1.1 THE NATURE OF COMPUTING Throughout history humans have relied mainly on their brains to perform calcula-tions; in other words, they were the computers [Boyer 1989]. As civilizationadvanced, a variety of computing tools were invented that aided, but did notreplace, manual computation. The earliest peoples used their fingers, pebbles, ortally sticks for counting purposes. The Latin words digitus meaning "finger" andcalculus meaning "pebble" have given us digital and calculate and indicate theancient origins of these computing concepts. Two early computational aids that were widely used until quite recently are theabacus and the slide rule, both of which are illustrated in Figure 1.1. The abacushas columns of pebblelike beads mounted on rods. The beads are moved by hand topositions that represent numbers. Manipulating the beads according to certain sim-ple rules enables people to count, add, and perform the other basic operations ofarithmetic. The slide rule, on the other hand, represents numbers by lengths markedon rulerlike scales that can be moved relative to one another. By adding a length aon a fixed scale to a length b on a second, sliding scale, their combined length c =a + b can be read off the fixed scale. The slide rule's main scales are logarithmic,so that the process of adding two lengths on these scales effectively multiplies two SECTION 1.1The Nature ofComputing B = 2.30 I • l t • 11 • ■ 11 r i ■ ■ | ■ 111111 < 11111111 i|iiii|ini|iiil|iiii|i T-m1"""'! |iiii|iiii|iiii|iiii|im|iiH|nii^™iiiiiiii| [in Lllllli mill.U l,l,lll,lil7,,il,,.il,l,,l.,,,L,l,,l,l,iiil,,,,l r ' ■: ■ i I'irl I'l'lTi'l""! '"l"" 8 9 1 2 ' i* * 19 8 7 6 01 llllllllllllllllllllllllllllllllllllllll lll|llll|llll|lltl|llll|IIIip 1 4 5 6 7Li.l.l.l.l.l.l.lNllllMl .I....I i, „l ,1 Ill Fri'i'i'i'i'H'ri'H i|ini|iiii|iiii|i|i| 67891 2 V* 4 S 6 7 8 9 I ll rilllllllllllllllllllllllllllllllllllllllllllllllllillllllllll7 J.,..l....i.,.,l l,..l...,l...,l....l...,l....l...l...,l...,T.I. I.I .1 .1. t.l.l ,1 .l,l,l,l.l.l,s Program Addre controlunit PCU Instructiondecoder —*■ Control77T signals 0 \£} M(0) 1 AR | 1 Mil) 2 M(2) 3 Mi?i 4 M(4) 5 M(5) *■ 1 1 ' IBR L_pc 1 , I Dataprocessing 1! ° unit DPL ii Arithmetic-logic unit ' 4,093 M.4,093) 4,094 M(4,094) M(4,095) 4,095 AC T ■ r + i — [MQ Mainmemory M Legend Program control unit PCUAR: Memory address registerIR: Instrucuon opcode registerIBR: Next- instruction buffer registerPC: Program counter Data processing unit DPI!AC: Accumulator registerDR: General-purpose data registerMQ: Multiplier- quotient register Figure 1.12 Organization of the CPU and main memory of the IAS computer. 21 CHAPTER 1Computing andComputers an instruction of the form ADD X (1.7) fetches the contents of the memory location X from main memory and adds it tothe contents of a CPU register known as the accumulator register AC. Theresulting sum is then placed in AC. Hence X and AC play the role of the threememory addresses A,, A2, and A3 appearing in (1.6).2. A program's instructions are stored in M in approximately the sequence inwhich they are executed. Hence the address of the next instruction word is usu-ally that of the current instruction plus one. Therefore, the EDVAC's next-instruction address A4 can be replaced by a CPU register (the program counterPC), which stores the address of the current instruction word and is incrementedby one when the CPU needs a new instruction word. Branch instructions areprovided to permit the instruction execution sequence to be varied. Figure 1.12 gives a programmer's perspective of the IAS, using modern nota-tion and terminology. One of the two main parts of the CPU is responsible forfetching instructions from main memory and interpreting them; this part is vari-ously known as the program control unit (PCU) or the I-unit (instruction unit). Thesecond major part of the CPU is responsible for executing instructions and isknown as the data processing unit (DPU), the datapath, or the E-unit (executionunit). The major components of the PCU are the instruction register IR, which storesthe opcode that is currently being executed, and the program counter PCwhichautomatically stores and keeps track of the address of the next instruction to be 22 SECTION 1.2The Evolution ofComputers fetched. The PCU has circuits to interpret opcodes and to issue control signals tothe DPU, M, and other circuits involved in executing instructions. The PCU canmodify the instruction execution sequence when required to do so by branchinstructions. There is also a 12-bit address register AR in the PCU that holds theaddress of a data operand to be fetched from or sent fo main memory. Because theIAS has the unusual feature of fetching two instructions at a time from M, it con-tains a second register, the instruction buffer register (IBR), for holding a secondinstruction. The main components of the DPU are the ALU, which contains the circuitsthat perform addition, multiplication, etc., as required by the possible opcodes, andseveral data registers to store data words temporarily during program execution.The IAS has two general-purpose 40-bit data registers: AC (accumulator) and DR(data register). It also has a third, special-purpose data register MQ (multiplier-quotient) intended for use by multiply and divide instructions. Main memory M is a 4096 word or 4096 x 40-bit array of storage cells. Eachstorage location in M is associated with a unique 12-bit number called its address,which the CPU uses to refer to that location. To read data from a particular mem-ory location, the CPU must have its address X (which it can store in PC or AR).The CPU accomplishes the read operation by sending the address X to M accompa-nied by control signals that specify "read." M responds by transferring a copy ofM(X), the word stored at address X, to the CPU, where it is loaded into DR. In asimilar way the CPU writes new data into main memory by sending to M the desti-nation address X, a data word D to be stored, and control signals that specify"write." Instruction set. The IAS machine had around 30 types of instructions. Thesewere chosen to provide a balance between application needs—the machine's focuswas on numerical computation for scientific applications—and computer hardwarecosts as they existed at the time. To represent instructions, we will use a notationcalled a hardware description language (HDL) or register-transfer language(RTL) that approximates the assembly language used to prepare programs for thecomputer; the designers of the IAS computer also used such a descriptive language[Burks, Goldstine, and von Neumann 1946]. The HDL introduced here and usedthroughout this book is largely self-explanatory. Storage locations in M or the CPUare referred to by acronym. The transfer of information is denoted by the assign-ment symbol :=, which suggests the left-going arrow is used to store acounting variable N and is initially set to 999. N is systematically decremented by oneafter each addition step: when it reaches -1, the program halts. The conditional branchinstruction in 5R performs this termination test. The three instructions in locations 3L.3R. and 4L are the key ones that implement (1.8). The address-modify instructions in8L. 9L. and 10L decrement the address parts of the three instructions in 3L.-3R. and zo Location Instruction or data Comment SECTION 1.2The Evolution of 0 999 Constant (count N). Computers 1 1 2 1000 Constant. 3L AC := M(2000) Load A(I) into AC. 3R AC := AC + M(3000) Compute A(I) + B(I). 4L M(4000) := AC Store sum C(I). 4R AC := M(0) Load count N into AC. 5L AC:=AC-M(1) Decrement count N by one. 5R if AC > 0 then go to M(6, 20:39) Test N and branch to 6R if nonnegative. 6L gotoM(6, 0:19) Halt. 6R M(0) := AC Update count N. 7L AC := AC + Mf 1) Increment AC by one. 7R AC := AC + M(2) Modify address in 3L. 8L M(3. 8:19):=AC(28:39) 8R AC := AC + M(2) 9L M(3, 28:39) :=AC(28:39) 9R AC := AC + M(2) 10L M(4, 8:19):=AC(28:39) 10R gotoM(3,0:19) Constant. • Modify address in 3R. Modify address in 4L. Branch to 3L. Figure 1.15 An IAS program for vector addition. 4L, respectively. Thus the program continuously modifies itself during execution. Fig-ure 1.15 shows the program before execution commences. At the end of the computation, the first three instructions will have changed to the following: 3L AC:=M(1001) 3R AC:=AC + M(2001) 4L M(3001):=AC Critique. In the years that have elapsed since the IAS computer was com-pleted, numerous improvements in computer design have appeared. Hindsightenables us to point out some of the IAS's shortcomings. 1. The program self-modification process illustrated in the preceding example fordecrementing the index I is inefficient. In general, writing and debugging a pro-gram whose instructions change themselves is difficult and error-prone. Further,before every execution of the program, the original version must be reloadedinto M. Later computers employ special instruction types and registers for indexcontrol, which eliminates the need for address-modify instructions. 2. The small amount of storage space in the CPU results in a great deal of unpro-ductive data-transfer traffic between the CPU and main memory M; it also addsto program length. Later computers have more CPU registers and a specialmemory called a cache that acts as a buffer between the CPU register? and M. 3. No facilities were provided for structuring programs. For example, the IAS hasno procedure call or return instructions to link different programs. 4. The instruction set is biased toward numerical computation. Programs for non-numerical tasks such as text processing were difficult to write and executedslowly. 5. Input-output (10) instructions were considered of minor importance—in fact,they are not mentioned in Burks, Goldstine, and von Neumann [1946] beyondnoting that they are necessary. IAS had two basic and rather inefficient 10instruction types [Estrin 1953]. The input instruction INPUT(X, N) transferredN words from an input device to the CPU and then to N consecutive main mem-ory locations, starting at address X. The OUTPUT(X, N) instruction transferredN consecutive words from the memory region with starting address X to an out-put device. 27 CHAPTER 1Computing andComputers 1.2.3 The Later Generations In spite of their design deficiencies and the limitations on size and speed imposedby early electronic technology, the IAS and other first-generation computers intro-duced many features that are central to later computers: the use of a CPU with asmall set of registers, a separate main memory for instruction and data storage, andan instruction set with a limited range of operations and addressing capabilities.Indeed the term von Neumann computer has become synonymous with a computerof conventional design. The second generation. Computer hardware and software evolved rapidlyafter the introduction of the first commercial computers around 1950. The vacuumtube quickly gave way to the transistor, which was invented at Bell Laboratories in1947, and a second generation of computers based on transistors superseded thefirst generation of vacuum tube-based machines. Like a vacuum tube, a transistorserves as a high-speed electronic switch for binary signals, but it is smaller,cheaper, sturdier, and requires much less power than a vacuum tube. Similarprogress occurred in the field of memory technology, with ferrite cores becomingthe dominant technology for main memories until superseded by all-transistormemories in the 1970s. Magnetic disks became the principal technology for sec-ondary memories, a position that they continue to hold. Besides better electronic circuits, the second generation, which spans thedecade 1954-64. introduced some important changes in the design of CPUs andtheir instruction sets. The IAS computer still served as the basic model, but moreregisters were added to the CPU to facilitate data and address manipulation. Forexample, index registers were introduced to store an index variable I of the kindappearing in the statement C(I):=A(I) + B(I) (1.9) 28 SECTION 1.2The Evolution ofComputers Index registers make it possible to have indexed instructions, which increment ordecrement a designated index I before (or after) they execute their main operation.Consequently, repeated execution of an indexed operation like (1.9) allows it tostep automatically through a large array of data. The index value I is stored in aCPU register and not in the program, so the program Itself does not change duringexecution. Another innovation was the introduction of two programcontrolinstructions, now referred to as call and return, to facilitate the linking of pro-grams; see also Example 1.5. "Scientific" computers of the second generation, such as the IBM 7094 whichappeared in 1962, introduced floating-point number formats and supportinginstructions to facilitate numerical processing. Floating point is a type of scientificnotation where a number such as 0.0000000709 is denoted by 7.09 X 10"8. Afloating-point number consists of a pair of fixed-point numbers, a mantissa Mand an exponent E, and has the value M X B~E. In the preceding example M =7.09, E = -8, and B = 10. In their computer representation M and E are encoded inbinary and embedded in a word of suitable size; the base B is implicit. Floating-point numbers eliminate the need for number scaling; floating-point numbers areautomatically scaled as they are processed. The hardware needed to implementfloating-point arithmetic instructions directly is relatively expensive. Conse-quently, many computers (then and now) rely on software subroutines to imple-ment floating-point operations via fixed-point arithmetic. Input-output operations. Computer designers soon realized that IO operations,that is, the transfer of information to and from peripheral devices like printers andsecondary memory, can severely degrade overall computer performance if doneinefficiently. Most IO transfers have main memory as their final source or destina-tion and involve the transfer of large blocks of information, for instance, moving aprogram from secondary to main memory for execution. Such a transfer can takeplace via the CPU, as in the following fragment of a hypothetical IO program: Location Instruction Comment LOOP AC := D(I) M(I) := ACI:=I+1if I < MAX go to LOOP Input word from IO device D into AC.Output word from AC to main memory.Increment index I.Test for end of loop. Clearly, the IO operation ties up the CPU with a trivial data-transfer task.Moreover, many IO devices transfer data at low speeds compared to that of theCPU because of their inherent reliance on electromechanical rather than electronictechnology. Thus the CPU is idle most of the time when executing an IO programdirected at a relatively slow device such as a printer. To eliminate this bottleneck,computers such as the IBM 7094 introduced input-output processors (IOPs), orchannels in IBM parlance, which are special-purpose processing units designedexclusively to control IO operations. They do so by executing IO programs (seepreceding sample), but channeling the data through registers in the IO processor,rather than through the CPU. Hence IO data transfers can take place independently of the CPU, permitting the CPU to execute user programs while 10 operations aretaking place. Programming languages. An important development of the mid-1950s wasthe introduction of "high level" programming languages, which are far easier touse than assembly languages because they permit programs to be written in a formmuch closer to a computer user's problem specification. A high-level language isintended to be usable on many different computers. A special program called acompiler translates a user program from the high-level language in which it is writ-ten into the machine language of the particular computer on which the program isto be executed. The first successful high-level programming language was FORTRAN (fromFORmula TRANslation), developed by an IBM group under the direction of JohnBackus from 1954 to 1957. FORTRAN permits the specification of numericalalgorithms in a form approximating normal algebraic notation. For example, thevector addition task in Figure 1.16 can be expressed by the following two-line pro-gram in the original version of FORTRAN: DO 5 1=1, 1000 5 C(I) = A(I) + B(I) FORTRAN has continued to be widely used for scientific programming and, likenatural languages, it has changed over the years. The version of FORTRAN knownas FORTRAN90 introduced in 1990 replaces the preceding DO loop with the sin-gle vector statement C(1:1000) = A(1:1000) + B(1:1000) (1.10) High-level languages were also developed in the 1950s for business applica-tions. These are characterized by instructions that resemble English statements andoperate on textual as well as numerical data. One of the earliest such languages wasCommon Business Oriented Language (COBOL), which was defined in 1959 by agroup representing computer users and manufacturers and sponsored by the U.S.Department of Defense. Like FORTRAN. COBOL has continued (in variousrevised forms) to be among the most widely used programming languages. FOR-TRAN and COBOL are the forerunners of other important high-level languages,including Basic, Pascal, C, and Java, the latter dating from the mid-1990s. EXAMPLE 1.5 A NONSTANDARD ARCHITECTURE: STACK COMPUTERS. Although most computers follow the von Neumann model, a few alternatives wereexplored quite early in the electronic era. In the stack organization illustrated in Fig-ure 1.16a a stack memory replaces the accumulator and other CPU registers used fortemporary data storage. A stack resembles the array of contiguous storage locationsfound in main memory, but it has a very different mode of access. Stack locations haveno external addresses; all read and write operations refer to one end of the stack calledthe top of the stack TOS. A push operation writes a word into the next unused locationTOS + 1 and causes this location to become the new TOS. A pop operation reads theword stored in the current TOS and causes the location TOS - 1 below TOS to becomethe new TOS. Hence TOS serves as a dynamic entry point to the stack, which expandsand contracts in response to push and pop operations, respectively. The region abovethe stack (shaded in Figure 1.16a) is unused, but it is available for future use. Among 29 CHAPTER 1Computing andComputers 30 SECTION 1.2The Evolution ofComputers Program PUSHWPUSH 3PUSH X -PUSHYSUBTRACTMULTIPLYADDPOPZ ... Controlunit — Arithmetic-logicunit sp| -4- i 1 / 1 *— / / Stack pointer * ... ^""~- Top of stack TOS Stack (a) ... z TOS w ... PUSHW ... Z TOS x - y 3 H' ... ... z TOS 3 w ... PUSH 3 ... Z TOS 3 x (x - y) w ... ... Z TOS X 3 IV ... PUSHX ... Z TOS w + 3 X (x - v) ... ... Z TOS y X 3 w ... PUSHY Z w + 3 x (x->) TOS w + 3 x (x-y) ... SUBTRACT MULTIPLY ADD POPZ (A) Figure 1.16 (a) Essentials of a stack processor; (b) stack states during the execution ofz := w + 3 x (x-y). the earliest stack computers was the Burroughs B5000, first delivered in 1963[Siewiorek. Bell, and Newell 1982]; a recent example is the Sun picoJava micropro-cessor designed for fast execution of compiled Java code [O'Connor and Tremblay1997]. In a stack machine an instruction's operands are stored at the top of the stack, sodata-processing instructions do not need to contain addresses as they do in a conventional, von Neumann computer. The add operation x + y is specified for a stackmachine by the following sequence of three instructions: PUSH*PUSHyADD The first PUSH instruction loads x into TOS. Execution of PUSH y causes x's locationto become TOS - 1 and places y in the new TOS immediately above x. To execute ADD.the top two words of the stack are popped into the ALU where they are added, and thesum is pushed back into the stack. Hence in the preceding program fragment, ADDcomputes x + y, which replaces x and y at the top of the stack. The electronic circuits thatcarry out these actions can be complicated, but they are hidden from the programmer. Akey component is a register called the stack pointer SP which stores the internal addressof TOS, and automatically adjusts the TOS for every push and pop operation. A pro-gram counter PC keeps track of instruction addresses in the usual manner. A stack computer evaluates arithmetic and other expressions using a formatknown as Polish notation, named after the Polish logician Jan Lukasiewicz (1878-1956). Instead of placing an operator between its operands as in x + y, the operator isplaced to the right of its operands as in x y +. A more complex expression such as z :=w + 3 x (x - y) becomes w 3 x y x + := (1-11) in Polish notation, and the expression is evaluated from left to right. Note that Polishnotation eliminates the need for parentheses. The Polish expression (1.11) leadsdirectly to the eight-instruction stack program shown in Figure 1.16a. The step-by-stepexecution of this code fragment is illustrated in Figure 1.16b. Here it is assumed thatw,x,y,z represent the values of operands stored at the memory addresses W,X,Y, and Z.respectively. Stack computers such as the B5000 employ a main memory M to store programsand data in much the same way as a conventional computer. For cost reasons, the CPUcontains only a small stack—a two-word stack in the B5000 case—-implemented byhigh-speed registers. However, the stack expands automatically into M by treatingsome main memory locations as if they were stack registers and coupling them withthose in the CPU. While stack processors can evaluate complex expressions such as(1.11) efficiently, they are generally slower than von Neumann machines, especiallywhen executing vector operations such as (1.10). Large stack computers were success-fully marketed for many years, notably by Burroughs Corp. However, the stack con-cept eventually became widely used in only two specialized applications: Pocket calculators sometimes employ a stack organization to take advantage of theconciseness of Polish notation when entering data and commands manually via akeypad. Stacks are included in most conventional computers to implement subroutine calland return instructions. In its basic form, a call-subroutine instruction takes the formCALL SUB. It first saves the current contents of PC—the calling routine's returnaddress—by pushing it into a stack region of M that is under the control of a stackpointer SP. Then SUB. the start address of the subroutine being called, is loadedinto PC, and its execution begins. Control is returned to the calling program whenthe subroutine executes a RETURN instruction, whose function is to pop the returnaddress from the top of the stack and load it back into PC. 1. 2. 31 CHAPTER 1Computing andComputers 32 SECTION 1.2The Evolution ofComputers System management. In the early days, all programs or jobs were run sepa-rately, and the computer had to be halted and prepared manually for each new pro-gram to be executed. With the improvements in 10 equipment and programmingmethodology that came with the second-generation machines, it became feasible toprepare a batch of jobs in advance, store them on magnetic tape, and then have thecomputer process the jobs in one continuous sequence, placing the results onanother magnetic tape. This mode of system management is termed batch process-ing. Batch processing requires the use of a supervisory program called a batchmonitor, which is permanently resident in main memory. A batch monitor is a rudi-mentary version of an operating system, a system program (as opposed to a user orapplication program)

designed to manage a computer's resources efficiently andprovide a set of common services to its users. Later operating systems were designed to enable a single CPU to process aset of independent user programs concurrently, a technique called multiprogram-ming. It recognizes that a typical program alternates between program executionwhen it requires use of the CPU, and IO operations when it requires use of anIOP. Multiprogramming is accomplished by the CPU temporarily suspending exe-cution of its current program, beginning execution of a second program, andreturning to the first program later. Whenever possible, a suspended program isassigned an IOP, which performs any needed 10 functions. Consequently, multi-programming attempts to keep a CPU (usually viewed as the computer's mostprecious resource) and any available IOPs busy by overlapping CPU and 10 oper-ations. Multiprogrammed computers that process many user programs concurrentlyand support users at interactive terminals or workstations are sometimes calledtime-sharing systems. The third generation. This generation is traditionally associated with the intro-duction of integrated circuits (ICs), which first appeared commercially in 1961, toreplace the discrete electronic circuits used in second-generation computers. Thetransistor continued as the basic switching device, but ICs allowed large numbersof transistors and associated components to be combined on a tiny piece of semi-conductor material, usually silicon. IC technology initiated a long-term trend incomputer design toward smaller size, higher speed, and lower hardware cost. Perhaps the most significant event of the third-generation period (which beganaround 1965) was recognition of the need to standardize computers in order toallow software to be developed and used more efficiently. By the mid-1960s a fewdozen manufacturers of computers around the world were each producingmachines that were incompatible with those of other manufacturers. The cost ofwriting and maintaining programs for a particular computer—the software cost—began to exceed that of the computer's hardware. At the same time many big usersof computers, such as banks and insurance companies, were creating huge amountsof application software on which their business operations were becoming verydependent. Switching to a different computer and making one's old software obso-lete was thus an increasingly unattractive proposition. Influenced by these considerations, IBM developed (at a cost of about $5 bil-lion) what was to be the most influential third-generation computer, the System/360, which it announced in 1964 and delivered the following year; see Figure 1.17.System/360 was actually a series of computers distinguished by model numbers 33 —*- Control Instruction decoder (may be ZV si?nals Program microprogrammed) control unit 1U IU devices devices PCU CHAPTER 1 4 t 1 1 » 1 AR 1 i i i System /360 1 IO interface Computing andComputers r IO Program status word PSW i 10processor(channel) processor (channel) SR PC i v II i n ii r H ii Main memorycontrol unit II " » II Sixteen Four 64-bit 32-bit general floatingpoint registers registers i 1 1 i n Main memory M r Floating-pointALU "I Fixed-pointALU DecimalALU Data-processingunit DPU Figure 1.17 Structure of the IBM System/360. and intended to cover a wide range of computing performance [Siewiorek, Bell,and Newell 1982; Prasad 1989]. The various System/360 models were designed tobe software compatible with one another, meaning that all models in the seriesshared a common instruction set. Programs written for one model could be runwithout modification on any other; only the execution time, memory usage, and thelike would change. Software compatibility enabled computer owners to upgradetheir systems without having to rewrite large amounts of software. The System/360models also used a common operating system. OS/360, and the manufacturer sup-plied specialized software to support such widely used applications as transactionprocessing and database management. In addition, the System/360 models hadmany hardware characteristics in common, including the same interface for attach-ing 10 devices. While the System/360 standardized much of IBM's own product line, it alsobecame a de facto standard for large computers, now referred to as mainframecomputers, produced by other manufacturers. The long list of makers of System/360-compatible machines includes such companies as Amdahl in the I oiled Statesand Hitachi in Japan. The System/360 series was also remarkably long-lived. Itevolved into various newer mainframe computer series introduced by IBM over theyears, all of which maintained software compatibility with the original System/ 34 SECTION 1.2The Evolution ofComputers 360; for example, the System/370 introduced in 1970, the 4300 introduced in 1979,and the System/390 introduced in 1990. The System/360 added only modestly to the basic principles of the von Neu-mann computer, but it established a number of widely followed conventions anddesign styles. It had about 200 distinct instruction'types (opcodes) with manyaddressing modes and data types, including fixed-point and floating-point numbersof various sizes. It replaced the small and unstructured set of data registers (AC,MQ, etc.) found in earlier computers with a set of 16 identical general-purpose reg-isters, all individually addressable. This is called the general-register organization.The System/360 had separate arithmetic-logic units for processing various datatypes; the fixed-point ALU was used for address computations including indexing.The 8-bit unit byte was defined as the smallest unit of information for data trans-mission and storage purposes. The System/360 also made 32 bits (4 bytes) themain CPU word size, so that 32 bits and "word" have become synonymous in thecontext of large computers. The CPU had two major control states: a supervisor state for use by the operat-ing system and a user state for executing application programs. Certain program-control instructions were "privileged" in that they could be executed only when theCPU was in supervisor state. These and other special control states gave rise to theconcept of a program status word (PS W) which was stored in a special CPU regis-ter, now generally referred to as a status register (SR). The SR register encapsu-lated the key information used by the CPU to record exceptional conditions such asCPU-detected errors (an instruction attempting to divide by zero, for example),hardware faults detected by error-checking circuits, and urgent service requests orinterrupts generated by IO devices. Architecture versus implementation. With the advent of the third generation, adistinction between a computer's overall design and its implementation detailsbecame apparent. As defined by System/360's designers [Prasad 1989], the archi-tecture of a computer is its structure and behavior as seen by a programmer work-ing at the assembly-language level. The architecture includes the computer'sinstruction set, data formats, and addressing modes, as well as the general design ofits CPU, main memory, and IO subsystems. The architecture therefore defines aconceptual model of a computer at a particular level of abstraction. A computer'simplementation, on the other hand, refers to the logical and physical design tech-niques used to realize the architecture in any specific instance. The term computerorganization also refers to the logical aspects of the implementation, but theboundary between the terms architecture and organization is vague. Hence we can say that the models of the IBM System/360 series have a com-mon architecture but different implementations. These differences reflect the exist-ence of physical circuit technologies with different cost/performance ratios forconstructing processing circuits and memories. To achieve instruction-set compati-bility across many models, the System/360 also used an implementation techniquecalled microprogramming. Originally proposed in the early 1950s by Maurice V.Wilkes at Cambridge University, microprogramming allows a CPU's programcontrol unit PCU to be designed in a systematic and flexible way [Wilkes andStringer 1953]. Low-level control sequences known as microprograms are placedin a special control memory in the PCU so that an instruction from the CPU's main instruction set is executed by invoking and executing the corresponding micropro-gram. A CPU with no floating-point arithmetic circuits can execute floatingpointinstructions (albeit slowly) if microprograms are written to perform the desiredfloating-point operations by means of fixed-point arithmetic circuits. Microprogramming allowed the smaller System/360 models to implement the full System/360 instruction set with less hardware than the larger, faster models, some of whichwere not microprogrammed. Other developments. The System/360 was typical of commercial computersaimed at both business and scientific applications. Efforts were also directed byvarious manufacturers towards the design of extremely powerful (and expensive)scientific computers, loosely termed supercomputers. Control Data Corp., forinstance, produced a series of commercially successful supercomputers beginningwith the CDC 6660 in 1964, and continuing into the 1980s with the subsequentCYBER series. These early supercomputers experimented with various types ofparallel processing to improve their performance. One such technique called pipe-lining involves overlapping the execution of instructions from the same programwithin a specially designed CPU. Another technique, which allows instructionsfrom different programs to be executed simultaneously, employs a computer withmore than one CPU; such a computer is called a multiprocessor. A contrasting development of this period was the mass production of small,low-cost computers called minicomputers. Their origins can be traced to theLINC (Laboratory Instrument Computer) developed at MIT in the early 1960s[Siewiorek, Bell, and Newell 1982]. This machine influenced the design of thePDP (Programmed Data Processor) series of small computers introduced by Dig-ital Equipment Corp. (Digital) in 1965, which did much to establish the mini-computer market. Minicomputers are characterized by short word size—CPUword sizes of 8 and 16 bits were typical—limited hardware and software facili-ties, and small physical size. Most important, their low cost made them suitablefor many new applications, such as the industrial process control where a com-puter is permanently assigned to one particular application. The Digital VAXseries of minicomputers introduced in 1978 brought general-purpose computingto many small organizations that could not afford the high cost of a mainframecomputer. 35 CHAPTER 1Computing andComputers 1.3 THE VLSI ERA Since the 1960s the dominant technology for manufacturing computer logic andmemory circuits has been the integrated circuit or IC. This technology has evolvedsteadily from ICs containing just a few transistors to those containing thousands ormillions of transistors; the latter case is termed very large-scale integration orVLSI. The impact of VLSI on computer design and application has been profound.VLSI allows manufacturers to fabricate a CPU. main memory, or even all the elec-tronic circuits of a computer, on a single IC that can be mass-produced at \ery lowcost. This has resulted in new classes of machines ranging from portable personalcomputers to supercomputers that contain thousands of CPUs. 36 SECTION 1.3The VLSI Era &^> (a) (b) (c) Figure 1.18 Some representative IC packages: (a) 32-pin small-outline J-lead (SOJ); (b) 132-pin plasticquad flatpack (PQFP); (c) 84-pin pin-grid array (PGA). [Courtesy of Sharp ElectronicsCorp.] 1.3.1 Integrated Circuits The integrated circuit was invented in 1959 at Texas Instruments and FairchildCorporations [Braun and McDonald 1982]. It quickly became the basic buildingblock for computers of the third and subsequent generations. (The designation ofcomputers by generation largely fell into disuse after the third generation.) An IC isan electronic circuit composed mainly of transistors that is manufactured in a tiny-rectangle or chip of semiconductor material. The IC is mounted into a protectiveplastic or ceramic package, which provides electrical connection points called pinsor leads that allow the IC to be connected to other ICs, to input-output devices likea keypad or screen, or to a power supply. Figure 1.18 depicts several representativeIC packages. Typical chip dimensions are 10 X 10 mm, while a package like that ofFigure 1.18b is approximately 30 X 30 X 4 mm. The IC package is often consider-ably bigger than the chip it contains because of the space taken by the pins. ThePGA package of Figure 1.18c has an array of pins (as many as 300 or more) pro-jecting from its underside. A multichip module is a package containing several ICchips attached to a substrate that provides mechanical support, as well as electricalconnections between the chips. Packaged ICs are often mounted on a printed cir-cuit board that serves to support and interconnect the ICs. A contemporary com-puter consists of a set of ICs, a set of IO devices, and a power supply. The numberof ICs can range from one IC to several thousand, depending on the computer'ssize and the IC types it uses. IC density. An integrated circuit is roughly characterized by its density,defined as the number of transistors contained in the chip. As manufacturing tech-niques improved over the years, the size of the transistors in an IC and their inter-connecting wires shrank, eventually reaching dimensions below a micron or 1 pm.(By comparison, the width of a human hair is about 75 ujn.) Consequently, IC den-sities have increased steadily, while chip size has varied very little. The earliest ICs—the first commercial IC appeared in 1961—contained fewerthan 100 transistors and employed small-scale integration or SSI. The termsmedium-scale, and very-large-scale integration (MSI, LSI and VLSI. • lG-bit ,•* a DRAM . „' 109 • • u lM-bit ./^DRAM ./• 64-bit c c C j 6_bjt ./* microprocessor 106 microprocessor A* 32-bit £ v^ microprocessor | lK-bit v» 8-bit microprocessorDRAM ./^ ■a y 103 1 •SSI _,— • 4-bit microprocessorMSI i i i i 1960 1970 1980 1990 Year 2000 2010 Figure 1.19 Evolution of the density of commercial ICs. 37 CHAPTER 1Computing andComputers respectively) are applied to ICs containing hundreds, thousands, and millions oftransistors, respectively. The boundaries between these IC classes are loose, andVLSI often serves as a catchall term for very dense circuits. Because their manu-facture is highly automated—it resembles a printing process—ICs can be manufac-tured in high volume at low cost per circuit. Indeed, except for the latest anddensest circuits, the cost of an IC has stayed fairly constant over the years, imply-ing that newer generations of ICs deliver far greater value (measured by computingperformance or storage capacity) per unit cost than their predecessors did. Figure 1.19 shows the evolution of IC density as measured by two of the dens-est chip types: the dynamic random-access memory (DRAM), a basic componentof main memories, and the single-chip CPU or microprocessor. Around 1970 itbecame possible to manufacture all the electronic circuits for a pocket calculator ona single IC chip. This development was quickly followed by single-chip DRAMsand microprocessors. As Figure 1.19 shows, the capacity of the largest availableDRAM chip was IK = 210 bits in 1970 and has been growing steadily since then,reaching 1M = 220 bits around 1985. A similar growth has occurred in the com-plexity of microprocessors. The first microprocessor, Intel's 4004, which wasintroduced in 1971, was designed to process 4-bit words. The Japanese calculatormanufacturer Busicom commissioned the 4004 microprocessor, but after Busi-com's early demise, Intel successfully marketed the 4004 as a programmable con-troller to replace standard, nonprogrammable logic circuits. As IC technologyimproved and chip density increased, the complexity and performance of one-chipmicroprocessors increased steadily, as reflected in the increase in CPU word size to8 and then 16 bits by the mid-1980s. By 1990 manufacturers could fabricate theentire CPU of a System/360-class computer, along with part of its main memory,on a single IC. The combination of a CPU, memory, and IO circuits in one IC (or asmall number of ICs) is called a microcomputer. SECTION 1.3The VLSI Era IC families. Within IC technology several subtechnologies exist that are dis-tinguished by the transistor and circuit types they employ. Two of the most impor-tant of these technologies are bipolar and unipolar; the latter is normally referred toas MOS (metal-oxide-semiconductor) after its physical structure. Both bipolar andMOS circuits have transistors as their basic elements! They differ, however, in thepolarities of the electric charges associated with the primary carriers of electricalsignals within their transistors. Bipolar circuits use both negative carriers (elec-trons) and positive carriers (holes). MOS circuits, on the other hand, use only onetype of charge carrier: positive in the case of P-type MOS (PMOS) and negative inthe case of N-type MOS (NMOS). Various bipolar and MOS IC circuit types or ICfamilies have been developed that provide trade-offs among density, operatingspeed, power consumption, and manufacturing cost. An MOS family that effi-ciently combines PMOS and NMOS transistors in the same IC is complementaryMOS or CMOS. This technology came into widespread use in the 1980s and hasbeen the technology of choice for microprocessors and other VLSI ICs since thenbecause of its combination of high density, high speed, and very low power con-sumption [Weste and Eshragian 1992]. EXAMPLE 1.6 A ZERO-DETECTION CIRCUIT EMPLOYING CMOS TECH-NOLOGY. To illustrate the role of transistors in computing, we examine a smallCMOS circuit whose function is to detect when a 4-bit word x0xlx2xi becomes zero.The circuit's output z should be 1 when x0x]x2xi = 0000; it should be 0 for the other 15combinations of input values. Zero detection is quite a common operation in data pro-cessing. For example, it is used to determine when a program loop terminates, as in theif statement (location 5R) appearing in the IAS program of Figure 1.15. Figure 1.20 shows a particular implementation ZD of zero detection using a repre-sentative CMOS subfamily known as static CMOS. The circuit is shown in standardsymbolic form in Figure 1.20a. It consists of equal numbers of PMOS transistorsdenoted 5,:57 and NMOS transistors denoted SS:SU. Each transistor acts as an on-offswitch with three terminals, where the center terminal c controls the switch's state.When turned on, a signal propagation path is created between the transistor's upper andlower terminals; when turned off, that path is broken. An NMOS transistor is turned onby applying 1 to its control terminal c; it is turned off by applying 0 to c. A PMOS tran-sistor, on the other hand, is turned on by c - 0 and turned off by c = 1. Each set of input signals applied to ZD causes some transistors to switch on andothers to switch off, which creates various signal paths through the circuit. In Figure1.20 the constant signals 0 and 1 are applied at various points in ZD. (These signals arederived from ZD's electrical power supply.) The 0/1 signals "flow" through the circuitalong the paths created by the transistors and determine various internal signal values,as well as the value applied to the main output line z. Figure 1.20b shows the signalsand signal transmission paths produced by x0xix2x3 - 0001. The first input signal x0 = 0is applied to PMOS transistor 5, and NMOS transistor 5g; hence S, is turned on and 5gis turned off. Similarly, x, = 0 turns S2 on and S9 off. A path is created through S, andS2, which applies 1 to the internal line y,, as shown by the left-most heavy arrow in Fig-ure 1.20b. In the same way the remaining input combinations make y2 = 0 and y3 = 1.The latter signal is applied to the two right-most transistors turning S7 off and 514 on,which creates a path from the zero source to the primary output line via 514, so z = 0 asrequired. If we change input x3 from 1 to 0 in Figure 1.20b, the following chain of eventsoccurs: 54 turns on and 5,, turns off, changing y2 to 1. Then 5I3 turns on and S6 turnsoff, making y3 = 0. Finally, the new value of y3 turns 57 on and S]4 off, so z becomes 1. Output Inputs PMOS transistor NMOS transistor (a) ; = 0 xQ = 0 xl = 0 x2 - 0 *3 = 1 Transistor switched on Transistor switched off (b) Figure 1.20 (a) CMOS circuit ZD for zero detection; (b) state of ZD with input combination xQxlx2x} = 0001 making z = 0. Hence the zero input combination x0xlx2x3 = 0000 makes c = 1 as required. It canreadily be verified that no other input combination does this. 39 CHAPTER 1Computing andComputers A transistor circuit like that of Figure 1.20 models the behavior of a digitalcircuit at a low level of abstraction called the switch level. Because many of theICs of interest contain huge numbers of transistors, it is rarely practical to analyzetheir computing functions at the switch level. Instead, we move to higher abstrac-tion levels, two of which are illustrated in Figure 1.21. At the gate or logicAexe\illustrated by Figure 1.21a. we represent certain common subcircuits by symbolic 40 SECTION 1.3The VLSI Era NOR gates NAND gate (a) NOT gate(inverter) Zerodetector 00 Figure 1.21 The zero-detection circuit of Figure 1.20 modeled at (a) the gate level and (b) the regis-ter level of abstraction. components called (logic) gates. This particular logic circuit comprises four gatesA, B, C, and D of three different types as indicated; note that each gate type has adistinct graphic symbol. In moving from the switch level, we collapse a multi-transistor circuit into a single gate and discard all its internal details. A key advan-tage of the logic level is that it is technology independent, so it can be used equallywell to describe the behavior of any IC family. In dealing with computer design,we also use an even higher level of abstraction known as the register or register-transfer level. It treats the entire zero-detection circuit as a primitive or indivisiblecomponent, as in Figure 1.21b. The register level is the level at which we describethe internal workings of a CPU or other processor as, for example, in Figures 1.2and 1.17. Observe that the primitive components (represented by boxes) in thesediagrams include registers, ALUs, and the like. When we treat an entire CPU,memory, or computer as a primitive component, we have moved to the highestlevel of abstraction, which is called the processor or system level. 1.3.2 Processor Architecture By 1980 computers were classified into three main types: mainframe computers,minicomputers, and microcomputers. The term mainframe was applied to the tradi-tional "large" computer system, often containing thousands of ICs and costing mil-lions of dollars. It typically served as the central computing facility for anorganization such as a university, a factory, or a bank. Mainframes were thenroom-sized machines placed in special computer centers and not directly accessibleto the average user. The minicomputer was a smaller (desk size) and slower ver-sion of the mainframe, but its relatively low cost (hundreds of thousands of dollars)made it suitable as a "departmental" computer to be shared by a group of users—ina small business, for example. The microcomputer was even smaller, slower, andcheaper (a few thousand dollars), packing all the electronics of a computer into ahandful of ICs, including microprocessor (CPU), memory, and IO chips. Personal computers. Microcomputer technology gave rise to a new class ofgeneral-purpose machines called personal computers (PCs), which are intended fora single user. These small, inexpensive computers are designed to sit on an officedesk or fold into a compact form to be carried. The more powerful desktop com-puters intended for scientific computing are referred to as workstations. A typical PC has the von Neumann organization, with a microprocessor, a multimegabytemain memory, and an assortment of 10 devices: a keyboard, a video monitor orscreen, a magnetic or optical disk drive unit for high-capacity secondary memory,and interface circuits for connecting the PC to printers and to other computers. Per-sonal computers have proliferated to the point that, in the more developed societ-ies, they are present in most offices and many homes. Two of the main applicationsof PCs are word processing, where personal computers have assumed and greatlyexpanded all the functions of the typewriter, and data-processing tasks like finan-cial record keeping. They are also used for entertainment, education, and increas-ingly, communication with other computers via the World Wide Web. Personal computers were introduced in the mid-1970s by a small electronicskit maker, MITS Inc. [Augarten 1984]. The MITS Altair computer was builtaround the Intel 8008, an early 8-bit microprocessor, and cost only $395 in kitform. The most successful personal computer family was the IBM PC series intro-duced in 1981. Following the precedent set by earlier IBM computers, it quicklybecome the de facto standard for this class of machine. A new factor also aided thestandardization process—namely, IBM's decision to give the PC what came to becalled an open architecture, by making its design specifications available to othermanufacturers of computer hardware and software. As a result, the IBM PCbecame very popular, and many versions of it—the so-called PC clones—wereproduced by others, including startup companies that made the manufacture oflow-cost PC clones their main business. The PC's open architecture also providedan incentive for the development of a vast amount of applicationspecific softwarefrom many sources. Indeed a new software industry emerged aimed at the mass-production of low-cost, self-contained programs aimed at specific applications ofthe IBM PC and a few other widely used computer families. The IBM PC series is based on Intel Corp.'s 80X86 family of microprocessors,which began with the 8086 microprocessor introduced in 1978 and was followedby the 80286 (1983), the 80386 (1986), the 80486 (1989), and the Pentium2 (1993)[Albert and Avnon 1993]; the Pentium II appeared in 1997. The IBM PC series isalso distinguished by its use of the MS/DOS operating system and the Windowsgraphical user interface, both developed by Microsoft Corp. Another popular per-sonal computer series is Apple Computer's Macintosh, introduced in 1984 andbuilt around the Motorola 680X0 microprocessor family, whose evolution from the68000 microprocessor (1979) parallels that of the 80X86/Pentium [Farrell 1984|.In 1994 the Macintosh CPU was changed to a new microprocessor known as thePowerPC. Figure 1.22 shows the organization of a typical personal computer from themid-1990s. Its legacy from earlier von Neumann computers is apparent —compareFigure 1.22 to Figure 1.17. At the core of this computer is a single-chip micropro-cessor such as the Pentium or PowerPC. As we will see, the microprocessor's inter-nal (micro) architecture usually contains a number of speedup features not found inits predecessors. A system bus connects the microprocessui to a main memor)based on semiconductor DRAM technology and to an IO subsystem. A separate IObus, such as the industry standard PCI (peripheral component interconnect) "'local'" 41 CHAPTER 1Computing andComputers 2A legal ruling that microprocessor names that are numbers cannot have trademark protection, resulted in the80486 being followed by a microprocessor called the Pentium rather than the 80586. 42 SECTION 1.3The VLSI Era Microprocessor CPU Cache Bus interface unit Main memory M Secondary (hard disk) memory Videomonitor Keyboard Hard diskcontrol —r Videocontrol Communicationnetwork Keyboardcontrol UTI T IO devices Networkcontrol IO expansionslots r u zr 'Li 2 IO (local) bus Peripheral (IO) interface control unit irT System bus Figure 1.22 A typical personal computer system. bus, connects directly to the IO devices and their individual controllers. The IO busis linked to the system bus, to which the microprocessor and memory are attachedvia a special bus-to-bus control unit sometimes referred to as a bridge. The IOdevices of a personal computer include the traditional keyboard, a CRT-based orflat-panel video monitor, and disk drive units for the hard and flexible (floppy) diskstorage devices that constitute secondary memory. More recent additions to the IOdevice repertoire include drive units for CD-ROMs (compact disc read-only mem-ories), which have extremely high capacity and allow sound and video images tobe stored and retrieved efficiently. Other common audiovisual IO devices in per-sonal computers are microphones, loudspeakers, video scanners, and the like,which are referred to as multimedia equipment. Performance considerations. As processor hardware became much less expen-sive in the 1970s, thanks mainly to advances in VLSI technology (Figure 1.19),computer designers increased the use of complex, multistep instructions. Thisreduces N, the total number of instructions that must be executed for a given task,since a single complex instruction can replace several simpler ones. For example, amultiply instruction can replace a multiinstruction subroutine that implements mul-tiplication by repeated execution of add instructions. Reducing N in this way tendsto reduce overall program execution time T, as well as the time that the CPUspends fetching instructions and their operands from memory. The same advancesin VLSI made it possible to add new features to old microprocessors, such as newinstructions, data types, instruction sets, and addressing modes, while retaining theability to execute programs written for the older machines. The Intel 80X86/Pentium series illustrates the trend toward more complexinstruction sets. The 1978-vintage 8086 microprocessor chip, which contained amere 20,000 transistors, was designed to process 16-bit data words and had noinstructions for operating on floating-point numbers [Morse et al. 1978]. Twenty-five years later, its direct descendant, the Pentium, contained over 3 million transis-tors, processed 32-bit and 64-bit words directly, and executed a comprehensive setof floating-point instructions [Albert and Avnon 1993]. The Pentium accumulated most of the architectural features of its various predecessors in order to enable it toexecute, with little or no modification, programs written for earlier 80X86seriesmachines. Reflecting these characteristics, the 80X86, 680X0, and most older com-puter series have been called complex instruction set computers (CISCs).3 By the 1980s it became apparent that complex instructions have certain disad-vantages and that execution of even a small percentage of such instructions cansometimes reduce a computer's overall performance. To illustrate this condition,suppose that a particular microprocessor has only fast, simple instructions, each ofwhich requires k time units, to execute. Thus the microprocessor can execute 100instructions in 100k time units. Now suppose that 5 percent of the instructions areslow, complex instructions requiring 2lk time units each. To execute an averageset of 100 instructions therefore requires (5x21+ 95)k = 200k time units, assum-ing no other factors are involved. Consequently, the 5 percent of complex instruc-tions can, as in this particular example, double the overall program execution time. Thus while complex instructions reduce program size, this technology does notnecessarily translate into faster program execution. Moreover, complex instructionsrequire relatively complex processing circuits, which tend to put CISCs in the larg-est and most expensive IC category. These drawbacks were first recognized by JohnCocke and his colleagues at IBM in the mid-1970s, who developed an experimentalcomputer called 801 that aimed to achieve very fast overall performance via astreamlined instruction set that could be executed extremely fast [Cocke and Mark-stein 1990]. The 801 and subsequent machines with a similar design philosophyhave been called reduced instruction set computers (RISCs). A number of commer-cially successful RISC microprocessors were introduced in the 1980s, including theIBM RISC System/6000 and SPARC, an "open" microprocessor developed by SunMicrosystems and based on RISC research at the University of California, Berkeley[Patterson 1985]. Many of the speedup features of RISC machines have found theirway into other new computers, including such CISC microprocessors as the Pen-tium. Indeed, the term RISC is often used to refer to any computer with an instruc-tion set and an associated CPU organization designed for very high performance:the actual size of the instruction set is relatively unimportant. A computer's performance is also strongly affected by other factors besidesits instruction set, especially the time required to move instructions and databetween the CPU and main memory M and, to a lesser extent, the time required tomove information between M and IO devices. It typically takes the CPU aboutfive times longer to obtain a word from M than from one of its internal registers.This difference in speed has existed since the first electronic computers, despitestrenuous efforts by circuit designers to develop memory devices and processor-memory interface circuits that are fast enough to keep up with the fastest micro-processors. Indeed the CPU-M speed disparity has become such a feature of stan-dard (von Neumann) computers that is sometimes referred to as the von Neumannbottleneck. RISC computers usually limit access to main memory to a few loadand store instructions; other instructions, including all data-processing and pro-gram-control instructions, must have their operands in CPU registers. This so43 CHAPTER IComputing andComputers 3The public became aware of CISC complexity when a design flaw affecting the floating-point divisioninstruction of the Pentium was discovered in 1994. The cost to Intel of this bug. including the replacementcost of Pentium chips already installed in PCs. was about $475 million. 44 called load-store architecture is intended to reduce the impact of the von Neusection 3 mann bottleneck by reducing the total number of the memory accesses made by The VLSI Era the CPU. Performance measures. A rough indication of CPU speed is the number of"basic" operations that it can perform per unit of time. A typical basic operation isthe fixed-point addition of the contents of two registers Rl and R2, as in the sym-bolic instruction Rl :=R1 +R2 Such operations are timed by a regular stream of signals (ticks or beats) issued by acentral timing signal, the system clock. The speed of the clock is its frequency /measured in millions of ticks per second; the units for this are megahertz (MHz).Each tick of the clock triggers a basic operation; hence the time required to executethe operation is 1//microseconds ((is). This value is called the clock cycle or clockperiod Tdock. For example, a computer clocked at 250 MHz can perform one basicoperation in the clock period Tdock = 1/250 = 0.004 (is. Complicated operationssuch as division or operations on floating-point numbers can require more than oneclock cycle to complete their execution. Generally speaking, smaller electronic devices operate faster than larger ones,so the increase in IC chip density discussed above has been accompanied by asteady, but less dramatic, increase in clock speed. For example, from 1981 to 1995microprocessor clock speeds increased from about 10 MHz to 100 MHz. Clockspeeds of 1 gigahertz (1 GHz or 1000 MHz) and beyond are feasible using fasterversions of current CMOS technology. It might therefore seem possible to achieveany desired processor speed simply by increasing the CPU clock frequency. How-ever, the rate at which clock frequency is increasing due to IC technology improve-ments is relatively slow and may be approaching limits determined by the speed oflight, power dissipation, and similar physical considerations. Extremely fast cir-cuits also tend to be very expensive to manufacture. The CPU's processing of an instruction involves several steps, each of whichrequires at least one clock cycle: 1. Fetch the instruction from main memory M. 2. Decode the instruction's opcode. 3. Load (read) from M any operands needed unless they are already in CPU regis-ters. 4. Execute the instruction via a register-to-register operation using an appropriatefunctional unit of the CPU, such as a fixed-point adder. 5. Store (write) the results in M unless they are to be retained in CPU registers. The fastest instructions have all their operands in CPU registers and can be exe-cuted by the CPU in a single clock cycle, so steps 1 to 3 all take one clock cycle.The slowest instructions require multiple memory accesses and multiple register-to-register operations to complete their execution. Consequently, measures ofinstruction execution performance are based on average figures, which are usuallydetermined experimentally by measuring the run times of representative or bench-mark programs. The more representative the programs are, that is, the more accu-rately they reflect real applications, the better the performance figures they provide. Suppose that execution of a particular benchmark program or set (suite) ofsuch programs Q on a given CPU takes T seconds and involves the execution of atotal of N machine (object) instructions. Here N is the actual number of instructionsexecuted, including repeated executions of the same instruction; it is not the num-ber of instructions appearing in Q. As far as the typical computer user is concerned,the key performance goal is to minimize the total program execution time T. WhileT can be determined accurately only by measurement of 'J+\ =(.v, + */y,)/2 Here; = 1, 2, 3, ..., and >•„ is an initial approximation to x]f2. Assuming that IAS pro-cesses real (floating-point) numbers directly, construct a program in the style of Figure 1.15 to calculate the square root of a given positive number x according to thisformula. 1.20. Early computer literature describes the IAS and other first-generation computers as"parallel." unlike some of their predecessors. In what sense was the IAS a parallel com-puter? What forms of parallelism do modern computers have that are lacking in theIAS? 1.21. The IAS had no call or return instructions designed for transferring control betweenprograms, (a) Describe how call and return can be programmed using the IAS's 59 CHAPTER 1Computing andComputers 60 original instruction set. (b) What feature would you suggest adding to the IAS to support call and return operations?SECTION 1.5Problen^ 1-22. Construct both a Polish expression and a stack program of the kind given in Figure 1.16a to evaluate the following expression: f:=(4x(a2 + b + c)-d)/(e+fxg) (1.14) 1.23. From the data presented in Figure 1.19, estimate how long it takes, on average, for thedensity of leading-edge ICs to double. This doubling rate, which has remained remark-ably constant over the years, is referred to as Moore's law, after Gordon E. Moore, acofounder of Intel Corp., who formulated it in the 1960s. 1.24. Using the circuit of Figure 1.20 as an illustration, discuss and justify the followinggeneral properties of CMOS circuits: (a) Power consumption is very low and most ofit occurs when the circuit is changing state (switching), (b) The logic signals 0 and 1correspond to electrical voltage levels, (c) The subcircuits that constitute logic gatesdraw their power directly from the global power supply rather than from the external(primary) input signals: hence the gates perform signal amplification. 1.25. The CMOS zero-detection circuit of Figures 1.20 and 1.21 can be implemented as asingle four-input logic gate. Identify the gate in question and redesign the circuit in themore compact single-gate form. 1.26. Design a CMOS ones-detection circuit in the multigate style of Figure 1.20. It shouldproduce the output z = 1 if and only if x0x]x2x3 - 1111. Give both a transistor (switch-level) circuit and a gate-level circuit for your design. 1.27. Discuss the impact of developments in computer hardware technology on the evolutionof each of the following: (a) the logical complexity of the smallest replaceable compo-nents; (b) the operating speed of the smallest replaceable components; and (c) the for-mats used for data and instruction representation. 1.28. Define the terms software compatibility and hardware compatibility. What role havethey played in the evolution of computers? 1.29. Identify and briefly describe three distinct ways in which parallelism can be introducedinto the microarchitecture of a computer in order to increase its overall instruction ex-ecution speed. 1.30. Compare and contrast the IAS and PowerPC processors in terms of the complexity ofwriting assembly-language programs for them. Use the vector addition programs ofFigures 1.15 and 1.27 to illustrate your answer. 1.31. A popular microprocessor of the 1970s was the Intel 8085, a direct ancestor of the80X86/Pentium series, which has the structure shown in Figure 1.32. The data wordsize in the CPU and M is 8 bits, while the address size is 16 bits. Because the 8085'sIC package has only 40 pins, the lines AD for transmitting addresses and data betweenthe CPU and M are shared (multiplexed) as indicated. AD is used to attach IO devicesas well as M to the 8085; there is also a separate serial (two line) IO port. The 8085 hasabout 70 different instruction types. Its most complex arithmetic instructions are addi-tion and subtraction of 8-bit fixed-point (binary and decimal) numbers. There are six8-bit registers designated B, C, D, E, H, and L, which, with the accumulator A, form ageneral-purpose CPU register file. The register-pairs BC, DE, and HL serve as 16-bitaddress registers. A program counter PC maintains the address of the next instructionbyte required from M in the usual manner. The 8085 also has stack pointer SP thatpoints to the top of a user-defined stack area in M. (a) What is the maximum capacity Serial 10 devices Li Serial10 port B C D E H L *—8—► «— 8—»• Data/ AddressAddress low high Control System bus(to M and 10) 8-bit internal data bus Accumu-lator A Statusregister SR 8-bit ALU 8/16-bit register file Figure 1.32 Structure of the Intel 8085 microprocessor. Instructionregister IR Programcontrol Stack pointer SP Program counter PC 61 CHAPTER 1Computing andComputers Location Instruction Comment ADDEC: LOOP: LXI D, NUM1 LXI H, NUM2 MVI C, 16 LDAX D ADC M DAA MOV M,A DCX D DCX H DCR C JNZ LOOP + 16 Initialize address: DE := NUM1 + 16.+ 16 Initialize address: HL := NUM2 + 16. Initialize count: C := 16. Load data: D := M(DE). A := A + CY + M(HL). Update CY flag. Convert sum in A to decimal. Store data: M(HL) := A. Decrement address: DE := DE - 1. Decrement address: HL := HL - 1. Decrement count: C := C - 1. Update Z flag. Jump to LOOP if Z * 1. Figure 1.33 An 8085 program to add two 32-digit decimal integers. of the 8085's main memory? (b) What is the size of PC? (c) What is the purpose of SP?(d) Identify three common features of more recent microprocessors that the 8085 lacks. 1.32. Consider the Intel 8085 described in the preceding problem. A taste of its software canbe found in Figure 1.33, which lists a program ADDEC written in 8085 assembly lan-guage that performs the addition of two long (n digit) decimal numbers NUM1 andNUM2. The numbers are added two digits (8 bits) at a time using the instructions ADC(add with carry) and DAA (decimal adjust accumulator). ADC takes a byte from Mand, treating it as an 8-bit binary number, adds it and a carry bit CY to the contents of SECTION 1.6References 62 the A register. DAA then changes the binary sum in A to binary-coded decimal form. This calculation uses several flag bits of the status register SR: the carry flag CY, whichis set to 1 (0) whenever the 9th bit resulting from an 8-bit addition is 1 (0); and the zeroflag Z, which is set to 1 (0) when the result of an arithmetic instruction such as add ordecrement is 0 (non-0), (a) From the information given here, determine the size n ofthe numbers being added and the (symbolic) location in M where the sum NUM1 +NUM2 is stored, (b) Ignoring the size of the 8085's instruction set, would you classifyit as CISC or RISC? Justify your answers. 1.33. The performance of a 100 MHz microprocessor P is measured by executing10,000,000 instructions of benchmark code, which is found to take 0.25 s. What are thevalues of CPl and MIPS for this performance experiment? Is P likely to be superscalar? 1.34. Suppose that a single-chip microprocessor P operating at a clock frequency of 50 MHzis replaced by a new model P , which has the same architecture as P but has a clockfrequency of 75 MHz. (a) If P has a performance rating of p MIPS for a particularbenchmark program Q, what is the corresponding MIPS rating p for P ? (b) P takes250 s to execute Q in a particular personal computer system C. On replacing P by P inC, the execution time of Q drops only to 220 s. Suggest a possible reason for this dis-appointing performance improvement. 1.35. {a) What are the usual definitions of the terms CISC and RISC? Identify two key archi-tectural features that distinguish recent RISC and CISC machines, (b) When develop-ing the RISC/6000, the direct predecessor of the PowerPC, IBM viewed the word RISCto mean "reduced instruction set cycles." Explain why this meaning might be more ap-propriate for the PowerPC than the usual one. 1.6REFERENCES 1. Albert, D. and D. Avnon. "Architecture of the Pentium Microprocessor." IEEE Micro,vol. 13 (June 1993) pp. 11-21. 2. Augarten, S. Bit by Bit: An Illustrated History of Computers. New York: Ticknor andFields, 1984. 3. Barwise, J. and J. Etchemendy. Turing's World 3.0: An Introduction to ComputabilityTheory. Stanford, CA: CSLI Publications, 1993. 4. Boyer, C. B. A History of Mathematics. 2nd ed. New York: Wiley, 1989. 5. Braun, E. and S. MacDonald. Revolution in Miniature. The History and Impact of Semi-conductor Electronics. 2nded. Cambridge, England: Cambridge University Press, 1982. 6. Burks, A. W., H. H. Goldstine, and J. von Neumann. "Preliminary Discussion of theLogical Design of an Electronic Computing Instrument." Report prepared for U.S. ArmyOrdnance Department, 1946. (Reprinted in Ref. 26, vol. 5, pp. 34-79.) 7. Cocke, J. and V. Markstein. "The Evolution of RISC Technology at IBM." IBM Journalof Research and Development, vol. 34 (January 1990) pp. 4-11. 8. Cormen, T. H., C. E. Leiserson, and R. L. Rivest. Introduction to Algorithms. MIT Press,Cambridge, MA, and McGraw-Hill, New York, 1990. 9. Diefendorf, K., R. Oehler, and R. Hochsprung. "Evolution of the PowerPC Architec-ture." IEEE Micro, vol. 14 (April 1994) pp. 34-^9. 10. Estrin, G. "The Electronic Computer at the Institute for Advanced Studies." Mathemat-ical Tables and Other Aids to Computation, vol. 7 (April 1953) pp. 108-14. 11. Farrell, J. J. "The Advancing Technology of Motorola's Microprocessors and Micro-computers." IEEE Micro, vol. 4 (October 1984) pp. 55-63. 12. Garey, M. R. and D. S. Johnson. Computers and Intractability. San Francisco: W. H.Freeman, 1979. 13. Goldstine, H. H. and J. von Neumann. "Planning and Coding Problems for an ElectronicComputing Instrument." Part II, vols. 1 to 3. Three reports prepared for U.S. Army Ord-nance Department, 1947-1948. (Reprinted in Ref. 26, vol. 5, pp. 80-235.) 14. Hwang, K. Advanced Computer Architecture. New York: McGraw-Hill, 1993. 15. Morrison, P. and E. Morrison (eds.). Charles Babbage and His Calculating Engines.New York: Dover, 1961. 16. Morse, S. P. et al. "Intel Microprocessors: 8008 to 8086." Santa Clara, CA: Intel, 1978.(Reprinted in Ref. 24, pp. 615-46.) 17. Motorola Inc. PowerPC 601 RISC Microprocessor User's Manual. Phoenix, AZ, 1993.(Also published by IBM Microelectronics, Essex Junction, VT, 1993). 18. O'Connor, J. M. and M. Tremblay. "picoJava-I: The Java Virtual Machine in Hard-ware." IEEE Micro, vol. 17 (March/April 1997) pp. 45-53. 19. Patterson, D. "Reduced Instruction Set Computers." Communications of the ACM, vol.28, (January 1985) pp. 8-21. 20. Poppelbaum, W. J. et al. "Unary Processing." Advances in Computers, vol. 26, ed. M.Yovits. New York: Academic Press, 1985, pp. 47-92. 21. Prasad, N. S. IBM Mainframes: Architecture and Design. New York: McGraw-Hill,1989. 22. Randell, B. (ed.) The Origins of Digital Computers: Selected Papers. 3rd ed. Berlin:Springer-Verlag, 1982. 23. Russell, R. M. "The CRAY-1 Computer System." Communications of the ACM, vol. 21(January 1978), pp. 63-78. (Reprinted in Ref. 24, pp. 743-52.) 24. Siewiorek, D. P., C. G. Bell, and A. Newell. Computer Structures: Readings and Exam-ples. New York: McGraw-Hill, 1982. 25. Swade, D. D. "Redeeming Charles Babbage's Mechanical Computer." Scientific Amer-ican, vol. 268 (February 1993) pp. 86-91. 26. von Neumann, J. Collected Works, ed. A. Taub, 6 vols. New York: Pergamon, 1963. 27. Weiss, S. and J. E. Smith. Power and PowerPC. San Francisco, CA: Morgan Kaufmann,1994. 28. Weste, N. and K. Eshragian. Principles of CMOS VLSI Design. 2nd ed. Reading, MA:Addison-Wesley, 1992. 29. Wilkes, M. V. and J. B. Stringer. "Microprogramming and the Design of Control Cir-cuits in an Electronic Digital Computer." Proc. Cambridge Phil. Soc, pt. 2, vol. 49(April 1953) pp. 230-38. (Reprinted in Ref. 24, pp. 158-63.) 63 CHAPTER 1 Computing andComputers CHAPTER 2 Design Methodology This chapter views the design process for digital systems at three basic levels ofabstraction: the gate, the register, and the processor levels. It discusses the natureof the design process, examines design at the register and processor levels in detail,and briefly introduces computer-aided design (CAD) and analysis methods. 2.1 SYSTEM DESIGN A computer is an example of a system, which is defined informally as a collec-tion—often a large and complex one—of objects called components, that are con-nected to form a coherent entity with a specific function or purpose. The functionof the system is determined by the functions of its components and how the compo-nents are connected. We are interested in information-processing systems whosefunction is to map a set A of input information items (a program and its data, forexample) into output information B (the results computed by the program acting onthe data). The mapping can be expressed formally by a mathematical function/from A to B. If/maps element a of A onto element b of B, we write b = /(a) or b :=f(a). We also restrict membership of A and B to digital or discrete quantities, whosevalues are defined only at discrete points of time. 2.1.1 System Representation A useful way of modeling a system is a graph. A (directed) graph consists of aset of objects V = {v,^^,...^,,} called nodes or vertices and a set of edges Ewhose members are (ordered) pairs of nodes taken from the set {(vl,v2),(V!,v3),...,(vn _,v„)}of all such pairs. The edge e = (v,-,yp joins or connects nodev, to node v.-. A graph is often defined by a diagram in which nodes are repre- 64 sented by circles, dots, or other symbols and edges are represented by lines: thisdiagram is synonymous with the graph. The ordering implied by the notation(v,,v) may be indicated in the diagram by an arrowhead pointing from v, to v as,for instance, in Figure 2.1. The systems of interest comprise two classes of objects: a set of information-processing components C and a set of lines S that carry information signalsbetween components. In modeling the system by a graph G, we associate C withthe nodes of G and S with the edges of G; the resulting graph is often called ablock diagram. This name comes from the fact that it is convenient to draw eachnode (component) as a block or box in which its name and/or its function can bewritten. Thus the various diagrams of computer structures presented in Chapter 1—Figure 1.29, for instance—are block diagrams. Figure 2.2 shows a block diagramrepresenting a small gate-level logic circuit called an EXCLUSIVE-OR or modulo-2adder. This circuit has the same general form as the more abstract graph of Fig-ure 2.1. 65 CHAPTER 2 Design Methodology Structure versus behavior. Two central properties of any system are its struc-ture and behavior; these very general concepts are often confused. We define thestructure of a system as the abstract graph consisting of its block diagram with nofunctional information. Thus Figure 2.1 shows the structure of the small system ofFigure 2.2. A structural description merely names components and defines theirinterconnection. A behavioral description, on the other hand, enables one to deter-mine for any given input signal a to the system, the corresponding output/(a). Wedefine the function/to be the behavior of the system. The behavior/may be repre-sented in many different ways. Figure 2.3 shows one kind of behavioral descriptionfor the logic circuit of Figure 2.2. This tabulation of all possible combinations ofinput-output values is called a truth table. Another description of the sameEXCLUSIVE-OR behavior can be written in terms of mathematical equations asfollows, noting that/(a) =/(x,^c2): /(0,0) = 0 /(0,1)=1 /(1,0) = 1 /(U) = 0 Figure 2.1 A graph with eight nodes and nine edges. 66 SECTION 2.1System Design AND *1 NOT • OR NOT AND p x2 o " »o x, © x2 Figure 2.2 A block diagram representing an EXCLUSIVE-OR logic circuit. The structural and behavioral descriptions embodied in Figures 2.1 and 2.3 areindependent: neither can be derived from the other. The block diagram of Figure2.2 serves as both a structural and behavioral description for the logic circuit inquestion, since from it we can derive Figures 2.1 and 2.3. In general, a block diagram conveys structure rather than behavior. For exam-ple, some of the block diagrams of computers in Chapter 1 identify blocks as beingarithmetic-logic units or memory circuits. Such functional descriptions do notcompletely describe the behavior of the components in question; therefore, we can-not deduce the behavior of the system as a whole from the block diagram. If weneed a more precise description of system behavior, we generally supply a separatenarrative text, or a more formal description such as a truth table or a list of equa-tions. Hardware description languages. As we have seen, we can fully describe asystem's structure and behavior by means of a block diagram—the term schematicdiagram is also used—in which we identify the functions of the components. Wecan convey the same detailed information by means of a hardware description lan-guage (HDL), a format that resembles (and is usually derived from) a high-levelprogramming language such as Ada or C. The construction of such description lan-guages can be traced back at least as far as Babbage [Morrison and Morrison1961]. Babbage's notation, of which he was very proud, centered around the use ofspecial symbols such as —> to represent the movement of mechanical components.In modern times Claude E. Shannon [Shannon 1938] introduced Boolean algebra Input a Output x\ x2 fia) 0 0 0 0 1 1 1 0 1 1 1 0 Figure 2.3 Truth table for the EXCLUSIVE-OR function. as a concise and rigorous descriptive method for logic circuits. Beginning in the1950s, academic and industrial researchers developed many ad hoc HDLs. Theseeventually evolved into a few widely used languages, notably VHDL and Verilog,1which were standardized in the 1980s and 90s [Smith 1996; Thomas and Moorby1996]. Hardware description languages such as VHDL have several advantages. Theycan provide precise, technology-independent descriptions of digital circuits at vari-ous levels of abstraction, primarily the gate and register levels. Consequently, theyare widely used for documentation purposes. Like programming languages, HDLscan be processed by computers and so are suitable for use with computer-aideddesign (CAD) programs which, as discussed later, play an important role in thedesign process. For example, an HDL description of a processor P can beemployed to simulate the behavior of P before all the details of its design havebeen specified. On the negative side, HDL descriptions are often long and verbose;they lack the intuitive appeal and rapid insights that circuit diagrams and less for-mal descriptive methods provide. 67 CHAPTER 2 Design Methodology EXAMPLE 2.1 VHDL DESCRIPTION OF A HALF ADDER. To illustrate the use of HDLs, we give in Figure 2.4a a VHDL description of a simple logic componentknown as a half adder. Its purpose is to add two 1-bit binary numbers x and y to form a2bit result consisting of a sum bit sum and a carry bit carry. For example, if x = y= 1.the half adder should produce carry = 1, sum = 0, corresponding to the binary number10, that is, two. A VHDL description has two main parts: an entity part and an architecture part.The entity part is a formal statement of the system's structure at the highest level, thatis, as a single component. It describes the system's interface, which is the "face" pre-sented to external devices but says nothing about the system's behavior or its internalstructure. In this example the entity statement gives the half adder's formal namehalf_adder and the names assigned to its input-output (IO) signals; 10 signals arereferred to in VHDL by their connection terminals or ports. Inputs and outputs are entity half_adder is port (x.y: in bit; sum. earn-, out bit);end half judder; architecture behavior of half_adder isbegin sum alpha); NAND2: nand_gate port map (d => alpha, e => alpha./=> carry);end structure: (a) 69 CHAPTER 2 Design Methodology xor_circuitXOR c nand_gateNAND1 / half_adder alpha c nand_gateNAND2 / Figure 2.5 Half adder: (a) structural VHDL description; (b) block diagram. states that half_adder has a component called NAND1. which is of type nand_gate andhas its d, e, and/ports (terminals) mapped (connected) to the signals x, v, and alpha,respectively. 2.1.2 Design Process Given a system's structure, the task of determining its function or behavior istermed analysis. The converse problem of determining a system structure thatexhibits a given behavior is design or synthesis. Design problem. We can now state in broad terms the problem facing the com-puter designer or, indeed, any system designer. Given a desired range of behavior and a set of available components, determine astructure (design) formed from these components that achieves the desired behav-ior with acceptable cost and performance. While assuring the correctness of the new design's behavior is the overriding goalof the design process, other typical requirements are to minimize cost as measured 70 SECTION 2.1System Design by the cost of manufacture and to maximize performance as measured by the speedof operation. There are some other performance- and cost-related constraints to satisfy such as high reliability, low power consumption, and compatibility with exist-ing systems. These multiple objectives interact in poorly understood ways thatdepend on the complexity and novelty of the design. Despite careful attention to detail and the assistance of CAD tools, the initialversions of a new system often fail to meet some design objective, sometimes insubtle and hard-to-detect ways. This failure can be attributed to incomplete specifi-cations for the design (some mode of behavior was overlooked), errors made byhuman designers or their CAD tools (which are also ultimately due to humanerror), and unanticipated interactions between structure, performance, and cost. Forexample, increasing a system's speed to a desired level can make the cost unac-ceptably high. The complexity of computer systems is such that the design problem must bebroken down into smaller, easier tasks involving various classes of components.These smaller problems can then be solved independently by different designers ordesign teams. Each major design step is often implemented via the multistep oriterative process depicted by a flowchart in Figure 2.6. An initial design is created,perhaps in ad hoc fashion, by adapting an existing design of a similar system. Theresult is then evaluated to see if it meets the relevant design objectives. If not, thedesign is revised and the result reevaluated. Many iterations through the redesignand evaluation steps of Figure 2.6 may be necessary to obtain a satisfactory design. Computer-aided design. The emergence of powerful and inexpensive desktopcomputers with good graphics interfaces provides designers with a range of pro-grams to support their design tasks. CAD tools are used to automate, at least in (Begin J Construct aninitial design Evaluate its costand performance Modify the designto meet the goals Figure 2.6 Flowchart of an iterativedesign process. part, the more tedious design and evaluation steps and contribute in three importantways to the overall design process. • CAD editors or translators convert design data into forms such as HDL descrip-tions or schematic diagrams, which humans, computers, or both can efficientlyprocess. • Simulators create computer models of a new design, which can mimic thedesign's behavior and help designers determine how well the design meets vari-ous performance and cost goals. • Synthesizers automate the design process itself by deriving structures that imple-ment all or part of some design step. Editing is the easiest of these three tasks, and synthesis the most difficult.Some synthesis methods incorporate exact or optimal algorithms which, even ifeasy to program into CAD tools, often demand excessive amounts of computingresources. Many synthesis approaches are therefore based on trial-and error meth-ods and experience with earlier designs. These computationally efficient but inex-act methods are called heuristics and form the basis of most practical CA© t»«ls. Design levels. The design of a complex system such as a computer is carriedout at several levels of abstraction. Three such levels are generally recognized incomputer design, although they are referred to by various different names in the lit-erature: • The processor level, also called the architecture, behavior, or system level. • The register level, also called the register-transfer level (RTL). • The gate level, also called the logic level. As Figure 2.7 indicates we are naming each level for a key component treated asprimitive or indivisible at that level of abstraction. The processor level correspondsto a user's or manager's view of a computer. The register level is approximatelythe level of detail seen by a programmer. The gate level is primarily the concern ofthe hardware designer. These three design levels also correspond roughly to themajor subdivisions of integrated-circuit technology into VLSI, MSI, and SSI com-ponents. The boundaries between the levels are far from clear-cut, and it is com-mon to encounter descriptions that mix components from more than one level. 71 CHAPTER 2 Design Methodology It Information Level Components density units Time units Gate Logic gates, flip-flops. SSI Bits 10-'2to 10"9s Register Registers, counters,combinational circuits,small sequential circuits. MSI Words lfr'toio^s Processor CPUs, memories, 10 devices. VLSI Blocks ofwords ur'io io-'s Figure 2.7 The major computer design levels. 72 A few basic component types from each design level are listed in Figure 2.7. section 2 ^e ^°^c £ates rec°gnized as primitive at the gate level include AND, OR, System Design NAND, NOR, and NOT gates. Consequently, the EXCLUSIVE-OR circuit of Figure 2.2 is an example of a gate-level circuit composed of five gates. Thecomponent marked XOR in Figure 2.5b performs the EXCLUSIVE-OR functionand so can be thought of as a more abstract or higher-level view of the circuitof Figure 2.2, in which all internal structure has been abstracted away. Similarly,the half-adder block of Figure 2Ab represents a higher-level view of the three-component circuit of Figure 2.5b. We consider a half adder to be a register-levelcomponent. We might regard the circuit of Figure 2.5b as being at the registerlevel also, but because NAND is another gate type and XOR is sometimes treatedas a gate, this circuit can also be viewed as gate level. Figure 2.7 indicates some further differences between the design levels. Theunits of information being processed increase in complexity as one goes from thegate to the processor level. At the gate level individual bits (Os and Is) are pro-cessed. At the register level information is organized into multibit words or vec-tors, usually of a small number of standard types. Such words represent numbers,instructions, and the like. At the processor level the units of information are blocksof words, for example, a program or a data set. Another important difference lies inthe time required for an elementary operation; successive levels can differ by sev-eral orders of magnitude in this parameter. At the gate level the time required toswitch the output of a gate between 0 and 1 (the gate delay) serves as the time unitand typically is a nanosecond (ns) or less. A clock cycle of, say, 10 ns, is a com-monly used unit of time at the register level. The time unit at the processor levelmight be a program's execution time, a quantity that can vary widely. System hierarchy. It is customary to refer to a design level as high or low; themore complex the components, the higher the level. In this book we are primarilyconcerned with the two highest levels listed in Figure 2.7, the processor and regis-ter levels, which embrace what is generally regarded as computer architecture. Theordering of the levels suggested by the terms high and low is, in fact, quite strong.A component in any level L, is equivalent to a (sub) system of components takenfrom the level L, _ , beneath it. This relationship is illustrated in Figure 2.8. For-mally speaking, there is a one-to-one mapping ht between components in L, anddisjoint subsystems in level L,-.,;a system with levels of this type is called a hier-archical system. Thus in Figure 2.8 the subsystem composed of blocks 1, 3, and 4in the low-level description maps onto block A in the high-level description. Fig-ures 2Ab and 2.5b show two hierarchical descriptions of a half-adder circuit. Complex systems, both natural and artificial, tend to have a well-defined hier-archical organization. A profound explanation of this phenomenon has been givenby Herbert A. Simon [Simon 1962]. The components of a hierarchical system ateach level are self-contained and stable entities. The evolution of systems fromsimple to complex organizations is greatly helped by the existence of stable inter-mediate structures. Hierarchical organization also has important implications in thedesign of computer systems. It is perhaps most natural to proceed from higher tolower design levels because this sequence corresponds to a progression of succes-sively greater levels of detail. Thus if a complex system is to be designed usingsmall-scale ICs or a single IC composed of standard cells, the design process mightconsist of the following three steps. x\ 1 2 * 1 A 5 1 rl 3 ►4 to * (a) Figure 2.8 5 Two descriptions of a hierarchical system: (a) low level; (b) high level. 1. Specify the processor-level structure of the system. 2. Specify the register-level structure of each component type identified in step 1. 3. Specify the gate-level structure of each component type identified in step 2. This design approach is termed top down; it is extensively used in both hardwareand software design. If the foregoing system is to be designed using medium-scaleICs or standard cells, then the third step, gate-level design, is no longer needed. As might be expected, the design problems arising at each level are quite dif-ferent. Only in the case of gate-level design is there a substantial theoretical basis(Boolean algebra). The register and processor levels are of most interest in com-puter design, but unfortunately, design at these levels is largely an art that dependson the designers' skill and experience. In the following sections we examine designat the register and processor levels in detail, beginning with the better-understoodregister level. We assume that the reader is familiar with binary numbers and withgate-level design concepts [Armstrong and Gray 1993; Hayes 1993; Hachtel andSomenzi 1996], which we review in the next section. 73 CHAPTER 2 Design Methodology 2.1.3 The Gate Level Gate-level (logic) design is concerned with processing binary variables whose pos-sible values are restricted to the bits (binary digits) 0 and 1. The design componentsare logic gates, which are simple, memoryless processing elements, and flip-flops,which are bit-storage devices. Combinational logic. A combinational film rum, also referred to as a logic, or aBoolean function, is a mapping from the set of 2" input combinations of n binaryvariables onto the output values 0 and 1. Such a function is denoted by r(.v,. v: 74 SECTION 2.1System Design xn) or simply by z. The function z can be defined by a truth table, which specifiesfor every input combination (jc1, x2,..., xn) the corresponding value of z{xx, x2,...,xn). Figure 2.9a shows the truth table for a pair of three-variable functions, s0(xq,v'o c_,) and c0(xq, Vq, c_,), which are the sum and carry outputs, respectively, of alogic circuit called a full adder. This useful logic circuit computes the numericalsum of its three input bits using binary (base 2) arithmetic: c&0 = xQphisy0plusc_] (2.2) For example, the last row of the truth table of Figure 2.9a expresses the fact thatthe sum of three Is is CqS0 = 112, that is, the base-2 representation of the numberthree. When discussing logic circuits, we will normally reserve the plus symbol (+)for the logical OR operation, and write out plus for numerical addition. We willalso use a subscript to identify the number base when it is not clear from the con-text; for example, twelve is denoted by 1210 in decimal and by 11002 in binary. A combinational function z can be realized in many different ways by combi-national circuits built from the standard gate types, which include AND, OR, Inputs Outputs xo >'o C-i co 5o 0 0 0 0 0 0 0 1 0 1 0 1 0 0 1 0 1 1 1 0 1 0 0 0 1 1 0 1 1 0 0 1 0 1 1 1 1 1 1 EXCLUSIVE-OR gate >o Half adder Half adder OR gate CO NAND gate NAND gate used as an inverter (a) (b) x0 < Vn ,_ --1 xi) .'0 C 1 ,2'>'i'>'o) and computes their sum S = (s3,s2,S\,s0y, it also accepts an inputcarry signal c_, and produces an output carry c3. A multibit adder is treated as aprimitive component at the register level, as shown Figure 2.10b, at which point itsinternal structure or logic design may no longer be of interest. Flip-flops. By adding memory to a combinational circuit in the form of 1-bitstorage elements called flip-flops, we obtain a sequential logic circuit. Flip-flopsrely on an external clock signal CK to synchronize the times at which they respond 2This design, which is known as a ripple-carry adder, and other types of binary adders are examined in detailin Chapter 4. *3 >3 x2 >'2 77 x y cuFull adder Cnnr S x y cuFull adder x y c,,Full adder c y c«Full adder -' CHAPTER 2DesignMethodology c0 (b) Figure 2.10 Four-bit ripple-carry: (a) logic structure; (b) high-level symbol. to changes on their input data lines. They are also designed to be unaffected bytransient signal changes (noise) produced by the combinational logic that feedsthem. An efficient way to meet these requirements is edge triggering, which con-fines the flip-flop's state changes to a narrow window of time around one edge (the0-to-l or l-to-0 transition point) of CK. Figure 2.11 summarizes the behavior of the most common kind of flip-flop, anedge-triggered D {delay) flip-flop. (Another well-known flip-flop type, the JK flip-flop, is discussed in problem 2.11.) The output signal y constitutes the stored dataor state of the flip-flop. The D flip-flop reads in the data value on its D line whenthe 0-to-l triggering edge of clock signal CK arrives; this D value becomes the newvalue of y. The triangular symbol on the clock's input port in Figure 2.1 la specifiesedge triggering; its omission indicates level triggering, in which case the flip-flop(then usually referred to as a latch) responds to all changes in signal value on D.Since there is just one triggering edge in each clock cycle, there can be just onechange in y per clock cycle. Hence we can view the edge-triggered flip-flop as tra-versing a sequence of discrete state values v(/), one for every clock cycle i. The input data line D can be varied independently and so can go through sev-eral changes in any clock cycle i. However, only the data value D{i) present justbefore the arrival of the triggering edge of CK determines the next state y{i + 1).To change the flip-flop's state, the D signal must be held steady for a minimumperiod known as the setup time Tselup before the flip-flop is triggered. For exam-ple, in Figure 2.1 lc, which shows a sample of the D flip-flop's behavior, we haveD(l) = 1 and v(l) = 0 in clock cycle 1. At the start of the next clock cycle, ychanges to 1 in response to D(l) = 1. making v(2) = 1. In clock cycle 3, y changes 78 SECTION 2.1System Design Data Clock PRED >CK — > Input Di i)0 1 CLR State 0 0 10 1 Next statevO'+l) 1 (a) (b) T Triggen ng edge Glitch/ 'setup 1 01 1 0 0 \ / 1 1 II Time ►■ 1/ 1 0 1 21 D 30 0 40 1 51 6 1 Clock CKDataD State y Cycle iDataD(/)State >■(/) (c) Figure 2.11 D flip-flop: (a) graphic symbol; (b) state table; (c) timing diagram. back to 0, making y(3) = 0. Even though D = 1 for most of clock cycle 3, D(3) = 0during the critical setup phase of cycle 3, thus ensuring that y(4) = 0. Observe thatthe spurious pulse or glitch affecting D in cycle 5 has no effect on y. Hence edge-triggered flip-flops have the very useful property of filtering out noise signalsappearing at their inputs. When a flip-flop is first switched on. its state y is uncertain unless it is

explic-itly brought to a known initial state. It is therefore desirable to be able to initialize(reset) the flip-flop asynchronously, that is, independently of the clock signal CK,at the start of operation. To this end, a flip-flop can have one or two asynchronouscontrol inputs, CLR (clear) and PRE (preset), as shown in Figure 2.11a. Each isdesigned to respond to a brief input pulse that forces y to 0 in the case of CLR or to1 in the case of PRE. In normal synchronous operation with a clock that is matched to the timingcharacteristics of its flip-flops, we can be sure that one well-defined change of statetakes place in a sequential circuit during each clock cycle. We do not have to worryabout the exact times at which signals change within the clock cycle. We can there-fore consider the actions of a flip-flop, and hence of any sequential circuit employ-ing it, to occur at a discrete sequence of points of time /= 1, 2, 3, ... In effect, theclock quantizes time into discrete, technology-independent time steps, each ofwhich represents a clock cycle. We can then describe a D flip-flop's next-statebehavior by the following characteristic equation: y(/+l) = D(/) (2.5) which simply says that y takes the value of D delayed by one clock cycle, hence theD flip-flop's name. Figure 2.1 \b shows another convenient way to represent the flip-flop's next-state behavior. This state table tabulates the possible values of the next state y{i + 1)for every possible combination of the present input D(i) and the present state y(i). Itis not customary (or necessary) to include clock-signal values explicitly in charac-teristic equations or state tables. The clock is considered to be the implicit generatorof time steps and so is always present in the background. Asynchronous inputs arealso omitted as they are associated only with initialization. 79 CHAPTER 2 Design Methodology Sequential circuits. A sequential circuit consists of a combinational circuitand a set of flip-flops. The combinational logic forms the computational or data-processing part of the circuit. The flip-flops store information on the circuit's pastbehavior; this stored information defines the circuit's internal state Y. If the pri-mary inputs are X and the primary outputs are Z, then Z is a function of both X andY, denoted Z(X,Y). It is usual to supply a sequential circuit with a precisely con-trolled clock signal that determines the times at which the flip-flops change state;the resulting circuit is said to be clocked or synchronous. Each tick (cycle orperiod) of the clock permits a single change in the circuit's state Y as discussedabove; it can also trigger changes in the primary output Z Reflecting the impor-tance of state behavior, the term finite-state machine (FSM) is often applied to asequential circuit. The behavior of a sequential circuit can be specified by a state table thatincludes the possible values of its primary outputs and its internal states. Figure2.12a shows the state table of a small but useful sequential circuit, a serial adder,which is intended to add two unsigned binary numbers X, and X2 of arbitrarylength, producing their sum Z = X{ plus X2. The numbers are supplied serially, thatis, bit by bit, and the result is also produced serially. In contrast, the combinational Input x 1*2 00 01 10 11 Present S0(y = 0) 50.0 50.1 50.1 s,.o state 5,.0 5,.l S,(y=l) s0.i S,,0 Next Presentstate output (a) Figure 2.12 (a) State table; (b) logic circuit for a serial adder. D nip-Hop Clock 80 adder of Figure 2.10 is a "parallel" adder, which, ignoring its internal-signal propagation delays, adds all bits of the input numbers simultaneously. In one clock cycle System Design '"' ^ se"a^ adder receives 2 input bits Xy(i) and x2(i) and computes 1 bit z(i) of Z It also computes a carry signal c(i) that affects the addition in the next clock cycle.Thus the output computed in clock cycle i is c(i)z(i) = x^Oplus x2(i)plus c(i - 1) (2.6) where c(i - 1) must be determined from the adder's present state S(i). Observe that(2.6) is equivalent to the expression (2.2) for the full-adder function defined earlier.It follows that two possible internal states exist: 50, meaning that the previous carrysignal c(i - 1) = 0, and Sx, meaning that c(i - 1) = 1. These considerations lead tothe twostate state table of Figure 2.12a. An entry in row 5(0 and column x^x^i)of the state table has the format S(i + 1), z(i), where S(i + 1) is the next internal statethat the circuit must have when the present state is 5(0 and the present primaryinput combination is xl(i)x2(i); z(i) is the corresponding primary output signal thatmust be generated. Because the serial adder has only two internal states, its memory consists of asingle flip-flop storing a state variable y. There are only two possible ways toassign 0s and Is to y. We select the "natural" state assignment that has y = 0 for 50and y = 1 for Sx, since this equates >(/) with the stored carry signal c(i - 1). Assumethat we use an edgetriggered D flip-flop (Figure 2.11) to store y. The combina-tional logic C then must generate two signals: the primary output z(i) and a second-ary output signal D(i) that is applied to the D flip-flop's data input. The flip-flop'sbehavior is defined by its characteristic equation (2.5); that is, y(i + 1) = D(i).Hence we have D(i) = c(0 It follows from the above discussion that C can be implemented directly by a full-adder circuit such as that of Figure 2.9b, whose sum output is z and whose carryoutput is D; see Figure 2.12b. Before entering two new numbers to be added, it isnecessary to reset the serial adder to the 50 state. The easiest way to do so is toapply a reset pulse to the flip-flop's asynchronous clear (CLR) input. Example 2.2 involves a similar, but more complex sequential circuit and dem-onstrates the use of CAD tools in its design. example 2.2 design of a 4-bit-stream serial adder. Consider another typeof serial adder that adds four number streams instead of the two handled by a conven-tional serial adder (Figure 2.12). The new adder has four primary input lines jc,, x2, x3,x4 and a single primary output z. To determine the circuit's state behavior—often themost difficult part of the design process—we first identify the information to be stored.As in the standard serial adder case, the circuit must remember carry information computed in earlier clock cycles. The current 2-bit sum SUM(i) = c(i)z(i) is given by SUM(i) = xx{i)plus x2(i)plus x3(/)/?/us x4(i)plus c(i -1) where c(i -1) is the carry computed in the preceding clock cycle. If c(i -1) is 0 andeach xfi) = 1, then SUM(i) = 1 plus 1 plus 1 plus 1 plus 0 = 4 = 1002, so c(i) = 102.With c(i -1) = 102, SUM{i) becomes 6 = 1102, making c{i) = 112. Finally, c(i - 1) =112 makes SUM(i) = 1112 and c(i) = 112, which is the maximum possible value of c.The carry data to be stored is a binary number ranging from 002 to 112, which implies that the adder needs four states and two flip-flops. We will denote the four states by50, 5,, S2, S3, where 5, represents a stored carry of (decimal) value i. Figure 2.13a shows the adder's state table, which has four rows and 16 columns.For present state S(i) and input combination j, the next-state/output entry Sk,z isobtained by adding i2 and the 4 input bits that determine 7 to form SUM(i) = (k2k]k0)2. Itfollows that k = (k2ki)2 and z = k0. For example, with present state S2 and present input7, SUM(i) = 0 plus 1 plus 1 plus 1 plus 102 = 1012, so z = 1 and A: =102 = 2, making S-,the next-state. Following this pattern, it is straightforward to construct the adder's statetable. With D flip-flops, the next-state values >',(/ + l)y2(i + I) coincide with the flip-flops' data input values D{(i)D2(i). The adder thus has the general structure shown inFigure 2.13£>. A truth table for the combinational logic C appears in Figure 2.13c. It is deriveddirectly from Figure 2.13a with the states assigned the four bit patterns of >',y2 as follows: S0 = 00, 5, = 01, S2 = 10, and 53= 11. Suppose we want to design Cas a two-level il CHAPTER 2 Design Methodology Present inputs xlx2xix4 (decimal) 0 1 2 So 3 4 5 6 7 8 9 10 S0,0 S,,. 1 S0, 1 s,.o S0.1 S,.0 S,.0 S,.l S0.1 S[,0 S,,0 11 12 13 14 15 S,,l S,,0 S,,l S,.l S2,0 Present 5i s0,1 s,,o s,.o S,.l S,,0 S,, 1 S,, 1 S2,0 S,,0 S„ 1 S,,l state S-> S,,0 S,.l S,,l s2,o S,. 1 S2.0 S2,0 S2.1 S,.l S2.0 S2,0 S2.l s2.0 s2,1 s2,1 s3,0 s2.0 s2.1 s2.1 S3.0 s2.1 s3,0 s3.0 s3.1 S„ 1 5:.0 S2,0 (a) Combinationallogic C CK< CK< Dy Reset Clock Present Present Secondary- Primary inputs state outputs output A" j X2 Xy X4 >'l >'2 £», D2 z 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 2 0 0 0 0 1 0 0 1 0 3 0 0 0 0 1 1 0 1 1 4 0 0 0 1 0 0 0 0 1 5 0 0 0 1 0 1 0 1 0 6 0 0 0 1 1 0 0 1 1 7 0 0 0 1 1 1 1 0 0 8 0 0 0 10 0 0 0 0 1 59 1 1 1 0 1 1 1 1 0 S2.0 S,, 1 S2.0 S2.0 S2,1 60 1 1 1 1 0 0 1 0 0 61 1111 0 1 1 0 1 62 1 1 1 1 0 1 1 1 0 63 1 1 1 1 1 1 1 1 1 (b) (c) Figure 2.13 Four-bit-stream serial adder: (a) state table; (b) overall structure; (c) truth table for 82 % espresso -Dexact Example 2.2 SECTION 2.1System Design .1 6 .o 3. 26 1010-1 001 .p 51 27 0110-1 001 1 -00010 010 28 1001-1 001 2 0-0010 010 29 0101-1 001 3 00-010 010 30 0011-1 001 4 000-10 010 31 -11111 010 5 00001- 010 32 1-1111 010 6 1000-0 001 33 11-111 010 7 0100-0 001 34 111-11 010 8 0010-0 001 35 11111- 010 9 0001-0 001 36 1111-1 001 10 0000-1 001 37 -111-1 100 11 -11000 010 38 1-11-1 100 12 1-0100 010 39 11-1-1 100 13 01-100 010 40 111—1 100 14 101-00 010 41 1111— 100 15 0-1001 010 42 —Ill 100 16 10-001 010 43 —1-11 100 17 -00101 010 44 -1—11 100 18 010-01 010 45 1—11 100 19 11000- 010 46 —111- 100 2 0 00110- 010 47 -1-11- 100 21 1110-0 001 48 1—11- 100 22 1101-0 001 49 -11-1- 100 23 1011-0 001 50 1-1-1- 100 24 0111-0 001 51 11—1- 100 25 1100-1 001 . e Figure 2.14 ///I \ \AX Minimal two-level (SOP) design for C com- x, x2 x3 x4 >>, y2 D, £>2 z puted by ESPRESSO. circuit like that of Figure 2.9c, using the minimum number of gates. Manual minimiza-tion methods [Hayes 1993] are painfully slow in this case without computer aid. Wehave therefore used a logic synthesis program called Espresso [Brayton et al. 1984;Hachtel and Somenzi 1996] to obtain a two-level SOP design. To instruct Espresso tocompute the minimum-cost SOP design on a UNIX-based computer requires issuing acommand like ^espresso seradd4 where seradd4 is a file containing the truth table of Figure 2.13c or an equivalentdescription of C. Espresso responds with the table of Figure 2.14, which specifies anSOP design containing the fewest product terms (these are in a minimal form calledprime implicants [Hayes 1993]), in this case, 51. For example, row 26, which has theformat x]x2x3x4yiy2 DiD2z= 1010-1 001 states that output z (but not the outputs D, or D2) has xix2xix4y2 as one of its chosenproduct terms. The dash in 1010-1 indicates a literal, in this case ylt that is not includedin the term in question. Similarly, row 51 (11 - -1 - 100) states that xix2yi is a term ofDy We conclude from Figure 2.14 that an SOP realization of C for the fourstreamadder has 51 product terms, none of which happen to be shared among the output func-tions. This conclusion implies a two-level circuit containing the equivalent of at least54 gates (51 ANDs and three ORs), some—especially the OR gates—with very highfan-in, which makes this type of two-level design expensive and impractical for manyIC technologies. Example 2.6 in section 2.2.3 shows an alternative approach that leadsto a lower-cost, multilevel design for this adder. Minimizing the number of gates in a sequential circuit is difficult because it isaffected by the flip-flop types, the state assignment, and, of course, the way inwhich the combinational subcircuit C is designed. Other design techniques exist tosimplify the design process at the expense of using more logic elements. It isimpractical to deal with complete binary descriptions like state tables if they con-tain more than, say, a dozen states. Consequently, large, sequential circuits aredesigned by heuristic techniques whose implementations use reasonable but non-minimal amounts of hardware [Hayes 1993; Hachtel and Somenzi 1996]. Thesecircuits are often best designed at the more abstract register level rather than thegate level. 83 CHAPTER 2 Design Methodology 2.2 THE REGISTER LEVEL At the register or register-transfer level, related information bits are grouped intoordered sets called words or vectors. The primitive components are small combina-tional or sequential circuits intended to process or store words. 2.2.1 Register-Level Components Register-level circuits are composed of word-oriented devices, the more importantof which are listed in Figure 2.15. The key sequential component, which gives thislevel of abstraction its name, is a (parallel) register, a storage device for words.Other common sequential elements are shift registers and counters. A number ofstandard combinational components exist, ranging from general-purpose devices,such as word gates, to more specialized circuits, such as decoders and adders. Type Component Functions Combinational Sequential Word gates.Multiplexers.Decoders and encoders.Adders. Arithmetic-logic units.Programmable logic devices. (Parallel) registers.Shift registers. Logical (Boolean) operations. Data routing: general combinational functions. Code checking and conversion. Addition and subtraction. Numerical and logical operations. General combinational functions. Information storage. Information storage; serial-parallel converCounters. Programmable logic devices. Control/timing signal generation.General sequential functions. Figure 2.15 The major component types at the register level. 84 SECTION 2.2The Register Level Register-level components are linked to form circuits by means of word-carryinggroups of lines, referred to as buses. Types. The component types of Figure 2.15 are generally useful in register-level design; they are available as MSI parts in various IC series and as standardcells in VLSI design libraries. However, they cannot be identified a priori based onsome property analogous to the functional completeness of gate-level operations.For example, we will show that multiplexers can realize any combinational func-tion. This completeness property is incidental to the main application of multiplex-ers, which is signal selection or path switching. There are no universally accepted graphic symbols for register-level compo-nents. They are usually represented in circuit diagrams by blocks containing anabbreviated description of their behavior, as in Figure 2.16. A single signal line in adiagram can represent a bus transmitting m > 1 bits of information in parallel; m isindicated explicitly by placing a slash (/) in the line and writing m next to it (seeFigure 2.16). A components's 10 lines are often separated into data and controllines. An m-bit bus may be given a name that identifies the bus's role, for example,the type of data transmitted over a data bus. A control line's name indicates theoperation determined by the line in its active, enabled, or asserted state. Unlessotherwise indicated, the active state of a bus occurs when its lines assume the logi-cal 1 value. A small circle representing inversion is placed at an input or outputport of a block to indicate that the corresponding lines are active in the 0 state andinactive in the 1 state. Alternatively, the name of a signal whose active value is 0includes an overbar. The input control lines associated with a multifunction block fall into twobroad categories: select lines, which specify one of several possible operations thatthe unit is to perform, and enable lines, which specify the time or condition for aselected operation to be performed. Thus in Figure 2.16, to perform some operationFx, first set the select line F to a bit pattern denoting F{ and then activate the edge-triggered enable line £by applying a O-to-1 edge signal. Enable lines are often con-nected to clock sources. The output control signals, if any, indicate when or howthe unit completes its processing. Figure 2.16 indicates termination by 5 = 0. Thearrowheads are omitted when we can infer signal direction from the circuit struc-ture or signal names. Function kselect F Enable E Controlinput lines Data input linesAi A-? A-i i /f m /T m X Z, Z2 Data output lines Control output lines Figure 2.16 Generic block representation of aregister-level component. Operations. Gate-level logic design is concerned with combinational func-tions whose signal values are from the two-valued set B = {0,1} and form a Bool-ean algebra. We can extend these functions to functions whose values are takenfrom Bm, the set of 2m m-bit words, rather than from B. Let z(xx,x2,...,xn) be anytwo-valued combinational function. Let Xx,X2,...,Xn denote m-bit binary wordshaving the form X, = (xiti,xi^,...,xi^) for / = 1,2,...,«. We define the word opera-tion z as follows: z(Xl,X2,...,Xn) = [z(xl {,x2 u...,xn l)^(xl2,x22,...,xn2),...,zixljn,x2jn,...,xnjn]] (2.7) This definition simply generalizes the usual Boolean operations, AND, NAND,and so forth, from 1-bit to m-bit words. If z is the OR function, for instance, wehave Xl+X2+- +Xn = (*lfl + x2A + ■■■+ xnAsh2 + x22 + ■■■+ xn2, ■ • •' X\jn + xljm + '•• + xn,m) which applies OR bitwise to the corresponding bits of n m-bit words.2mnThe set of all 2 combinational functions defined on n m-bit words forms a Boolean algebra with respect to the word operations for AND, OR, and NOT. Thisgeneralization of Boolean algebra to multibit words is analogous to the extensionof the ordinary algebra from single numbers (scalars) to vectors. Pursuing thisanalogy, we can treat bits as scalars and words as vectors, and obtain more com-plex logical operations, such as yX=(yxl,yx2, ...,yxjy + X = (y + x],y + x2, ...,y + xj (2.8) Word-based logical operations of this type are useful in some aspects of register-level design. However, they do not by themselves provide an adequate design the-ory for several reasons. • The operations performed by some basic register-level components are numeri-cal rather than logical; they are not easily incorporated into a Boolean frame-work. • Many of the logical operations associated with register-level components arecomplex and do not have the properties of the gates —interchangeability ofinputs, for example—that simplify gate-level design. • Although a system often has a standard word length w based on the width ofsome important buses or registers, some buses carry signals with a differentnumber of bits. For example, the outcome of a test on a set 5 of vv-bit words(does S have property PI) is 1 bit rather than w. The lack of a uniform word sizefor all signals makes it difficult to define a useful algebra to describe operationson these signals. Lacking an adequate general theory, register-level design is tackled mainly withheuristic and intuitive methods. We next introduce the major combinational and sequential components used indesign at the register level. (Refer to Figure 2.15). 85 CHAPTER 2 Design Methodology 86 SECTION 2.2The Register Level Word gates. Let X = (xux2,...,xm) and Y = (yi,y2,...,y„,) be two m-bit binarywords. As noted already, it is useful to perform gate operations bitwise on X and Yto obtain another m-bit word Z = (zi,z2,...,Zm). We coin the term word-gate opera-tions for logical functions of this type. In general, if/is any logic operator, wewrite Z=f(X,Y) if z, =/(jc,,y,) for i = l,2,...,m. For example, Z= XY denotes the m-bit NAND operation defined by Z=(zl,z2,...,zm) = (xly1,x2y2, ...,xmym) This generalized NAND is realized by the gate-level circuit in Figure 2.17a. It isrepresented in register-level diagrams by the two-input NAND symbol of Figure2.17b, which is an example of a word gate. It is also useful to represent scalar-vector operations by a single gate symbol. For example, the operation y + Xdefined by (2.8) and realized by the circuit of Figure 2.18a can be represented bythe register-level gate symbol of Figure 2.18b. Word gates are universal in that they suffice to implement any logic circuit;moreover, word-gate circuits can be analyzed using Boolean algebra. In practice,however, the usefulness of word gates is severely limited by the relative simplicityof the operations they perform and by the variability in word size found at the reg-ister level. *i y\ x2 >'2 X Y m ,' / m (a) V z (b) Figure 2.17 Two-input, m-bit NAND word gate: (a) logic diagram and (b) symbol. (a) X y m / / 1 Z (b) Figure 2.18 OR word gate implementing y + X: (a) logic diagram; (b) symbol. Multiplexers. A multiplexer is a device intended to route data from one ofseveral sources to a common destination; the source is specified by applyingappropriate control (select) signals to the multiplexer. If the maximum number ofdata sources is k and each 10 data line carries m bits, the multiplexer is referred toas a k-input (or k-way), mbit multiplexer. It is convenient to make k = 2P, so thatdata source selection is determined by an encoded pattern or address of p bits.The 2P addresses then cover the range 00...0, 00...1, ..., 11...1 = 2P - 1. A multi-plexer is easily denoted by a suitably labeled version of the generic block symbolof Figure 2.16; the tapered block symbol shown in Figure 2.19, where the narrowend indicates the output side, is also common. Let a{ = 1 when we want to select the m-bit input data bus X, = (jc,-^*, (,...,xi,m-\) °f me multiplexer of Figure 2.19. Then at = 1 when we apply the word cor-responding to the binary number i to the select bus 5. The binary variable a,denotes the selection of input data bus X,—a, is not a physical signal. The dataword on X, is then transferred to Z when e = 1. The operation of the 2^-input w-bitmultiplexer is therefore defined by m sum-of-product Boolean equations of theform Zj= (x0ja0 + xljal+ ••• +x2p_i ja2p_i)e for;' = 0, 1, ...,m- 1 (2.9) or by the single word-based equation Z= (X0a0 + Xlal + {a2P-i)e Figure 2.20 shows a typical gate-level realization of a two-input, 4-bit multiplexer.Several &-input multiplexers can be used to route more than k data paths byconnecting them in the treelike fashion shown in Figure 2.21. A g-level tree cir-cuit of this type forms a ^-input multiplexer. A distinct select line is associatedwith every level of the tree and is connected to all multiplexers in that level. Thuseach level performs a partial selection of the data line X, to be connected to theoutput Z. Multiplexers as function generators. Multiplexers have the interesting prop-erty that they can compute any combinational function and so form a type of uni-versal logic generator. Specifically, a 2"-input, 1-bit multiplexer MUX cangenerate any ^-variable function z(v0,v,,...,v„_,). This is accomplished by apply-ing the n input variables v0,v,,...,vn_, to the n select Ymes s0,s],...,sn_]of MUX, and2" function-specific constant values (0 or 1) to MUX's 2" input data lines .v0,.v, 87 CHAPTER 2 Design Methodology Data in X, 1 %2Pi- m r ... m)r Select S Enable e P \ 0 1 Multiplexer(MUX) Data out Z Figure 2.19 A 2/'-input, m-bit multiplexer. Data in xn ft x o.o *i.o *o. 1 xi, 1 *0,2 ^1.2 x0.3 xl.l SECTION 2.2The Register Level Select s Enable e Data out z0 Figure 2.20 Realization of a two-input, 4-bit multiplexer. Data in An X< X2 X3 X4 X5 X6 Xn ^'—To T7 I—To i7 |—To i7 I—To 1 A Mux / 1 A Mux / 1 A Mux / 1 A Mux Select •< ^i —» Enable e 0 11 4 Mux Mux 1 A Mux / Data out Z Figure 2.21 An eight-input multiplexer constructed from two-input multiplexers. jc2n_j. The output of MUX is then Z= (x0a0 + xla1 + ••• +x2"_,a2«_1)e (2.10) as defined by (2.9), where again a, denotes the selection of input data bus jc,.Clearly, a, corresponds to the z'th row in z's truth table with respect to the inputvariables v0, v^..., vn_,. With e= 1, setting xt = 1 (0) if row i of the truth table for zis 1 (0) makes (2.10) into a sum-of-products expression for z. Hence by connectingeach input data line to the appropriate logic value 0 or 1, we can realize any of the2 possible logic functions of n variables. EXAMPLE 2.3 USING A MULTIPLEXER TO IMPLEMENT A FULL ADDER. As we saw in section 2.1, a full adder is a three-input, two-output circuit that adds 3 bits x0, y0, andc_] (the carry in) to obtain a 2-bit result consisting of s0 (the sum bit) and c0 (the carryout). It is the basic component of a serial adder (Figure 2.12) and has various gate-levelrealizations such as those of Figure 2.9. A multiplexer MUXX with m = 2 and n = 2P = 8,that is, an eight-input, 2-bit multiplexer, can implement the full adder, as shown in Fig-ure 2.22b. The adder's input variables are applied to the three select lines, not as mightbe expected, to the multiplexer's data input buses. Instead constant values 0 or 1 areapplied to the data inputs as indicated. Each pattern i of x^qC^ selects a specific inputdata bus X, and routes its 2-bit word to the output bus z = s0c0. Observe how this proce-dure effectively maps the truth table for .s0 and c0 (Figure 2.22a) directly onto M£/X,'sinput data lines. If one input variable of the full adder, say c_,, is available in both true and comple-mented form, we can implement the adder with the smaller, four-input, 2-bit multi-plexer MUX2 shown in Figure 2.22c. The two inputs x0, y0 are applied to M£/X2's selectlines as before, but we apply one of c_x, c_1? 0, or 1 to each line Xq of data bus X,. NowXjj must realize two rows of the form x$>00 and x^qI in the adder's truth table. If, forexample, these rows have the same fixed value a for the output (s0 or c0) of interest,then we apply a tox^-. If the rows have different values, then either c_, or c_, is applied 89 CHAPTER 2 Design Methodology Inputs Outputs *0 yo c-\ so co 0 0 0 0 0 0 0 1 1 0 0 1 0 1 0 0 1 1 0 1 1 0 0 1 0 1 0 1 0 1 1 1 0 0 1 1 1 1 1 1 (fl) Sum s0Carry c0 Sum sQ— Carry c0 (c) Figure 2.22 Multiplexer-based full adder: (a) truth table; (b) first version; (c) second version. 90 SECTION 2.2The Register Level to Xn, as appropriate. We see from this example that a 2"-input, m-bit multiplexer canrealize any (n + Invariable, w-output logic function. Decoders. A l-out-of-2" or 1/2" decoder is a'combinational circuit with ninput lines X and 2" output lines Z such that each of the 2" possible input combina-tions Aj applied to X activates a corresponding output line z(. Figure 2.23 shows a1/4 decoder. Several 1/2" decoders can be used to decode more than n lines byconnecting them in a tree configuration analogous to the multiplexer tree of Figure2.21. The main application of decoders is address decoding, where A, is interpretedas an address that selects a specific output line Z; or some circuit attached to z,. Forexample, decoders are used in RAMs to select storage cells to be read from orwritten into. Another common application of decoders is that of routing data from a com-mon source to one of several destinations. A circuit of this kind is called a demulti-plexer, since it is, in effect, the inverse of a multiplexer. In this application thecontrol input e (enable) of the decoder is viewed as a 1-bit data source to be routedto one of 2" destinations, as determined by the address applied to the decoder. Thusa 1/2" decoder is also a 2"-output, 1-bit demultiplexer. A £:-output, m-bit demulti-plexer can be readily constructed from a network of decoders. Figure 2.24 shows afour-output, 2-bit demultiplexer that employs two 1/4 decoders of the type in Fig-ure 2.23. Encoders. An encoder is a circuit intended to generate the address or index ofan active input line; it is therefore the inverse of a decoder. Most encoders have 2input data lines and k output data lines. For example, when k = 3, entering a data Enable e 1/4decoder Z0 Z\ z2 z3(b) Figure 2.23 A 1/4 decoder: (a) logic diagram; (b) symbol. Data in Select(address) 1/4decoder 1/4decoder Data out Z: Figure 2.24 A four-output, 2-bit demultiplexer. 91 CHAPTER 2 Design Methodology pattern such as x0xix2x3x4x5x6x1 = 00000010 into an eight-input encoder shouldproduce the response z2Z\Zq =110, denoting the number 6, and indicating that x6 =1. Additional (control) outputs are necessary to distinguish the input jc0 active andno input active states. Moreover, it is also necessary to assign priorities to the inputlines and design the encoder so that the output address is always that of the activeinput line with the highest priority. A circuit of this type is called a priority-encoder; see Figure 2.25. A fixed priority is assigned to each input line such that a,has higher priority than x if / >j. We leave the logic design of this priority encoderas an exercise (problem 2.22). Arithmetic elements. A few fairly simple arithmetic functions, notably addi-tion and subtraction of fixed-point numbers, can be implemented by combinationalregister-level components. Most forms of fixed-point multiplication and divisionand essentially all floating-point operations are too complex to be realized by sin-gle components at this design level. However, adders and subtracters for fixed-point binary numbers are basic register-level components from which we canderive a variety of other arithmetic circuits, as we will see later. Figure 2.26ashows a component that adds two 4-bit data words and an input carry bit: it iscalled a 4-bit adder. (A full adder is sometimes called a 1-bit adder.) The adder'scarry-in and carry-out lines allow several copies of this component to be chainedtogether to add numbers of arbitrary size; note, however, that the addition timeincreases with the number size. (See Chapter 4 for coverage of the design of addersand more-complex arithmetic circuits). Another useful arithmetic component is amagnitude comparator, whose function is to compare the magnitudes of* twobinary numbers. Figure 2.26b shows the overall structure of a 4-bit comparator. 92 SECTION 2.2The Register Level Input active Inputs Outputs e x0 x\ x2 xl x4 *s x6 *7 ^2 Zl zo ia 0 d d d d d d d d (1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 d rf rf d d d d 1 1 1 1 1 d rf d d d d 1 0 1 I 0 1 d d d d d l 0 0 1 0 1 1 d d d d 1 0 0 0 1 0 0 1 rf d d 1 0 0 0 0 0 1 1 1 d d 1 0 0 0 0 0 0 0 1 0 1 zm-?>>--->zOx) := (zm-\'Zm-2>---'Z\'Zo) In each case a bit of stored data is lost from one end of the shift register, while anew data bit x is brought in at the other end. In its simplest form, an m-bit shift reg-ister consists of m flip-flops each of which is connected to its left or right neighbor.Data can be entered 1 bit at a time at one end of the register and can be removed(read) 1 bit at a time from the other end; this process is called serial input-output.Figure 2.30 shows a 4-bit shift register built from D flip-flops. A right shift isaccomplished by activating the SHIFT enable line connected to the clock input CKof each flip-flop. In addition to the serial data lines, m input or output lines areoften provided to permit parallel data transfers to or from the shift register. Addi-tional control lines are required to select the serial or parallel input modes. A fur-ther refinement is to permit both left- and right-shift operations. 95 CHAPTER 2 DesignMethodology 4 ,' LOAD 1 0 2-way,5 multiplexer X 4 / LOADCLOCKCLEAR > Register Z Z(*) Figure 2.29 A 4-bit D register with parallel load: (a) logic diagram; (b) symbol. 96 SECTION 2.2The Register Level SHIFTCLEAR (a) SHIFT CLEAR Shift register (b) Figure 2.30 A 4-bit, right-shift register: (a) logic diagram; (b) symbol. Shift registers are useful design components in a number of applications,including storage of serial data and serial-to-parallel or parallel-to-serial data con-version. They can also be used to perform certain arithmetic operations on binarynumbers, because left- (right-) shifting corresponds to multiplication (division) bytwo. The instruction sets of most computers include shift operations. Counters. A counter is a sequential circuit designed to cycle through a prede-termined sequence of k distinct states 50,5,,..., Sk_j in response to signals (1 -pulses)on an input line. The k states represent k consecutive numbers, so the state transitionscan be described by the statement SM := 5, plus 1 (modulo k) Each 1-input increments the state by one; the circuit can therefore be viewed ascounting the input Is. Counters come in many different varieties depending on thenumber codes used, the modulus k, and the timing mode (synchronous or asynchro-nous). Figure 2.31 shows a counter designed to count 1-pulses applied to its COUNTENABLE input line. The counting is modulo-2"; that is, the counter's modulus k = 2", and it has 2" states Sn, S, '2--1The output is an n-bit binary number COUNT = Sj, and the count sequence is either up or down, as determined by thecontrol line DOWN. In the up-counting mode (DOWN= 0), the counter's behavior is S,+1 := 5, plus 1 (modulo 2") COUNT ENABLECLEARDOWN Modulo-2"up-downcounter COUNT Figure 2.31 A modulo-2'1 up-down counter. 97 CHAPTER 2 Design Methodology whereas in the down-counting mode (DOWN = 1), the behavior becomes5,+1 := S minus 1 (modulo 2") In some counters modulus-select control lines can alter the modulus; such countersare termed programmable. Counters have several applications in computer design. They can store thestate of a control unit, as in a program counter. Incrementing a counter provides anefficient means of generating a sequence of control states. Counters can also gener-ate timing signals and introduce precise delays into a system. Buses. A bus is a set of lines (wires) designed to transfer all bits of a wordfrom a specified source to a specified destination on the same or a different IC; thesource and destination are typically registers. A bus can be unidirectional, that is,capable of transmitting data in one direction only, or it can be bidirectional.Although buses perform no logical function, a significant cost is associated withthem, since they require logic circuits to control access to them and, when usedover longer distances, signal amplification circuits (drivers and receivers). The pinrequirements and gate density of an IC increase rapidly with the number of externalbuses connected to it. If these buses are long, the cost of the wires or cables usedmust also be taken into account. To reduce costs, buses are often shared, especially when they connect manydevices. A shared bus is one that can connect one of several sources to one of sev-eral destinations. Bus sharing reduces the number of connecting lines but requiresmore complex bus-control mechanisms. Although shared buses are relativelycheap, they do not permit simultaneous transfers between different pairs ofdevices, which is possible with unshared or dedicated buses. Bus structures areexplored further in Chapter 7. 2.2.2 Programmable Logic Devices Next we examine a class of components called programmable logic devices orPLDs, a term applied to ICs containing many gates or other general-purpose cellswhose interconnections can be configured or "programmed" to implement anydesired combinational or sequential function [Alford 1989]. PLDs are relativelyeasy to design and inexpensive to manufacture. They constitute a key technologyfor building application-specific integrated circuits (ASICs). Two techniques, areused to program PLDs: mask programming, which requires a few special steps in 98 SECTION 2.2The Register Level the IC chip-manufacturing process, and field programming, which is done bydesigners or end users "in the field" via small, low-cost programming units. Somefieldprogrammable PLDs are erasable, implying that the same IC can be repro-grammed many times. This technology is especially convenient when developingand debugging a prototype design for a new product. Programmable arrays. The connections leading to and from logic elements ina PLD contain transistor switches that can be programmed to be permanentlyswitched on or switched off. These switches are laid out in two-dimensional arraysso that large gates can be implemented with minimum IC area. The programmablelogic gates of a PLD array are represented abstractly in Figure 2.32b, with x denot-ing a programmable connection or crosspoint in a gate's input line. The absence ofan x means that the corresponding connection has been programmed to the off (dis-connected) state. The gate structures of Figure 232b can be combined in various ways to imple-ment logic functions. The programmable logic array (PLA) shown in Figure 2.33is intended to realize a set of combinational logic functions in minimal SOP form.It consists of an array of AND gates (the AND plane), which realize a set of prod-uct terms (prime implicants), and a set of OR gates (the OR plane), which formvarious logical sums of the product terms. The inputs to the AND gates are pro-grammable and include all the input variables and their complements. Hence it ispossible to program any desired product term into any row of the PLA. For exam-ple, the top row of the PLA in Figure 2.33 is programmed to generate the termx2x3x4y}y2, which is used in computing the output D2\ the last row is programmedto generate xxx2yx for output D,. The inputs to the OR gates are also programma-ble, so each output column can include any subset of the product terms producedby the rows. The PLA in Figure 2.33 realizes the combinational part C of the 4-bit-stream adder specified in Figure 2.13. The AND plane generates the 51 six-vari-able product terms according to the SOP design given in Figure 2.14. :L>(a) X X > x\ x2 *\ *2 (b) Figure 2.32 AND and OR gates: (a) normal notation; (b) PLD notation. AND planevy w • w w w 1 A i OR plane > >> \ f / ^ V s,,...,P7 are referredto as partial products. When the current multiplier bit x1_i is 1, (2.18) becomes P, :=P, + YM\ when x7_, = 0, (2.18) becomes P, := P, + 0. Hence step (2.18) requires add-ing either the multiplicand YM or 0 to the current partial product />,. The factor 2~l in(2.19) indicates that P, is right-shifted by 1 bit after each addition; this factor is equiv-alent to division by 2. Note that each add-and-shift step appends 1 bit to the partialproduct, which therefore grows from 7 to 15 bits (including the sign bit p0) over thecourse of the multiplication. With these preliminaries, we can now specify the main components needed formultiplier8. Two 8-bit registers, conventionally denoted Q (for multiplier-quotient) andM (for multiplicand), are required to store X and Y, respectively. A double-length, 16-bit register A (for accumulator) stores the P,'s; this standard length is more convenientthan the actual 15-bit maximum size of P. A 7-bit combinational adder is used for theaddition specified by (2.18) (The serial adder of Figure 2.12 could also be used, but itwould be about seven times slower.) The adder must have its output and one input con-nected to A, while its other input must be switched between M and zero. The 1-bitright-shift function (2.19) can be conveniently obtained by constructing A from a right-shift register with parallel IO. As specified by (2.18), addition is controlled by bit x1_i, which is stored in the Qregister. The multipliers control unit must be able to scan the contents of Q from rightto left in the course of the multiplication. If Q is a right-shift register, then x1_i canalways be obtained from Q's right-most flip-flop Q[l] by right-shifting Q before thenext x1_l is needed. Consequently, XSi is gradually reduced from 7 to 0 bits while Pt isexpanding from 7 to 14 bits, also by right-shifting. Hence we can combine A and Q intoa single 16-bit, right-shift register, the left half of which is A while the right half is Q.The multiplier is completed by the inclusion of external data buses INBUS and OUTBUS and a control unit, which contains a 3-bit iteration counter named COUNT. Theresulting circuit has the structure depicted in Figure 2.41. A complete HDL descriptionof the multiplication algorithm developed above appears in Figure 2.39. At the core of our design is the adder and the A.Q register that implement (2.18)and (2.19). respectively. The output-carry signal cOVT of the adder is the most signifi-cant bit of an 8-bit sum and so is connected to the data input of A[0}. The counterCOUNT is incremented and tested at the end of each add-shift step to determine if theaddshift phase should terminate. When COUNT is found to contain 7, PSi occupiesbits 1:14 of the register-pair A.Q; that is, bits A[1:7].Q[0:6]. The sign bit p0 is thencomputed from x0 and y0, which are stored in Q[l] and M[0], respectively, and p0 isplaced inA[0]. At the same time 0 is written into Q[l] to expand the final product from15 to 16 bits. Figure 2.42 shows the complete step-by-step multiplication process fortwo sample fractions X = 10110011 and Y = 01010101. The sign bit x0 = 1 of X (indi-cating that it is a negative number) is marked by an underline. The data in A.Q to theleft of .v0 is the current partial product P,. The control unit of Figure 2.41 is designed by first identifying from the formaldescription (Figure 2.39) all the control signals and control points needed to implementthe specified register-transfer operations. Figure 2.43 lists a possible set of control 113 CHAPTER 2 Design Methodology Controlsignal Operation controlled Clear accumulator A (reset to 0). Clear counter COUNT (reset to 0). LoadA[0]. Load multiplicand register M from INBUS. Load multiplier register Q from INBUS. Load main adder outputs into A[1:7]. Select M or 0 to apply to right input of adder. Right-shift A. Q. Increment counter COUNT. Select COUTot Af[0] xor Q[l\ to load into A[0]. Clear Q[l\. Transfer contents of A to OUTBIS Transfer contents of Q to OUTBUS. Figure 2.43 Control signals for multipliers. 114 SECTION 2.3 The Processor Level M[l:7] OLTBUS Figure 2.44 Implementation of some control points of multiplier8. signals for the multiplier. In some cases several control signals implement a particularoperation. For instance, the add operation employs c6 to select the adder's right inputoperand, c9 to select cOUT for loading into A[0], and c2 and c5 to actually load the 8-bitsum into v4[0:7]. The number of distinguished control signals will vary with the detailsof the logic used to implement the control unit. Figure 2.44 shows a straightforwardimplementation of the control logic associated with the accumulator and adder subcir-cuits using the control signals defined in Figure 2.43. 2.3 THE PROCESSOR LEVEL The processor or system level is the highest in the computer design hierarchy. It isconcerned with the storage and processing of blocks of information such as pro-grams and data files. The components at this level are complex, usually sequential,circuits that are based on VLSI technology. Processor-level design is very much aheuristic process, as there is little design theory at this level of abstraction. 2.3.1 Processor-Level Components 115 The component types recognized at the processor level fall into four main groups:processors, memories, IO devices, and interconnection networks; see Figure 2.45.In this section we give only a brief summary of the characteristics of processor-level components; they are examined individually and in much greater depth inlater chapters. CHAPTER 2 Design Methodology Central processing unit. We define a CPU to be a general-purpose, instruc-tion-set processor that has overall responsibility for program interpretation andexecution in a computer system. The qualifier general-purpose distinguishes CPUsfrom other, more specialized processors, such as IO processors (IOPs), whosefunctions are restricted. An instruction-set processor is characterized by the factthat it operates on word-organized instructions and data, which the processorobtains from an external memory that also stores results computed by the proces-sor. Most contemporary CPUs are microprocessors, implying that their physicalimplementation is a single VLSI chip. Figure 2.46 shows the essential internal organization of a CPU at the registerlevel. The CPU contains the logic needed to execute its particular instruction setand is divided into datapath and control units. The control part (the I-unit) gener-ates the addresses of instructions and data stored in external memory. In this par-ticular system a cache memory is interposed between the main memory M and theCPU. The cache is a fast buffer memory designed to hold an active portion of thesystem's address space; it is often placed, wholly or in part, on the same IC asthe CPU. Each memory request generated by the CPU is first directed to the cache.If the required information is not currently assigned to the cache, the request is re-directed to M and the cache is automatically updated from M. The I-unit fetchesinstructions from the cache or M and decodes them to derive the control signalsneeded for their execution. The CPU's datapath (E-unit) has the arithmetic-logiccircuits that execute most instructions; it also has a set of registers for temporarydata storage. The CPU manages a system bus, which is the main communicationlink among the CPU-cache subsystem, main memory, and the IO devices. Micro-processor(CPU) Mainmemory Interconnection network(system bus) Input/output devices(keyboard, video display,secondary memory, etc.) Figure 2.45 Major components of a computer system. 116 Main memory M and IO system SECTION 2.3 The Processor Level System bus Li j_ *!' i\ Cache If /i /i IF Programcounter PC Instructionregister IR Arithmetic-logic unit Registerfile t i Addressgeneration |#—T Instructiondecoding Prun ogram ccit (I-unit J i >n) i i ■■ 1 > 1 1 1 i_ — tro s Control signal Datapath(E-unit) Figure 2.46 Internal organization of a CPU and cache memory. The CPU is a synchronous sequential circuit whose clock period is the com-puter's basic unit of time. In one clock cycle the CPU can perform a register-transferoperation, such as fetching an instruction word from M via the system bus and load-ing it into the instruction register IR. This operation can be expressed formally by IR := M(PC); where PC is the program counter the CPU uses to hold the expected address of thenext instruction word. Once in the I-unit, an instruction is decoded to determine theactions needed for its execution; for example, perform an arithmetic operation ondata words stored in CPU registers. The I-unit then issues the sequence of controlsignals that enables execution of the instruction in question. The entire process offetching, decoding, and executing an instruction constitutes the CPU's instructioncycle. Memories. CPUs and other instruction-set processors operate in conjunctionwith external memories that store the programs and data required by the proces-sors. Numerous memory technologies exist, and they vary greatly in cost and per-formance. The cost of a memory device generally increases rapidly with its speedof operation. The memory part of a computer can be divided into several majorsubsystems: 1. Main memory M, consisting of relatively fast storage ICs connected directly to,and controlled by, the CPU. 2. Secondary memory, consisting of less expensive devices that have very highstorage capacity. These devices often involve mechanical motion and so aremuch slower than M. They are generally connected indirectly (via M) to theCPU and form part of the computer's 10 system. 3. Many computers have a third type of memory called a cache, which is posi-tioned between the CPU and main memory. The cache is intended to furtherreduce the average time taken by the CPU to access the memory system. Someor all of the cache may be integrated on the same IC chip as the CPU itself. Main memory M is a word-organized addressable random-access memory(RAM). The term random access stems from the fact that the access time for everylocation in M is the same. Random access is contrasted with serial access, wherememory access times vary with the location being accessed. Serial access memo-ries are slower and less expensive than RAMs; most secondary-memory devicesuse some form of serial access. Because of their lower operating speeds and serial-access mode, the manner in which the stored information is organized in secondarymemories is more complex than the simple word organization of main memory.Caches also use random access or an even faster memory-accessing method calledassociative or content addressing. Memory technologies and the organization ofstored information are covered in Chapter 6. IO devices. Input-output devices are the means by which a computer commu-nicates with the outside world. A primary function of 10 devices is to act as datatransducers, that is, to convert information from one physical representation toanother. Unlike processors, 10 devices do not alter the information content ormeaning of the data on which they act. Since data is transferred and processedwithin a computer system in the form of digital electrical signals, input (output)devices transform other forms of information to (from) digital electrical signals.Figure 2.47 lists some widely used 10 devices and the information media theyinvolve. Many of these devices use electromechanical technologies; hence theirspeed of operation is slow compared with processor and main-memory speeds.Although the CPU can take direct control of an IO device it is often under theimmediate control of a special-purpose processor or control unit that directs theflow of information between the IO device and main memory. The design of 10systems is considered in Chapter 7. Interconnection networks. Processor-level components communicate byword-oriented buses. In systems with many components, communication may becontrolled by a subsystem called an interconnection network; terms such as switch-ing network, communications controller, and bus controller are also used in thiscontext. The function of the interconnection network is to establish dynamic com-munication paths among the components via the buses under its control. For costreasons, these paths are usually shared. Only two communicating devices canaccess and use a shared bus at any time, so contention results when several systemcomponents request use of the bus. The interconnection network resolves suchcontention by selecting one of the requesting devices on some priority basis andconnecting it to the bus. The interconnection network may place the other request-ing devices in a queue. 117 CHAPTER 2 Design Methodology 118 SECTION 2.3 The Processor Level Type Medium to/from which IO device IO device Input Output transforms digital electrical signals Analog-digital converter X Analog (continuous) electrical signals CD-ROM drive X Characters ^nd coded images) on optical disk Document scanner/reader X Dot-matrix display panel Keyboard/keypad Images on paper X X Images on screen Characters on keyboard Laser printer X Images on paper Loudspeaker X Spoken words and sounds Magnetic-disk drive X X Characters (and coded images) on magnetic disk Magnetic-tape drive X X Characters (and coded images) on magnetic tape Microphone X Spoken words and sounds Mouse/touchpad X Spatial position on pad Figure 2.47 Some representative IO devices. Simultaneous requests for access to some unit or bus result from the fact thatcommunication between processor-level components is generally asynchronous inthat the components cannot be synchronized directly by a common clock signal.This synchronization problem can be attributed to several causes. • A high degree of independence exists among the components. For example,CPUs and IOPs execute different types of programs and interact relatively infre-quently and at unpredictable times. • Component operating speeds vary over a wide range. CPUs operate from 1 to 10times faster than main-memory devices, while main-memory speeds can bemany orders of magnitude faster than IO-device speeds. • The physical distance separating the components can be too large to permit syn-chronous transmission of information between them. Bus control is one of the functions of a processor such as a CPU or an IOP. AnIOP controls a common IO bus to which many IO devices are connected. The IOPis responsible for selecting a device to be connected to the IO bus and from there tomain memory. It also acts as a buffer between the relatively slow IO devices andthe relatively fast main memory. Larger systems have special processors whosesole function is to supervise data transfers over shared buses. 2.3.2 Processor-Level Design Processor-level design is less amenable to formal analysis than is design at the reg-ister level. This is due in part to the difficulty of giving a precise description of thedesired system behavior. To say that the computer should execute efficiently allprograms supplied to it is of little help to the designer. The common approach to design at this level is to take a prototype design of known performance and modifyit where necessary to accommodate new technologies or meet new performancerequirements. The performance specifications usually take the following form: • The computer should be capable of executing a instructions of type b per second. • The computer should be able to support c memory or 10 devices of type d. • The computer should be compatible with computers of type e. • The total cost of the system should not exceed/ Even when a computer is closely based on a known design, it may not be pos-sible to predict its performance accurately. This is due to our lack of understandingof the relation between the structure of a computer and its performance. Perfor-mance evaluation must generally be done experimentally during the design pro-cess, either by computer simulation or by measurement of the performance of acopy of the machine under working conditions. Reflecting its limited theoreticalbasis, only a small amount of useful performance evaluation can be done via math-ematical analysis [Kant 1992]. Prototype structures. We view the design process as involving two majorsteps: First select a prototype design and adapt it to satisfy the given performanceconstraints. Then determine the performance of the proposed system. If unsatisfac-tory, modify the design and repeat this step; continue until an acceptable design isobtained. This conservative approach to computer design has been widely followedand accounts in part for the relatively slow evolution of computer architecture. It israre to find a successful computer structure that deviates substantially from thenorm. The need to remain compatible with existing hardware and software stan-dards also influences the adherence to proven designs. Computer owners areunderstandably reluctant to spend money retraining users and programmers, orreplacing well-tested software. The systems of interest here are general-purpose computers, which differ fromone another primarily in the number of components used and their autonomy. Thevariety of interconnection or communication structures used is fairly small. Wewill represent these structures by means of block diagrams that are basically graphs(section 2.1.1). Figure 2.48 shows the structure that applies to first-generation com-puters and many small, modern microprocessor-based systems. The addition ofspecial-purpose 10 processors typical of the second and subsequent generations is 119 CHAPTER 2 Design Methodology Centralocessing unit CPU M Main memory Systembus IO devices ICN D, D2 D* Figure 2.48 Basic computer structure 120 SECTION 2.3 The Processor Level Centralprocessing unit Cachememory Systembus IO processors IO devices CPU CM IOP, Mainmemory ICN IOP, Figure 2.49 Computer with cache and IOprocessors. shown in Figure 2.49. Here ICN denotes an interconnection (switching) networkthat controls memory-processor communication. Figure 2.50 shows a prototypestructure employing two CPUs; it is therefore a multiprocessor. The uniprocessorsystems of Figures 2.48 and 2.49 are special cases of this structure. Even morecomplex structures such as computer networks can be obtained by linking severalcopies of the foregoing prototype structures. Performance measurement. Many performance figures for computers arederived from the characteristics of its CPU. As observed in section 1.3.2, CPU Centralprocessing units Cachememories Crossbarswitchingnetwork CPU, —| CM, CPU, —i CM, Main memory M, M2 ICN IOdevices IOprocessors IOP, D, IOP2 D2 IOP„ D3 D* Figure 2.50 Computer with multiple CPUs and main memory banks. speed can be measured easily, but roughly, by its clock frequency/in megahertz.Other, and usually better, performance indicators are MIPS, which is the averageinstruction execution speed in millions of instructions per second, and CPI, whichis the average number of CPU clock cycles required per instruction. As discussedin section 1.3.2, these performance measures are related to the average time 7* inmicroseconds (us) required to execute N instructions by the formula NxCPI Hence the average time tE to execute an instruction is tE=T/N= CPI/f us While / depends mainly on the IC technology used to implement the CPU, CPIdepends primarily on the system architecture. We can get another perspective on tE by considering the distribution of instruc-tions of different types and speeds in typical program workloads. Let /,, I2, ..., /„be a set of representative instruction types. Let f, denote the average execution time(us) of an instruction of type /, and let pi denote the occurrence probability of type-/, instructions in representative object code. Then the average instruction executiontime tE is given by 121 CHAPTER 2 Design Methodology ■ I PA us (2.20) The /, figures can be obtained fairly easily from the CPU specifications, but accu-rate Pj data must usually be obtained by experiment. The set of instruction types selected for (2.20) and their occurrence probabili-ties define an instruction mix. Numerous instruction mixes have been publishedthat represent various computers and their workloads [Siewiorek, Bell, and Newell1982]. Figure 2.51 gives some recent data collected for two representative Probability ol occurrence Program A Program B Instruction type (commercial) (scientific) Memory load 0.24 0.29 Memory store 0.12 0.15 Fixed-point operations 0.27 0.15 Floating-point operations 0.00 0.19 Branch 0.17 0.10 Other 0.20 0.12 Figure 2.51 Representative instruction-mix data.Source: McGrory, Carlton, and Askins 1992. SECTION 2.3 The Processor Level 122 programs running on computers employing the Hewlett-Packard PA-RISC architecture under the UNIX operating system [McGrory, Carlton, and Askins 1992].The execution probabilities are derived from counting the number of times aninstruction of each type is executed while running each program; instructions fromboth the application program and the supporting system code are included in thiscount. Program A is a program TPC-A designed to represent commercial on-linetransaction processing. Program B is a scientific program FEM that performsfinite-element modeling. In each case, memory-access instructions (load andstore) account for more than a third of all the instructions executed. The computa-tion-intensive scientific program makes heavy use of floating-point instructions,whereas the commercial program employs fixed-point instructions only. Condi-tional and unconditional branch instructions account for 1 in 6 instructions in pro-gram A and for 1 in 10 instructions in program B. Other published instructionmixes suggest that as many as 1 in 4 instructions can be of the branch type. A few performance parameters are based on other system components, espe-cially memory. Main memory and cache size in megabytes (MB) can provide arough indication of system capacity. A memory parameter related to computingspeed is bandwidth, defined as the maximum rate in millions of bits per second(Mb/s) at which information can be transferred to or from a memory unit. Memorybandwidth affects CPU performance because the latter's processing speed is ulti-mately limited by the rate at which it can fetch instructions and data from its cacheor main memory. Perhaps the most satisfactory measure of computer performance is the cost ofexecuting a set of representative programs on the target system. This cost can bethe total execution time T, including contributions from the CPU, caches, mainmemory, and other system components. A set of actual programs that are represen-tative of a particular computing environment can be used for performance evalua-tion. Such programs are called benchmarks and are run by the user on a copy(actual or simulated) of the computer being evaluated [Price 1989]. It is also usefulto devise artificial or synthetic benchmark programs, whose sole purpose is toobtain data for performance evaluation. The program TPC-A providing the data forprogram A in Figure 2.51 is an example of a synthetic benchmark. EXAMPLE 2.8 PERFORMANCE COMPARISON OF SEVERAL COMPUTERS [MCLELLAN 1993]. Figure 2.52 presents some published data on the performanceof three machines manufactured by Digital Equipment Corp. in the early 1990s,based on various versions of its 64-bit Alpha microprocessor. The SPEC (StandardPerformance Evaluation Cooperative) ratings are derived from a set of benchmarkprograms that computer companies use to compare their products. The SPECint92and SPECfp92 parameters indicate instruction execution speed relative to a standard-ized 1-MIPS computer (a 1978-vintage Digital VAX 11/780 minicomputer) whenexecuting benchmark programs involving integer (fixed point) and floating-pointoperations, respectively. Hence the SPEC figures approximate MIPS measurementsfor two major classes of application programs like those of Figure 2.51. The remain-ing data in Figure 2.52 are relative performance figures for executing some otherwell-known benchmark programs, most aimed at scientific computing. Data of this sort are better suited to measuring relative rather than absolute perfor-mance. For example, suppose we wish to compare the performance of the Digital 3000and 10000 machines listed in Figure 2.52. The ratio of their SPECint92 MIPS numbersis 104.3/63.8 = 1.65. The corresponding ratios for the other five benchmarks range DEC 3000 DEC 4000 DEC 10000 Performance measure Model 400 Model 610 Model 610 CPU clock frequency (MHz) 133 160 200 Cache size (MB) 0.5 1 4 SPECint92 63.8 81.2 104.3 SPECfp92 112.2 143.1 200.4 Linpack 1000 x 1000 90 114 155 Perfect BM suite 18.1 22.9 28.6 Cernlib 16.9 21.0 26.0 Livermore loops 18.7 22.9 28.1 Figure 2.52 Performance comparison of three computers based on the Digital Alpha processor. Source: McLellan 1993. from 1.50 to 1.79, suggesting that the Digital 10000 is about two-thirds faster than theDigital 3000. Note also that the ratio of their clock frequencies is 200/133 = 1.50. Queueing models. In order to give a flavor of analytic performance modeling,we outline an approach based on queueing theory. The origins of this branch ofapplied probability theory are usually traced to the analysis of congestion in tele-phone systems made by the Danish engineer A. K. Erlang (1878-1929) in 1909.Our treatment is quite informal; the interested reader is referred to [Allen 1980;Robertazzi 1994] for further details. The queueing model that we will consider is the single-queue, single-servercase depicted in Figure 2.53; this is known as the M/M/l model for historical rea-sons. It represents a "server" such as a CPU or a computer with a set of tasks (pro-grams) to be executed. The tasks are activated or arrive at random times and arequeued in memory until they can be processed or "serviced" by the CPU on a first-come first-served basis. The key parameters of the model are the rate at whichtasks requiring service arrive and the rate at which the tasks are serviced, both mea-sured in tasks/s. The mean or average arrival and service rates are conventionallydenoted by A (lambda) and p (mu), respectively. The actual arrival and servicerates vary randomly around these mean values and are represented by probability 123 CHAPTER 2 Design Methodology Sharedresource " Items 1r Queue Serviced Server items Quel leing sy stem Figure 2.53 Simple queueingmodel of;. computer. The Processor Level 124 distributions. The latter are chosen to approximate the actual behavior of the sys_____., _ , tem being modeled; how well they do so must be determined by observation and SECTION 1.5 J J measurement. The symbol p (rho) denotes A/p and represents the mean utilization of theserver, that is, the fraction of time it is busy, on average. For example, if an averageof two tasks arrive per second (X = 2) and the server can process them at an averagerate of eight tasks per second (p = 8), then p = 2/8 = 0.25. The arrival of tasks at the system is a random process characterized by theinterarrival time distribution px{t) defined as the probability that at least one taskarrives during a period of length t. The M/M/l case assumes a Poisson arrival pro-cess—named after the French mathematician Simeon-Denis Poisson (1781-1840)—for which the probability distribution is Pl(t) = l-e~h This exponential distribution has px(t) = 0 when t = 0. As t increases, px(t) increasessteadily toward 1 at a rate determined by X. Exponential distributions characterizethe randomness of many queueing models quite well. They are also mathematicallytractable and lead to simple formulas for various performance-related quantities ofinterest. It is therefore usual to model the behavior of the server (the service pro-cess) by an exponential distribution also. Let ps(t) be the probability that the ser-vice required by a task is completed by the CPU in time t or less after its removalfrom the queue. Then the service process is characterized by ps(t)=\-e^' Various performance parameters can characterize the steady-state performanceof the single-server queueing system under the foregoing assumptions. • The utilization p = A/p of the server, that is, the average fraction of time it isbusy. • The average number of tasks queued in the system, including tasks waiting forservice and those actually being served. The parameter is called the mean queuelength and is denoted by /Q. It can be shown [Robertazzi 1994] that /Q = p/(1-P) (2.21) • The average time that arriving tasks spend in the system, both waiting for serviceand being served, which is called the mean waiting time tQ. The quantities rQ and/q are related directly as follows. An average task X passing through the systemunder steady-state conditions should encounter the same number of waiting tasks/q when it enters the system as it leaves behind when it departs from the systemafter being serviced. The number left behind is Xtq, which is the number of tasksthat enter the system at rate X during the period tQ when X is present. Hence weconclude that /Q = Xtq, in other words, tQ = Iq/X (2.22) Equation (2.22) is called Little's equation. It is valid for all types of queueing sys-tems, not just the M/M/l model. Combining (2.21) and (2.22), we get tQ = l/(p - X) (2.23) The quantities /Q and tQ refer to tasks that are either waiting for access to theserver or are actually being served. The mean number of tasks waiting in thequeue excluding those being served is denoted by /w, while rw denotes the meantime spent waiting in the queue, excluding service time. (The subscript W standsfor "waiting.") The mean utilization of the server in an M/M/l system, that is, themean number of tasks being serviced, is X/\i; hence subtracting this from /Qyields /w: ^2 'w = 'o - P = \l(\L-X) (2.24) Similarly 125 CHAPTER 2 Design Methodology 'w = 'o - 1/M- = H(H-k) (2.25) where 1/p is the mean time it takes to service a task. Comparing (2.24) and (2.25)we see that fw = lw/X; therefore, Little's equation holds for both the Q and the Wsubscripts. To illustrate the use of the foregoing formulas, consider a server computer thatis processing jobs in a way that can be approximated by the M/M/l model. Arrivingjobs are queued in main memory until they are fully executed in one step by theCPU, which therefore is the server. New jobs arrive at an average rate of 10 perminute, and the computer is, on the average, idle 25 percent of the time. We ask twoquestions: What is the average time T that each job spends in the computer? What isthe average number of jobs N in main memory that are waiting to begin execution?To answer, we assume that steady-state conditions prevail, from which it followsthat T is tq, and N is /w. Since the system is busy 75 percent of the time, p = X/]i =0.75. We are given that X = 10 jobs/min; hence the service rate p. is 40/3 jobs/min.Substituting into (2.23) yields T= tQ= 1/(40/3 - 10) = 0.3 min. From Little's equa-tion, N=lQ = XtQ = 3; hence by (2.25), /w = 3 - 0.75 = 2.25 jobs. EXAMPLE 2.9 ANALYSIS OF SHARED COMPUTER USAGE [ALLEN 1980]. A small company has a computer system with a single terminal that is shared by its engineeringstaff. An average of 10 engineers use the terminal during an eight-hour work day, andeach user occupies the terminal for an average of 30 minutes, mostly for simple androutine calculations. The company manager feels that the computer is underutilized,since the system is idle an average of three hours a day. The users, however, complainthat it is overutilized, since they typically wait an hour or more to gain access to the ter-minal; they want the manager to purchase new terminals and add them to the system.We will now attempt to analyze this apparent contradiction using basic queueing the-ory. Assume that the computer and its users are adequately represented by an M/M/lqueueing system. Since there are 10 users per eight hours on average, we set X =10/8 users/hour = 0.0208 users/min. The system is busy an average of five out ofeight hours; hence the utilization p = 5/8, implying that u = 1/30 = 0.0333. Substitut-ing these values for X and u into (2.25) yields fw = 50 mm, which confirms theusers' estimate of their average waiting time for terminal access. The manager is now convinced that the company needs additional terminals andagrees to buy enough to reduce rw from 50 to 10 min. The question then arises: Howmany new terminals should he buy? We can approach this problem by representing 126 each terminal and its users by an independent M/M/l queueing system. Let m be the minimum number of terminals needed to make tw < 10 or, equivalents, tn < 40. TheSECTION 2 4 arriving users are assumed to divide evenly into m queues, one for each terminal. The arrival rate X* per terminal is taken to be X/m = 0.0208/m users/min. If, as indicatedabove, the computer's CPU is lightly utilized, then a few additional terminals shouldnot affect the response time experienced at a terminal*, hence we assume that each ter-minal's mean service rate is u* = p. = 0.0333 users/min. To meet the desired perfor-mance goal, we require t*Q = l/(u* - X*) = i/(n - X/m) < 40 from which it follows that m > 2.5. Hence three terminals are needed, so two new ter-minals should be acquired. This result is pessimistic, since the users are unlikely toform three separate queues for three terminals or to maintain the independence of thequeues by not jumping from one queue to another whose terminal has become avail-able. Nevertheless, this simple analysis gives the useful result that m should be 2 or 3. 2.4SUMMARY The central problem facing the digital system designer is to a devise a structure (acircuit, network, or system) from given components that exhibits a specifiedbehavior or performs a specified range of operations at minimum cost. Variousmethods exist for describing structure and behavior, including block diagrams (forstructure), truth and state tables (for behavior), and HDLs (for behavior and struc-ture). Computer systems can be viewed at several levels of abstraction, where eachlevel is determined by its primitive components and information units. Three levelshave been presented here: the gate, register, and processor levels, whose compo-nents process bits, words, and blocks of words, respectively. Design at all levels isa complex process and depends heavily on CAD tools. The gate level employs logic gates as components and has a well-developedtheory based on Boolean algebra. A combinational circuit implements logic orBoolean functions of the form z{xx, x2, ..., xn), where z and the x,'s assume the val-ues 0 and 1. The circuit can be constructed from any functionally complete set ofgate types such as {AND, OR, NOT} or {NAND}. Every logic function can berealized by a two-level circuit that can be obtained using exact or heuristic minimi-zation techniques. Sequential circuits implement logic functions that depend ontime; unlike combinational circuits, sequential circuits have memory. They arebuilt from gates and 1-bit storage elements (flip-flops) that store the circuit's stateand are synchronized by means of clock signals. Register-level components include combinational devices such as word gates,multiplexers, decoders, and adders, as well as sequential devices such as (parallel)registers, shift registers, and counters. Various general-purpose programmable ele-ments also exist, including PLAs, ROMs, and FPGAs. Little formal theory existsfor the design and analysis of register-level circuits. They are often described byHDLs whose fundamental construct is the register-transfer statement cond: Z := F,(X1,X2,...,Xit); denoting the conditional transfer of data from registers Xl,X2,...,Xk to register Z viaa combinational processing circuit F{. Register-level circuits often consist of adatapath unit and a control unit. The first step in register-level design is to con-struct a formal (HDL) description of the desired behavior from which the compo-nents and connections for the datapath unit can be determined. The logic signalsneeded to control the datapath are then identified. Finally, a control unit is designedthat generates these control signals. The components recognized at the processor level are CPUs and other proces-sors, memories, 10 devices, and interconnection networks. The behavior of proces-sor-level systems is complex and is often specified in approximate terms usingaverage or worst-case behavior. Processor-level design is heavily based on the useof prototype structures. A prototype design is selected and modified to meet thegiven performance specifications. The actual performance of the system is thenevaluated, and the design is further modified until a satisfactory result is achieved.Typical performance measures are millions of instructions executed per second(MIPS) and clock cycles per instruction (CPI). A few analytical methods for perfor-mance evaluation exist—notably queueing theory—but their usefulness is limited.Instead, experimental approaches using computer-based simulation or performancemeasurements on an actual system are used extensively. 127 CHAPTER 2 Design Methodology 2.5PROBLEMS 2.1. Explain the difference between structure and behavior in the

digital system context. Il-lustrate your answer by giving (a) a purely structural description and (b) a purely be-havioral description of a half-subtracter circuit that computes the 1-bit difference d =x - v and also generates a borrow signal b whenever x < y. 2.2. (a) Following the example of Figure 2.4, construct a behavioral VHDL description ofthe full-adder circuit of Figure 2.9b. (b) Following Figure 2.5, construct a structuralVHDL description of the full adder. 2.3. Construct both structural and behavioral descriptions in VHDL of the EXCLUSIVE-OR circuit appearing in Figure 2.2. 2.4. Figure 2.54 describes a half adder in the widely used Verilog HDL. The Verilog sym-bols for the logic operations AND, OR, EXCLUSIVE-OR, and NOT are &, I. \ and ~.respectively, (a) Is this description behavioral or structural? (b) Construct a similar de-scription in Verilog for a full adder. module half judder (xQ, v0, s0, co)'Input x0. yy; output s0, c0; assign s0 = x0 A y0; assign c0 = x0 & y0;endmodule Figure 2.54 Verilog description of a half adder. 128 Inputs Outnuts SECTION 2.5 *i y{ Kl 4 bi Problems 0 0 0 0 0 0 1 11 0 1 0 11 0 1 1 01 1 0 0 10 1 0 1 00 11 11 0 00 1 11 Figure 2.55 Truth table of a full subtracter. 2.5. Assign each of the following components to one of the three major design levels—pro-cessor, register, or gate—and justify your answers, (a) A multiplier of two n-bit num-bers jV, and N2. (b) An identity circuit that outputs a 1 if all its n inputs (which representa number AO are the same; it outputs a 0 otherwise, (c) A negation circuit that convertsN to -N. (d) A first-in first-out (FIFO) memory, that stores a sequence of numbers inthe order received; it also outputs the numbers in the same order. 2.6. Certain very small-scale ICs contain a single two-input gate. The ICs are manufacturedin three varieties—NAND, OR, and EXCLUSIVE-OR—as indicated by a printed labelon the ICs package. By mistake, a batch of all three varieties is manufactured withouttheir labels, (a) Devise an efficient test that a technician can apply to any IC from thisbatch to determine which gate type it contains, (b) Suppose the batch of unlabeled ICscontains NOR gates, as well as NAND, OR, and EXCLUSIVE-OR. Devise an efficienttesting procedure to determine each ICs gate type. 2.7. Construct a logic circuit implementing the 1-bit (full) subtracter defined in Figure 2.55using as few gates as you can. 2.8. {a) Obtain an efficient all-NAND realization for the following four-variable Booleanfunction: fx(a,b,c,d) = a(b + c)d + a(b + d)(b + c)(c + d)+ b c d (b) Construct an efficient all-NOR design ioxfx{a,b,c,d). 2.9. Design a two-level combinational circuit in the sum-of-products style that computesthe 3-bit sum of two 2-bit binary numbers. The circuit is to be implemented using ANDand OR gates. 2.10. Consider the D flip-flop of Figure 2.11. (a) Explain why the glitch does not affectthe flip-flop's state y. (b) This flip-flop is said to be positive edge-triggered becauseit triggers on the positive (rising or 0 to 1) edge of the clock CK. A negative edge-triggered flip-flop triggers on the negative (falling or 1 to 0) edge of CK, which isindicated by placing an inversion bubble at the CK input like that at the y output.Redraw the y part of Figure 2.11 for a negative edge-triggered flip-flop. 2.11. Figure 2.56 defines a 1-bit storage device called a JK flip-flop. It has the same edge-triggered clocking as the D flip-flop of Figure 2.11 but has two data inputs insteadof one. The J input is activated to store a 1 in the flip-flop; that is, JK = 10 sets y = SetClock -Reset — J yCKK Inputs JK00 01 10 11 State 0>'(') 1 0 0 11 Next state ioio y(' + D 129 CHAPTER 2 Design Methodology (a) Figure 2.56 JK flip-flop: (a) graphic symbol; (b) state table. (b) 1. Similarly, the K input is activated to store a 0 in the flip-flop; that is, JK = 01 re-sets >• to 0. The input combination JK = 00 leaves the state unchanged, while JK =11 always changes, or toggles, the state, (a) What is the characteristic equation for aJK flip-flop, analogous to (2.5)? (b) Show how to build a JK flip-flop from a D flip-flop and a few NAND gates. 2.12. Derive a state table for a synchronous sequential circuit that acts as a serial incre-menter. An unsigned number N of arbitrary length is entered serially on input line x,causing the circuit to output serially the number N + \ on its output line z. Give theintuitive meaning of each state and identify the reset state. 2.13. An alternative to a state table for representing the behavior of a sequential circuitSC is a state diagram or state transition graph, whose nodes denote states{S^Sj,...^} and whose edges, which are indicated by arrows, denote transitionsbetween states. A transition arrow from 5, to & is labeled XJZV if, when SC is instate 5, and input Xu is applied, the (present) output Zv is produced and SC's nextstate is Sj. (a) Construct a state table equivalent to the state diagram for SC appear-ing in Figure 2.57. (b) How many flip-flops are needed to implement 5C? 2.14. Design the sequential circuit SC whose behavior is defined in Figure 2.57 using Dflip-flops and NAND gates. SC has a single primary input line and a single primaryoutput line. Your answer should include a complete logic diagram for SC. Use asfew gates and flip-flops as you can in your design. 2.15. 2.16, Implement the sequential circuit SC specified in the preceding problem, this timeusing JK flip-flops (see problem 2.11) and NOR gates. Derive a logic diagram forSC and use as few gates and flip-flops as you can. Design a serial subtracter analogous to the serial adder. The subtracter's inputs aretwo unsigned binary numbers nx and n2; the output is the difference n, - n2. Construct Reset Figure 2.57 State diagram for a sequential circuit SC. 130 a state table, an excitation table, and a logic circuit that uses JK flip-flops and NOR gates only. SECTION 2.5 problems 2.YI. Design a sequential circuit that multiplies an unsigned binary number N of arbitrary length by 3. N is entered serially via input line x with its least significant bit first.The result representing 3/V emerges serially from the circuit's output line z. Con-struct a state table for your circuit and give a complete logic circuit that uses D flip-flops and NAND gates only. 2.18. An important property of gates is functional completeness, which ensures that acomplete gate set is adequate for all types of digital computation, (a) It has been asserted that functional completeness is irrelevant at the register level when dealingwith components such as multiplexers, decoders, and PLDs. Explain concisely whythis is so. (b) Suggest a logical property of sets of such components that might besubstituted for completeness as an indication of the components' general usefulnessin digital design. Give a brief argument supporting your position. 2.19. Redraw the gate-level multiplexer circuit of Figure 2.20 at the register level usingword gates. Use as few such gates as you can and mark all bus sizes. Observe that asignal such as e that fans out to m lines can be considered to create an m-bit bus car-rying the w-bit word£ = (e,e,...,e). 2.20. Figure 2.55 gives the truth table for a full subtracter, which computes the differenceXj - >', - bi_i, where bt_x denotes the borrow-in bit. The subtracter's outputs are bt,d{, where b{ denotes the borrow-out bit. Show how to use (a) an eight-input multi-plexer and (b) a four-input multiplexer to realize the full subtracter. 2.21. Show how to design a 1/16 decoder using the 1/4 decoder of Figure 2.236 as yoursole building block. 2.22. Describe how to implement the priority encoder of Figure 2.25 by (a) a two-levelAND-OR circuit and (b) a multiplexer of suitable size. Demonstrate that one designis much less costly than the other and derive a logic diagram for the less expensivedesign. 2.23. Design a 16-bit priority encoder using two copies of an 8-bit priority encoder. Youmay use a few additional gates of any standard types in your design, if needed. 2.24. A magnitude-comparator circuit compares two unsigned numbers X and Y and pro-duces three outputs z,, z2, and z3, which indicate X= Y,X>Y, and X < Y, respectively.(a) Show how to implement a magnitude comparator for 2-bit numbers using a single16-input, 3-bit multiplexer of appropriate size, (b) Show how to implement the samecomparator using an eight-input, 2-bit multiplexer and a few (not more than five)two-input NOR gates. 2.25. Commercial magnitude comparators such as the 74X85 have three control inputsconfusingly labeled X = Y, X > Y. and X < Y, like the comparator's output lines.These inputs permit an array of k copies of a 4-bit magnitude comparator to be ex-panded to form a Ak-hil magnitude comparator as shown in Figure 2.58. Modify the4-bit magnitude comparator of Figure 2.27 to add the three new control inputs andexplain briefly how they work. [Hint: The unused carry input lines denoted cin inFigure 2.27 play a central role in the modification.] 2.26. Show how to connect n half adders (Figure 2.5) to form an «-bit combinational in-crementer whose function is to add one (modulo 2") to an «-bit number X. For example, if X = 10100111. the incrementer should output Z = 10101000; if X =11111111, it should output Z = 00000000. 2.27. Show how the register circuit of Figure 2.29 can be simplified by using theLOAD line to enable and disable the register's clock signal CLOCK. Explain clear2.28. 2.29. ly why this gated-clocking technique is often considered a violation of good de-sign practice. A useful operation related to shifting is called rotation. Left rotation of an ra-bitregister is defined by the register-transfer statement 131 (Zm-2>Zn (2.26) •'^0'Zm-l) :_ (Zm-l»Zin_2v.>Zi.Zo) (a) Give an assignment statement similar to (2.26) that defines right rotation. Showhow the 4-bit right-shift register SR of Figure 2.30 can easily be made to implementright rotation, (b) Using as few additional components and control lines as possible,show how to extend SR to implement both right shifting and right rotation. Design an 8-bit counter using only the following component types: 4-bit D-type reg-isters, half adders, full adders, and two-input NAND gates. The counter's inputs area CLEAR signal that resets it to the all-0 state and a COUNT signal whose 0-to-l(positive) edge causes the current count to be incremented by one. Use as few com-ponents as you can, assuming for simplicity that each component type has the samecost. 2.30. Assuming that input variables are available in true form only, show how to makethe Actel FPGA cell of Figure 2.35a realize two-input versions of the NAND, NOR,and EXLCLUSIVE-OR functions. 2.31. (a) Assuming that input variables are available in true form only, what is the fan-inof the largest NAND gate that can be implemented with a single Actel FPGA cell(Figure 2.35a)? (b) What is the largest NAND if both true and complemented inputsare available and we allow some or all of the inputs to the NAND to be inverted? 2.32. Show how to implement the full subtracter defined in Figure 2.55 using as few cop-ies as you can of the Actel C-module. Again assume that the input variables are sup-plied in true form only. 2.33. Figure 2.59 shows the Actel FPGA S-module, which adds a D flip-flop to the out-put of the C-module discussed in the text. Show how to use one copy of this cell toimplement the edge-triggered JK flip-flop defined in problem 2.11, assuming onlythe true output y is needed and that either one of the flip-flop's J or K inputs can becomplemented. CHAPTER 2 Design Methodology Figure 2.58 Expansion of a 4-bit magnitude comparator to form a 16-bit comparator. 4 *3 X4 Y, 4 *7 Yi 4 Y*- X\2 4 *15 Yn r,5 > \ Y 4-bitmagnitudecomparator Y 4-bitmagnitudecomparator X Y 4-bitmagnitudecomparator X X>Y X>Y X X X>Y X>YX=Y X=YXYX=Y X=YXYX=Y X=YX 0 andCMPA sets the zero flag Z to 0, indicating a nonzero result. (It also sets various otherflags not used by this program). When A0 finally reaches 1001, A0 - 1001 = 0, soCMPA sets Z to 1. Now the last instruction BNE, which stands for branch if not equalto zero, is a conditional branch instruction whose operation is described by ifZ*l then PC: = START It therefore transfers execution back to the ABCD instruction in location START aslong as A0 > 1001. When A0 finally reaches 1001, Z becomes 1, and PC is incrementednormally to exit from the program. It is interesting to compare this 680X0 program with the similar programs givenearlier for the IAS (Figure 1.15) and PowerPC (Figure 1.27) computers. Coprocessors. The built-in instruction repertoire of the 68020 includes fixed-point multiplication and division and stack-based instructions for transferring con-trol between programs. Hardware-implemented floating-point instructions are notavailable directly; however, they are provided indirectly by means of an auxiliaryIC, the 68881 floating-point coprocessor. (The ARM6 also has provisions forexternal coprocessors.) In general, a coprocessor P is a specialized instruction exe-cution unit that can be coupled to a microprocessor so that instructions to be exe-cuted by P can be included in programs fetched by the microprocessor. Thus thecoprocessor serves as an extension to the microprocessor and forms part of theCPU as indicated in Figure 3.14. The 68881 (and the similar but faster 68882) contains a set of eight 80-bitregisters for storing floating-point numbers of various formats, including 32- and64-bit numbers conforming to the standard IEEE 754 format (presented later).Additional control registers in the 68881 allow it to communicate with the68020. A set of coprocessor instructions are defined for the 68020; they containcommand fields specifying floating-point operations that the 68881 can execute.When the 68020 fetches and decodes such an instruction, it transfers the com-mand portion to the coprocessor, which then executes it. Further exchanges takeplace between the main processor and the coprocessor until the coprocessor com-pletes execution of its current operation, at which point the 68020 proceeds toits next instruction. The commands executed by the 68881 include the basic 159 CHAPTER 3Processor Basics CPU 68020micro-processor Systembus floating-pointcoprocessor TTT 32-bit address bus Main memory Read-onlymemory(ROM) Input-output interface circuit (IO port) 32-bit data bus Control lines Input-output interface circuit (IOport) Read-writememory(RAM) "J-! IO device Figure 3.14 68020-based microcomputer with floating-point coprocessor. IO device 160 arithmetic operations (add. subtract, multiply, and divide), square root, logarithms, and trigonometric functions. Other types of coprocessors may be Data Representation attached to the 68020 in similar fashion. Later members of the 680X0 familytake advantage of advances in VLSI to integrate a floating-point (co)processorinto the CPU chip. Other design features. Like the IBM System/360-370 and the ARM6, the CPUhas a supervisor state intended for operating system use and a user state for appli-cation programs. As Figures 3.11 and 3.12 indicate, certain "privileged" controlregisters and instructions can be used only in the supervisor state. User and super-visory programs are thus clearly separated—for example, they employ differentstack pointers—thereby improving system security. 680X0-based computers arealso designed to allow easy implementation of virtual memory, whereby the oper-ating system makes the main memory appear larger to user programs than it reallyis. Hardware support for virtual memory is provided by the 68851 memory man-agement unit (MMU), another 680X0 coprocessor. Provided they meet certain independence conditions, up to three 68020 instruc-tions can be processed simultaneously in pipeline fashion. This pipelining is com-plicated by the fact that instruction lengths and execution times vary, a problem thatRISCs try to eliminate. Another speedup feature found in the 68020 is a smallinstruction-only cache (I-cache). The 68020 prefetches instructions from mainmemory while the system bus is idle; the instructions can subsequently be readmuch more quickly from the on-chip cache than from the off-chip main memory.An unusual feature of the 68020 noted in Figure 3.11 is its use of two levels ofmicroprogramming to implement the CPU's control logic. For the manufacturer,this feature increases design flexibility while reducing IC area compared with con-ventional (one-level) microprogrammed control. 3.2 DATA REPRESENTATION The basic items of information handled by a computer are instructions and data.We now examine the methods used to represent such information, focusing on theformats for numerical data. 3.2.1 Basic Formats Figure 3.15 shows the fundamental division of information into instructions (oper-ation or control words) and data (operands). Data can be further subdivided intonumerical and nonnumerical. In view of the importance of numerical computation,computer designs have paid a great deal of attention to the representation of numbers. Two main number formats have evolved: fixed-point and floating-point. Thebinary fixed-point format takes the form bAb^)c.. .bK, where each bx is 0 or 1 and abinary point is present in some fixed but implicit position. A floating-point num-ber, on the other hand, consists of a pair of fixed-point numbers M,E, whichdenote the number M x BE, where B is a predetermined base. The many formatsused to encode fixed-point and floating-point numbers will be examined later in Binary 161 Instructions ^ Fixed-point CHAPTER 3 Information Bm,2>Bm,l>Bm,0 (3-8) Suppose we store these 4(m + 1) bytes in M using the "natural" order defined by(3.8); that is, we assign a sequence of increasing memory addresses adr0, adrx, adr2, adr3, ..., adr4m+2, adr4m+3 to the bytes as listed in (3.8). This storage sequence, which is illustrated in Figure3.18a, is a byte-storage convention called big-endian.2 It is so named because themost significant (biggest) byte Bj3 of word Wt is assigned the lowest address and theleast significant byte BiQ is assigned the highest address. In other words, the big-endian scheme assigns the highest address to byte 0. The alternative byte-storagescheme called little-endian assigns the lowest address to byte 0. This corresponds to ^0,^,...,!^ = #o.O'fiO,l'fiO,2'fiO,3'5l,0'5l,l'fl1.2'#1.3 BmfrBm,\iBm2'Bm3 and is illustrated by Figure 3.18&. Interestingly, computer manufacturers have never agreed on this issue, so boththe big-endian and little-endian conventions are in widespread use. For example,the Motorola 680X0 uses the big-endian method, whereas the Intel 80X86 series islittle-endian. Some computers including the ARM family can switch between thetwo endian conventions. Tags. In the von Neumann computer, instruction and data words are storedtogether in main memory and are indistinguishable from one another—this is theclassic "stored program" concept. An item plucked at random from memory cannotbe identified as an instruction or data. Different data types such as fixed-point andfloating-point numbers also cannot be distinguished by inspection. A word's typeis determined by the way a processor interprets it. In principle, the same word canbe treated as an instruction and data at different times, for example, the word X in 2The allusion is to an argument appearing in Gulliver's Travels on whether an egg should be opened at it> bigor little end [Cohen 1981]. 164 SECTION 3.2Data Representation ...ooc Byte 3.3 ...00B Byte 2,0 ...OOA Byte 2,1 ...009 Byte 2,2 ...008 Byte 2,3 ...007 Byte 1,0 ...006 Byte 1,1 ...005 Byte 1,2 ...004 Byte 1.3 ...003 Byte 0,0 ...002 Byte 0,1 ...001 Byte 0.2 ...000 Byte 0.3 Higheraddresses Byteaddress 02 01 00 Loweraddresses Wordaddress .OOC Byte 3,0 .OOB £yte 2,3 OOA Byte 2,2 .009 Byte 2.1 .008 Byte 2,0 .007 Byte 1.3 .006 Byte 1,2 .005 Byte 1,1 .004 Byte 1,0 .003 Byte 0,3 .002 Byte 0,2 .001 Byte 0,1 .000 Byte 0,0 Byte address Figure 3.18 Basic byte storage methods: (a) big-endian and (b) little-endian. 02 01 00 Wordaddress the instruction sequence X:=X + Y; go to X; It is the programmer's (and compiler's) responsibility to ensure that data are notinterpreted as instructions, and vice versa. A reason for this deliberate indistinguishability of data and instructions can be seen in the design of the IAS computer(section 1.2.2). The LAS's address-modify instructions alter stored instructions inmain memory. The ability to modify instructions in this way—in effect, treatingthem as data—is useful when processing indexed variables, as illustrated in Exam-ple 1.4. However, this type of instruction modification in memory became obsoletewith the introduction of address-indexing hardware. A few computer designers have argued that the major information types shouldbe assigned formats that identify them [Feustel 1973; Myers 1982], This can bedone by associating with each information word a group of bits, called a tag, thatidentifies the word's type. The tag may be considered as a physical implementationof the type declaration found in some high-level programming languages. One ofthe earliest machines to use tags was the 1960s-vintage Burroughs B6500/7500series, which employed a 3-bit tag field in every word so that eight word typescould be distinguished. The 52-bit word format of the B6500/7500 and the inter-pretation of its tag appear in Figure 3.19. Tagging simplifies instruction specification. In conventional, nontagged com-puters, an instruction's opcode must explicitly or implicitly specify the type of dataon which it operates. The PCU must know the operand types in order to route them 47 Parity- Tagcheck bit VInformation bits Tag Interpretation 000 Single-precision number. 001 Indirect reference word. 010 Double-precision number on Segment descriptor. 100 Step-index control word. 101 Data descriptor. 110 Uninitialized operand. 111 Instruction. Figure 3.19 Tagged-word format of the Burroughs B6500/750O series. to the proper arithmetic circuits and registers. It is therefore necessary to providedistinct instructions for each data type; for example, add binary word, add binaryhalfword, add BCD word, add floating-point word, and add floating-point doubleword. If, on the other hand, tags distinguish the operand types, then a single ADDopcode suffices for all cases. The processor merely has to inspect an operand's tagto determine its type. Furthermore, tag inspection permits the hardware to checkfor software errors, such as an attempt to add operands whose types are incompati-ble. Tags have a serious cost disadvantage, however. They increase memory sizeand add to the system hardware costs without increasing computing performance.This fact has severely restricted the use of tagged architectures. Error detection and correction. Various factors like manufacturing defects andenvironmental effects cause errors in computation. Such errors frequently appearwhen information is being transmitted between two relatively distant points withina computer or is being stored in a memory unit. "Noise" in the communication linkcan corrupt a bit x that is being sent from A to B so that B receives x instead of x.To guard against errors of this type, the information can be encoded so that speciallogic circuits can detect, and possibly even correct, the errors. A general way to detect or correct errors is to append special check bits toevery word. One popular technique employs a single check bit c0 called a parity-bit. The parity bit is appended to an n-bit word X = (x0, xu . .., *„_,) to form the(n + l)-bit word X* = (x0, *,, . . . , *„_,,%); see Figure 3.19. Bit c0 is assigned thevalue 0 or 1 that makes the number of ones in X* even, in the case of even-paritycodes, or odd, in the case of odd-parity codes. In the even-parity case, c0 isdefined by the logic equation Cn = Xn © X, © ... © X n—1 (3.9) where © denotes EXCLUSIVE-OR, while in the odd-parity case Cr\ — Xft Suppose that the information X is to be transmitted from A to B. The value of c0 isgenerated at the source point A using, say, (3.9), and X* is sent to B. Let B receivethe word X' = (x'Q, x\, . . . , xn_vc'Q). B then determines the parity of the receivedword by recomputing the parity bit according to (3.9) thus: 165 CHAPTER 3Processor Basics C*n = x'n © X , ©•*'„-! 166 SECTION 3.2Data Representation Error source (memory unit. Inpudata comunication link, —*- Output—»- data(corrected ifnecessary) etc.) Errorcorrector r ,r 1 ...I • • a i\ Check-bitgenerator Errordetector ' r Check-bitgenerator Figi ire 3.20 Erro r detection and correction logic. The received parity bit c'0 and the reconstituted parity bit c*0 are then compared. Ifc'0 * c*0, the received information contains an error. In particular, if exactly 1 bit ofX* has been inverted during the transmission process (a single-bit error), then c'0 *c*0. If c'0 = c*0, it can be concluded that no single-bit error occurred, but the possi-bility of multiple-bit errors is not ruled out. For example, if a 0 changes to 1 and a 1changes to 0 (a double error), then the parity of X is the same as that of X* and theerror will go undetected. The parity bit c0 therefore provides single-errordetection. It does not detect all multiple errors, much less provide any informationabout the location of the erroneous bits. The parity-checking concept can be extended to the detection of multipleerrors or to the location of single or multiple errors. These goals are achieved byproviding additional parity bits, each of which checks the parity of some subset ofthe bits in the word X*. By appropriately overlapping these subsets, the correctnessof every bit can be determined. Suppose, for instance, that we can deduce from theparity checks the identity of the bit x, responsible for a single-bit error. It is then asimple matter to introduce logic circuits to replace xi by Jc,, thus providing single-error correction. Let c be the number of check bits required to achieve single-errorcorrection with n-b\t data words. Clearly the check bits have 2C patterns that mustdistinguish between n + c possible error locations and the single error-free case.Hence c must satisfy the inequality 2C > n + c + 1 (3.10) For n = 16, (3.10) implies that c > 5, while for n = 32 we have c > 6. A variety ofpractical single-error-correcting parity-check codes meet the lower bound on cimplied by (3.10) [Siewiorek and Swarz 1992]. Some of these codes can also detectdouble errors and so are called single-error-correcting double-error-detecting(SECDED) codes. As the main memories of computers have increased in storagecapacity and decreased in physical size, they have become more prone to transientfailures that are often correctable via SECDED codes. Figure 3.20 shows the struc-ture of a typical error detection and correction scheme used with a computer's mainmemory. 3.2.2 Fixed-Point Numbers 167 In selecting a number representation to be used in a computer, the following factorsshould be taken into account: • The number types to be represented; for example, integers or real numbers. • The range of values (number magnitudes) likely to be encountered. • The precision of the numbers, which refers to the maximum accuracy of the repre-sentation. • The cost of the hardware required to store and process the numbers. The two principal number formats are fixed-point and floating-point. Fixed-pointformats allow a limited range of values and have relatively simple hardwarerequirements. Floating-point numbers, on the other hand, allow a much larger rangeof values but require either costly processing hardware or lengthy software imple-mentations. Binary numbers. The fixed-point format is derived directly from the ordinary(decimal) representation of a number as a sequence of digits separated by a decimalpoint. The digits to the left of the decimal point represent an integer; the digits tothe right represent a fraction. This is positional notation in which each digit has afixed weight according to its position relative to the decimal point. If i > 1, the /thdigit to the left (right) of the decimal point has weight 10,_I (10"')- Thus the five-digit decimal number 192.73 is equivalent to 1 x 102 + 9 x 101 + 2 x 10° + 7 x 10"1 + 3 x 1(T2 More generally, we can assign weights of the form r\ where r is the base or radixof the number system, to each digit. The most fundamental number representation used in computers employs abase-two positional notation. A binary word of the form bN...b-ib2bxbQ. b_xb_2b_ib^...bM (3.11) represents the number 2V When unclear from the context, the base r being used will be indicated by append-ing r as a subscript to the number. Thus 10102 denotes the binary equivalent of thedecimal number 1010, whereas 102 denotes 210. The format of (3.11) is an exampleof a fixed-point binary number and is used to denote unsigned numbers. Severaldistinct methods used for representing signed (positive and negative) numbers arediscussed below. Suppose that an n-bit word is to contain a signed binary number. One bit isreserved to represent the sign of the number, while the remaining bits indicate itsmagnitude. To permit uniform processing of all n bits, the sign is placed in the left-most position, and 0 and 1 are used to denote plus and minus, respectively. This CHAPTER 3Processor Basics 168 leads to the format SECTION 3.2 xn-\xn-2xn-2 ■ ■ ■ *2*1*0 (3-12) Data Representation | *■ Y ' Sign Magnitude < The precision allowed by this format is n - 1 bits, which is equivalent to (n - 1)log 210 decimal digits. The binary point is not explicitly represented; instead, it isimplicitly assigned to some fixed location in the word. The binary point's positionis not very important from the point of view of design. In many situations the num-bers being processed are integers, so the binary point is assumed to lie immediatelyto the right of the least significant bit jc0. Monetary quantities are often expressed asintegers; for instance, S54.30 might be expressed as 5430 cents. Using an /i-bitinteger format, we can represent all integers N with magnitude \N\ in the range 0 mlZero extension alone is sometimes used for this purpose, but it does not allow them-bit address to refer to all 2" possible addresses. The usual solution found inCISCs as well as in RISCs is to treat a short memory address as a modifier, or off-set, which is added (in zero-extended form) to a full-length memory address storedin a designated CPU register, called a base register. The RISC 1 uses its Rs registerfor this purpose, with S2 serving as the offset. The following store-byte instruction STB Rs,Rd(S2) (3.24) is designed to copy the byte from the right end of register Rs to the memory loca-tion whose address is Rd +S2zero.extended. In practice, sign extension is often implicitand Rd +S2zer0_extended is written simply as Rd +S2. Hence (3.24) is equivalent to M(Rd + S2):=Rs[24:31] The final memory address Rd + S2 is an example of an effective address. As wewill see shortly in our discussion of addressing modes, many other techniques areemployed for constructing effective addresses. EXAMPLE 3.5 INSTRUCTION FORMATS OF THE MIPS RX000 SERIES [Kane and heinrich 1992]. MIPS Computer Systems (now a division of SiliconGraphics) introduced the MIPS RX0O0 series of microprocessors in 1986. The firstmembers of the series, the MIPS R2000 and R3000, are 32-bit machines that have mostof the classic RISC features: a streamlined instruction set, a load/store architecture, andan instruction pipeline to support a performance target of one instruction completedevery clock cycle. Later RX000 machines, such as the R10000 announced in 1994. addvarious extensions to the "MIPS I" architecture implemented in the R2000 and R3000;we will confine our discussion to the MIPS I case. The RX000 is noteworthy for its simple and regular instruction formats, which wenow examine in detail. As seen from Figure 3.29, all the RX000 instructions are oneword (32 bits) in length and contain a 6-bit opcode in a fixed position. The remaining 31 J-typeformat 25 Opcodei i i i 1 31 I-typeformat Opcode 0 i : i 25 Rs Rt i i 20 Immediate operand IMM Branch address ADR i i i i i i i i i i i i i i i i 15 0 i i i i i i i i i vRegister addresses (2) 183 CHAPTER 3 Processor Basics R-typeformat 25 20 10 Opcode J I I I L R^ Rt Rd Shift amount_l I I i_ FunctionJ I I I L Register addresses (3) Figure 3.29 Instruction formats of the MIPS RX000. 26 bits are used in various ways, depending on the instruction type. Any operandsincluded in the instruction must be less than a full word in length, so some way isneeded to extend them to a full-size memory address or a twos-complement number. In the case of a J-type (jump or branch) instruction, the 26 operand bits form amemory address ADR, which is the target or branch address. For example, a simpleunconditional branch instruction has the J-type format J ADR (3.25) meaning go to ADR. Since RX000 memory addresses are 32 bits long, the PCU mustextend the 26-bit address field ADR in (3.25) to 32 bits. This is done automatically bythe following two-step process: Temp:=PC[31:28].ADR.OO;PC := Temp; First the four high-order bits from the program counter PC are placed in front ofADR and 00 is appended to it. Then the resulting 32-bit word is made the new con-tents of PC. The above address-extension method confines the possible branch addresses to a226-word region of memory space near the location of the current branch instruction.However, this is not as restrictive as it might appear. First of all, recall that a 32-bitmemory address refers to just one byte. Only 230 instructions can be placed in a 2 -bytememory, so only 30 bits are really needed to locate an instruction. The RX000 and sim-ilar machines always assign instructions to memory word locations with addresses thatend in 00: that is, all instructions are aligned with the natural word boundaries in M.Moreover, while the 26-bit address field ADR is still 4 bits short of 30. the size of theaccessible region for branching (226= 6.71 X 107 different addresses) is more than ade-quate for most programming purposes—and can be increased by software means, ifnecessary. The other two formats shown in Figure 3.29 specify register addresses usingeither two or three 5-bit fields. The RX000 has 25 = 32 general-purpose registers in itsregister file, so register addresses can be fully specified vuth no difficult). The second Instruction Sets 184 (I type) format is used by ALU-immediate instructions such as SECTION 3.3 ADDI Rs,Rt,IMM which adds the contents of the instruction's immediate address field, that is, bits 15:0of the instruction, to the contents of register Rs and places the result in register Rt. Toconvert the immediate operand FMM from 16 to 32 bits, it is sign-extended to 32 bitsby duplicating its left-most bit to obtain bits 31:16. The third (R type) format of the RX000 is used by data-processing instructionsthat have a natural three-address format to define operations of the form X}:=op(X2,X3). For instance, the add-register instruction ADD Rd,Rs,Rt performs the 32-bit addition Rd := Rs + Rt using the contents of the named registers. Since the register addresses occupy only 15bits of the instruction format, the remaining 11 bits are used in various ways to increase(and complicate) the range of operations that can be performed. In effect, they serve asextensions to the opcode. For example, there are six shift-register instructions, all ofwhich use instruction bits 10:6 to specify the amount by which the target register's con-tents are to be shifted. The shift-left logical instruction SLL Rd,Rt,Shamt shifts the contents of register Rt left by Shamt (shift amount) bits; it inserts 0s in thevacated positions on the right and places the result in Rd. In other words, Rt := Rd[31-Shamt:0].0Shamt where 0* denotes a string of k 0s. For load and store instructions, the RX000 uses the typical RISC technique of pro-viding a short address in the instruction, which serves as an offset to a fulllengthaddress stored in a CPU register. The I-type format of Figure 3.29 is used for load andstore instructions. In this case Rs serves as the base register, and Rt serves as the datasource (for store) or destination (for load). The instruction that loads a word into theCPU has the assembly-language format LW Rt, IMM(Rs) which causes the 16-bit immediate address EMM, that is, the offset, to be sign-extendedto 32 bits and added to the contents of Rs to form the effective address. This address isthen used to read a word of data from M into register Rt. In HDL terms Rt := M(Rs + MM) Addressing modes. The purpose of an address field is to point to the currentvalue V(X) of some operand X used by an instruction. This value can be specifiedin various ways, which are termed addressing modes. The addressing mode of Xaffects the following issues: • The speed with which V(X) can be accessed by the CPU. • The ease with which V(X) can be specified and altered. Access speed is influenced by the physical location of V(X)—normally the CPUor the external memory M. Operand values located in CPU registers, such as the general-register file and the program counter PC, can be accessed faster thanoperands in M. It is therefore usual to favor instructions that address CPU regis-ters, both in the design of instruction sets and in their use in computer programs.An operand's accessibility is also affected by the directness of its addressingmode: The address field X itself can be V(X), it can specify directly the locationof V(X), or it can identify a location that specifies directly the location of V(X).We can thus distinguish the number of levels of indirection associated with anaddress. The advantage of indirection, as we will see, is increased programmingflexibility. We can achieve further flexibility by providing addresses that areautomatically altered or indexed, for example, to step through an array of consec-utive addresses. If the value V(X) of the target operand is contained in the address field itself,then X is called an immediate operand and the corresponding addressing mode isimmediate addressing. By implication X is a constant, since it is very undesirableto modify instruction fields during execution.4 More often than not, X is a variablein the usual mathematical sense, and the corresponding address field identifies thestorage location that contains the required value V(X). Thus X corresponds to avariable, and its value V(X) can be varied without modifying the instructionaddress field. Operand specification of this type is called direct addressing. The addressing modes of the operands appearing in a machine-languageinstruction, which can vary from operand to operand, are defined in the instruc-tion's opcode. Some assembly languages allow addressing modes to be similarlydefined by distinct opcodes. For example, the assembly language of the Intel 8085series has the opcode MOV (move) to specify data transfers involving directaddressing only. Therefore, the register-to-register transfer A := B, for instance, isspecified by MOV A,B (3.26) The A and B operands of (3.26) are considered to be directly addressed, since thecontents of the named registers are the desired operand values. In contrast, to spec-ify the operation A := 99, where 99 is an immediate operand, the 8085 instruction MVI A, 99 (3.27) with the opcode MVI (wove /mmediate) must be used. Note that (3.27) uses boththe direct and immediate addressing modes. Most assembly languages take a different approach by specifying the address-ing modes in the operand fields. For example, the Motorola 680X0 equivalents of(3.26) and (3.27), with Dl = A and D2 = B are and MOVE D2,D1MOVE #99. Dl (3.28) respectively. (Note that the Motorola operand order is reversed with respect to theIntel convention.) In (3.28) the prefix # indicates that the immediate addressingmode is to be used for the operand in question. Deleting the # from (3.28) causes 4Self-modifying programs like the IAS code shown in Figure 1.15 (section 1.2.2) reflect the madeqithe addressing modes available in the earliest computers. 185 CHAPTER 3 Processor Basics 186 SECTION 3.3Instruction Sets the first operand to refer to the data in memory location 99, that is, M(99), whichwould be an instance of direct memory addressing. It is sometimes useful to change the location (as opposed to the value) of Xwithout changing the address fields of any instructions that refer to X. This may beaccomplished by indirect addressing, whereby the instruction contains the addressW of a storage location, which in turn contains the address X of the desired operandvalue V(X). By changing the contents of W, the address of the operand valuerequired by the instruction is effectively changed. While direct addressing requiresonly one fetch operation to obtain an operand value, indirect addressing requirestwo. Figure 3.30 illustrates these different ways of specifying operands in the caseof three load instructions that transfer the number 999 to the CPU register AC. The ability to use all addressing modes in a uniform and consistent way withall opcodes of an instruction set or assembly language is a desirable feature termedorthogonality. Orthogonal instruction sets simplify programming both by reducingthe number of distinct opcodes needed and by simplifying the rules for operandaddress specification. Many CISC computers like the 680X0 have little orthogo-nality, since processor costs can be reduced (at the expense of programming costs)by restricting instructions to a few frequently used addressing modes that varyfrom instruction to instruction. LOADI 999 AC Memory (a) LOAD X AC ■• 999 Memory (*) LOADN W AC 999 Memory (c) Figure 3.30 Three basic addressing modes: (a) immediate; (b) direct; (c) indirect. Relative addressing. Absolute addressing, conceptually the simplest mode of 187direct address formation, requires the complete operand address to appear in theinstruction operand field. This address is used without modification (except, per-haps, zero or sign extension in the case of a short address field) to access the desireddata item. Frequently, only partial addressing information is included in the instruc-tion, so the CPU must construct the complete (absolute) address. One of the com-monest address construction techniques is relative addressing, in which theoperand field contains a relative address, also called an offset or displacement D.The instruction also implicitly or explicitly identifies other storage locations R{, R2 Rk (usually CPU registers) containing additional addressing information. The effective address A of an operand is then some function f(D,R]tR2,...,Rk). In mostcases of interest, each operand is associated with a single address register R from aset of general-purpose address registers, and A is computed by adding D to the con-tents of R. that is, A:=R + D R may also be a special-purpose address register such as the program counter PC.There are several reasons for using relative addressing. 1. Since all the address information need not be included in the instructions,instruction length is reduced. 2. By changing the contents of R, the processor can change the absolute addressesreferred to by a block of instructions B. This address modification permits theprocessor to move (relocate) the entire block B from one region of main mem-ory to another without invalidating the addresses in B. When used in this way, Rmay be referred to as a base register and its contents as a base address. 3. R can be used for storing indexes to facilitate the processing of indexed data. Inthis role R is called an index register. The indexed items X(0), X(\),...,X(k) arestored in consecutive addresses in memory. The instruction-address field D con-tains the address of the first item X(0), while the index register R contains theindex i. The address of item X(i) is D + R. By changing the contents of the indexregister, a single instruction can be made to refer to any item X(i) in the givendata list. The main drawbacks of relative addressing are the extra logic circuits and process-ing time needed to compute addresses. So far we have assumed that each operand is a single memory word and cantherefore be specified by a single address. If an instruction must process variable-length data consisting of many words, each operand specification is divided intotwo parts: an address field that points to the location of the first word of the oper-and and a length field L that indicates the number of words in the operand. TheCPU automatically increments the instruction address field as successive words ofthe operand are accessed. The access is complete when L words have beenaccessed. Indexed items are frequently accessed sequentially so that a reference to X(k)stored in memory location A is immediately followed by a reference to X(k + 1) orX(k-l) stored in location A + 1 or A -1. respectively. To facilitate stepping througha sequence of items in this manner, addressing modes that automatically incrementor decrement an address can be defined; the resulting address-modification process CHAPTER 3 Processor Basics 188 is called autoindexing. In the case of the Motorola 680X0 series [Motorola 1989], the address field -(A3) appearing in an assembly-language instruction indicates I stm ti n s ts tnat me con,:ents °ftne designated address register A3 should be decremented automatically before the instruction is executed; this process is called predecrementing.Similarly, (A3)+ specifies that A3 should be incremented automatically after thecurrent instruction has been executed ipostincrementing). In each case the amountof the address increment or decrement is the length in bytes of the indexed oper-ands. Most processors have only a few, simple addressing modes for CPU registers,principally direct and immediate addressing. Immediate addresses represent datavalues that come with the instruction fetch and are placed in the instruction registerIR. In register direct addressing, the address (name) R of the register containingthe desired value V(R) appears in the instruction. The Motorola 680X0 instruction MOVE #99, Dl which means "move the constant 99 to data register D1," uses immediate address-ing for 99 and register direct (or simply direct) addressing for Dl. The term register indirect addressing refers to indirect addressing with a regis-ter R name in the address field. It is often used to access memory, in which case Rbecomes a memory address register. For example, MOVE.B (A0),D1 uses parentheses to indicate that (A0) is an indirect address involving the 680X0'sA0 addresss register. This move-byte instruction—the opcodes's .B suffix speci-fies a 1byte operand—corresponds to D1[7:0]:=M(A0) and copies the byte addressed by A0 into the low-order byte position of data regis-ter Dl. (The other three bytes of Dl are unchanged.) An extension of this address-ing mode is register indirect with offset, which can also be viewed as a type of baseor indexed addressing. This mode is the only memory addressing mode employedby the MIPS RX000 series (Example 3.5). The RXOOO's store-word instruction, forexample, is written as SW Rt, OFFSET(Rs) (3.29) where Rs is the base register and OFFSET is a number acting as an (immediate)offset operand. Instruction (3.29) is equivalent to the HDL statement M(Rs + OFFSET) := Rt where the offset is sign-extended before adding it to Rs to obtain the effectiveaddress Rs + OFFSET. The PowerPC has two addressing modes: register indirectwith offset as described above (but called register indirect with immediate index)and a second mode (called register indirect with index) in which the effectiveaddress is Rs + Ri, where Ri is a register name. The Motorola 680X0, like other CISC-style architectures, has many address-ing modes, including the following: immediate, register direct, register indirect,register indirect with postincrement, register indirect with predecrement, registerindirect with offset, register indirect with index, absolute short, absolute long, PC with offset, and PC with index. Its autoindexing features are illustrated in the fol- 189lowing example. EXAMPLE 3.6 STACK CONTROL IN THE MOTOROLA 680X0 [GILL. CORWIN and logar 1987; motorola 1989]. A stack is a sequence of storage locationsthat are accessible from only one end referred to as the top of the stack. A write opera-tion addressed to a stack, termed a push operation, stores a new item at the top of thestack, while a read operation, termed a pop operation, removes the item stored at thetop of the stack. Push or pop changes the position of the stack top by an amount thatdepends on the length of the operand pushed or popped. A stack is controlled by anaddress register called the stack pointer SP. This register stores the address of the lastoperand placed in the stack; that address is automatically adjusted after a push or popoperation so that SP contains the address of the new stack top. Some computers—the Intel 80X86, for example—have special instructions andhardware for handling stacks that are intended as communication areas for program-control instructions like call and return. A few early computers such as the BurroughsB6500/7500 even employed stacks in place of general-register files; see Example 1.5(section 1.2.3). The Motorola 680X0 has no explicit hardware for stack support, but, aswe now show, its various addressing modes make it easy to treat any contiguous regionof its external memory M as a stack. Suppose that the programmer designates the address register A2 of the 680X0 tobe a stack pointer and that the stack grows toward the low addresses of M. To push thecontents of a data register, say, D6, into the stack requires the single instruction MOVE.L D6,-(A2) (3.30) The input operand is the 4-byte contents of D6, which is directly addressed in (3.30),while the output operand, which is the new contents of the top of the stack, is designated by -(A2), which denotes indirect addressing with predecrementing using addressregister A2. This push instruction is equivalent to the following HDL operations: A2 := A2 - 4; M(A2) := D6; Figure 3.31 shows the state of the affected parts of the CPU and M immediately before(Figure 3.31a) and immediately after (Figure 3.316) execution of instruction (3.30).Observe how the data bytes are stored in M according to the big-endian convention.It is easily seen that the pop instruction corresponding to (3.30) is MOVE.L (A2)+,D6 (3.31) which is equivalent to D6 := M(A2); A2 := A2 + 4; In this case the operand (A2)+ employs the register indirect with postincrementaddressing mode. Number of addresses. Some computers, notably CISCs like the 680X0, haveinstructions of several different lengths containing various numbers of addresses.A source of controversy in the early days was the question of how many explicitoperand addresses to include in instructions. Clearly the fewer the addresses, theshorter the instruction format needed. However, limiting the number of addressesalso limits the range of operations that an instruction can perform. Roughlyspeaking, fewer addresses mean more primitive instructions and therefore longer CHAPTER 3 Processor Basics 190 SECTION 3.3Instruction Sets CPU D6 = stack data register B(0,3) B(0,2) B(0.1) B(0.0) A2 = stack pointer register OF FF 78 54 1 word ■ (a) CPU D6 = stack data register B(0,3J B(0,2) B,, P2. and P3, where P, is an /-address machine.P0 is a zero-address stack machine, while />,. P2, and P3 are conventional computerseach with 16 general-purpose registers R0:R15 for data and address storage. All fourprocessors have instructions with the (assembly language) opcodes ADD. SUB, MUL. 220 SECTION 3.5Problems 31 23 15 7 0 Ra Ba3 Ba2 Bal BaO Rb Bb3 Bb2 Bbl BbO Register file (a) Figure 3.44 Snapshot of RX000 state. Byte address (hex) 100101102103104105106107108109 BaO Bal Ba2 Ba3 BbO Bbl Bb2 Bb3 M (b) Byte resslex) 100 101 BaO 102 Bal 103 Ba2 104 Ba3 105 BbO 106 Bbl 107 Bb2 108 Bb3 109 M (c) and DIV to implement the operations +, -, X, and /, respectively, (a) Using as few in-structions as you can, write a program for each of the four machines to evaluate the fol-lowing arithmetic expression: X := (A/B + CX D)/(D XE-F+ C/A) + G (3.44) Use standard names for any additional instructions that you need, for example, LOADor PUSH, (b) Calculate the total object-program size in bits for each of your four programs assuming the following data on machine-language instruction formats: opcodes(which contain no addressing information) are 8 bits long; memory-address length is16 bits; and register-address length is 4 bits. (For example, the two-address instructionLOAD R7,B for P2, which denotes R7 := M(B), occupies 8 + 4 + 16 = 28 bits.) 3.46. Figure 3.44a shows the byte-by-byte contents of two registers in the RX000 generalregister file, (a) Construct a short program that transfers the data in question from theregister file to memory M exactly as indicated in Figure 3.446. (b) Suppose that thesame two words must be stored as shown in Figure 3.44c, where they are not alignedwith memory word boundaries. Suggest two methods for performing the two-wordstorage operation in this case. 3.47. Show how each of the following macroinstructions can be implemented by a singlemachine instruction from the RX000 instruction set. (a) LI Rdest.IMM ; Load immediate: load IMM (sign-extended) into register Rdest (b) MOVE Rdest,Rsource ; Move contents of register Rsource to register Rdest (c) NOP ; No operation: execute an instruction cycle that does not change the : CPU's state 3.48. A new microprocessor is being designed with a conventional architecture employingsingle-address instructions and 8-bit words. Due to physical size constraints, only eight distinct 3-bit opcodes are allowed. The use of modifiers or the address field toextend the opcodes is forbidden, (a) Which eight instructions would you implement? Specify the operations performed by each instruction as well as the location of its op-erands, (b) Demonstrate that your instruction set is functionally complete in some rea-sonable sense; or if it is not, describe an operation that cannot be programmed usingyour instruction set. 221 CHAPTER 3 Processor Basics 3.49. Write a short code segment for the RXOOO to implement the following common macro,which computes the absolute value of the contents of register Rsource and puts the re-sult in register Rdest. ABS Rdest.Rsource 3.50. There are few well-defined general principles concerning hardware-software trade-offs in processor design. Two principles of this type are given below. Write a brief noteon each, illustrating it with examples, (a) "Whenever there is a system function that isexpensive and slow in all its generality, but where software can recognize a frequentlyoccurring degenerate case (or can move the entire function from run time to compiletime) that function [should be] moved from hardware to software, resulting in lowercost and improved performance." (George Radin, 1983) (b) "Simple, frequent, andhighly-skew conditional branches [e.g., tests for arithmetic overflow] should be imple-mented in hardware [rather than software]." (Brian Randell, 1985) 3.51. (a) Explain how directives differ from other assembly-language instructions, (b) Listthe criteria for using macros instead of subroutines to structure assemblylanguage pro-grams. 3.52. A program called a disassembler is sometimes useful for debugging programs. It is de-signed to convert object code to assembly-language format, thus reversing the work ofan assembler. However, a disassembler cannot recover all the structure of the originalassembly-language code. Explain in detail why this is so. 3.53. Consider the processor and memory state depicted in Figure 3.40 and suppose that ex-ecution of the subroutine continues to completion. Let the subroutine's RETURN in-struction be stored in memory location 2500 (decimal). Draw a diagram similar toFigure 3.40 that shows the system state at the same three points during the executionof RETURN. 3.6REFERENCES 1. Circello, J. et al. "The Superscalar Architecture of the MC68060." IEEE Micro, vol.15 (April 1995) pp. 10-21. 2. Cohen, D. "On Holy Wars and a Plea for Peace." IEEE Computer, vol. 14 (October1981)pp.48-54. 3. Colwell, R. P. et al. "Computers, Complexity, and Controversy." IEEE Computer, vol.18 (September 1985) pp. 8-19. 4. Feustel, E. A. "On the Advantages of Tagged Architecture." IEEE Transactions on Com-puters, vol. C-12 (July 1973) pp. 644-56. 5. Furber, S. B. VLSI RISC Architecture and Organization. New York: Marcel Dekker.1989. 222 6. Gill, A., E. Corwin, and A. Logar. Assembly Language Programming for the 68000. Englewood Cliffs, NJ: Prentice-Hall, 1987. SECTION 3.6 7 Goldberg, D. "What Every Computer Scientist Should Know about Floating-Point References Arithmetic." ACM Computing Surveys, vol. 23 (March 1991) pp. 5-48. 8. Hamming, R. W. Coding and Information Theory. 2nd ed. Englewood Cliffs, NJ: Pren-tice-Hall, 1986. 9. IEEE Inc. IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985), New York, August 1985. 10. Intel Corp. MCS-80/85 Family User's Manual. Santa Clara, CA, 1979. 11. Kane, G. and J. Heinrich. MIPS RISC Architecture. Englewood Cliffs, NJ: Prentice-Hall,1992. 12. Motorola Inc. M68000 Family Programmer's Reference Manual. Phoenix, AZ, 1989. 13. Myers, G. J. Advances in Computer Architecture. 2nd ed. New York: Wiley-Inter-science, 1982. 14. Patterson, D. A. and C.H. Sequin. "A VLSI RISC." IEEE Computer, vol. 15 (September1982) pp. 8-21. 15. Siewiorek, D. P. and R. S. Swarz. Reliable Computer Systems. 2nd ed. Burlington, MA:Digital Press, 1992. 16. van Someren A. and C. Atack. The ARM RISC Chip. Wokingham, England: Addison-Wesley, 1994. CHAPTER 4 Datapath Design An instruction-set processor consists of datapath (data processing) and controlunits. This chapter addresses the register-level design of the datapath unit, whileChapter 5 covers the control unit. The focus is on the arithmetic algorithms and cir-cuits needed to process numerical data. These circuits are examined first for fixed-point numbers (integers) and then for floating-point numbers. The use of pipeliningto speed up data processing is also discussed. 4.1 FIXED-POINT ARITHMETIC The design of circuits to implement the four basic arithmetic instructions for fixed-point numbers—addition, subtraction, multiplication, and division—is the maintopic of this section. It also discusses the implementation of logic instructions andALU design. 4.1.1 Addition and Subtraction Add and subtract instructions for fixed-point binary numbers are found in theinstruction set of every computer. In smaller machines such as microcontrollersthey are the only available arithmetic instructions. As we have seen in earlier chap-ters, addition and subtraction hardware (Example 2.7) or software (Example 3.1)can be used to implement multiplication and, in fact, any arithmetic operation.Beginning with Charles Babbage, computer designers have devoted considerableeffort to the design of high-speed adders and subtracters. As we will see. thesebasic circuits can be designed in many different ways that involve various.trade-offs between operating speed and hardware cost. 223 224 SECTION 4.1 Fixed-Point Arithmetic Basic adders. First consider the design of a circuit to add two n-bit unsignedbinary numbers, a topic discussed in section 2.1.3. The fastest such adder is, inprinciple, a two-level combinational circuit in which each of the n sum bits isexpressed as a (logical) sum of products or product of sums of the n input vari-ables. In practice, such a circuit is feasible for very Small values of n only, as itrequires c{n) gates with fan-in f(n), where both c(n) and f(n) grow exponentiallywith n. Practical adders take the form of multilevel combinational or, occasionally,sequential circuits. They sacrifice operating speed for a reduction in circuit com-plexity as measured by the number and size of the components used. In general, theaddition of two /7-bit numbers X and Y is performed by subdividing the numbersinto stages X, and Yt of length nt, where n> «, > 1. Xi and K, are added separately,and the resulting partial sums are combined to form the overall sum. The formationof this sum involves assimilation of carry bits generated by the partial additions. The sum zi,ci of two 1-bit numbers x, and v, can be expressed by the half-adderlogic equations z, = x, 0 >>,where zt is the sum bit, c, is the carry-out bit, © denotes EXCLUSIVE-OR, andjuxtaposition denotes AND. If we introduce a third input bit c,_, denoting a carry-insignal, we obtain the following full-adder equations: c, = jr,y,+ x,r,_1 + y,c,_1 (4.1) (Note that + denotes logical OR—not plus—here.) A full adder, also called a 1-bit adder, can be directly implemented from these equations in various ways, asdemonstrated by Figure 2.9 (section 2.1.1). Figure 4.1 shows a fast AND-ORrealization of a 1-bit adder, along with an appropriate circuit symbol for use inregister-level designs. The least expensive circuit in terms of hardware cost for adding two «-bitbinary numbers is a serial adder, the design of which was covered in Example 2.2.A serial adder adds the numbers bit by bit and so requires n clock cycles to com-pute the complete sum of two n-bit numbers. As Figure 4.2 indicates, a serial adderconsists of a full adder realizing Equations (4.1) and a flip-flop to store c,. One sumbit is generated in each clock cycle; a carry is also computed and stored for use dur-ing the next clock cycle. Figure 4.2 presents a high-level view of a serial adder thathas a D flip-flop as the carry store. Although this adder is slow, its circuit size isvery small and is independent of n. Circuits that, in one clock cycle, add all bits of two «-bit numbers, as well asan external carry-in signal cin, are called n-bit parallel adders or simply n-bitadders. The simplest such adder is formed by connecting n full adders as in Figure4.3. Each 1-bit adder stage supplies a carry bit to the stage on its left. A 1 appear-ing on the carry-in line of a 1-bit adder can cause it to generate a 1 on its carry-out line. Hence carry signals propagate through the adder from right to left, givingrise to the name ripplecarry adder. In the worst case a carry signal can ripplethrough all n stages of the adder. The input carry signal cm is normally set to 0for addition. The maximum signal propagation delay of an «-bit ripple-carryadder, which in synchronous circuit design determines the operating speed, is nd, fc> Sum;,Carrv out c, (a) (b) Figure 4.1 A 1-bit (full) adder: (a) two-level AND-OR logic circuit and (b) symbol. where d is the delay of a full-adder stage. Unlike a serial adder, the amount ofhardware in a ripple-carry adder increases linearly with n, the word size of thenumbers being added. Subtracters. Adders like those of Figures 4.2 and 4.3 operate correctly onboth unsigned and positive numbers because the 0 sign bit of a positive numberhas the same effect as a leading zero in an unsigned number. The best way to add 225 CHAPTER 4Datapath Design Data Carry Sum. Reset Clock Figure 4.2 A serial binary adder. 226 SECTION 4.1 Fixed-Point Arithmetic L J 1-bitadder 1-bitadder cn-i rr Ti -^n-l >'n-I xn-l Vn-l 1-bitadder *0 ^0 Figure 4.3 An n-bit ripple-carry adder composed of n 1-bit (full) adders. negative numbers—these have 1 as the sign bit—depends on the number code inuse. Adding -X to Y is equivalent to subtracting X from Y, so the ability to add neg-ative numbers implies the ability to do subtraction. Subtraction is relatively simple with twos-complement code because negation(changing X to -X) is very easy to implement. As discussed in section 3.2.2, if X =xn_x xn_2...x0 is a twos-complement integer, then negation is realized by -X = -l*n-2. . Xn + 1 (4.2) where + denotes addition modulo 2". An efficient way to obtain the ones-comple-ment portion X = xn_xxn_2... x0 of -X in (4.2) uses the word-based EXCLUSIVE-OR functionX® s with a control variable s. When s = 0, X© s = X, but when 5=1,X © 5 = X. Suppose that Y and X © s are now applied to the inputs of an n-bitadder. The addition of 1 required by (4.2) to change X to -X can be realized byapplying s to the carry input line of the adder. In the resulting circuit shown in Fig-ure 4.4, the control line s selects theaddition operation Y + X when 5 = 0 and thesubtraction operation Y — X = Y + X + 1 when 5=1. Thus extending a paralleladder to perform twoscomplement subtraction as well as addition merely requiresconnecting n two-input EXCLUSIVE-OR gates to the adder's inputs; these gatesare represented by a single rcbit word gate in Figure 4.4. z = Y±X 1 \ n Carry Carry ^ cn-\ rc-bit paralleladder Cin in out i ". i / -i n, * (\ ".,- Subtract s Figure 4.4 An n-bit twos-complement adder-subtracter. As an example, let X = 11101011 and Y = 00101000, denoting -2110 and 4010, 227 respectively, in twos-complement code. Bit-by-bit addition produces v J v J y CHAPTER 4 Z = X+Y= 11101011 +00101000 = 00010011 (4.3) Datapath Design which corresponds to -21,0 + 4010 = +1910. (Observe that the output carry c„_, = 1in (4.3) is ignored.) To subtract X from Y, we first compute -X= 1110 10 11+ 1=00010101 and then the sum Z = (-X) + Y = 00010101 + 00101000 = 00111101 which corresponds to 2110 + 4010 = +6110. Subtraction is not so readily implemented in the case of unsigned or sign-mag-nitude numbers. It is sometimes useful to construct a subtracter for such numbersbased on the full (1-bit) subtracter function z, = y, - xt - bt_x. This operation isdefined by the logic equations: Z;=X,@y,0&;_l b, = xiyi+xibi_i+>>,_! Here z, is the difference bit, while b,_, and b{ are the borrow-in and borrow-out bits,respectively, n-bit serial or parallel binary subtracters are constructed in essentiallythe same way as the corresponding adders with carry signals replaced by borrows.Subtracters are of minor interest compared with adders, because, as we have justseen, an adder suffices for both addition and subtraction when twos-complementnumber code is used. Overflow. When the result of an arithmetic operation exceeds the standardword size n, overflow occurs. With n-bit unsigned numbers, overflow is indicatedby an output carry bit c„_, = 1. For example, adding the unsigned numbers X =11101011 = 23510 and Y= 00101010 = 4210 using an adder like that of Figure 4.3yields Z=X+ Y= 11101011 +00101010 = 00010101 (4.4) with c„_[= c7 =1. Now Zcorresponds to 2110, which is 235]0 + 4210 (modulo 256)and is the result of addition that "wraps around" when the largest number 2" - 1. inthis case 11111111 = 25510, is exceeded. On appending c7 to Z, we get c7Z =100010101 = 27710 = 25610 + 2110, which is the sum in ordinary (modulo infinity)arithmetic. Unsigned arithmetic operations are often viewed as modulo-2" opera-tions only, and overflow is not explicitly detected. This is the case when computingmemory addresses in a computer, for instance, where addresses simply wraparound to zero after the highest address is reached. Overflow is indicated by a flag bit v in operations involving signed numbers;this flag is found in CPU status (condition code) registers. If we reinterpret thenumbers in the preceding example as twos-complement rather than as unsigned,then X = 11101011 denotes -2110, while Y = 00101010 denotes +421(). The result Zcomputed in (4.4) now denotes +2110, and the fact that cn_l = 1 does not indicateoverflow. In fact, we can never have overflow on adding a positive to a negativenumber. Overflow in modulo-2" twos-complement addition can only result fromadding two positive numbers or two negative numbers. In the first case overflow 228 SECTION 4.1 Fixed-Point Arithmetic is indicated by a carry bit into the sign position, that is, by c„_2 = 1, since this indi-cates that the magnitude of the sum exceeds the n - 1 bits allocated to it. A littlethought shows that overflow from adding two negative numbers is indicated bycn_2 = 0. We can thus conclude (as we did earlier in section 3.2.2) that the over-flow condition is specified by the logic expression L7!-l->7!-lLn-2 + X n-Vn-\Ln-2 (4.5) Now c„_,, the carry output signal from the sign position, is defined by xn_lyn_l +xn_iCn_2 + }'„_icn-2' fr°m which it follows that v = c„ (4.6) Either (4.5) or (4.6) can be used to design overflow detection logic for twos-complement addition or subtraction. Overflow detection in the case of sign-magnitude numbers is similar and is left as an exercise (problem 4.6). High-speed adders. The general strategy for designing fast adders is to reducethe time required to form carry signals. One approach is to compute the input carryneeded by stage i directly from carrylike signals obtained from all the precedingstages i - l,i - 2,...,0, rather than waiting for normal carries to ripple slowly fromstage to stage. Adders that use this principle are called carry-lookahead adders. An/i-bit carry-lookahead adder is formed from n stages, each of which is basically afull adder modified by replacing its carry output line c, by two auxiliary signalscalled gj and /?,, or generate and propagate, respectively, which are defined by thefollowing logic equations: &=*# Pi=xi+yt (4-7) The name generate comes from the fact that stage i generates a carry of 1 (c, = 1)independent of the value of ct_x if both x, and y, are 1; that is. if x,v, = 1. Stage ipropagates cM; that is, it makes c, = 1 in response to c,_, =1 if x, or y, is 1—in otherwords, if Xj + y, = 1. Now the usual equation c, = jc,v,+ *,', O-i Dataout 53 S2 5[S0 Select 5 Figure 4.29 An n-bit logic unit that realizesall 16 two-variable functidns. 254 SECTION 4.2 Arithmetic-Logic Units gates. The complete 4-bit ALU can therefore be expected to contain more than 100gates of various kinds and have depth 9 or so. By judicious sharing of functionsbetween the two main subunits, both of these figures can be reduced by a third, asthe next example shows. EXAMPLE 4.4 DESIGN OF A COMBINATIONAL ARITHMETIC-LOGIC UNIT [Hansen and hayes 1995]. We now examine the structure of a well-known com-binational ALU design that is found in many commercial products including me74181, an IC referred to as a 4-bit ALU/function generator [Texas Instruments 1988].Like the circuit of Figure 4.29, this design implements all 16 two-variable logic func-tions, as well as 16 arithmetic functions (some of which, like X Y plus A, are of ques-tionable value). Its standard realization has about 60 gates and depth 6; see problem4.21. We will describe its structure at the register level, following the model developedin [Hansen and Hayes 1995]. The main internal features of the 74181 appear in Figure 4.30. The key arithmeticoperation of twos-complement addition is implemented by the carry-lookaheadmethod. As in the design of Figure 4.6. the adder consists of propagate-generate logicfeeding a lookahead circuit that computes carries, and a set of XOR gates that computethe final sum. The 74181's carry-lookahead generator is the same as that given earlierwith the addition of propagate and generate outputs (denoted p and g) for extensionpurposes. However, the pg and sum circuits are also designed to be shared with thelogic unit in an efficient, but nonobvious fashion. The modules labeled M] and M2 gen-erate a pair of 4-bit signals IP and IG that serve as internal propagate and generate,respectively, in the arithmetic mode and as minterm sources in the logic mode. FromFigure 4.30 we see that each data output function Fi is defined by Ft = IPf® IG/® (10^ + M) (4.34) Select 5 Carry in clr Mode M Figure 4.30 A register-level view of the 74181 4-bit ALU. for 3 > i > 0, where IC denotes the set of four internal carries produced by the carry-lookahead generator. The IP and IG functions are defined by IP, = A, + B,S0+B,Sl (4.35) IG^AiBfo + AjBfo (4.36) (See Figure 4.64 in this chapter's problem set for the gate-level implementation ofthese functions.) In the logic mode of operation, M - 1, so (4.34) becomes F, = IP, © Jg , (4.37) On substituting (4.35) and (4.36) into (4.37) and simplifying, we obtain Fi =A ,B,SQ+AfB,S i + AtBS2 + Afifo (4.38) This expresses F,(A,,fi() in sum-of-minterms form, with a distinct (possibly comple-mented) select variable controlling each minterm. It therefore produces a differentlogic function for each of the 16 possible combinations of the 5 variables, and so isessentially the same as (4.33). Hence with M= 1, the 74181 acts as a universal functiongenerator capable of producing any two-variable Boolean function F(A,B).In the arithmetic mode M = 0, and (4.34) changes to F^/^e/G,©/^., This has the general form of a sum (or difference) output—compare Equation (4.11).We can interpret the entire output function F = F3F2F]F0 more easily using the arithmetic expression F = IP plus IG plus c(4.39) which is implied by (4.35) to (4.37) when M - 1. Here plus denotes twos-complementaddition to distinguish it from + denoting logical OR. When S - 1001, Equations (4.35)and (4.36) imply that IP, and IG, become the usual propagate and generate functions,IPj = Aj + Bj and /G, = Afi^ respectively. Hence the control settings M - 1 and S = 1001make the 74181 behave like a carry-lookahead adder that computes F = A plus B plus cjn Changing 5 to 0110 produces the twos-complement subtraction F = A minus B minus cin and effectively reconfigures the ALU as shown in Figure 4.4. The various combinations of 5 produce a total of 16 different functions in thearithmetic mode, only a few of which are useful. For example, with S = 0100. Equation(4.39) becomes F = 1111 plus 0000 plus cin which is 1111 when cin = 0, that is, the constant minus-one in twos-complement code.When cjn = 1, F changes to 0000, since we are adding plus-one to minus-one. The abil-ity to generate constants like ±1 and 0 in this way is useful for implementing sometypes of instructions. The74181's/J, g, and coul outputs are intended to allow k copies of the 74181 to becombined either using ripple-carry propagation or carry-lookahead to form a 4£-bitALU. Figure 4.31 shows a 16-bit ALU composed of four 74181 stages, with ripple-carry propagation between stages; compare Figure 4.3. Note how the 5 and AT control 255 CHAPTER 4Datapath Design 256 SECTION 4.2 Arithmetic-Logic Units lines are shared, while the data lines are separate. Note too that no interstage connec-tions are needed for the logic operations because of their bitwise, word-oriented nature.Another interesting feature of the 74181 is its ability to act as a magnitude comparatorin conjunction with the carry output cout; see problem 4.23. The electronic circuits driv-ing the 74181's (A = B) output are

designed so that "when several (A = B) lines arewired together as in Figure 4.31, the wired connection outputs the AND function of allits input signals. In other words, the overall (A = B) output signal is 1 if and only if each74181 slice produces (A = B)= 1. This type of technology-specific connection is calleda wired AND. No extra gates or other "glue" logic are needed for ripple-carry expan-sion of the 74181. 4.2.2 Sequential ALUs Although, as we have seen, both multiplication and division can be implementedby combinational logic, it is generally impractical to merge these operations withaddition and subtraction into a single, combinational ALU. The reason is twofold.Combinational multipliers and dividers are costly in terms of hardware. They arealso much slower than addition and subtraction circuits, a consequence of theirmany logic levels. An n-bit combinational multiplier or divider is typically com-posed of n or more levels of add-subtract logic, making multiplication and divisionat least n times slower than addition or subtraction. The number of gates in themultiply-divide logic is also greater by a factor of about n. Hence except when n isvery small, complete ALUs are usually constructed from low-cost sequential cir-cuits where add and subtract each take one clock cycle, while multiplication anddivision are multicycle operations. Basic design. Figure 4.32 shows a widely used sequential ALU design thataims at minimizing hardware costs. This ALU organization is found in the IAScomputer (Figure 1.11) and in many computers built after IAS. It is intended to (A = B) (A = B) F> Aw'.Aa SiiiSo Fn.Fi (A=B) Fv-Fn 741814-bitALU 4,< 4,> A-,:A4 By.Bt 741814-bitALU ALU •*—| A A 4.- 4,A3A0 ByB0 -is— SM Figure 4.31 A 16-bit combinational ALU composed of four 74181s linked by ripple-carry propagation. Systembus Accumulator AC Multiplier-quotientregister MQ Parallel adder andlogic circuits (Memory) dataregister DR Flags Control unit Figure 4.32 Structure of a basic sequential ALU. 257 CHAPTER 4Datapath Design implement multiplication and division using one of the sequential digit-by-digitshift-and-add/subtract algorithms discussed earlier. Three one-word registers areused for operand storage: the accumulator AC, the multiplier-quotient register MQ.and the data register DR. AC and MQ are organized as a single register AC.MQcapable of leftand right-shifting. Additional data processing is provided by acombinational ALU capable of addition, subtraction, and logical operations; wewill refer to this unit as the add-subtract unit. This unit derives its inputs from ACand DR and places its results in AC. The MQ register is so-called because it storesthe multiplier during multiplication and the quotient during division. DR stores themultiplicand or divisor, while the result (product or quotient and remainder) isstored in the register-pair AC.MQ. The role of these registers is defined conciselyas follows: Addition Subtraction Multiplication Division AND OR EXCLUSIVE-OR NOT AC := AC + DRAC := AC - DRAC.MQ := DR x MQAC.MQ := MQ/DRAC := AC and DRAC := AC or DRAC := AC xor DRAC := not(AC) DR can serve as a memory data register to store data addressed by an instructionaddress field ADR. Then DR can be replaced by M(ADR) in the above list of ALUoperations, resulting in a one-address memory-referencing format. Register files. Modern CPUs retain special registers like the multiplier-quo-tient register MQ for multiplication and division, but the accumulator AC and thedata register DR are usually replaced by a set of general-purpose registers R(,:Rm_| known as a register file RF. Each register R, in RF is individually addressable—itsaddress is the subscript /—so that arithmetic-logic instructions can take the generictwoand three-address forms SECTION 4.2 Arithmetic-Logic Units R2 :=/(R!,R2) . (4.40) R3:=/(R1,R2) (4.41) respectively. Hence the processor can retain intermediate results in fast, easilyaccessed registers, rather than having to pack them off to external memory M.Clearly RF functions as a small random-access memory (RAM) and, in fact, isoften implemented using a fast RAM technology. RF differs from M in one impor-tant respect: RF requires two or three operands to be accessible simultaneously.For example, to implement (4.40) as a single-cycle instruction, we must be able toread R, and R2, and write to R2 in the same clock cycle. RF then needs severalaccess ports for simultaneously reading from or writing to several different regis-ters. Hence a register file is often realized as a multiport RAM. A standard RAMhas just one access port with an associated address bus ADR and data bus D. Thisport can be used to read or write the data word in the single word location wedenote by M(ADR). To build a multiport register file requires a set of registers of the appropriatesize and several multiplexers and demultiplexers that allow data words to besteered from any desired registers to the various output ports (read operations) orfrom the various input ports to registers (write operations). Of course, we don'twant several devices writing to the same register R, simultaneously, although theymay read from several R/s simultaneously. Figure 4.33 shows a three-port registerfile that supports simultaneous reads from two ports A and 5, while writing cantake place via a third port C. This file contains four 16-bit registers and meets thedata access requirements of (4.40) and (4.41). In the two-address case (4.40), theaddress of R, is applied to port A, while that of R2 is applied to ports B and C. Figure 4.34 shows a representative datapath unit for implementing logical andfixed-point operations; it is often referred to as an integer or fixed-point unit. Itcontains a register file RF and a (combinational) ALU capable at least of additionand subtraction. Often specialized circuitry is added for multiplication and divi-sion because the longer delay of these operations and their use of double-lengthoperands make it difficult to include their registers in RF. Also shown are linksthat connect the datapath unit to the external memory M (a cache or main mem-ory) and the IO system. These links can also connect to other functional units suchas a floating-point unit. ALU expansion. It is quite feasible to manufacture an entire sequential ALUfor fixed-point w-bit numbers on a single IC chip. Moreover, the ALU can easilybe designed for expansion to handle operands of size n = km, or indeed any wordsize n > m, in two ways: 1. Spatial expansion: Connect k copies of the m-bit ALU in the manner of a rip-ple-carry adder to form a single ALU capable of processing km-b\t wordsdirectly. The resulting array-like circuit is said to be bit sliced because eachcomponent ALU concurrently processes a separate "slice" of m bits from eachkm-b\\. operand. Data in C16L Address C Address A PortC Register fileRF Portal |Portfi Address B Writeaddress C Tel ieT Data out A Data out B(a) Data in C16L J- 4-way 16-bitdemultiplexer 16J, 16| 16 L 16 L 16-bit register R3 16 X 16-bit register R2 16 JL. 16-bit register R, 16 16-bit register Rq 16 Readaddress A 4-way 16-bitmultiplexer Data out A 4-way 16-bit / 2 Readmultiplexer s/ address B 16LData out B (b) Figure 4.33 A register file with three access ports: (a) symbol and (b) logic diagram. 259 CHAPTER 4Datapath Design 2. Temporal expansion: Use one copy of the m-bit ALU chip in the manner of aserial adder to perform an operation on /cm-bit words in k consecutive steps(clock cycles). In each step the ALU processes a separate m-bit slice of eachoperand. This processing is called multicycle or multiple-precision processing. The 16-bit ALU in Figure 4.31 composed of four copies of the 4-bit 74181 ICis an example of a bit-sliced combinational ALU. The hardware cost of a bit-slicedALU such as this increases directly with k, the number of slices, but the ALU'sperformance measured, say, in cycles per instruction (CPI), remains essentiallyconstant. The cycle period does increase slowly with k, however. In a multicycleALU, on the other hand, the performance decreases directly with k. but the amountof hardware remains constant. A multicycle ALU must be controlled by a (micro)program that repeatedly applies the same basic instruction to all slices of the oper-ands, which must be supplied serially (slice by slice) to the ALU. 260 SECTION 4.2 Arithmetic-Logic Units (Micro) program control unit To M and IO system Figure 4.34 A generic datapath unit with an ALU and a register file. Figure 4.35 shows how a 16-bit ALU can be constructed from four 4-bitsequential ALU slices. The data buses and register files of the individual slices areeffectively juxtaposed to increase their size from 4 to 16 bits. The control lines thatselect and sequence the operations to be performed are connected to every slice sothat all slices execute the same actions in lockstep with one another. Each slice thusperforms the same operation on a different 4-bit part (slice) of the input operandsand produces only the corresponding part of the results. The required control sig-nals are derived from an external control unit, which can be hardwired or micro-programmed. Certain operations require information to be exchanged betweenslices. For example, to implement a shift operation, each slice must be able to senda bit to, and receive a bit from, its left or right neighbors. Similarly, when perform-ing addition or subtraction, carry bits must be transmitted between neighboringslices. For this purpose horizontal connections are provided between the slices asshown in Figure 4.35. A multicycle implementation of the 16-bit ALU of Figure 4.35 would requirethe basic 4-bit ALU to store internally all the information that needs to beexchanged between slices. Add and shift operations require only modest changeslike extra flip-flops to store the output carry and shift signals, as well as (micro)instructions of the add-withcarry type that make use of these stored signals. Multi-plication and division require more significant changes. EXAMPLE 4.5 THE ADVANCED MICRO DEVICES 2901 BIT-SLICED ALU [MICK AND brick 1980). AMD introduced the 2900 series of ICs for bit-slicedprocessor design in the mid-1970s. Its elegant design has been widely imitated, and its principal members are included in recent VLSI cell libraries [AT&T Microelectronics1994]. The 2901 IC is the simplest of several 4-bit ALU slices in the 2900 family. Ithas the internal organization depicted in Figure 4.36 and executes a small set of opera-tions usually specified by microinstructions. A combinational arithmetic-logic circuitC performs three arithmetic operations (twos-complement addition and subtraction)and five logical operations on 4-bit operands. The particular operation to be carried outby C is defined by a 9-bit (micro) instruction bus I intended to be driven by an externalcontrol unit. A pair of combinational shifters allow results generated by C to be left- orright-shifted to facilitate the implementation of multiplication, division, and so on viashift-and-add/subtract algorithms. The 2901 has a general-register organization withsixteen 4-bit registers organized as a 16 x 4-bit register file R[0:15], referred to as "theRAM." An additional register designated Q is designed to act as the multiplierquotientregister when implementing multiplication or division. C obtains its inputs either fromthe RAM, Q, or an external input data bus D; all-0 constant input operands may also be 261 CHAPTER 4Datapath Design Data 16, Shift *■signals _ Carry outand flags Slice [15:12] Register file—ICombinationalALU Controlcircuits Slice [11:8] ^\ Register file CombinationalALU Controlcircuits Slice [7:4] Register file CombinationalALU Controlcircuits Slice [3:0] Register file CombinationalALU Controlcircuits Shiftsignals Carry in Control Figure 4.35 Sixteen-bit ALU composed of four 4-bit slices. 262 SECTION 4.2 Arithmetic-Logic Units RAM3 >*Q34 A 7^RAMaddresses 4B *Carry out cout -«Carrylookahead Sign F3 -*Overflow OVR -*Zero Z -*Data in D4/ RAM shifter 4,' B 16 x 4-bitregister file (RAM)A B 0 Q shifter 4: r Q register 1 1 1 I 1 1 1 Multiplexer 4/ Multiplexer 4/ 4-bit arithmetic-logic circuit C Instruction I 7*Decoder 4/ RAMq Qo Carry in cn \ Multiplexer / Data out Y Figure 4.36 Organization of the 2901 4-bit ALU slice. specified. The RAM registers to be used as operand sources or destinations are speci-fied by the 4-bit A and B address buses, which are also derived from an external micro-instruction. The results generated by C can be stored internally in the 2901 and/orplaced on the external output data bus Y. A set of k 2901s can be interconnected according to the one-dimensional arraystructure of Figure 4.35 to form a processor with essentially the same properties as the2901 but handling 4/c-bit instead of 4-bit data. The instruction bus I and the RAMaddress buses A and B are the main control lines that are connected in common to allslices. Direct connections between the shifters on adjacent slices permit shifting to beextended across the entire processor array. Each slice produces a carry-out signal cout that can be connected to the carry-in line cin of the slice on its left, allowing arithmeticoperations to be extended across the array via the bit-sliced scheme of Figure 4.7. Ripple-carry connections between slices have the drawback that carry-propagationtime increases rapidly with the number of slices. Consequently, the 2901 and other bitsliced ALUs also support the implementation of carry lookahead in the style of Figure4.5. To this end. the 2901 produces (in complemented form) the g and p signalsrequired for carry lookahead, and an external carry-lookahead circuit generates the cinsignals for the slices (except the right-most one) from the g's and p's of all precedingslices. The 2900 series has an IC for this purpose, namely, the 2902 4-bit carry-lookahead generator, which is a fast, two-level logic circuit that implements Equations(4.10). The 2901 also produces three flag signals providing status information on thecurrent result F from the arithmetic-logic circuit C. The zero flag Z indicates whetherthe all-0 result F = 0000 occurred; the overflow flag OVR indicates whether overflowoccurred during arithmetic operations; and the sign flag F3 is the value of the left-mostbit of F. A 16-bit ALU composed of four copies of the 2901 appears in Figure 4.37.This circuit employs carry lookahead. and also shows how the flag signals for the arrayare produced (compare Figure 4.31). The 290l's 9-bit control bus I contains three 3-bit fields—Is, IF, and ID—whichspecify the operand sources, the ALU function, and the result destinations, respec-tively; see Figure 4.38. ID is also used to control shifting of the result; this is indicatedby multiplication by 2 (left shift) or division by 2 (right shift) in the figure. The variouspossible combinations of the three I fields define the 290l's microinstruction set andenable a large number of distinct register-transfer operations to be specified. For exam-ple, the subtraction R[6]:=R[7]-R[6] 263 CHAPTER 4Datapath Design F3 OVR ** Z nM), then the shifting process toalign one of the mantissas, say, XM in AC, will result in AC = 0 after nM steps. Con-tinued shifting to make E = 0 will not affect the result, which in this case will beYM. Note also that it is more efficient to terminate the shifting after nM steps insteadof IEI steps, as is done in Figure 4.42. Figure 4.43 shows the step-by-step application of the addition algorithm ofFigure 4.42 to two 32-bit floating-point numbers. The numbers have the 32-bit for-mat of the IEEE Standard 754 described in Example 3.4. In this format each num-ber N has a 23-bit fractional mantissa M with a hidden bit. an 8-bit exponent E inexcess-127 code, and a base B = 2. The value of /Vis therefore given by the formula The numbers to be added in this instance are AT = 0 01111111 1000000000000000000000Y=0 10000111 00101011010000000000000 270 SECTION 4.3Advanced Topics register AC Kf-LO], DR^m-IiO]. E[/»e-1:0], El[nE-l:0], E2[nE-l:0], AC_OVERFLOW, ERROR;BEGIN: AC_OVERFLOW := 0, ERROR := 0. LOAD: El := XE, AC := XM: E2 := YE, DR := YM; {Compare and equalize exponents} COMPARE: E:=E1-E2; EQUALIZE: if E < 0 then AC := right-shift(AC), E := E + 1, go to EQUALIZE; else if E > 0 then DR := right-shift(DR), E := E - 1,go to EQUALIZE; {Add mantissas} ADD: AC := AC + DR, E := max(El,E2); {Adjust for mantissa overflow and check for exponent overflow} OVERFLOW: if AC_OVERFLOW = 1 then begin if E = EMAX then go to ERROR:AC := right-shift(AC), E := E + 1, go to END; end {Adjust for zero result} ZERO: if AC = 0 then E := 0. go to END; {Normalize result} NORMALIZE: if AC is normalized then go to END; UNDERFLOW: if E > EMIN then AC := left-shift(AC), E := E - 1, go to NORMALIZE; {Set error flag indicating overflow or underflow} ERROR: ERROR := 1; END: Figure 4.42 Algorithm for floating-point addition. which denote +1.510and +299.25,0, respectively. The exponent subtraction XE - YEin the COMPARE step is done using excess-127 code and produces 11110111 =-810. Note that a 0 in the left-most bit position of E always indicates a negativenumber in this code (see Figure 3.25). Now the EQUALIZE step is executed, caus-ing E to be incremented and AC, which contains the mantissa of X (including itshidden bit), to be right-shifted. After eight shifts, E reaches zero, indicated by itsleft-most bit changing from 0 to 1. Then the mantissa addition takes place, and thelarger exponent is transferred from El to E. The sum appearing in AC is normal-ized, so the final result X + Y = 300.7510 has its exponent in E and its mantissa inAC. The sum is eventually stored in the following standard format. X+Y=0 10000111 00101100110000000000000 EXAMPLE 4.6 FLOATING-POINT ADD UNIT OF THE IBM SYSTEM/360 model 91 [Anderson et al. 1967]. We now briefly describe the floating-point 271 Exponent registers Mantissa registers CHAPTER 4 Datapath El Step AC UK LOAD 01111111 = *E 10000111 E2 E 00000000 11000000000000... 00 10010101101000... 00 = ^E = 1-*M =1-^1 Design COMPARE 01110111= XE-YE EQUALIZE 01111000 01100000000000... 00 01111001 00110000000000. .00 01111010 00011000000000. .00 01111011 00001100000000. .00 01111100 00000110000000. .00 01111101 00000011000000. .00 01111110 00000001100000. .00 01111111 00000000110000. .00 10000000 10000111 ADD = >E Result 10000111= (X+50e 10010110011000... 00= AC + DR 10010110011000... 00 = i.(X+y)M Figure 4.43 Illustration of the floating-point addition algorithm of Figure 4.42. adder of the IBM System/360 Model 91, a mainframe computer of the mid-1960swhose advanced design features, including caches and several types of instruction-level parallelism, were very influential. Figure 4.44 shows the datapath of the Model91 's add unit. It adds or subtracts 32-bit and 64-bit numbers having the floatingpointformat specific to the System/360 family and its successors (see section 3.2.3). Thegeneral algorithm of Figure 4.42 is used with some changes to increase speed. In par-ticular, the shifting needed to align the mantissas and subsequently to normalize theirsum is carried out by combinational logic (barrel shifters) rather than by shift registers.These shifters allow k hexadecimal digits (recall that the base B is 16) to be shiftedsimultaneously. The corresponding subtraction of k from the exponent required fornormalization is also done in one clock cycle by using an extra adder (adder 31. The operation of this floating-point adder unit is as follows. The exponents of theinput operands are placed in registers El and E2, and the corresponding mantissas areplaced in Ml and M2. Next E2 is subtracted from El using adder 1: the result is used toselect the mantissa to be right-shifted by shifter 1 and also to determine the length ofthe shift. For example, if El > E2 and El - E2 = k, M2 is right-shifted by k digit posi-tions, that is. 4k bit positions. The shifted mantissa is then added to or subtracted fromthe other mantissa via adder 2, a 56-bit parallel adder with several levels of carry look-ahead. The resulting sum or difference is placed in a temporary register R where it isexamined by a special combinational circuit, the zero-digit checker. The output z ofthis circuit indicates the number of leading 0 digits (or leading Fs in the case of nega-tive numbers) of the number in R. The number z is then used to control the final nor-malization step. The contents of R are left-shifted z digits by shifter 2. and the result isplaced in register M3. The corresponding adjustment is made to the exponent by sub-tracting z using adder 3. In the event that R = 0, adder 3 can be used to set all bits of E3to 0, which denotes an exponent of -64. 272 Data SECTION 4.3Advanced Topics El E2 Ml M2 "3_r Adder1 E1-E2 Shifter 1 Adder 2 " \\ ll Adder3 Zero-digitchecker . „ Shifter 2 E3 Data Exponent comparison and mantissa alignment Mantissaaddition-subtraction Resultnormalization M3 Figure 4.44 Floating-point add unit of the IBM System/360 Model 91. Coprocessors. Complicated arithmetic operations like exponentiation and trig-onometric functions are costly to implement in CPU hardware, while softwareimplementations of these operations are slow. A design alternative is to use auxil-iary processors called arithmetic coprocessors to provide fast, low-cost hardwareimplementations of these special functions. In general, a coprocessor is a separateinstruction-set processor that is closely coupled to the CPU and whose instructionsand registers are direct extensions of the CPU's. Instructions intended for thecoprocessor are fetched by the CPU, jointly decoded by the CPU and the coproces-sor, and executed by the coprocessor in a manner that is transparent to the pro-grammer. Specialized coprocessors like this are used for tasks such as managingthe memory system or controlling graphics devices. The MIPS RX000 series, forexample, was designed to allow the CPU to operate with up to four coprocessors[Kane and Heinrich 1992]. One of these is a conventional floating-point processor,which is implemented on the main CPU chip in later members of the series. Coprocessor instructions can be included in assembly or machine code justlike any other CPU instructions. A coprocessor requires specialized control logic tolink the CPU with the coprocessor and to handle the instructions that are executedby the coprocessor. A typical CPU-coprocessor interface is depicted in Figure4.45. The coprocessor is attached to the CPU by several control lines that allow the Coprocessoraddressdecoder Select CPU Coprocessor ■" Busy Interrupt request Synchronization signals k i 1 r 1 System bus . To main memoryJ and 10 devices Figure 4.45 Connections between a CPU and a coprocessor. activities of the two processors to be coordinated. To the CPU, the coprocessor is apassive or slave device whose registers can be read and written into in much thesame manner as external memory. Communication between the CPU and copro-cessor to initiate and terminate execution of coprocessor instructions occurs auto-matically as coprocessor instructions are encountered. Even if no coprocessor isactually present, coprocessor instructions can be included in CPU programs,because if the CPU knows that no coprocessor is present, it can transfer programcontrol to a predetermined memory location where a software routine implement-ing the desired coprocessor instruction is stored. This type of CPU-generated inter-ruption of normal program flow is termed a coprocessor trap. Thus thecoprocessor approach makes it possible to provide either hardware or software sup-port for certain instructions without altering the source or object code of the pro-gram being executed. A coprocessor instruction typically contains the following three fields: anopcode F0 that distinguishes coprocessor instructions from other CPU instructions,the address F] of the particular coprocessor to be used if several coprocessors areallowed, and finally the type F2 of the particular operation to be executed by thecoprocessor. The F2 field can include operand addressing information. By havingthe coprocessor monitor the system bus, it can decode and identify a coprocessorinstruction at the same time as the CPU; the coprocessor can then proceed to exe-cute the coprocessor instruction directly. This approach is found in some earlycoprocessors but has the major drawback that the coprocessor, unlike the CPU,does not know the contents of the registers defining the current memory addressingmodes. Consequently, it is common to have the CPU partially decode every copro-cessor instruction, fetch all required operands, and transfer the opcode and oper-ands directly to the coprocessor for execution. This is the protocol followed in680X0-based systems employing the 68882 floating-point coprocessor, which isthe topic of the next example. 273 CHAPTER 4 Datapath Design EXAMPLE 4.7 THE MOTOROLA 68882 FLOATING-POINT COPROCESSOR [motorola 1989]. The Motorola 68882 coprocessor extends 680X0-series CPUs 274 Type Opcode Operation specified FMOVE Move word to/from coprocessor data or control register SECTION 4.3 Advanced Topics Data transfer FMOVECR Move word to/from ROM storing constants (0.0,7t, e, etc.) FMOVEM Data processing FADD Move multiple words to/from coprocessor Add FCMP Compare FDIV Divide FMOD Modulo remainder FMUL Multiply FPvEM Remainder (IEEE format) FSCAJLE Scale exponent FSGLMUL Single-precision multiply FSGLDIV Single-precision divide FSUB Subtract FABS Absolute value FACOS Arc cosine FASIN Arc sine FATAN Arc tangent FATANH Hyperbolic arc tangent FCOS Cosine FCOSH Hyperbolic cosine FETOX e to the power of x FETOXMI (e to the power of x) minus 1 FGETEXP Extract exponent FGETMAN Extract mantissa FINT Extract integer part FINTPvZ Extract integer part rounded to zero FLOGN Logarithm of x to the base e FLOGNP1 Logarithm of x + 1 to the base e FLOG 10 Logarithm to the base 10 FLOG2 Logarithm to the base 2 FNEG Negate FSIN Sine FSINCOS Simultaneous sine and cosine FSINH Hyperbolic sine FSQRT Square root FT AN Tangent FTANH Hyperbolic tangent FTENTOX 10 to the power of x FTWOTOX 2 to the power of x FLOGN Logarithm of x to the base e Program control FBcc Branch if condition code (status) cc is 1 FDBcc Test, decrement count, and branch on cc FNOP No operation FRESTORE Restore coprocessor state FSAVE Save coprocessor state FScc Set (cc = 1) or reset (cc = 0) a specified byte FTST Set coprocessor condition codes to specified values FTRAPcc Conditional trap Figure 4.46 Instruction set of the Motorola 68882 floating-point coprocessor. like the 68020 (section 3.1.2) with a large set of floating-point instructions. The 68882and the 68020 are physically coupled along the lines indicated by Figure 4.45. Whiledecoding the instructions it fetches during program execution, the 68020 identifiescoprocessor instructions by their distinctive opcodes. After identifying a coprocessorinstruction, the 68020 CPU "wakes up" the 68882 by sending it certain control signals.The 68020 then transmits the opcode to a predefined location in the 68882 that servesas an instruction register. The 68882 decodes the instruction and begins its execution,which can proceed in parallel with other instructions executed within the CPU proper.When the coprocessor needs to load or store operands, it asks the CPU to carry out thenecessary address calculations and data transfers. The 68882 employs the IEEE 754 floating-point number formats described inExample 3.4 with certain multiple-precision extensions; it also supports a decimalfloatingpoint format. From the programmer's perspective, the 68882 adds to the CPUa set of eight 80-bit floating-point data registers FP0:FP7 and several 32-bit controlregisters, including instruction (opcode) and status registers. Besides implementing awide range of arithmetic operations for floating-point numbers, the 68882 has instruc-tions for transferring data to and from its registers, and for branching on conditions itencounters during instruction execution. Figure 4.46 summarizes the 68882's instruc-tion set. These coprocessor instructions are distinguished by the prefix F (floating-point) in their mnemonic opcodes and are used in assembly-language programs justlike regular 680X0-series instructions; see Fig. 3.12. The status or condition codes ccgenerated by the 68882 when executing floating-point instructions include invalidoperation, overflow, underflow, division by zero, and inexact result. Coprocessor sta-tus is recorded in a control register, which can be read by the host CPU at the end of aset of calculations, enabling the CPU to initiate the appropriate exception-processingresponse. As some coprocessor instructions have fairly long (multicyle) executiontimes, the 68882 can be interrupted in the middle of instruction execution. Its statemust then be saved and subsequently restored to complete execution of the interruptedinstruction. 275 CHAPTER 4 Datapath Design The appearance of coprocessors stems in part from the fact that until the 1980sIC technology could not provide microprocessors of sufficient complexity toinclude on-chip floating-point units. Once such microprocessors became possible,arithmetic coprocessors began to migrate onto CPU chips, losing some of their sep-arate identity in the process —especially in the case of CISC processors. For exam-ple, the 1990-vintage Motorola 68040 microprocessor integrates a 68882-stylefloating-point coprocessor with a 68020-style CPU in a single microprocessor chip[Edenfield et al. 1990]. Arithmetic coprocessors provide an attractive way of aug-menting the performance of a RISC CPU without affecting the simplicity and effi-ciency of the CPU itself. The multiple function (execution) units in superscalarmicroprocessors like the Pentium resemble coprocessors in that each unit has aninstruction set that it can execute independently of the program control unit and theother execution units. 4.3.2 Pipeline Processing Pipelining is a general technique for increasing processor throughput withoutrequiring large amounts of extra hardware [Kogge 1981; Stone 1993]. It is appliedto the design of the complex datapath units such as multipliers and floating-point 276 SECTION 4.3Advanced Topics adders. It is also used to improve the overall throughput of an instruction set pro-cessor, a topic to which we return in Chapter 5. Introduction. A pipeline processor consists of a sequence of m data-processing cir-cuits, called stages or segments, which collectively perform a single operation on astream of data operands passing through them. Some processing takes place ineach stage, but a final result is obtained only after an operand set has passedthrough the entire pipeline. As illustrated in Figure 4.47, a stage 5, contains a multi-word input register or latch R:, and a datapath circuit C, that is usually combina-tional. The /?,-'s hold partially processed results as they move through the pipeline;they also serve as buffers that prevent neighboring stages from interfering with oneanother. A common clock signal causes the /v,'s to change state synchronously.Each Rj receives a new set of input data D,_, from the preceding stage 5,_! exceptfor R\ whose data is supplied from an external source. D,_, represents the resultscomputed by Ci_] during the preceding clock period. Once Dj_l has been loadedinto Rh Cj proceeds to use D,_, to compute a new data set Dt. Thus in each clockperiod, every stage transfers its previous results to the next stage and computes anew set of results. At first sight a pipeline seems a costly and slow way to implement the targetoperation. Its advantage is that an m-stage pipeline can simultaneously process upto m independent sets of data operands. These data sets move through the pipelinestage by stage so that when the pipeline is full, m separate operations are being exe-cuted concurrently, each in a different stage. Furthermore, a new, final resultemerges from the pipeline every clock cycle. Suppose that each stage of the m-stage pipeline takes T seconds to perform its local suboperation and store itsresults. Then 7" is the pipeline's clock period. The delay or latency of the pipeline,that is, the time to complete a single operation, is therefore mT. However, thethroughput of the pipeline, that is, the maximum number of operations completedper second is 1/7/. Equivalently, the number of clock cycles per instruction or CPIis one. When performing a long sequence of operations in the pipeline, its perfor-mance is determined by the delay (latency) T of a single stage, rather than by thedelay mT of the entire pipeline. Hence an m-stage pipeline provides a speedup fac-tor of m compared to a nonpipelined implementation of the same target operation. Control unit 1 r)ata R 'p i r ■r ii 'i C. R- c. "*..."* R C in Dataout —v V Stage S| Stage S2 VStage S„ Figure 4.47 Structure of a pipeline processor. Any operation that can be decomposed into a sequence of suboperations ofabout the same complexity can be realized by a pipeline processor. Consider, forexample, the addition of two normalized floating-point numbers x and y, a topicdiscussed in section 4.3.1. This operation can be implemented by the followingfour-step sequence: compare the exponents, align the mantissas (equalize the expo-nents), add the mantissas, and normalize the result. These operations require thefour-stage pipeline processor shown in Figure 4.48. Suppose that x has the normal-ized floating-point representation (xM,xE), where xM is the mantissa and xE is theexponent with respect to some base B = 2k. In the first step of adding x = (xMjcE) toy = (yM,yE), which is executed by stage S{ of the pipeline, xE and yE are compared,an operation performed by subtracting the exponents, which requires a fixed-pointadder (see Example 4.6). S{ identifies the smaller of the exponents, say, xE, whosemantissa xM can then be modified by shifting in the second stage S2 of the pipelineto form a new mantissa x'M that makes (x'M,yE) = (xM,xE). In the third stage themantissas x'M and yM, which are now properly aligned, are added. This fixed-pointaddition can produce an unnormalized result; hence a fourth and final step isneeded to normalize the result. Normalization is done by counting the number k ofleading zero digits of the mantissa (or leading ones in the negative case), shiftingthe mantissa k digit positions to normalize it, and making a corresponding adjust-ment in the exponent. Figure 4.49 illustrates the behavior of the adder pipeline when performing asequence of N floating-point additions of the form x(+ y, for the case N = 6. Addsequences of this type arise when adding two yV-component real (floating-point)vectors. At any time, any of the four stages can contain a pair of partially processedscalar operands denoted Qt,,y,) in the figure. The buffering of the stages ensures thatS, receives as inputs the results computed by stage 5,_, during the preceding clockperiod only. If Tis the pipeline's clock period, then it takes time 4T to compute thesingle sum x, + y,; in other words, the pipeline's delay is AT. This value is approxi-mately the time required to do one floating-point addition using a nonpipelinedprocessor plus the delay due to the buffer registers. Once all four stages of the pipe-line have been filled with data, a new sum emerges from the last stage S4 every Tseconds. Consequently, N consecutive additions can be done in time (N + 3)T,implying that the four-stage pipeline's speedup is 277 CHAPTER 4 Datapath Design S(4) = 4NN+3 x - (xM. xE) >' = (>'m.Ve) Data Exponent Ri Exponentadder *: Mantissashifter c2 R3 Mantissaadder c3 adder and R> mantissa shifter Q Stage 5,(Exponentcomparison) Stage S2 (Mantissaalignment) 'VStage S3(Mantissaaddition) Stage 54(Normalization) Figure 4.48 Four-stage floating-point adder pipeline. Dataout 278 SECTION 4.3Advanced Topics (*6- >6> (x5,y5) (x6,y6) (x4, y4) (x5, y5) (x6, y6) (xj.yj) (xA,y4) (x5,y5) (x6,y6) (x2,y2) (x3,y3) (x4,y4) (xs,y5) (x6,y6) (*i.yi) ~1~ (*2- -v2) (x3, y3) nrri (^i.yi) X (*2- >2) Ui.yi) Current result Time t * * * (*4- >'4) (*i. ^5) (*6- >'6> • ♦ * C*3- >'3> (*4- >'*) (*5- >5> ♦ ♦ * C*2. >2) (*3- >3) (*4- >4) ft (r,,y,) (x2. \;l '3> t t ♦ C*5. >s) 1 * * ♦ ♦ • ♦ • ♦ * • (*e- ye) * (*5->'5> 6) ♦ ♦ 1 (x4,y4) •*1+>1 -'2 x3 + yi x\ + y* *5 + >5 *6 + >'6 *i+yi ■«2+>'2 •»3+>,3 -«4+>'4 xs+y$ X|+y, X2 + >2 ^ + y3 Jt4 + >'4 *,+>', *2 + >'2 *3 + :y3 x, + y, x2 + >2 5 6 7 8 9 10 Stage5, *2 *3 54 Figure 4.49 Operation of the four-stage floating-point adder pipeline. For large N, 5(4) = 4 so that results are generated at a rate about four times that of acomparable nonpipelined adder. If it is not possible to supply the pipeline with dataat the maximum rate, then the performance can fall considerably, an issue to whichwe return in Chapter 5. Pipeline design. Designing a pipelined circuit for a function involves firstfinding a suitable multistage sequential algorithm to compute the given function.This algorithm's steps, which are implemented by the pipeline's stages, should bebalanced in the sense that they should all have roughly the same execution time.Fast buffer registers are placed between the stages to allow all necessary data items(partial or complete results) to be transferred from stage to stage without interfer-ing with one another. The buffers are designed to be clocked at the maximum ratethat allows data to be transferred reliably between stages. Figure 4.50 shows the register-level design of a floating-point adder pipelinebased on the nonpipelined design of Figure 4.44 and employing the four-stageorganization of Figure 4.48. The main change from the nonpipelined case is theinclusion of buffer registers to define and isolate the four stages. A further modifi-cation has been made to implement fixed-point as well as floating-point addition. i Exponent, comparison : Mantissa1 alignment Data J— 1 . . Mantissaaddition/subtraction 279 i Normalization ji _L _ ' 1 Si Ml M2 LHAF1EK 4Datapath i El E2 _ i II I Design Adder 1 1 1 , 1, u f II ji E4 E5 M4 s2 M5 1 T 1 1 Shifter 1 i "' "1 u \ 1 r t E6 11 --?" i v * M6 *3 ii M7 1 . iii Adder 2 ji ■ E7 w Adder3 Zero-d er ' check r 54 1 R i r Shifter 2 u 1 I E3 M3 _|_ out Figure 4.50 Pipelined version of the floating-point adder of Figure 4.44. The circuits that perform the mantissa addition in stage 53 and the correspondingbuffers are enlarged, as shown by broken lines in Figure 4.50. to accommodatefull-size fixed-point operands. To perform a fixed-point addition, the input oper-ands are routed through 53 only, bypassing the other three stages. Thus the circuitof Figure 4.50 is an example of a multifunction pipeline that can be configuredeither as a four-stage floating-point adder or as a one-stage fixed-point adder. Ofcourse, fixed-point and floating-point subtraction can also be performed by this cir-cuit; subtraction and addition are not usually regarded as distinct functions in thiscontext, however. 280 The same function can sometimes be partitioned into suboperations in several section 4 3 different ways, depending on such factors as the data representation, the style of Advanced Topics tne '°^c design, an(* the need to share stages with other functions in a multifunction pipeline. A floating-point adder can have as few as two stages and as many assix. For example, five-stage adders have been built in which the normalizationstage (54 in Figure 4.50) is split into two stages: one to count the number k of lead-ing zeros (or ones) in an unnormalized mantissa and a second stage to perform thek shifts that normalize the mantissa. Whether or not a particular function or set of functions F should be imple-mented by a pipelined or nonpipelined processor can be analyzed as follows. Sup-pose that F can be broken down into m independent sequential steps Fl,F2,.-.,Fm sothat it has an m-stage pipelined implementation Pm. Let F, be realizable by a logiccircuit C, with propagation delay (execution time) 7*,. Let TR be the delay of eachstage Sj due to its buffer register 7?, and associated control logic. The longest 7",times create bottlenecks in the pipeline and force the faster stages to wait, doing nouseful computation, until the slower stages become available. Hence the delaybetween the emergence of two results from Pm is the maximum value of 7",. Theminimum clock period (the pipeline period) Tc is defined by the equation Tc = max{r,} + TR for i = 1,2,...,m (4.44) The throughput of Pm is \ITC = l/(max{r,} + 7"R). A nonpipelined implementationPx of F has a delay of Z/=j Ti or, equivalently, a throughput of 1/(XJ=1 TX Weconclude the m-stage pipeline Pm has greater throughput than Px\ that is, pipeliningincreases performance if Equation (4.44) also implies that it is desirable for all 7", times to be approximatelythe same; that is, the pipeline stages should be balanced. Feedback. The usefulness of a pipeline processor can sometimes beenhanced by including feedback paths from the stage outputs to the primary inputsof the pipeline. Feedback enables the results computed by certain stages to be usedin subsequent calculations by the pipeline. We next illustrate this important con-cept by adding feedback to a four-stage floating-point adder pipeline like that ofFigure 4.50. example 4.8 summation by a pipeline processor . Consider the prob-lem of computing the sum of ./V floating-point numbers bl,b2,---,bN- It can be solved byadding consecutive pairs of numbers using an adder pipeline and storing the partialsums temporarily in external registers. The summation can be done much more effi-ciently by modifying the adder as shown in Figure 4.51. Here a feedback path has beenadded to the output of the final stage 54, allowing its results to be fed back to the firststage 5|. A register/? has also been connected to the output of S4, so that stage's resultscan be stored indefinitely before being fed back to 5,. The input operands of the modified pipeline are derived from four separate sources: a variable X that is typicallyobtained from a CPU register or a memory location; a constant source K that can applysuch operands as the all-0 and all-1 words; the output of stage S4, representing theresult computed by S4 in the preceding clock period; and, finally, an earlier result com-puted by the pipeline and stored in the output register R. Input tf 4 i V 1 \ 1 i r ' Multiplexer r* - -\ Multiplexer /- Input X r 1 I Stage 5, «--- -4 3 1 1 Stage 52 ♦ -- -4 ' r Stage S3 ■« -4 i ' Stage 54 «--- -4 1 Register /? r ^ Control Output Figure 4.51 Pipelined adder with feedback paths. The jV-number summation problem is solved by the pipeline of Figure 4.51 in thefollowing way. The external operands bx,b2,...,bN are entered into the pipeline in a continuous stream via input X. This process requires a sequence of register or memoryfetch operations, which are easily implemented if the operands are stored in contiguousregister/memory locations. While the first four numbers bx,b2,bi,bA are being entered,the all-0 word denoting the floating-point number zero is applied to the pipeline inputK, as illustrated in Figure 4.52 for times t = 1:4. After four clock periods, that is, at timet = 5, the first sum 0 + bx = 6, emerges from 54 and is fed back to the primary inputs ofthe pipeline. At this point the constant input K = 0 is replaced by the current result 54 =bv The pipeline now begins to compute bx + b$. At t = 6, it begins to compute b2 + b6;at / = 7, computation of fc3 + bn begins, and so on. When bx + b5 emerges from the pipe-line at t = 8, it is fed back to 5, to be added to the latest incoming number b9 to initiatecomputation of bt + b5 + b9. (This case does not apply to Figure 4.52, where b% = bs isthe last item to be summed.) In the next time period, the sum b2 + bb emerges from thepipeline and is fed back to be added to the incoming number bw. Thus at any time, thepipeline is engaged in computing in its four stages four partial sums of the form 281 CHAPTER 4 Datapath Design bi + b1 + bu+b^ + ... fc4 +&■ + &,, + fe.fi + ... (4.45) 282 SECTION 4.3Advanced Topics 0 b. i L J 1 (0.M ii » T (0, b-,) HZ (O.fc,) z / = 1 f = 2 f = 3 1 1 (0, i3) 11 (CAJ 1 (0,6,) 1 J L (0, bA) JL_ (0, b3) (0. fc,) HZ (0J>.i r = 4 / = 5 ' (*i.*s) 1 (0, 64) * (0. fej) I rI (0. fc2) ' r *1 1 I ib* b6) T (fci.65) ZIZ (0. fc4) HZ (0, fc3) r = 6 Figure 4.52 Summation of an eight-element vector. r = 7 ' r I (by b- 1 1 (fe2, fcfi) 1 (*i. *5) I (0. fc4) ' 1 &? (** *8) JL (*>3, *7> J_ HZ (fc,.fc5) „ + b3 + b7 emerges from 54 at t - 16. At this point theoutputs of 54 and R are fed back to Sv The final result is produced four time periodslater—at t = 20 in the case of N = 8. i I (0,0) 4 (b4, b%) * (&,. b,) 4 (*2, b6) ' f bl + b5 / = 9 0 0 1 1 (0,0) 1 (b4 + bs, b} + b-j) 4 (0.0) 4 (fc, + fc5 + b2 + fc6) i r 0 f= 13 Figure 4.52 (continued) (fc, + 65, b2 + b6) z (0.0) ziz ZEZ (by b-j) b2 + b6 t= 10 0 0 1 1 (0.0) 1 (0,0) ♦ (£>4+ b8, fc^ + b-,) 4 (0.0) ' b} + b5 + b2 + b6 f= 14 (0,0) 4 (bl + /?5. b-> + />6) 41 (0.0) r 1 (ft4, fcg) ' ' b^ + b-. t= 11 ! I (0,0) * (0,0) 4 (0,0) 4 (b4 + b% + b-< + b-,) ' r b^ + b5 + b2 + b6 f= 15 (b4 + bg, b3 + b7) (0.0) JL (fc, + fc5, b2 + b6 (0,0) *4 + *8 f= 12 ZJl. (bt + b5 + b2 + b(,b4 + bg + b3 + b7) 4 (0,0) HZ (0.0) ~4~ (0,0) 1ZZ b,+b< + bT + b(. t= 16 283 CHAPTER 4 Datapath Design It is easily seen that for the general case of TV operands, the scheme of Figure 4.52can compute the sum of N > 4 floating-point numbers in time (N + 11)7", where T is thepipeline's clock period, that is. the delay per stage. Since a comparable nonpipelinedadder requires time 4NT to compute SUM. we obtain a speedup here of about 4N/(N +11), which approaches 4 as N increases. The foregoing summation operation can be invoked by a single vector instruc-tion of a type that characterized the vector-processing, pipeline-based "supercom-puters" of the 1970s and 1980s [Stone 1993]. For instance. Control Data Corp.'sSTAR-100 computer [Hintz and Tate 1972] has an instruction SUM that computes 284 SECTION 4.3Advanced Topics the sum of the elements of a specified floating-point vector B = (bx, b2,...,bN) ofarbitrary length N and places the result in a CPU register. The starting (base)address of B, which corresponds to a block of main memory, the name C of theresult register, and the vector length N are all specified by operand fields of SUM.We can see from Figure 4.52 that a relatively complex pipeline control sequence isneeded to implement a vector instruction of this sort. This complexity contributessignificantly to both the size and cost of vector-oriented computers. Moreover, toachieve maximum speedup, the input data must be stored in a way that allows thevector elements to enter the pipeline at the maximum possible rate—generally onenumber-pair per clock cycle. The more complex arithmetic operations in CPU instruction sets, includingmost floating-point operations, can be implemented efficiently in pipelines. Fixed-point addition and subtraction are too simple to be partitioned into suboperationssuitable for pipelining. As we see next, fixed-point multiplication is well suited topipelined design. Pipelined multipliers. Consider the task of multiplying two n-bit fixed-pointbinary numbers X = xn_lxn_2.. .x0 and Y =)'„_1y„_2- • • v0. Combinational array multi-pliers of the kind described in section 4.1.2 are easily converted to pipelines by theaddition of buffer registers. Figure 4.53 shows a pipelined array multiplier thatemploys the 1-bit multiply-and-add cell M of Figure 4.19 and has n = 3. Each cellM computes a 1-bit product xy and adds it to both a product bit from the precedingstage and a carry bit generated by the cell on its right. Thus the n cells in each stageSit 0 < i < n - 1, compute a partial product of the form />,■ = />,-_,+jri2,r (4.46) y a x carryout M M X2 JT[X0 A i L Register/?, M =1 iRegister R2 M 4M ^iL ^^ *-3 Register /f3 M M r i M Ps Pa Pi Pi P\ Figure 4.53 Multiplier pipeline using ripple-carry propagation. Xt M Pq with the final product Pn_x = XY being computed by the last stage. In addition tostoring the partial products in the buffer registers denoted /?,, the multiplicand Yand all hitherto unused multiplier bits must also be stored in /?,-. An /i-stage multiplier pipeline of this type can overlap the computation of nseparate products, as required, for example, when multiplying fixed-point vectors,and can generate a new result every clock cycle. Its main disadvantage is the rela-tively slow speed of the carry-propagation logic in each stage. The number of Mcells needed is n2, and the capacity of all the buffer registers is approximately 3«2(see problem 4.31); hence this type of multiplier is also fairly costly in hardware.For these reasons, it is rarely used. Multipliers often employ a technique called carry-save addition, which is par-ticularly well suited to pipelining. An n-bit carry-save adder consists of n disjointfull adders. Its input is three /7-bit numbers to be added, while the output consists ofthe n sum bits forming a word 5 and the n carry bits forming a word C. Unlike theadders discussed so far, there is no carry propagation within the individual adders.The outputs 5 and C can be fed into another «-bit carry-save adder where, as shownin Figure 4.54, they can be added to a third n-bit number W. Observe that the carryconnections are shifted to the left to correspond to normal carry propagation. Ingeneral, m numbers can be added by a treelike network of carry-save adders to pro-duce a result in the form (5,C). To obtain the final sum, S and C must be added bya conventional adder with carry propagation. Multiplication can be performed using a multistage carry-save adder circuit ofthe type shown in Figure 4.55; this circuit is called a Wallace tree after its inven-tor [Wallace 1964]. The inputs to the adder tree are n terms of the form M, =xiY2k. Here M, represents the multiplicand Y multiplied by the /th multiplier bitweighted by the appropriate power of 2. Suppose that Mi is In bits long and thata full double-length product is required. The desired product P is Z£Tq M;. Thissum is computed by the carry-save adder tree, which produces a 2«-bit sum and a 285 CHAPTER 4 Datapath Design *3 x2 Xd y3 ' z 1' 3 y2 ' z 2 1 Y\ z i I >'o z x y z II i i 1 CS adder w3 / w2 / wl / "0 w / 1 / i C S T V T " r T r / 1 1 T T T CS adder r ▼ « / / i » C 5' $3 $2 *1 -^O c'-i ci c\ Figure 4.54 At WO -sta ge ca irn, -Si ive i iddei 286 SECTION 4.3 I 1. Advanced Topics Multiplier decodingand multiplicand gating \Y\, then set v„_, to 0 and subtract Yfrom X (modulo 2"). 3. X negative; Ypositive: If in < LX1, subtract Y from X (modulo 2"). If in > 1X1, set.r„_, to 0 andsubtract X from Y (modulo 2"). 4. X and Y both negative: Add X and Y (modulo 2") and set z„_, to 1. Figure 4.61 Algorithm for subtracting sign-magnitude numbers. 295 CHAPTER 4 Datapath Design 1 n-bit adder-subtracter "} "} SUB Figure 4.62 An n-bit adder-subtracter circuit. lines appearing in the figure can be used to compute v. Construct a suitable logic circuitfor v. 4.6. Consider again the adder-subtracter of Figure 4.62, assuming now that it has beendesigned for sign-magnitude numbers. It computes Z = X + Y when SUB = 0 and Z =X- Y when SUB = 1. Assume that the circuit contains an n-bit ripple-carry adder anda similar «-bit ripple-borrow subtracter and that you have access to all internal lines.Derive a logic equation that defines an overflow flag v for this circuit. 4.7. Give an informal interpretation and proof of correctness of the two expressions (4.12)for/? and g that define the propagate and generate conditions, respectively, for a 4-bitcarry-lookahead generator. 4.8. Show how to extend the 16-bit design of Figure 4.8 to a 64-bit adder using the sametwo component types: a 4-bit adder module and a 4-bit carry-lookahead generator. 4.9. Stating your assumptions and showing your calculations, obtain an good estimate foreach of the following for both an n-bit carry-lookahead adder and an n-bit ripplecarryadder: (a) the total number of gates used; (b) the circuit depth (number of levels): and(c) the maximum gate fan-in. 4.10. Another useful technique for fast binary addition is the conditional-sum method. Itand a closely related method called carry-select addition are based on the idea of si-multaneously generating two versions of each sum bit s\: a version s], which assumesthat its input carry c,_, = 1. and a second version s?, which assumes that ct_{ = 0. Amultiplexer controlled by