Computer Architecture and Organization

John P. Hayes

Computer architecture and organisation

Computer architecture and organisation

Hayes, John P. (John Patrick), 1944-
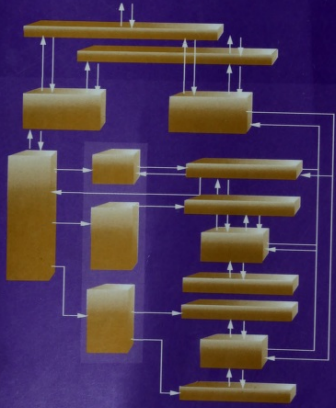
Computer Architecture andOrganization

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

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 compre-hensive 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

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 llllllllllllllllllllllllllllllllllllllllllllllllllllllll

lll|llll|llll|lltl|llll|IIIip

1 4 5 6 7Li.l.l.l.l.l.llNllllMl .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 rillllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllliillllllll7 J.,..l....i..,.,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,<.I.I.I.U.l.l.l.i.i.l.l.i.l I i~....i....i....i....i....f.,,.i....i....i....i....i.

R

rr|·„·.|„„|,„,M,.·,|„„r,r,r,rr„|,|,l,|,| |,|.|,|,,,|,|,|.|,|,|,|1|,|,|,|r|T|.|,|T|T|,,,,|,,,,|..,.|....|....|....r.T,..rr|.,.,!„„!

9 I A 2 A3 4 S 6 7 8 9 1

0 • 2 3 4 ■ s b 1 8 ' °

l,l,l.l,l,l.l.l.1.l.l.l.l.l.B.I.I.Ll,l,l,l,l.l.1.l.l,i,lj.l.l.l.l.l.l.l.l,1.l.l.l,l.l.l.l.l,l.l,lg . „l,l,l,l,', ,'.,!,1,1,1,1.1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1

" .""i""i""T"'i""r"i""i'ri'ri,n,i'iriln

4» ♦ 5

,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,!,!,!,1,1,1,1,1.1.1.Ill III

A = 1.30

C = 2.99

(b)

Figure 1.1

(a) Japanese abacus (soroban) displaying the number 0011234567890;

(b) slide rule illustrating the multiplication 1.30 x 2.30 = 2.99.

numbers.1 Slide rules are marked with various other scales that allow ay experi-enced user to evaluate complicated expressions such as 2.15 X 17.9_50sin7t in sev-eral steps.

As the size and complexity of the calculations being carried out increases, twoserious limitations of manual computation become apparent.

• The speed at which a human computer can work is limited. A typical elementaryoperation such as addition or multiplication takes several seconds or minutes.Problems requiring billions of such operations could never be solved manually ina reasonable period of time or at reasonable cost. Fortunately, modern computersroutinely tackle and quickly solve such problems.

• Humans are notoriously prone to error, so long calculations done by hand areunreliable unless elaborate precautions are taken to eliminate mistakes. Mostsources of human error (distraction, fatigue, and the like) do not affect machines,so they can provide results that are, within broad limits, free from error.

The English computer pioneer Charles Babbage (1792-1871) often cited thefollowing example to justify construction of his first automatic computing

'Logarithms are defined by the relation $10° = A$, where $a = log]0A$. A length marked A on a log scale isproportional to $log10i4 = a$. When we add two lengths marked A and Bona slide rule, we are actually add-ing $a = \ogl0A$ and $b = log10B$. Therefore, the result c represents $log10A + log,0B$. Now $10° x 10* = 10***$implies $c = log10 A + log 10 B = log 10 (A x B)$, so if we read c from the first scale, we will obtain the numberwhose log is c, that is. Ax B.

machine, the Difference Engine [Morrison and Morrison 1961]. In 1794 the Frenchgovernment began a project to compute entirely by hand an enormous set of math-ematical tables. Among the many required tables were the logs of the numbersfrom 1 to 200,000 calculated to 19 decimal places. The entire project took twoyears to complete and employed about 100 people. The mathematical abilities ofmost of these human computers were limited to addition and subtraction, and theyperformed their calculations using pen and paper. A few skilled mathematiciansprovided the instructions. To minimize errors, each number was calculated inde-pendently by two human calculators. The final set of tables occupied 17 large vol-umes. The log table alone contained about 8 million digits.

CHAPTERlComputing andComputers

1.1.1 The Elements of Computers

Every computer, human or artificial, contains the following components: a proces-sor able to interpret and execute programs; a memory for storing the programs andthe data they process; and input-output equipment for transferring informationbetween the computer and the outside world.

The brain versus the computer. Consider the actions involved in a manual cal-culation using pencil and paper—for example, filling out an income tax return. Thepurpose of the paper is information storage. The information stored can include alist of instructions—more formally called a program, algorithm, ox procedure—tobe followed in carrying out the calculation, as well as the numbers or data to beused. During the calculation intermediate results and ultimately the final results arerecorded on the paper. The data processing takes place in the human brain, whichserves as the {central} processor. The brain performs two distinct functions: a con-trol function that interprets the instructions and ensures that they are performed inthe proper sequence and an execution function that performs specific steps such asaddition, subtraction, multiplication, and division. A pocket calculator often servesas an aid to the brain. Figure 1.2a illustrates this view of human computation.

A computer has several key components that roughly correspond to those justmentioned; see Figure \.2b. The main memory corresponds to the paper used in themanual calculation. Its purpose is to store instructions and data. The computer'sbrain is its central processing unit (CPU). It contains a program control unit (alsoknown as an instruction unit) whose function is to fetch instructions from memoryand interpret them. An arithmetic-logic unit (ALU), which is part of the CPU'sdata-processing or execution unit, carries out the instructions. The ALU is so calledbecause many instructions specify either arithmetic (numerical) operations or vari-ous forms of nonnumerical operations that loosely correspond to logical reasoningor decision making.

There are important similarities and differences between human beings andartificial computers in the way in which they represent information. In both casesinformation is usually in digital or discrete form. This is contrasted with analog orcontinuous information as used, for example, in the slide rule of Figure \.\b. Dis-tance is a continuous quantity, and on a slide-rule scale it represents, or sen es as ananalog for, a continuous sequence of numbers. The problem is that such analogquantities have very limited accuracy. The numbers on a slide rule, for example,

SECTION 1.1The Nature ofComputing

(«)

Centralprocessing   Instructions

unit   Mainmemory   Input-outputequipment

Programcontrol

'   r

Arithmetic-logic unit

Data
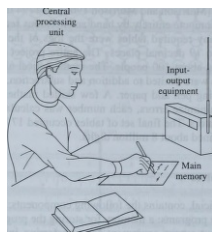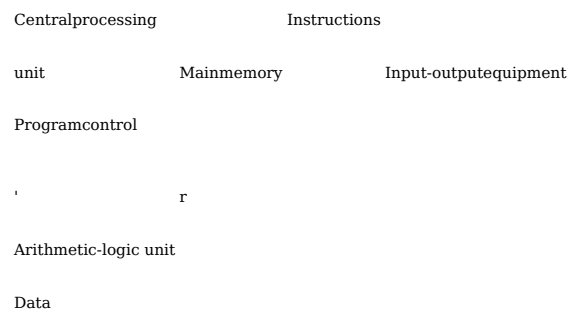
(b)

Figure 1.2

Main components of (a) human computation and (b) machine computation.

cannot be read to more than three decimal places. On the other hand, a digitaldevice can easily handle a large number of digits. Even the simple abacus of Figure1.1a can display a number—admittedly just one—to 13 places of accuracy. Thisadvantage of digital data representation over analog is also seen in the higher fidel-ity of the sound recorded on a compact disc (CD), a digital device, compared to anold-fashioned record (LP), which is an analog device.

Humans employ languages with a wide range of digital symbols, and they usu-ally represent numbers in decimal (base 10) form. It is not practical to build com-puters to handle symbolic or decimal data directly. Instead, computers process datain binary form, that is, using the two symbols 0 and 1 called bits (binary digits).Computers are built from electronic switches that have two natural states: off (0)and on (1). Hence the internal "language" of computers comprises forbidding-looking strings of bits such as 10010011 11011001. To provide communication
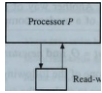
Read-write head

I

\

Memory tapeM

Figure 1.3

A Turing machine.

between a computer and its human users, a means of translating informationbetween human and machine (binary) formats is necessary. The input-outputequipment shown in Figure 1.2b performs this task.

An abstract computer. We are interested in the computational abilities of gen-eral-purpose digital computers. One might raise the following question at the out-set: Are there any computations that a "reasonable" computer can never perform?Three notions of reasonableness are widely accepted.

• The computer should not store the answers to all possible problems.

• The computer should only be required to solve problems for which a solutionprocedure or program can be given.

• The computer should process information at a finite speed.

A reasonable computer can therefore solve a particular problem only if it is sup-plied with a program that can generate the answer in a finite amount of time.

In the 1930s the English mathematician Alan M. Turing (1912-54) introducedan abstract model of a computer that satisfies all the foregoing criteria [Barwiseand Etchemendy 1993]. This model, now called a Turing machine, has the struc-ture shown in Figure 1.3. As we noted earlier two essential elements of any com-puter are a memory and a processor. The memory of a Turing machine is a tape Mwhich resembles that of a tape recorder. Unlike the tape recorder, however, theTuring machine's tape is of unbounded length and is divided lengthwise intosquares. Each square can be blank, or it can contain one of a small set of symbols.The Turing machine's processor P is a simple device with a small number of inter-nal configurations or states. It is linked to M by a read-write head that can read thecontents of one square Q and write a new symbol into Q to replace the old one in asingle time step. Instead of writing on the tape, the processor can also just read thecurrent symbol and move the tape one square to the left or right of the currentsquare Q.

We can view the Turing machine as having a set of instructions that we willwrite in the compact, four-part format

Sh Tt Oj Sk

This instruction is interpreted in the following way: If the present state of the pro-cessor P is Sh and the symbol it reads on the square of M under the read-writehead is T,, then perform the action (such as write a new symbol or move the tape)

specified by Oy and change the state of P to Sk. Another way of expressing thisinstruction, which is more in tune with the style of a modern computer program-ming language, is

if oldstate = Sh and input - Tt then output = 0} and newstate = Sk;

The output operation indicated by 0} can be any one of the following:

1. Oj= Tp meaning write the symbol T on the tape to replace the symbol Tt.

2. Oj = R, meaning move the tape so that the read-write head is over the square tothe right of the current square. (The tape is moved one square to the left.)

3. Oj = L, meaning move the tape so that the read-write head is over the square tothe left of the current square. (The tape is moved one square to the right.)

4. Oj = H, meaning halt the computation.

The foregoing apparently restricted form of instruction, with just a few differ-ent symbols to write on M and a few different states for P, turns out to be sufficientto define programs that can perform all reasonable computations. To determine thevalue of Z = F(X) via a Turing machine, where F is some function of interest, weproceed as follows: The input data X is placed in a suitably coded form on an other-wise blank tape M. The processor P is supplied with a program that specifies asequence of steps that are designed to compute F. The Turing machine is thenstarted and executes instruction after instruction, moving the tape M and writingintermediate results on it. Eventually, the Turing machine should halt, and the finalresult Z should be found on the tape.

EXAMPLE 1.1 A TURING MACHINE TO ADD TWO UNARY NUMBERS. Any

natural number n, that is, a positive integer selected from the set we usually write as 0,1, 2, 3, 4, 5,..., can be written in the unary form consisting of a sequence of n Is. Forexample, 5 can be written as 11111 and 13 as 1111111111111. When we record num-bers using tally or check marks only, we are using a unary notation. (Surprisingly,unary numbers still have a small place in computer design [Poppelbaum et al. 1985].)

We will now show how to program a Turing machine to compute the sum of twounary numbers n] and n2. The tape symbols needed are 1 and b, where b denotes ablank. We start with a blank tape (one containing b in every square) and write the twoinput numbers in the following format:

...bbbl 1 ... lbl 1 ... lbbb...

We position the read-write head over the blank square (underlined above) to the left ofthe left-most 1. Our Turing machine then computes nx + n2 by the simple expedient offinding the single blank that separates «, and n2 and replacing it with 1. The machinethen finds and deletes the left-most 1 of «,. The resulting pattern of Is and bs

...bbbbl 1 ... 1 1 1 1 1 ... lbbb.

n, + Hi

appearing on the tape is the required answer in the same unary format as the input data.The behavior of a seven-instruction Turing machine that implements this procedure isgiven with explanatory comments in Figure 1.4. Observe that although the tape M canhave an arbitrarily large number of states, the processor P has only the four states S0,5,, S2, and 53.

Instruction

Comment

SB

b R 5, Move read-write head one square to right.1 R Sj Move read-write head rightward across nx.b 1 S2 Replace blank between «, and n2 by 1.

L S2 Move read-write head leftward across «,.

R 53 Blank square reached; move one square to right.

b S3 Replace left-most 1 by blank.

H 53 Halt; the result nx + n2 is now on the tape.

Figure 1.4

Turing machine program to add two unary numbers.

One of Turing's most remarkable achievements was to prove that a universalTuring machine (not unlike the above unary adding machine) can by itself performevery reasonable computation. A universal Turing machine is essentially a simula-tor of Turing machines. If given a description of some particular Turing machineTM—a program description like that of Figure 1.4 will do—the universal machinesimulates all the operations performed by TM. A universal Turing machine needsonly t different tape symbols and s different processor states, where ts < 30, imply-ing that it can have a very small instruction set. Nevertheless, such a machine canperform any reasonable computation. It can therefore do anything that any real com-puter can do and so serves as an abstract model of the modern general-purpose com-puter. The universal Turing machine also captures a little of the flavor of reducedinstruction set computers (RISCs), which, despite having relatively few instructiontypes, are among the most powerful computing machines available today.

CHAPTER lComputing andComputers

1.1.2 Limitations of Computers

We turn next to the question of what problems computers can and cannot solve,either in principle or in practice [Barwise and Etchemendy 1993; Cormen and Leis-erson 1990; Garey and Johnson 1979].

Unsolvable problems. Problems exist that no Turing machine and therefore nopractical computer can solve. There are well-defined problems, some quite famous,for which no solutions or solution procedures are known. An example from puremathematics is Goldbach's conjecture, formulated by the mathematician ChristianGoldbach (1690-1764), winch states that every even integer greater than 2 is thesum of exactly two prime numbers. For instance, 8 = 3 + 5 and 108 = 37 + 71. Gold-bach's conjecture has been tested for an enormous number of even integers and istrue in all test cases. Nevertheless, it is not yet known if the conjecture is true forevery even integer, nor is any reasonable procedure known to determine whether theconjecture is true. The number of even integers is infinite, so a complete or exhaus-tive examination of all even integers and their prime factors is not feasible. „

Goldbach's conjecture is an example of an unsolved problem that may eventu-ally be solved—we just don't have a suitable solution procedure yet. Turing

8

machines have proven another class of problems to be unsolvable, so there is nohope of ever solving them; such problems are said to be undecidable. An exampleof an undecidable problem is to determine if an arbitrary polynomial equation of theform

a0 + axx + a^x2 + •" + an_xx"~x + a^ = b

has a solution consisting entirely of integers. This problem may be answerable forspecific equations, but a general procedure or program can never be constructed thatcan analyze any possible polynomial equation and decide if it has an integer solution.

Turing identified an undecidable problem that involves the basic nature of Tur-ing machines. Does a procedure exist to determine if an arbitrary Turing machinewith arbitrary input data will ever halt once it has been set in motion? Turingproved that the answer is no, so the Turing machine halting problem as this partic-ular problem is called, is also undecidable. This result has some practical implica-tions. A common and costly error made by inexperienced computer programmersis to write programs that contain infinite loops and therefore fail to halt under cer-tain input conditions. It would be useful to have a debugging program that coulddetermine whether any given program contains an infinite loop. The undecidabilityof the Turing machine halting problem implies that no such infinite-loop-detectingtool can ever be realized.

The Turing machine model of a computer has one unrealistic, if not unreason-able, aspect: The length of the tape memory, and hence the total number of states inthe Turing machine, is infinite. Real computers have a finite amount of memoryand are therefore referred to as finite-state machines. Therefore, Turing machinescan perform some computations that, in principle, finite-state machines cannot per-form. For example, a finite-state machine cannot multiply two arbitrarily largenumbers because it eventually runs out of the states needed to compute the product.The number of states of a typical computer is enormous, so this finiteness limita-tion has little significance. A typical general-purpose computer has billions ofstates and can quickly multiply numbers of any practical length.

Intractable problems. Real (finite-state) computers can solve most computa-tional problems to an acceptable degree of accuracy. The question then becomes:Can a computer of reasonable size and cost solve a given problem in a reasonableamount of time? If so, the problem is said to be tractable; otherwise, it is intracta-ble. Whether a given problem is tractable depends on several factors: the nature ofthe problem itself, the solution method or program used, and the computing speedor performance of the computer available to solve it. Figure 1.5 gives an indicationof the speed of modern computers. It shows how the number of basic operations,such as the addition of two numbers, that a CPU can perform has been evolvingwith advances in computer hardware.

Example 1.2 illustrates the impact of the solution method on problem diffi-culty.

EXAMPLE 1.2 FINDING AN EULER CIRCUIT IN A GRAPH. A well-known

problem associated with the Swiss mathematician Leonhard Euler (1707-1783) is thefollowing: Given a set of connected paths such as the aisles in an exhibition hall (Fig-ure 1.6a), is it possible to make a tour of the hall so that one walks along every aisleexactly once and ends up at the starting point? The problem can be representedabstractly by means of a graph, as shown in Figure \.6b. Each aisle is modeled by a

Component technology	Date	Number of basicoperations per second
Electromechanical: relays	1940	10 3
Electronic: vacuum tubes (valves)	1945	10 4
Electronic: transistors	1950	10
Small-scale integrated circuits	1960	10
Medium-scale integrated circuits	1980	10
Very large-scale integrated circuits	2000	10

Figure 1.5

Influence of hardware technology on computing speed.

CHAPTER 1Computing andComputers

line called an edge, and the junction of two or more aisles by a point called a node.The graph of Figure \.6b has five nodes A, B, C, D, and E and eight edges a, b, c, d, e,f, g, and h. Restated in graph terms, the walking-tour problem becomes that of findinga closed path around the graph that contains every edge exactly once; such a path isknown as an Euler circuit. We consider two possible ways to determine whether agraph contains an Euler circuit.

A "brute force" or exhaustive approach is to generate a list of the possible order-ings ox permutations of the edges of the graph. Each permutation then corresponds toapotential tour of the exhibition hall. The list of permutations can be written in the form

abcdefgh, acbdefgh, adbcefgh, aebcdfgh, afbcdegh,

(1.1)

We can search the permutation list and check each entry to see if it specifies an Euler circuit. Clearly, the list is huge, and most of its entries do not represent Euler circuits. For example, the first permutation abcdefgh does not represent an Euler circuit, because while it is possible to go from a to b and from b to c, it is not possible to go directly from c to d. A tour starting at node A that traverses a, b, and c must continue along g, at which point/or h may be followed. The permutation abcgfdhe appearing
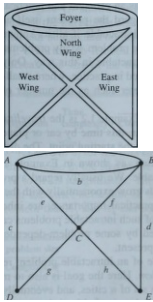
(a)

(b)

Figure 1.6

{a) Plan of the aisles in an exhibition hall and (b) the corresponding graph model.

SECTION 1.1 The Nature of Computing

somewhere down the list represents a circuit of the desired kind, as can be quickly ver-ified. Thus we conclude that the graph of Figure 1.6b does indeed contain an Euler cir-cuit.

The main drawback of this brute-force method is the length the permutation list; the time needed to generate, store, and check it is enqrmous. Most of the list's entries do not represent Euler circuits, but in the worst case, we might have to search the entire list to find an Euler circuit or prove that none exists. The number of possible permuta-tions of the eight edges in our example is 8!, which denotes eight factorial. Therefore

$$8! = 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 40,320$$

is the length of list (1.1). When q, the number of edges present, is large, the size of the permutation list q\ is approximated by

which shows that the size of the brute-force procedure in terms of storage requirements and computing speed increases exponentially with q. If q were 80 instead of 8, then we would have q\ = 80! = 7.16 x 10118. This huge number exceeds the estimated number (1010) of neurons in the human brain. A very fast computer capable of processing a tril-lion (1012) permutations per second would spend 2.27 x 10" years dealing with 80! permutations. We can therefore conclude with some confidence that the problem of finding an Euler circuit is intractable via the brute-force approach.

An alternative but very tractable solution procedure for the same problem depends on Euler's discovery that a graph has the desired circuit if and only if every node is the junction of an even number of edges. Intuitively, this result follows from the fact that every edge used to enter a node must be paired with an edge used to exit the node. Now the task of determining whether a graph contains an Euler cycle reduces to checking each node in turn and counting the edges that it connects. In the example of Fig-ure \.6b, nodes A, B, C, D, and E form the junctions of 4, 4, 4, 2, and 2 edges, respec-tively. It follows immediately that the graph has an Euler circuit. While the brute-force method requires a computation time and a storage capacity that grow exponentially with the number of edges q. the second method has a computational complexity that is proportional to q. The second method can easily solve problems with 80 or more edges.

Because the problem of finding an Euler circuit has an efficient and practical solution procedure, as shown in Example 1.2, we regard the problem itself as inherently tractable. We usually regard a problem as intractable if all its known solution methods grow exponentially with the size of the problem. Many problems, some of great practical importance, are inherently intractable in this way. Only small versions of such intractable problems can be solved in practice, where small-ness is measured by some problem-dependent parameter such as the number of input variables present.

An example of an intractable problem related to Example 1.2 is the Traveling salesman problem. Here the goal is also to make a tour, this time by car or plane through a given set of n cities, and eventually return to the starting point. The dis-tance between each pair of cities is known, and the problem is to determine a tour that minimizes the total distance traveled. Again it is convenient to use a graph model with nodes denoting cities and edges denoting intercity highways with dis-tances marked on them—the graph is tantamount to a roadmap. The best solution procedures known for this problem, although better than the brute-force approach

of listing all possible tours through the n cities, are exponential in n. Quite a few practical problems are closely related to the traveling salesman problem: The scheduling of airline flights, the routing of wires in an electronic circuit, and the sequencing of steps in a factory assembly line are examples. Such difficult comput-ing problems are a major motivation for the design and construction of bigger and faster computers.

An intractable problem can be solved exactly in a reasonable amount of time only when its size n is below some maximum value nMAX. The value of «MAX depends both on the problem itself and on the speed of the computers available to solve it. It might be expected that computer speeds could be increased to make "max any desired value. We now present arguments to indicate that this is highly unlikely.

CHAPTER 1 Computing and Computers

Speed limitations. An algorithm A has time complexity of order /(«), denoted 0(f(n)), if the number of basic operations—the precise nature of these operations is not important—A uses to solve a problem of size n is at most cj\n), where j{n) is some function of n and c is a constant. The function j\n) therefore indicates the rate at which the computing time that A needs to obtain a solution grows with the prob-lem size n.

To gauge the impact of computing speed on the size nMAX of the largest solv-able problem, we consider four algorithms Ax, A2, A3, and A4 of varying degrees of difficulty. Let the time complexities of Ax, A2, A3, and A4 be 0(n), 0(n2), 0(nm), and 0(2"), respectively. Because A4 has a time complexity that is exponential in n, it is the only obviously intractable procedure. Suppose that all four algorithms are programmed on a computer M having a speed of S basic operations per second. Let n{ denote the size of the largest problem that algorithm A, can solve in a fixed time period of T seconds. Let n{ denote the size of the largest problem that the same algorithm A, can solve in T seconds on a new computer M' that is 100 times faster than M; the speed of M' is therefore 100S operations per second. M' could be implemented by a different and faster hardware technology than M. It could also—at least in principle—be implemented by a "supercomputer" consisting of 100 cop-ies of M all working in parallel on the same problem, a technique referred to as parallel processing.

Figure 1.7 shows the values of n{ relative to ni for the four algorithms. In the case of the intractable algorithm A4, the increase in the size of the largest problem that can be handled on moving from M to M' is insignificant. This is also true for A3, even though it does not fall within the strict definition of intractability. To increase the size of the maximum problem that A} and A4 can solve in the given

Time

Maximum problem size

Algorithm complexity Computer M Computer M'

O(n) "i

0(n2) "2

0(nm) "3

0(2") "4

B,'a 100/1,

n2' = 10n2Bj'= 1.047n,n4 = w4 + 6.644

Figure 1.7

Effect of computer speedupby 100 on four algorithms.

time period by a factor of 100, we would need computers with speeds of 10 5 and10 °"4S, respectively. It is reasonable to expect that problems of these magnitudescan never be solved by the given algorithms on realistic computers.

Because so many important problems are intractable, we often devise approxi-mate or inexact methods to solve them. Two major techniques follow.

1. We replace the intractable problem Q with a tractable problem Q' whose solu-tion approximates that of Q.

2. We examine a relatively small set of possible solutions to Q using reasonable,intuitive, and often poorly understood selection criteria and take the "best" ofthese as the solution to Q. Methods that are designed to produce acceptable, ifnot optimal, answers using a reasonable amount of computing time are some-times called heuristic procedures.

To illustrate the heuristic approach, consider again the traveling salesman prob-lem. The salesman must visit n cities and return to his starting point. All intercitydistances are specified, and the objective of the problem is to find a tour that mini-mizes the total distance traveled by the salesman. We can represent the problem ona graph similar to that of Figure 1.6b, whose nodes denote cities and whose edgesdenote intercity links. A brute-force approach of the kind discussed in Example 1.2,which involves listing all n\ possible tours and their distances, is intractable, and noobviously tractable method to obtain a minimum-distance tour is known.

Real traveling salesmen often use the following simple heuristic: Qo to the pre-viously unvisited city that is closest to the current city and return to the start in thefinal leg of the tour. Hence for each of the n legs, the only computation needed is tocompare the distances between the current city and each of at most n - 1 other cities.The city that is the shortest distance away (if there are several such cities, select anyone of them) is visited next. Because this heuristic makes decisions that are optimalon a local basis only, it will not always find an overall optimum. Nevertheless, formost practical problems this heuristic provides a solution of minimum or near-min-imum length, but there is no guarantee that it will do so in any particular case.

Computers are continually being applied to new problems whose computa-tional requirements far exceed those of older problems. For example, the process-ing of high-quality speech and visual images for multimedia applications canrequire speeds measured in trillions of basic operations per second.-To meet theever-increasing demand for high-performance computation, we need better algo-rithms and heuristics, as well as faster computers. Although computers continue toincrease in speed because of advances in hardware technology, the rate of increase(see Figure 1.5) has not kept pace with demand. As a result, we still need to findnew ways to improve the performance of computers at reasonable cost—which isthe basic rationale for the study of computer architecture and organization.

1.2

THE EVOLUTION OF COMPUTERS

Calculating machines capable of performing the elementary operations of arith-metic (addition, subtraction, multiplication, and division) appeared in the 16thcentury, and perhaps earlier [Randell 1982; Augarten 1984]. These were clever

mechanical devices constructed from gears, levers, and the like. The French philos-opher Blaise Pascal (1623-62) invented an early and influential mechanical calcu-lator that could add and subtract decimal numbers. Decimal numerals wereengraved on counter wheels much like those in a car's odometer. Pascal's maintechnical innovation was a ratchet device for automatically transferring a carryfrom a digit di to the digit di+l on its left whenever dt passed from 9 to 0. In Ger-many, Gottfried Leibniz (1646-1716) extended Pascal's design to one that couldalso perform multiplication and division. Mechanical computing devices such asthese remained academic curiosities until the 19th century, when the commercialproduction of mechanical four-function calculators began.

1.2.1 The Mechanical Era

Various attempts were made to build general-purpose programmable computersfrom the same mechanical devices used in calculators. This technology posed somedaunting problems, and they were not satisfactorily solved until the introduction ofelectronic computing techniques in the mid-20th century.

Babbage's Difference Engine. In the 19th century Charles Babbage designedthe first computers to perform multistep operations automatically, that is, without ahuman intervening in every step [Morrison and Morrison 1961]. Again the technol-ogies were entirely mechanical. Babbage"s first computing machine, which hecalled the Difference Engine, was intended to compute and print mathematicaltables automatically, thereby avoiding the many errors occurring in tables that arecomputed and typeset by hand. The Difference Engine performed only one arith-metic operation: addition. However, the method of (finite) differences embodied inthe Difference Engine can calculate many complex and useful functions by meansof addition alone.

EXAMPLE 1.3 COMPUTING X2 BY THE METHOD OF DIFFERENCES. Con-sider the task of calculating a table of the squares y- = xK for Xj = 1,2,3,... using themethod of differences. To understand the underlying concept, suppose we alreadyhave the list of squares given in Figure 1.8<a. Subtract each square y, = xf from thenext value y;+1 = (xj + l)2 in the list. The result (x- + l)2 -xf = 2Xj + 1 is called the firstdifference of y and is denoted by A1 y ; die corresponding list of values in Figure 1.8ais 3, 5, 7, ... If we subtract two consecutive first-difference values, we obtain 2(xj + 1)+ 1 - (2xj + 1) = 2, which is the second difference A.2y; of y. Note that die second dif-ference is constant for ally.

The Difference Engine evaluates xr by taking the constant second difference A~and adding it to the first difference A1 y-. The result is

aVj+i = aVj+a2^

(1.2)

which is the next value of the first difference. At the same time, the engine calculates

r;+i

= v, + A

(1.3)

which is the next value of x2. By repeatedly executing the two addition steps (1.2) and(1.3), the Difference Engine can generate any desired sequence of consecutivesquares. It must be "primed" by manually inserting the initial values y, = 1. A y, = 3,

yj = xj: 1 4 9 16 25 36

Initial values

y,= i—«C

y,- register

+ ~1 Adder

A'y, = 3

] A1 ^register

First difference A1 yy. 3 5 7 9 11 13

+ I Adder

Second difference A2y: 2 2 2 2 2 2 A2y, = 2—M] A2y; register

(a)

(b)

Figure 1.8

Computing jt by the method of differences: (a) a representative computation and(b) the corresponding Difference Engine configuration.

2

and A y, = 2 for j; = 1, which appear at the left end of the corresponding lists in Fig-ure 1.8a. Then the Difference Engine computes A y2 =3 + 2 = 5 according to (1.2)and y2 = 1 + 3 =4 according to (1.3). It never has to recompute A~y , which remainsunchanged at 2 for all j. Once the values for j = 2 are known, the Difference Enginecan calculate A y3 and y3, and so on indefinitely. At the end of the computation illus-trated in Figure 1.8a, we have y6 = 36, A y6 = 13, and A y6 = 2. One more iterationyields A y1 = 13 + 2= 15 and y7 = 36 + 13 = 49, which is, of course, 72.

Figure 1.8& outlines the essential features of a small Difference Engine that exe-cutes the foregoing procedure. It contains several registers; these are memory devices,each of which stores a single number. Here we need three registers to store the threenumbers y-, A y •, and A y . The engine employs a pair of processing units calledadders to perform the addition steps specified by (1.2) and (1.3). Each adder takes thecontents of two registers, calculates their sum, and returns it to one of the registers sothat the sum becomes that register's new contents. The arrows in Figure l.Sb indicatethe manner in which information flows through the Difference Engine during operation.

We can easily show that the «th difference of ;c" is always a constant, fromwhich it follows the nth difference of any mh-order polynomial of the form

y(x) = a0 + axx + a-jjc2 + ••• + a^x"'1 + aj? (1.4)

is also a constant K. A Difference Engine can therefore calculate y(x) by evaluatinga set of n difference equations of the form

A ' A ' A ' + 1

A)) = Ay;_, + A yj_j

where 0 < i < n — 1, A0y = y.„ and A"y = K. Many useful functions encountered inscience and engineering are expressible as polynomials like (1.4) and therefore canbe evaluated by the method of differences. The trigonometric sine function, forinstance, can be written as

3 5 7 9 11

sinx = x-^ + X--X- + X--X— +

(1.5)

The first k terms of (1.5) form a (2k - l)th-order polynomial that approximatessin*. A higher-order polynomial will produce more accurate results.

Babbage constructed a small portion of his first Difference Engine in 1832,which served as a demonstration prototype. He later designed an improved version(Difference Engine No. 2), which was to handle seventh-order polynomials andhave 31 decimal digits of accuracy. Like some of his modern successors, Babbageconceived his computers on a grand scale that strained the limits of the technol-ogy—and funds—available to build them. He never completed Difference EngineNo. 2, mainly because of the difficulty of fabricating its 4000 or so high-precisionmechanical parts. The complexity of this 3-ton machine can be appreciated fromFigure 1.9, which is based on one of Babbage's own drawings. The vertical "fig-ure-wheel columns" constitute the registers for storing 31-digit numbers, while theadders are implemented by the rack-and-lever mechanism underneath. It was notuntil 1991 that a working version of Difference Engine No. 2 was actually con-structed (at a cost of around $500,000) by the Science Museum in London to cele-brate the bicentennial of Babbage's birth [Swade 1993].

15

The Analytical Engine. Another reason for Babbage's failure to complete hisDifference Engine was that he conceived of a much more powerful computingmachine that he called the Analytical Engine. This machine is considered to be thefirst general-purpose programmable computer ever designed.

The overall organization of the Analytical Engine is outlined in Figure 1.10. Itcontains in rudimentary form many of the basic features found in all subsequentcomputers— compare Figure 1.10 to Figure 1.2. The main components of the

HANDLE

PRINTER
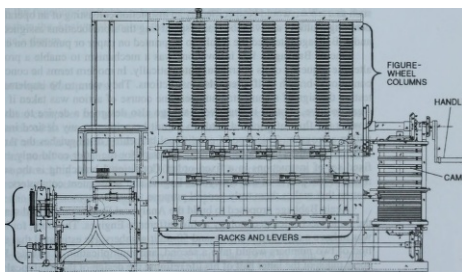
Figure 1.9

Diagram by Babbage of Difference Engine No. 2 [Courtesy of the National Science Museum/Science & SocietyPicture Library].

16

Input-outputequipment(printerand card *punch)

Arithmetic-logic unit(the mill) Data　　　Mainmemory(the store)

n　　　　　　　　　Instructions ii

Operationcards　　　　　　Variablecards

Proj　　　　　jam control unit

Figure 1.10

Structure of Babbage's Analytical Engine.

Analytical Engine are a memory called the store and an ALU called the mill; thelatter was designed to perform the four basic arithmetic operations. To control theoperation jt of the machine, Babbage proposed to use punched cards of a typedeveloped earlier for controlling the Jacquard loom. A program for the AnalyticalEngine was composed of two sequences of punched cards: operation cards used toselect the operation to be performed by the mill, and variable cards to specify thelocations in the store from which inputs were to be taken or results sent. An actionsuch as a x b = c would be specified by an instruction consisting of an operation carddenoting multiply and variable cards specifying the store locations assigned to a, b,and c. Babbage intended the results to be printed on paper or punched on cards.

One of Babbage's key innovations was a mechanism to enable a program toalter the sequence of its operations automatically. In modern terms he conceived ofconditional-branch or if-then-else instructions. They were to be implemented bytesting the sign of a computed number; one course of action was taken if the signwere positive, another if negative. Babbage also designed a device to advance orreverse the flow of punched cards to permit branching to any desired instructionwithin a program. This type of conditional branching distinguishes the AnalyticalEngine from the Difference Engine: a program for the latter could only execute afixed set of instructions in a fixed order. Conditional branching is the source ofmuch of the power of the Analytical Engine and subsequent computers; it is thefeature that makes them truly general purpose.

Again Babbage proposed to build the Analytical Engine on a grand scale usingthe same mechanical technology as his Difference Engines. The store, for instance,was to have a capacity of a thousand 50-digit numbers. He estimated that the addi-tion of two numbers would take a second, and multiplication, a minute. Babbagespent much of the latter half of his life refining the design of the Analytical Engine,but only a small part of it was ever constructed.

Later developments. Many improvements were made to the design of four-function mechanical calculators in the 19th century, which led to their widespread

use. The Comptometer, designed by the American Dorr E. Felt (1862-1930) in 171885, was one of the earliest calculators to use depressible keys for entering dataand commands; it also printed its results on paper. A later innovation was the use

r . j ■ i i • , , , • . Computing and

of electric motors to drive the mechanical components, thus making calculators Com UIers"electromechanical" and greatly increasing their speed. Another important devel-opment was the use of punched cards to sort and tabulate large amounts of data.The punched-card tabulating machine was invented by Herman Hollerith (1860—1929) and used to process the data collected in the 1880 United States census. In1896 Hollerith formed a company to manufacture his electromechanical equip-ment. This company subsequently merged with several others and in 1924 wasrenamed the International Business Machines Corp. (IBM).

No significant attempts to build general-purpose, program-controlled comput-ers were made after Babbage's death until the 1930s [Randell 1982]. In Germany,Konrad Zuse built a small mechanical computer, the Zl, in 1938, apparentlyunaware of Babbage's work. Unlike previous computers, the Zl used binary,instead of decimal, arithmetic. A subsequent Zuse machine, the Z3, which wascompleted in 1941, is believed to have been the first operational general-purposecomputer. Zuse's work was interrupted by the Second World War and had littleinfluence on the subsequent development of computers. Of great influence, how-ever, was a general-purpose electromechanical computer proposed in 1937 byHoward Aiken (1900-73), a physicist at Harvard University. Aiken arranged tohave IBM construct this computer according to his basic design. Work began onAiken's Automatic Sequence Controlled Calculator, later called the Harvard MarkI, in 1939; it became operational in 1944. Like Babbage's machines, the Mark Iemployed decimal counter wheels for its main memory. It could store seventy-two23-digit numbers. The computer was controlled by a punched paper tape, whichcombined the functions of Babbage's operation and variable cards. Although lessambitious than the Analytical Engine, the Mark I was in many ways the realizationof Babbage's dream.

1.2.2 Electronic Computers

A mechanical computer has two serious drawbacks: Its computing speed is limitedby the inertia of its moving parts, and the transmission of digital information bymechanical means is quite unreliable. In an electronic computer, on the other hand,the "moving parts" are electrons, which can be transmitted and processed reliablyat speeds approaching that of light (300,000 km/s). Electronic devices such as thevacuum tube or electronic valve, which was developed in the early 1900s, permitthe processing and storage of digital signals at speeds far exceeding those of anymechanical device.

The first generation. The earliest attempt to construct an electronic computerusing vacuum tubes appears to have been made in the late 1930s by John V. Atana-soff (1903-95) at Iowa State University [Randell 1982]. This special-purposemachine was intended for solving linear equations, but it was never completed. Thefirst widely known general-purpose electronic computer was the Electronic Numer-ical Integrator and Calculator (ENIAC) that John W. Mauchly (1907-80) and J.

Presper Eckert (1919-95) built at the University of Pennsylvania. Like Babbage'sDifference Engine, a motivation for the ENIAC was the need to construct mathe-matical tables automatically—this time ballistic tables for the U.S. Army. Work onthe ENIAC began in 1943 and was completed in 1946. It was an enormous machineweighing about 30 tons and containing more than 18,000 vacuum tubes. It was alsosubstantially faster than any previous computer. While the Harvard Mark I requiredabout 3 s to perform a 10-digit multiplication, the ENIAC required only 3 ms.

The ENIAC had a set of electronic memory units called accumulators with acombined capacity of twenty 10-digit decimal numbers. Each digit was stored in a10-bit ring counter, where the binary pattern 1000000000 denoted the decimal digit0, 0100000000 denoted 1, 0010000000 denoted 2, and so on. The ring counter wasthe electronic equivalent of the decimal counter wheel of earlier mechanical calcu-lators. Like counter wheels, the ENIAC's accumulators combined the function ofstorage with addition and subtraction. Additional units performed multiplication,division, and the extraction of square roots. The ENIAC was programmed by thecumbersome process of plugging and unplugging cables and by manually setting amaster programming unit to specify multistep operations. Results were punched oncards or printed on an electric typewriter. In computing ability, the ENIAC isroughly comparable to a modern pocket calculator!

Like the Analytical Engine, the Harvard Mark I and the ENIAC stored theirprograms and data in separate memories. Entering or altering the programs was atedious task. The idea of storing programs and their data in the same high-speedmemory—the stored-program concept—is attributed to the ENIAC's designers,notably the Hungarian-born mathematician John von Neumann (1903-57) whowas a consultant to the ENIAC project. The concept was first published in a 1945proposal by von Neumann for a new computer, the Electronic Discrete VariableComputer (EDVAC). Besides facilitating the programming process, the stored-program concept enables a program to modify its own instructions. (Such self-modifying programs have undesirable aspects, however, and are rarely used.)

The EDVAC differed from most of its predecessors in that it stored and pro-cessed numbers in true binary or base 2 form. To minimize hardware costs, datawas processed serially, or bit by bit. The EDVAC had two kinds of memory: a fastmain memory with a capacity of 1024 or IK words (numbers or instructions) and aslower secondary memory with a capacity of 20K words. Prior to their execution, aset of instructions forming a program was placed in the EDVAC s main memory.The instructions were then transferred one at a time from the main memory to theCPU for execution. Each instruction had a well-defined structure of the form

A, A2 A3 A4 OP

(1.6)

meaning: Perform the operation OP (addition, multiplication, etc.) on the contentsof main memory locations or "addresses" A, and A2 and then place the result inmemory location A3. The fourth address A4 specifies the location of the nextinstruction to be executed. A variant of this instruction format implements condi-tional branching, where the next instruction address is either A3 or A4, dependingon the relative sizes of the numbers stored in A, and A2. Yet another instructiontype specifies input-output operations that transfer words between main memoryand secondary memory or between secondary memory and a printer. The EDVACbecame operational in 1951.
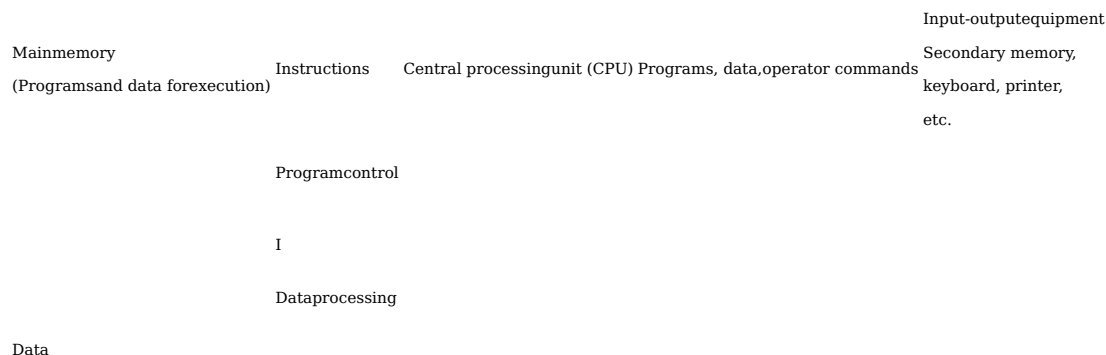
Mainmemory (Programsand data forexecution)	Instructions	Central processingunit (CPU) Programs, data,operator commands	Input-outputequipment Secondary memory, keyboard, printer, etc.

Programcontrol

I

Dataprocessing

Data

Figure 1.11

Organization of a first-generation computer.

Computing andComputers

In 1947 von Neumann and his colleagues began to design a new stored-pro-gram electronic computer, now referred to as the IAS computer, at the Institute forAdvanced Studies in Princeton. Like the EDVAC, it had the general structuredepicted in Figure 1.11, with a CPU for executing instructions, a main memory forstoring active programs, a secondary memory for backup storage, and miscella-neous input-output equipment. Unlike the EDVAC, however, the IAS machine wasdesigned to process all bits of a binary number simultaneously or in parallel. Sev-eral reports describing the IAS computer~were published [Burks. Goldstine, andvon Neumann 1946] and had far-reaching influence. In its overall design the IAS isquite modern, and it can be regarded as the prototype of most subsequent general-purpose computers. Because of its pervasive influence, we will examine the IAScomputer in more detail below.

In the late 1940s and 1950s, the number of vacuum-tube computers grew rap-idly. We usually refer to computers of this period as first generation, reflecting asomewhat narrow view of computer history [Randell 1982]. Besides those men-tioned already, important early computers included the Whirlwind I constructed atthe Massachusetts Institute of Technology and a series of machines designed atManchester University [Siewiorek, Bell, and Newell 1982], In 1947 Eckert andMauchly formed Eckert-Mauchly Corp. to manufacture computers commercially.Their first successful product was the Universal Automatic Computer (UNIVAC)delivered in 1951. IBM, which had earlier constructed the Harvard Mark I. intro-duced its first electronic stored-program computer, the 701, in 1953. Besides theiruse of vacuum tubes in the CPU, first-generation computers experimented withvarious technologies for main and secondary memory. The Whirlwind introducedthe ferrite-core memory in which a bit of information was stored in magnetic formon a tiny ring of magnetic material. Ferrite cores remained the principal technologyfor main memories until the 1970s.

The earliest computers had their instructions written in a binary code know n asmachine language that could be executed directly. An instruction in machine lan-guage meaning "add the contents of two memory locations" might take the form

00111011000000001001100100000111

Machine-language programs are extremely difficult for humans to write and soare very error-prone. A substantial improvement is obtained b\ allowing opera-tions and operand addresses to be expressed in an easily understood symbolic

20

SECTION 1.2The Evolution ofComputers

form such as

ADD XI, X2

This symbolic format, which is referred to as an assembly language, came into usein the 1950s, as computer programs were growing jn size and complexity. Anassembly language requires a special "system" program (an assembler) to translateit into machine language before it can be executed. First-generation computers weresupplied with almost no system software; often little more than an assembler wasavailable to the user. Moreover, assembly and machine languages varied widelyfrom computer to computer so first-generation software was far from portable.

The IAS computer. It is instructive to examine the design of the Princeton IAScomputer. Because of the size and high cost of the CPU's electronic hardware, thedesigners made every effort to keep the CPU, and therefore its instruction set,small and simple. Cost also heavily influenced the design of the memory sub-system. Because fast memories were expensive, the size of the main memory (ini-tially IK words but expandable to 4K) was less than most users would havewished. Consequently, a larger (16K words) but cheaper secondary memory basedon an electromechanical magnetic drum technology was provided for bulk storage.Essentially similar cost-performance considerations remain central to computerdesign today, despite vast changes over the years in the available technologies andtheir actual costs.

The basic unit of information in the IAS computer is a 40-bit word, which isthe standard packet of information stored in a memory location or transferred inone step between the CPU and the main memory M. Each location in M can beused to store either a single 40-bit number or else a pair of 20-bit instructions. TheIAS's number format is fixed-point, meaning that it contains an implicit binarypoint in some fixed position. Numbers are usually treated as signed binary frac-tions lying between -1 and +1, but they can also be interpreted as integers. Exam-ples of the IAS's binary number format are

01101000000 0000000000 0000000000 0000000000 = +.8125

10011000000 0000000000 0000000000 0000000000 = -0.8125

Numbers that lie outside the range ±1 must be suitably scaled for processing byIAS.

An IAS instruction consists of an 8-bit opcode (operation code) OP followedby a 12-bit address A that identifies one of up to 212 = 4K 40-bit words stored inM. The IAS computer thus has a one-address instruction format, which we repre-sent symbolically as

OP A

This format may appear very restrictive compared with the EDVAC's four-addressinstruction format (1.6). The IAS's shorter format clearly saves memory space.The fact that it does not restrict the machine's computational capabilities followsfrom two key aspects of the IAS's design that have been incorporated into all latercomputers:

1. The CPU contains a small set of high-speed storage devices called registers,which serve as implicit storage locations for operands and results. For example,

>s   Program

Addre controlunit PCU Instructiondecoder   —*■ Control77T signals

0   M(0)   \£}

1 AR |

1   Mil)

                                       1
2   M(2)   1

3   Mi?i   ' -

4   M(4)   IBR   L_pc

5   M(5)

      *■   1   ,

                              I

              Dataprocessing   1!

   º

   unit DPL   ii   '

              Arithmetic-logic unit

4,093 M.4,093)

4,094 M(4,094)   T ■   + i

      M(4,095)   r   —

4,095 AC   [MQ

      Mainmemory M

Legend

Figure 1.12

Organization of the CPU and main memory of the IAS computer.

21

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 PC-whichautomatically stores and keeps track of the address of the next instruction to be

22

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 <—. Hence, AC := MQ meanstransfer (copy) the contents of register MQ to register AC without altering the con-tents of MQ. Elements of main memory M are denoted by appending to M anaddress in parentheses. For example, M(X) denotes the 40-bit memory word withaddress X, while M(X,0:19) denotes the half-word consisting of bits 0 through 19ofM(X).

Figure 1.13 illustrates our descriptive notation for a simple three-instructionIAS program that adds two numbers. The numbers to be added are stored in themain memory locations with addresses 100 and 101; their sum is placed in memorylocation 102. Note the role played by the accumulator AC as an intermediatesource and destination of data.

The set of instructions defined for the IAS computer is given in Figure 1.14[Burks. Goldstine, and von Neumann 1946], omitting only those intended for

Instruction

Comment

AC := M(100) Load the contents of memory location 100 into the accumulator.

AC := AC + M(101) Add the contents of memory location 101 to the accumulator.M(102) := AC Store the contents of the accumulator in memory location 102.

Figure 1.13

An IAS program to add two numbers stored in main memory.

23

input-output operations. We have divided them into three categories: data-transfer,data-processing, and program-control instructions. Observe that some instructionshave all their operands in CPU registers; others have one operand in memory loca-tion M(X). The data-processing instructions do most of the "real" work; all theothers play supporting roles. Because only one memory address X can be speci-fied at a time, multioperand instructions such as add and multiply must use CPUregisters to store some of their operands. Consequently, it is necessary to precedeor follow a typical data-processing instruction by data-transfer instructions thatload input operands into CPU registers or transfer results from the CPU to mem-ory. This requirement is illustrated by the add operation in Figure 1.13, where twodata-transfer instructions and one add instruction are needed to accomplish a sin-gle addition operation. Hence the IAS like many of its successors contains quite afew data-transfer instructions whose purpose is to shuttle information unchanged(except possibly in sign) between CPU registers and memory. The IAS's data-pro-cessing instructions perform all the basic operations of arithmetic on signed 40-bitnumbers. The IAS can also perform nonnumerical operations, but with some diffi-culty, because it treats all its operands as numbers.

The group of instructions called program-control or branch instructions deter-mine the sequence in which instructions are executed. Recall that the programcounter PC specifies the address of the next instruction to be executed. Instructionsare normally executed in a fixed order determined by incrementing the programcounter PC. The program-control instructions are designed to change this order.The IAS has two unconditional branch instructions (also called "jump" or "go to"instructions), which load part of X into PC and cause the next instruction to betaken from the left half or right half of M(X). The two conditional branch instruc-tions permit a program branch to occur if and only if AC contains a nonnegativenumber. These instructions allow the results of a computation to alter the instruc-tion execution sequence and so are of

great importance.

The last two instructions listed in Figure 1.14 are "address-modify" instruc-tions that permit 12-bit addresses to be computed in the CPU and then inserteddirectly into instructions stored in M. Address-modify instructions allow a programto alter itself, enabling, for example, the same data-processing instruction to referto different operands at different times. Modifying programs during their executionis now considered obsolete and undesirable, but it was an important feature of earlycomputers like IAS.

Instruction execution. The IAS fetches and executes instructions in severalsteps that form an instruction cycle. Since two instructions are packed into a 40-bit

24

Instruction type Instruction

Description

Data transfer

AC := MQ

AC := M(X)M(X) := ACMQ := M(X)AC := -M(X)AC:= |M(X)IAC:=-|M(X)I

Transfer contents of register MQ to register AC.

Transfer contents of memory location X to AC.

Transfer contents of AC to memory location X.

Transfer M(X) to MQ.

Transfer minus M(X) to AC.

Transfer absolute value of M(X) to AC.

Transfer minus I M(X) I to AC.

Data processing AC := AC + M(X)AC := AC + |M(X)|AC := AC - M(X)AC:=AC- |M(X)|AC.MQ := MQ x M(X)

MQ.AC := AC - M(X)

AC := AC x 2AC := AC - 2

Add M(X) to AC putting the result in AC.

Add absolute value of M(X) to AC.

Subtract M(X) from AC.

Subtract |M(X)I from AC.

Multiply MQ by M(X) putting the double-wordproduct in AC and MQ.

Divide AC by M(X) putting the quotient in AC andthe remainder in MQ.

Multiply AC by two (1-bit left shift).

Divide AC by two (1-bit right shift).

Program control go to M(X, 0:19)go to M(X, 20:39)

if AC >0 then

gotoM(X, 0:19)

if AC > 0 thengo to M(X, 20:39)

M(X, 8:19):= AC (28:39)M(X, 28:39) := AC(28:39)

Take next instruction from left half of M(X)

Take next instruction from right half of M(X).

If AC contains a nonnegative number, then take nextinstruction from left half of M(X).

If AC contains a nonnegative number, then take nextinstruction from right half of M(X).

Replace left instruction address field in M(X) by 12right-most bits of AC.

Replace right instruction address field in M(X) by12 right-most bits of AC.

Figure 1.14

Instruction set of the IAS computer.

word, the IAS fetches two instructions in each instruction cycle. One instructionhas its opcode placed in the instruction register IR and its address field (if any)placed in the address register AR. The other instruction is transferred to the IBRregister for possible later execution. Whenever the next instruction needed by theCPU is not in IBR, the program counter PC is incremented to generate the nextinstruction address.

Once the desired instruction has been loaded into the CPU, its execution phasebegins. The PCU decodes the instruction's opcode, and the PCU's subsequentactions depend on the opcode's bit pattern. Typically, these actions involve one ortwo register-transfer (micro) operations of the form S ^/(S^Sj, ...,Sk), where the

5,'s are the locations of operands and/is a data-transfer or arithmetic operation.For example, the add instruction AC := AC + M(X) is executed by the followingtwo register-transfer operations:

DR := M(AR);

AC := AC + DR

First, the contents of the memory location M(AR) specified by the address registerAR are transferred to the data register DR. Then the contents of DR and the accu-mulator AC are added via the DPU's arithmetic-logic unit, and the result is placedin AC. The unconditional branch instruction go to M(X.0:19) has an address fieldcontaining some address X: after fetching this instruction, X is placed in AR. Thisinstruction is then executed via the single register-transfer operation PC := AR.which makes PC point to the desired next instruction stored in the half-wordM(X,0:19).

25

CHAPTER 1Computing andComputers

EXAMPLE 1.4 AN IAS PROGRAM TO PERFORM VECTOR ADDITION. Let

A = A(l), A(2), ..., A(1000) and B = B(l), B(2) B(IOOO) be two vectors, that is.

one-dimensional arrays, of numbers to be added. The desired vector sum C - A + B isdefined by

C(l), C(2), ..., C(1000) = A(l) + B(l), A(2) +B(2), ... A(1000) + B(1000)

For simplicity we will assume that the numbers processed by the IAS, including thevector elements A(I). B(I), and C(I) are 40-bit integers, and that the input vectors areprestored in the IAS"s main memory M. We need to perform the add operation

C(I) := A(I) + B(I)

1000 times, specifically for I = 1, 2, ..., 1000. Using the operations available in the IASinstruction set. the basic addition step above can be realized by the following three-instruction sequence (compare Figure 1.13):

AC := A(I)

AC := AC + B(I)C(I):=AC

(1.8)

Clearly, a program with 1000 copies of these three instructions, each with a differentindex I. would implement the vector addition. However, such a program, besides beingvery inconvenient to write, would not fit in M along with the three vectors A. B. and C.We need some type of loop or iterative program that contains one copy of (1.8) but canmodify the index I to step through all elements of the vectors.

Figure 1.15 shows such a program. The vectors A. B. and C are assumed to bestored sequentially, beginning at locations 1001, 2001, and 3001. respectively. Thesymbol to the left of each instruction in Figure 1.15 is its location in M. For instance.2L (2R) denotes the left (right) halt" of M(2). The first location M(0> 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	Constant. •
	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)	Modify address in 3R.
	9L		M(3, 28:39) :=AC(28:39)	
	9R		AC := AC + M(2)	Modify address in 4L.
	10L		M(4, 8:19):=AC(28:39)	
	10R		gotoM(3,0:19)	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 computa-tion, 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 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.

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)

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 program-controlinstructions, 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

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

        ...
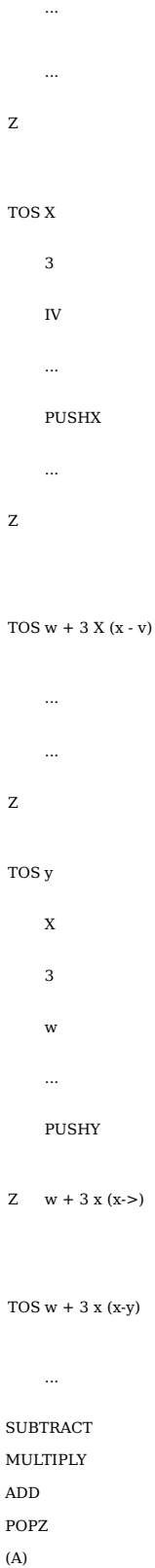
Z

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 conven-

tional, 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 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

Instruction decoder (may be microprogrammed)	—*- Control ZV si?nals Program control unit PCU	1U devices	IU devices

CHAPTER 1

	4	System /360	1 Computing andComputers
t	1	IO interface	r
1 »	1 AR 1		

			IO
i	i i	i	Program status word PSW	10processor(channel)	processor (channel)

SR PC

i	v	II
		i

ii

r	H	ii	Main memorycontrol unit

»   II   i   n

| Sixteen 32-bit general registers | Four 64-bit floating-point registers | Main memory M |

i   1
             1
r

Floating-pointALU

"I

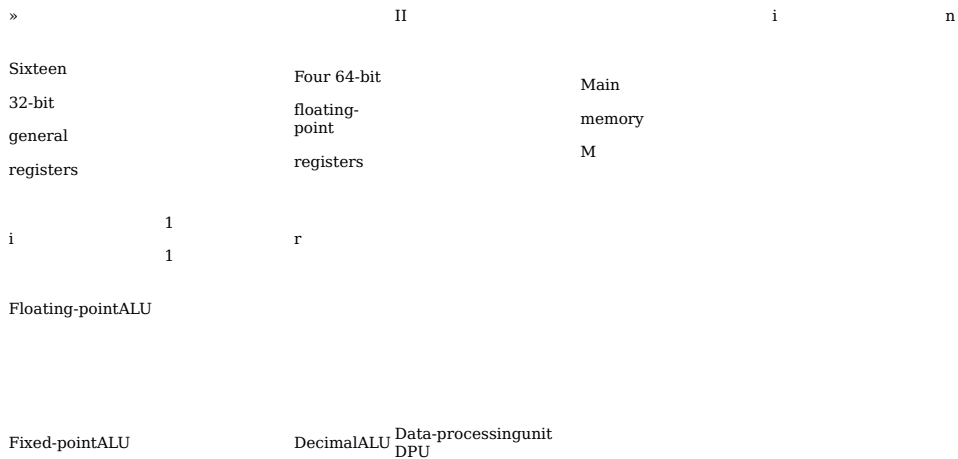Fixed-pointALU   DecimalALU   Data-processingunit DPU

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 floating-pointinstructions (albeit slowly) if microprograms are written to perform the desiredfloating-point operations by means of fixed-point arithmetic circuits. Micropro-gramming 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.

SECTION 1.3 The VLSI Era

&^>
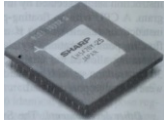
(a) (b) (c)

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, large-scale, and very-large-scale integration (MSI, LSI and VLSI.

•

lG-bit ,•*

a   109   DRAM . „'

• •

u

c   lM-bit ./^DRAM ./• 64-bit

c   j 6_bjt ./* microprocessor

C   106   microprocessor A* 32-bit

£   v^ microprocessor

| lK-bit v» 8-bit microprocessorDRAM ./^

■a

y   103

_,— • 4-bit microprocessorMSI
1   •SSI
    i i i i

1960

1970

1980 1990

Year

2000

2010
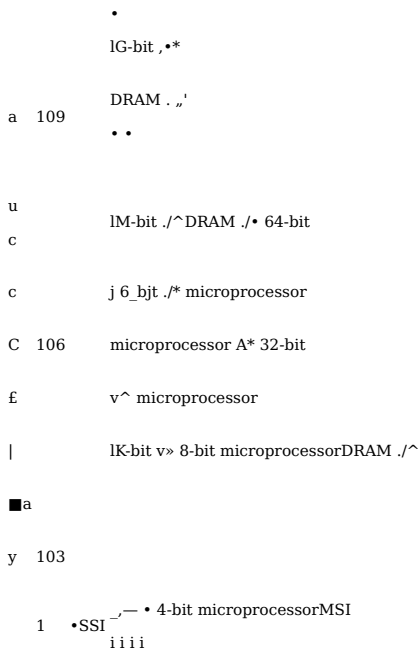
Figure 1.19

Evolution of the density of commercial ICs.

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 technology improved and chip density increased, the complexity and performance of one-chip microprocessors increased steadily, as reflected in the increase in CPU word size to 8 and then 16 bits by the mid-1980s. By 1990 manufacturers could fabricate the entire 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 a small number of ICs) is called a microcomputer.

SECTION 1.3 The 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 to as MOS (metal-oxide-semiconductor) after its physical structure. Both bipolar and MOS circuits have transistors as their basic elements! They differ, however, in the polarities of the electric charges associated with the primary carriers of electrical signals within their transistors. Bipolar circuits use both negative carriers (elec-trons) and positive carriers (holes). MOS circuits, on the other hand, use only one type of charge carrier: positive in the case of P-type MOS (PMOS) and negative in the case of N-type MOS (NMOS). Various bipolar and MOS IC circuit types or IC families have been developed that provide trade-offs among density, operating speed, power consumption, and manufacturing cost. An MOS family that effi-ciently combines PMOS and NMOS transistors in the same IC is complementary MOS or CMOS. This technology came into widespread use in the 1980s and has been the technology of choice for microprocessors and other VLSI ICs since then because 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 small CMOS circuit whose function is to detect when a 4-bit word $x_0 x_1 x_2 x_i$ becomes zero. The circuit's output z should be 1 when $x_0 x_1 x_2 x_i = 0000$; it should be 0 for the other 15 combinations 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 the if 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 standard symbolic form in Figure 1.20a. It consists of equal numbers of PMOS transistors denoted 5,:57 and NMOS transistors denoted SS:SU. Each transistor acts as an on-off switch 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 and lower terminals; when turned off, that path is broken. An NMOS transistor is turned on by 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 and others to switch off, which creates various signal paths through the circuit. In Figure 1.20 the constant signals 0 and 1 are applied at various points in ZD. (These signals are derived from ZD's electrical power supply.) The 0/1 signals "flow" through the circuit along 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 signals and signal transmission paths produced by $x_0 x_i x_2 x_3 - 0001$. The first input signal $x_0 = 0$ is applied to PMOS transistor 5, and NMOS transistor 5g; hence S, is turned on and 5g is turned off. Similarly, x, = 0 turns S2 on and S9 off. A path is created through S, and S2, 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 as required.

If we change input x3 from 1 to 0 in Figure 1.20b, the following chain of events occurs: 54 turns on and 5,, turns off, changing y2 to 1. Then 5I3 turns on and S6 turns off, making y3 = 0. Finally, the new value of y3 turns 57 on and S]4 off, so z becomes 1.
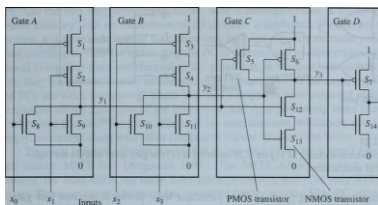
Output

Inputs

PMOS transistor NMOS transistor

(a)

; = 0
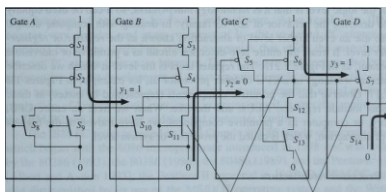
xQ = 0 xl = 0 x2 - 0 *3 = 1 Transistor switched on Transistor switched off

(b)

Figure 1.20

(a) CMOS circuit ZD for zero detection; (b) state of ZD with input combination

$x_0 x_1 x_2 x_3\} = 0001$ making z = 0.

Hence the zero input combination $x_0 x_1 x_2 x_3 = 0000$ makes c = 1 as required. It can readily be verified that no other input combination does this.

A transistor circuit like that of Figure 1.20 models the behavior of a digital circuit at a low level of abstraction called the switch level. Because many of the ICs of interest contain huge numbers of transistors, it is rarely practical to analyze their 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 logic Aexe\illustrated by Figure 1.21a. we represent certain common subcircuits by symbolic
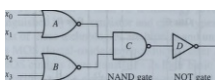
NOR gates

NAND gate

(a)

NOT gate (inverter)

Zero detector

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 application-specific 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

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

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 bus is linked to the system bus, to which the microprocessor and memory are attached via a special bus-to-bus control unit sometimes referred to as a bridge. The IO devices of a personal computer include the traditional keyboard, a CRT-based or flat-panel video monitor, and disk drive units for the hard and flexible (floppy) disk storage devices that constitute secondary memory. More recent additions to the IO device repertoire include drive units for CD-ROMs (compact disc read-only mem-ories), which have extremely high capacity and allow sound and video images to be 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. This reduces 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, a multiply instruction can replace a multiinstruction subroutine that implements mul-tiplication by repeated execution of add instructions. Reducing N in this way tends to reduce overall program execution time T, as well as the time that the CPU spends fetching instructions and their operands from memory. The same advances in VLSI made it possible to add new features to old microprocessors, such as new instructions, data types, instruction sets, and addressing modes, while retaining the ability to execute programs written for the older machines.

The Intel 80X86/Pentium series illustrates the trend toward more complex instruction sets. The 1978-vintage 8086 microprocessor chip, which contained a mere 20,000 transistors, was designed to process 16-bit data words and had no instructions 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 set of floating-point instructions [Albert and Avnon 1993]. The Pentium accumulated

most of the architectural features of its various predecessors in order to enable it to execute, with little or no modification, programs written for earlier 80X86-series machines. 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 can sometimes reduce a computer's overall performance. To illustrate this condition, suppose that a particular microprocessor has only fast, simple instructions, each of which requires k time units, to execute. Thus the microprocessor can execute 100 instructions in 100k time units. Now suppose that 5 percent of the instructions are slow, complex instructions requiring 2lk time units each. To execute an average set 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 not necessarily translate into faster program execution. Moreover, complex instructions require relatively complex processing circuits, which tend to put CISCs in the larg-est and most expensive IC category. These drawbacks were first recognized by John Cocke and his colleagues at IBM in the mid-1970s, who developed an experimental computer called 801 that aimed to achieve very fast overall performance via a streamlined instruction set that could be executed extremely fast [Cocke and Mark-stein 1990]. The 801 and subsequent machines with a similar design philosophy have been called reduced instruction set computers (RISCs). A number of commer-cially successful RISC microprocessors were introduced in the 1980s, including the IBM RISC System/6000 and SPARC, an "open" microprocessor developed by Sun Microsystems and based on RISC research at the University of California, Berkeley [Patterson 1985]. Many of the speedup features of RISC machines have found their way 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 besides its instruction set, especially the time required to move instructions and data between the CPU and main memory M and, to a lesser extent, the time required to move information between M and IO devices. It typically takes the CPU about five 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, despite strenuous 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 Neumann bottleneck. RISC computers usually limit access to main memory to a few load and store instructions; other instructions, including all data-processing and pro-gram-control instructions, must have their operands in CPU registers. This so-

43

CHAPTER I Computing and Computers

[3] The public became aware of CISC complexity when a design flaw affecting the floating-point division instruction of the Pentium was discovered in 1994. The cost to Intel of this bug. including the replacement cost 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 Neu-

section 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 is the 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 a central 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 execute the operation is 1//microseconds ((is). This value is called the clock cycle or clock period Tdock. For example, a computer clocked at 250 MHz can perform one basic operation in the clock period Tdock = 1/250 = 0.004 (is. Complicated operations such as division or operations on floating-point numbers can require more than one clock 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 a steady, but less dramatic, increase in clock speed. For example, from 1981 to 1995 microprocessor clock speeds increased from about 10 MHz to 100 MHz. Clock speeds of 1 gigahertz (1 GHz or 1000 MHz) and beyond are feasible using faster versions of current CMOS technology. It might therefore seem possible to achieve any 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 of light, 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 which requires 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 appropriate functional 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 of instruction execution performance are based on average figures, which are usually determined 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) of such programs Q on a given CPU takes T seconds and involves the execution of a total of N machine (object) instructions. Here N is the actual number of instructions executed, 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. While T can be determined accurately only by measurement of <2's run time in actual or simulated execution, we can relate T to some basic parameters of the computer's architecture and implementation. One such parameter is the (average) number of instructions executed per second, which we denote by IPS. Clearly, T = N/IPS s. Another common measure of the performance of a CPU is the average number of cycles per instruction or CPI needed to execute Q. Now CPI = (/ X \06)/IPS, where / is the CPU's clock frequency in MHz. Hence, the program execution time T is given by

T =

NxCPI/xlO6

(1.12)

It is also common to measure CPU performance in terms of millions of instruc-tions executed per second, denoted MIPS, where MIPS = IPS X 106. ClearlyMIPS =f/CPI.

Equation (1.12) indicates how the three separate factors software, architecture,and hardware technology jointly determine a computer's performance.

1. Software: The efficiency with which the programs are written and compiled intoobject code influences N, the number of instructions executed. Other factorsbeing equal, reducing N tends to reduce the overall execution time T.

2. Architecture: The efficiency with which individual instructions are processeddirectly affects CPI, the number of cycles per instruction executed. ReducingCPI also tends to reduce T.

3. Hardware: The raw speed of the processor circuits determines/, the clock fre-quency. Increasing/tends to reduce T.

In general, the complex instruction sets of CISC processors aim to reduce N at theexpense of CPI, whereas RISC processors aim to reduce CPI at the expense of N.Advances in VLSI technology affecting all types of computers tend to increase/

Speedup techniques. A number of speed-enhancing features have been incor-porated into the design of computers in recent years [Hwang 1993]; they are sum-marized in Figure 1.23. These methods were defined as far back as the 1960s and1970s for use in mainframe computers. A cache is a memory unit placed betweenthe CPU and main memory M and used to store instructions, data, or both. It hasmuch smaller storage capacity than M, but it can be accessed (read from or writteninto) more rapidly and is often placed (at least partly) on the same chip as the CPU.The cache's effect is to reduce the average time required to access an instruction ordata word, typically to just a single clock cycle. Special hardware and softwaretechniques support the complex flow of information among M, the cache, and theregisters of the CPU.

Another important speedup technique known as pipelining allows the process-ing of several instructions to be partially overlapped Pipelining is most easily done

Feature Objective Description

SECTION 1.3 —

The VLSI Era Cache To provide the CPU with faster A cache is a memory unit inserted between

memory access to instructions and data. the CPU and main memory M. It is faster

than Mtmt has less storage capacity.

Pipehned To increase performance by allowing The CPU is constructed from independentprocessing the processing of several instructions subunits (stages), which can hold several

to be partially overlapped. instructions in different stages of execution.

Superscalar To increase performance by allowing Multiple (pipelined) units are provided forprocessing several instructions to be processed instruction processing. Instructions can bein parallel (full overlapping). issued simultaneously to each unit.

Figure 1.23

Some important speedup features of modern computers.

for a sequence of instructions of the same or similar types that employ a single E-unit, such as a floating-point processor. However, all the common steps involved ininstruction processing by the CPU can be pipelined: instruction fetching (IF),instruction decoding (ID), operand loading (OL), execution (EX), and operandstoring (OS). A pipelined system is often compared to an assembly line on whichmany products are in various stages of manufacture at the same time. In a nonpipe-lined CPU, instructions are executed in strict sequence, as depicted in Figure 1.24a.Pipelining permits the situation shown in Figure 1.24/?, where each major step of

Instruction /[Instruction A Instruction /

^i r

Instruction fetch IF: |lF,| |lF:| |lF?|

Instruction decode ID: [id7| flDTI flD^

Operand load OL: |OL,| |OL:| |OL3

Execution EX: Ex] |EX,|

Operand store OS: |OS,[

Time (clock cycles): 12 3 4 5 6 7 8 9 10 11 12 13 14 15

(a)

Instruction fetch IF: [JfT|[7f~]|1f7]\^I\| || IF? |[1f7|| IF7 || || || |[TF^[rr~]|JF^[JF^]Instruction decode ID:[|[roj]|lD:||lD, |[idT1| I["^1 ["^1 [">] 1 II II 1 ["^1 ["^1HOperand load OL: | || |[OL^[5C\6u\|OL4|| ^|qLT|[qL7|[qlT]| || || ||OLg|[OL^|Execution EX: | || || ^[E^[E^|EX;||Exl1[E3q[E^[eq[E)ri| || || |fEX^

Operand store OS: | |j [| \[[|OS, |[qS^[p"s311 |[osl][qsT][oS^| || || || |

Time (clock cycles): 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Figure 1.24

Instruction processing: (a) sequential or nonpipelined and (b) pipelined.

instruction processing is assigned to, and handled independently by, a separate sub-unit (stage) of the CPU pipeline. In this example, up to five instructions can be overlapped, provided the necessary pipeline stages are available. Note that perfor-mance-reducing delays occur, as in the case of instruction 74 (shaded), which mustuse the EX stage for two consecutive cycles. A similar problem occurs in the caseof branch instructions like 77 in Figure 1.24b, where the outcome of 77's EX stepmust be known before the location of the next instruction (78) to be processed canbe identified.

A microprocessor's effective MIPS rate can also be increased by replicatingvarious instruction-processing circuits so that several instructions can be in thesame processing phase at the same time. This makes it possible to start the process-ing of, or issue, two or more instructions simultaneously or in parallel; in otherwords, the instructions can be completely overlapped. CPUs with this capability aresaid to be superscalar. (Note that two instructions in the same pipeline must beissued sequentially rather than in parallel.) For example, if the logic needed for theIF, ID, OL, EX, and OS steps is duplicated (with or without pipelining), then twoinstructions can be issued simultaneously. However, if the instructions are not inde-pendent, for example, if they share the same operands or one takes as input a resultcomputed by the other, then delays not unlike those illustrated in Figure 1.247? canoccur. Pipelining and superscalar design are both instances of instruction-level par-allelism. The logic circuits needed to deal with parallelism of this kind add consid-erable complexity to the CPU's program control and execution units.

EXAMPLE 1.7 THE POWERPC MICROPROCESSOR SERIES [MOTOROLA

19 93]. In the early 1990s Apple, IBM, and Motorola jointly developed the PowerPC.It is a family of single-chip microprocessors, including the 601, 603, and other models,which share a common architecture derived from the POWER architecture used inIBM's RISC System/6000 [Diefendorf, Oehler, and Hochsprung 1994; Weiss andSmith 1994]. Although it is also designated a RISC, the PowerPC has a large numberof instructions—more than 200 distinct types, in fact—and its design is far from sim-ple. Nevertheless, it exhibits the following features that are typical of contemporaryRISC-style designs:

1. Instructions have a fixed length (32 bits or one word) and employ just a few opcodeformats and addressing modes.

2. Only load and store instructions can access main memory; all other instructionsmust have their operands in CPU registers. This load/store architecture reduces thetime devoted to accessing memory. This time is further reduced by the use of one ormore levels of cache memory.

3. Instruction processing is heavily pipelined. For example, the PowerPC has an E-unitfor integer (fixed-point) operations that has the four pipeline stages: fetch, decode,execute, and write results. Hence if an E-unit's pipeline can be kept full, a newresult emerges from it every clock cycle, thus achieving the ideal performance levelof one fully executed instruction per clock cycle.

4. The CPU contains several E-units—the number depends on the model—whichallow it to issue several instructions simultaneously and puts the PowerPC in

thesuperscalar category.

The organization shown in Figure 1.25 is typical of the early PowerPC models,such as the 601 and 603, which have three E-units: an integer execution unit, a float-ing-point unit, and a branch processing unit, allowing up to three instructions no be

Syste m bus

A

II

Cache

<

r

control Instruction

unit   queue   '

'1

1 ,

1   1   "   1

General-

Branch- Integer   Floating- purpose

processing execution   point   and

unit   unit   unit   floating-

(pipeline) (pipeline) (pipeline) pointregisters

i   r   I 1

I- 1

Figure 1.25

Overall organization of the PowerPC.

issued in the same clock cycle. The integer unit executes all fixed-point numerical andlogic operations, including those associated with load-store instructions. Although partof the CPU's program control unit, the branch processing unit is considered an E-unitfor branch instructions. Each PowerPC chip also contains a cache memory, whose sizeand organization vary with the model. For example, the PowerPC 603, which wasintroduced in 1995 and is aimed at low-power applications like laptop computers, hasa 16 KB cache, half of which stores data while the other half stores instructions. A hintof the complexity of the 603 can be seen from Figure 1.26. It contains 1.6 million tran-sistors in an IC chip of area 7.4 x 11.5 mm (in its earliest versions) and consumes lessthan 3 watts of power.

To illustrate the PowerPC's instruction set, consider the vector addition discussedearlier and expressed by the FORTRAN90 statement

C(1: 1000) = A(l: 1000) + B(l: 1000)

Assume that each vector consists of 1000 double-precision (64-bit), floating-pointnumbers. An assembly-language program for the PowerPC that carries out this vectoroperation appears in Figure 1.27. (We have slightly simplified the language syntaxhere.) The last five instructions form the program's main loop and are executed 1000times. The key data-processing instruction in this loop has the opcode fadd, and per-forms a double-precision, floating-point addition. All fadd's operands are in 64-bitfloating-point registers, of which the PowerPC has 32, denoted fr0:fr31 here. Theprogram communicates with memory via the instructions lw (load word), lfdu (loadfloating-point double-precision with update), and stfdu (store floating-point double-precision with update); these are just a few of the PowerPC's many types of load-store
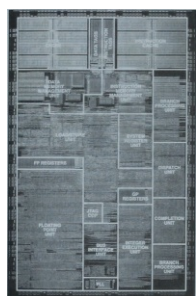
Figure 1.26

Photomicrograph of the PowerPC 603 micro-processor chip. [Courtesy of Motorola Inc.]

instructions. The PowerPC has 32 general-purpose registers r0:r31, several of whichserve as memory address registers in our program. The update option, indicated by theu suffix on lfdu and stfdu invokes a kind of automatic indexing, which causes the con-tents of the memory address register to be initially incremented. For example, theinstruction

lfdu frl, l(r5)

invokes the following two operations: increment the address register r5 and then loadthe data register frl. In other words

r5 := r5 + 1: frl := mem(r5):

(1.13)

Location Instruction

Comment

mtspr CTR, #1000 Move vector length N = 1000 to special register CTR.

Load start address of vector A into general register r5.

Load start address of vector B into general register r6.

Load start address of vector C into general register r7.

LOOP lfdu frl, Kr5) Load A(i + 1) into floating-point register frl: update r5.

Load B(i + 1) into floating-point register fr2; update r6.Perform floating-point addition frl := frl + fr2Store frl as C(i + 1); update r7.Decrement CTR. then branch to LOOP if CTR * 0.

lw   r5, #A

lw   r6. #B

lw   r7,#C

lfdu frl, Kr5)

lfdu fr2. I(r6)

fadd frl, fr2, frl

stfdu frl. I(r7)

bne   LOOP

Figure 1.27

A PowerPC program for vector addition.

SECTION 1.3The VLSI Era

50 The memory data denoted by mem(r5) in (1.13) is normally in the PowerPC's

cache memory which, at any time, mimics a portion of the main memory M that is inactive use. Thus if the current memory address defined by r5 is assigned to the cache,the data required by lfdu is fetched from the cache, rather than from M, where a "mas-ter" copy of the same data resides. Similarly, the store instruction stfdu writes its datainto a cache location, although (eventually) the corresponding data in M must beupdated. Should mem(r7) not be currently assigned to the cache, the PowerPC's elabo-rate memory access control automatically transfers data between M and the cache toassign the relevant portion of the processor's address space to the cache. The lastinstruction bne (branch if not equal) appearing in Figure 1.27 is a powerful conditionalbranch instruction. First bne automatically decrements the "special" register calledCTR (counter) and tests it for zero. If CTR * 0, then the next instruction executed is theone stored in location LOOP. When CTR reaches zero, the vector addition terminatesand the instruction following bne is executed. Observe that the five-instruction pro-gram loop typically resides in the cache for the duration of the program's execution.

As Figure 1.25 indicates, the Power PC has three (more in some models) separateE-units for executing integer, floating-point, and branch instructions. This superscalardesign allows up to three separate instructions to be dispatched (issued) for executionin every clock cycle. Moreover, these E-units are pipelined to varying degrees, so thatan active E-unit can contain several consecutive instructions in various stages of execu-tion. Hence, for our vector addition task, we would expect to find the CPU concur-rently executing several operations of the form

C(j) := A(j) + B(j), C(j + 1) := A(j + 1) + B(j + 1), C(j + 2) := A(j + 2) + B(j + 2), ...

The concurrency achieved, and therefore the execution time of the program, depend onvarious implementation details and cannot be determined from inspection of the pro-gram code alone.

The vector addition programs for the IAS (Figure 1.15) and the PowerPC (Fig-ure 1.27) reflect the evolution of computer architecture over a 50-year period. Thetwo programs are fundamentally similar in that each program is designed to loop TVtimes through the three basic steps: load data from M, add data in CPU registers,and store results in M. The computers share the same basic features of the vonNeumann architecture. However, the IAS machine has far fewer data types, amuch weaker instruction set (especially in the area of program control), and essen-tially no instruction-level parallelism. The IAS lacks floating-point data formatsand instructions, so a much more complicated IAS program would be required tohandle double-precision, floating-point numbers comparable to those assumed inFigure 1.27. The IAS also lacks the following features of the PowerPC's instruc-tion set: indexed addressing modes; conditional branch instructions that can decre-ment and test a variable; and powerful arithmetic instructions such as multiply,divide, and multiply-and-add. Note also the vast differences in physical size, per-formance, and cost between the IAS and PowerPC.

1.3.3 System Architecture

We next review the overall organization of contemporary computer systems,including those formed by linking computers together into large networks.

Central

processing unit

CPU

Main

memory

M

Instructions

£

Data

Input-outputports
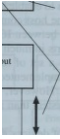
Stored programsand data

Svstem bus

Input-outputtransfers

Input-output devices

(keyboard, video display, secondary memories,

multimedia devices, etc.)

Figure 1.28

Overview of computer system operation.

CHAPTER 1Computing andComputers

Basic organization. A stand-alone computer system, which is most commonlyseen as a desktop machine (a PC or workstation) intended for a single user, has thebasic organization illustrated by Figure 1.28; see also Figure 1.22. This organiza-tion has changed little from that found in earlier generations, despite the massiveimprovements in implementation technologies that have occurred in recent years.The computer's main hardware components continue to be a CPU. a main memory,and an 10 subsystem, which communicate with one another over a system bus. Itsmain software component is an operating system that performs most system man-agement functions.

The key hardware element is a single-chip microprocessor, embodying a mod-ern version of the von Neumann architecture. The microprocessor serves as the com-puter's CPU and is responsible for fetching, decoding, and executing instructions.Data and instructions are typically composed of 32-bit words, which constitute thebasic information units processed by the computer. The CPU is characterized by aninstruction set containing up to 200 or so instruction types, which perform datatransfer, data processing, and program control operations that have changed littleover the years. The CPU may be augmented by on-chip or off-chip coprocessors thatimplement such specialized functions as managing the graphical user interface(GUI).

The role of the computer's main or primary memory M is to store programsand data as they are being processed by the CPU. M is a random-access memory(RAM) comprising a linear store of items (usually 8-bit bytes), each of which isassigned a unique address that permits the CPU to read or change (write) its con-tents via load or store instructions, respectively. M is backed up by a much largerbut slower secondary memory, typically implemented by hard disks employing

SECTION 1.3The VLSI Era

magnetic or optical storage technology and forming part of the 10 subsystem. As inthe PowerPC (Figure 1.25), an intermediate memory called a cache may also beinserted between the CPU and M. Thus we find a hierarchy of memory devicescomposed of the CPU's registers, the cache, the main memory, and the secondarymemory. This complex structure results from the /act that the fastest memorydevices are also the most costly. The memory hierarchy is intended to provide theCPU with fast access to large amounts of data at a fairly low cost.

The purpose of the 10 system is to enable a user to communicate with the com-puter. 10 devices are attached to the host computer by means of 10 ports, whosefunction is to control data transfers between 10 devices and main memory. Activeprograms communicate with IO ports in much the same way as they communicatewith M. An IO device is assigned a set of memory-like addresses, which allowinput and output instructions to be implemented in essentially the same way as loadand store instructions, respectively. However, the CPU usually takes much longerto access a word stored in the 10 system than to access a word stored in M—most10 operations are quite slow.

The traditional input and output devices are a keyboard and screen (providedby a CRT or a flat-panel display), respectively, which are convenient for handlingtextual information. Adding a pointing device like a mouse makes a display screeninto an input device, permitting communication between the user and the computervia graphical images. Special software, such as the Windows interface found inpersonal computers, supports GUIs. Audio interfaces for speech generation andrecognition extend the computer into a multimedia system. A major component ofmost 10 systems is a set of secondary memory devices that provide bulk storage ofprograms and data. Rapid transfer of information between primary and secondarymemories is often a key factor in a system's overall performance.

Microcontrollers. Their small size and low cost have made it feasible to useminiature general-purpose computers, referred to as microcontrollers, for tasks thatpreviously employed either special-purpose control circuits or had no control logicat all, for example, controlling a home washing machine or the ignition system of acar. Programs stored in a read-only memory (ROM) that forms a part of the mainmemory tailor a microcontroller to a particular application. The microcontroller isbuilt into, or embedded in, the controlled device, often in a way that is invisible tothe end user. Hence an embedded microcontroller that has been programmed tohandle the application in question can replace application-specific control circuits,often at substantial cost savings. Furthermore, by bringing the power of a computerto bear on relatively mundane applications, manufacturers can readily introducemany new features to improve flexibility, performance, or ease of use. As a result,most computers in operation today are microcontrollers in embedded systems.

Figure 1.29 shows one of the first applications of a microcontroller: a point-of-sale (POS) terminal that has replaced cash registers in retail stores. The microcon-troller has a conventional computer organization built around a system bus towhich are attached a microprocessor (the CPU), one or more ROM chips for pro-gram storage, and one or more RAM chips for data and working storage. All 10devices are also connected to the system bus using IO ports with standard inter-faces. The 10 devices in a typical POS terminal are a keyboard, a receipt printer, avisual display, a product-code scanner, and a credit-card reader. The latter is used

Centralcomputer

CPU(microprocessor)

H

RAM

E

ROM

Totelephonenetwork

10

port

ik

53

CHAPTER 1

Computing andComputers

10

port

Systembus

Microcontroller

10

port

n

10

port

10

port

Product-codescanner

Keyboard

10port

Printer

anddisplay

Credit-cardreader

Figure 1.29

A microcontroller-based point-of-sale terminal.

for credit authorization and requires a connection to the telephone system. Thefinal component is a link to a central computer used to provide pricing information,perform inventory control, and so forth.

Computer networks. The computer in Figure 1.29 is linked directly to a centralcomputer and indirectly to a potentially huge number of computers via the tele-phone network. The linking of computers to form networks of various types hasbecome an increasingly important feature of modern computing; see Figure 1.30. A

Servercomputer

Personalcomputer   D

[ ]   ( )   ( )   n

'   1 1   1 1   : :   i

c

Gatewaycomputer

Communication cables

Links to othercomputer networks
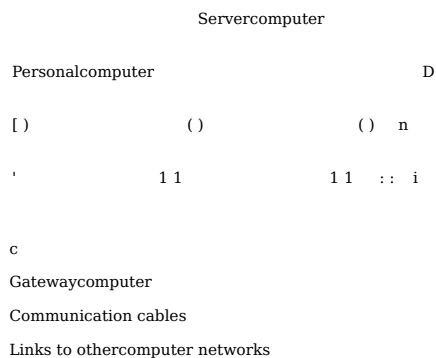
Figure 1.30

A local-area computer network.

54 computer in an office or industrial environment is typically linked to other comput-

„„_·..., , ers in the same organization via communication links that can be thought of as an

SECTION 1.3 °

The vlsi Era extension to the system bus. The linked computers then form a small, closed com-

puter network known as a local-area network (LAN) or intranet. The physicallinks between the computers can be built in various ways, including electricalcables, optical fibers, and radio (wireless) links. Special 10 programs (communica-tion software) enable the computers on the network to exchange information andaccess common computing resources called servers.

Computer networks have several advantages over the large, centralized (main-frame) computers that they have come to replace. The individual user has directaccess to a computer (his or her personal computer) that can quickly and conve-niently handle many routine computing tasks. Users can also access computingfacilities that they need less frequently, for example a high-performance supercom-puter or costly 10 equipment, via the computer network. Many widely dispersedusers can share such specialized equipment via the network, thus lowering its costto individual users. Furthermore, a computer network provides useful new servicessuch as electronic mail, remote library services, and on-line shopping.

Several LANs can be linked together by various means including the telephonenetworks, which increasingly are designed to accommodate digital data transmis-sion, including video data, as well as the traditional (digitized) voice communication.In Figure 1.30, one computer serves as a gateway device that manages communica-tion between the LAN and other computer networks. A collection of linked LANsforms a large computer network that can be worldwide in scope. In the early 1990sa network of this sort known as the Internet emerged, which because of its huge sizeand global reach—an estimated 16 million server sites in 180 countries with 72 mil-lion users in 1997—has had a profound impact on the way people compute and com-municate.

The Internet had its origins in a computer network called the ARPANET spon-sored by the Advanced Research Projects Agency of the U.S. Department ofDefense around 1970. This experimental network was originally designed to con-nect research institutions in the United States via leased lines; Figure 1.31 shows thestructure of the ARPANET at an early stage in its evolution (1972) when it linked26 research organizations in the United States. The ARPANET pioneered an infor-mation-transmission technique called packet switching, which divides both longand short messages into packets of fixed length that can be transmitted indepen-dently from source to destination via variable numbers of intermediate nodes. Eachnode contains a server that is responsible for sorting the packets from the variousmessages and forwarding them to the appropriate next destinations. Different pack-ages can be sent by different routes determined by the network traffic conditions. Atthe final destination, a message is reassembled from its constituent packets. Thecommunication software designed for the ARPANET and known as TCP/IP(Transmission Control Protocol/Internet Protocol) defines the communicationstandards for the Internet.

In the early years the Internet was used almost exclusively to transfer text filessuch as electronic mail (e-mail) messages. This situation changed fundamentally in1989 when scientists at CERN (Centre Europeen pour la Recherche Nucleaire) inGeneva overlaid on TCP/IP a new, high-level protocol called http {hypertext trans-port protocol) and an associated programming language html {hypertext markup

55
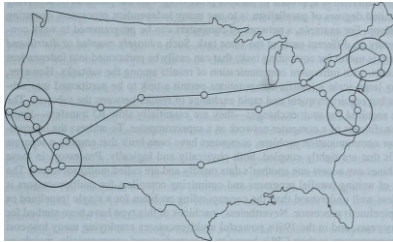
Figure 1.31

The ARPANET in 1972.

language) to permit the linking of diverse file types—text, still pictures, movies, sound, etc.—in an simple way. This combination enabled users to create multime-dia files easily and transmit them rapidly over the Internet. For example, using html, a text file can be tagged with commands that tell a computer where to find and insert visual images into the text file; the required image files can be located anywhere on the Internet. The human end user can access the information from a remote host via a simple point-and-click operation on a PC or workstation. The result is an enor-mously rich collection of easily accessible data that has come to be known as the World Wide Web.

Parallel processing. So-called supercomputers capable of executing many instructions in parallel have existed since the 1950s. Early commercial supercom-puters relied heavily on pipeline processing and had a single CPU organized around one or more multistage pipelines. This organization allows several instruc-tions to be in process simultaneously in each pipeline, resulting in a potential increase in performance of a factor of n per «-stage pipeline. The Cray-1 super-computer, first marketed by Cray Research Inc. in 1976, contained 12 pipeline pro-cessors for arithmetic-logic operations, several of which could operate in parallel [Russell 1978]. The Cray-1 could execute up to 160 million operations such as floating-point addition per second. Computers of this type have been most success-fully applied to scientific computations involving large amounts of vector and matrix calculations; consequently they are sometimes called vector processors. The degree of parallelism n possible with a pipeline is small, typically less than 10. As the PowerPC demonstrates (Example 1.7), pipeline processing of instructions is now a standard feature of microprocessors. Indeed, single-chip microprocessors reached the Cray-Fs level of performance in scientific computation in the mid-1990s.

56 An alternative approach to parallel processing with the potential of achieving

unlimited degrees of parallelism is to use many independent processors operating

Summary *n umson- F°r example, a network of computers can be programmed to work con-

currently on different parts of the same task. Such a loosely coupled or distributed system is useful for computing tasks that can easily be partitioned into independent subtasks, with infrequent communication of results among the subtasks. However, many large-scale scientific computations permit a task to be partitioned into sub-tasks but require frequent and rapid exchange of results between the subtasks. The time required for such exchanges—they are essentially slow 10 transfers—limits the usefulness of a computer network as a supercomputer. To address the interpro-cessor communication problem, computers have been built that employ n separate CPUs that are tightly coupled, both physically and logically. Processors in these machines can access one another's data rapidly and are called multiprocessors. The task of writing parallel programs and optimizing compilers for multiprocessors is far less well understood than the corresponding problem for a single (pipelined or nonpipelined) processor. Nevertheless, machines of this type have been studied for many years, and in the 1980s powerful multiprocessors employing many low-cost microprocessors as their CPUs began to be manufactured commercially, mainly as scientific computers.

Two types of multiprocessors are shared-memory and distributed-memory machines. In shared-memory machines all the processors have access to a common main memory through which they communicate to share programs and data. In distributed-memory machines each processor has only a private or local main memory and communicates with other processors by sending them messages through an 10 subsystem linking the processors. In each case a key issue is to design processor-to-memory or processor-to-processor interconnection networks that are of high-speed and reasonable cost. For small multiprocessors containing up to 30 or so processors, a fast bus can serve as an interconnection network. Ineffect, the basic organization of Figure 1.30 is used with multiple CPUs attached to a high-speed system bus. To construct massively parallel multiprocessors, that is, computers with hundreds or thousands of CPUs, various specialized interconnec-tion networks have been developed, which we will examine in Chapter 7. Mas-sively parallel multiprocessors are difficult to program and cannot run conventional (uniprocessor) programs efficiently. As a result, these machines have so far had a limited impact on the commercial computer marketplace.

1.4 SUMMARY

Humans have struggled with difficult computations since ancient times. Some of these problems are inherently unsolvable—they cannot be solved even in principle by a Turing machine, which is a simple, abstract, but completely general digital computer. Some theoretically solvable problems are intractable in that they cannot be solved within a reasonable amount of time by practical computers. However, given a suitable algorithm or solution method as well as a computer of sufficient power, many important problems can be satisfactorily solved. Designing practical computers that provide the highest possible performance at acceptable cost is the basic job of the computer architect.

The design of computing machines has evolved over a long period of time. Charles Babbage conceived the concept of a general-purpose, program-controlled computer in the mid-19th century. Such a machine was not completed until the 1940s, however, when the first electronic computers were successfully con-structed. Since then, progress has been dramatic, mainly driven by advances in computer hardware technology.

John von Neumann and others defined the basic organization of the modern computer. It comprises the following major components: a CPU responsible for fetching and executing instructions; a main memory used for instruction and data storage; and a set of input-output devices, such as user terminals, printers, and sec-ondary memory devices. Three main instruction types are found in every com-puter: data-transfer, data-processing, and program-control instructions. The instruction set and the way the instructions are processed define the power of a computer. The computer is typically programmed in a high-level language such as C++ or Java, which is automatically compiled into executable code (object pro-grams) built from its instruction set.

Integrated circuit technology has had a profound impact on computer design via the single-chip microprocessor and the high-capacity RAM chip. IC technol-ogy has enabled manufacturers to build very small, low-cost computers for gen-eral use (personal computers and workstations) as well as for special applications (embedded microcontrollers). IC technology has also been the driv-ing force in the proliferation of large-scale computer networks—the Internet, for example—and high-performance multiprocessors.

As the computer industry has matured, a few computer series have tended to become de facto architectural standards, notably IBM's System/360 mainframe family introduced in the 1960s and its PC personal computer family introduced in the 1980s. Recent computer families are distinguished by powerful RISC-style instruction sets and such performance-enhancing features as pipelining, instruction-level parallelism, and cache memories. Continuing advances in hard-ware and software technology, such as the introduction of multimedia comput-ing and the World Wide Web, suggest that major advances in computer design will continue into the foreseeable future.

57

1.5 PROBLEMS

1.1. To what extent does each of the following items play the role of processor and/or mem-ory when used in numerical computations: an abacus; a slide rule; an electronic pocket calculator?

1.2. Consider the Turing machine program of Figure 1.4, which adds two unary numbers n, and n2. A unary zero is represented by one or more blanks, which is an undesirable fea-ture of the unary system. Determine how the given Turing machine behaves (a) if n, =n2 = 0, that is, the initial tape is entirely blank; and (b) if/i, * 0 but ru = 0. In each case specify the final contents of the tape.

1.3. Design a a Turing machine that subtracts a unary number n: (rem another unary num-ber nx > n2. Assume that n,, n2, and the result n, - n: are stored in the formats described

58 in Example 1.1. That is, the tape initially contains only n, and n2 separated by a blank,

while the final tape should contain only n, - n2. Describe your machine by a program listing with comments, following the style used in Figure 1.4.

1.4. Construct a Turing machine program Countjup in the style of Figure 1.4 that incre-ments an arbitrary binary number by one. For example, if the number 10011 denoting 19 is initially on an otherwise blank tape, Count_up should replace it with 10100 de-noting 20. Assume that the read-write head starts and ends on the blank square imme-diately to the left of the number on the tape. Describe your machine by a program listing with comments, following the style used in Figure 1.4. [Hint: Fewer than 20 in-structions employing fewer than 10 states suffice for this problem.]

1.5. The number of possible sequences of moves (distinct games) in chess has been estimat-ed at around 10120. Is developing a surefire winning strategy for chess therefore an un-solvable problem?

1.6. Determine whether each of the following computational tasks is unsolvable. unde-cidable, or intractable. Explain your reasoning, (a) Determining the minimumamount of wire needed to connect any set of n points (wiring terminals) that are inspecified but arbitrary positions on a rectangular circuit board. Assume that at mosttwo wires may be attached to each terminal, (b) Solving the preceding wiring prob-lem when the n points and the wires that connect them are constrained to lie on theperiphery of the board; that is, the wire segments connecting the n points must lie ona fixed rectangle.

1.7. Most word-processing computer programs contain a spelling checker. An obviousbrute-force method to check the spelling of a word W is to search the entire on-line dic-tionary from beginning to end and compare W to every entry in the dictionary. Outlinea faster method to check spelling and compare its time complexity to that of the brute-force method.

1.8. Consider the four algorithms listed in Figure 1.7. With the given data, calculate themaximum problem size that each algorithm can handle on a computer M' that is 10,000times faster than M. Repeat the calculation for a computer M" that is 1,000.000 timesfaster than M.

1.9. The brute-force technique illustrated by the Euler-circuit algorithm in Example 1.2,which involves the enumeration and examination of all possible cases, is applicable tomany computing problems. To make the method tractable, problem-specific tech-niques are used to reduce the number of cases that need to be considered. For example,the eight-edge graph of Figure 1.6b can be simplified by replacing the edge-pair egwith a single edge because any Euler circuit that contains c must also contain g, andvice versa. Similarly, the pair dh can be replaced by a single edge. The problem thenreduces to checking for an Euler circuit in a six-edge graph. For the same problem, sug-gest another method that can sometimes substantially reduce the number of cases thatmust considered, illustrating it with a different graph example.

1.10. Consider the heuristic method to solve the traveling salesman problem discussed brief-ly in section 1.1.2. Construct a specific problem involving at most five cities, for whichthe total distance dhem traveled in the heuristic solution is not the minimum distancedmin. Conclude from your example (or from other considerations) that the heuristic so-lution can be made arbitrarily bad, that is, "worst case" problems can be contrived indmin can be made arbitrarily large.

1.11. Consider the computation of x2 by the method of differences covered in Example 1.3.Suppose we want to determine x2 forx = 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, that is, at intervalsof 0.5. Explain how to modify the method of Example 1.3 to accomplish this task.

1.12. Use the method of differences embodied in Babbage's Difference Engine to computex for integer values of * from 1 to 10.

1.13. Use the method of differences to compute x5, for integer values of x from 1 to 8. Whatis the smallest value of / for which the z'th difference of x5 is a constant? What is thevalue of that constant?

1.14. Consider the problem of computing a table of the natural logarithms of the integersfrom 1 to 200,000 to 19 decimal places, a task carried out manually in 1795. Select anymodern commercially available computer system with which you are familiar and es-timate the total time it would require to compute and print this table. Define all the pa-rameters used in your estimation.

1.15. Discuss the advantages and disadvantages of storing programs and data in the samememory (the stored program concept). Under what circumstances is it desirable to storeprograms and data in separate memories?

1.16. Computers with separate program and data memories implemented in RAMs andROMs, respectively, are sometimes called Hanard-class machines after the HarvardMark 1 computer. Computers with a single (RAM) memory for program and data stor-age are then called Princeton-class after the IAS computer. Most currently installedcomputers belong to one of these classes. Which one? Explain why the class you se-lected is the most widely used.

1.17. Write a program using the IAS computer's instruction set (Figure 1.14) to compute x2by means of the method of finite differences described in Example 1.3. For simplicity,assume that the numbers being processed are 40-bit integers and that the only data-processing instructions you may use are the IAS's add and subtract instructions. Theresultsx2, (x + l)2, (x + 2)2, ...,(x + k- l)2, should be stored in k consecutive memorylocations with starting address 3001.

1.18. A vector of 10 nonnegative numbers is stored in consecutive locations beginning in lo-cation 100 in the memory of the IAS computer. Using the instruction set of Figure 1.14.write a program that computes the address of the largest number in this array. If severallocations contain the largest number, specify the smallest address.

1.19. The designers of the IAS decided not to implement a square root instruction (ENIAChad one), citing the fact that y = xm can be computed iteratively—and very efficiently—via the following formula known in ancient Babylon:

>'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 Fig-ure 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

Location	Instruction	Comment
ADDEC:	LXI D, NUM1	+ 16 Initialize address: DE := NUM1 + 16.
	LXI H, NUM2	+ 16 Initialize address: HL := NUM2 + 16.
	MVI C, 16	Initialize count: C := 16.
LOOP:	LDAX D	Load data: D := M(DE).
	ADC M	A := A + CY + M(HL). Update CY flag.
	DAA	Convert sum in A to decimal.
	MOV M,A	Store data: M(HL) := A.
	DCX D	Decrement address: DE := DE - 1.
	DCX H	Decrement address: HL := HL - 1.
	DCR C	Decrement count: C := C - 1. Update Z flag.
	JNZ LOOP	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

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 f from A to B. If f maps element a of A onto element b of B, we write $b = f(a)$ 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 $V = \{v, \hat{}\hat{}, ...\hat{},,\}$ 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-

sented by circles, dots, or other symbols and edges are represented by lines: this diagram 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 signals between components. In modeling the system by a graph G, we associate C with the nodes of G and S with the edges of G; the resulting graph is often called a block diagram. This name comes from the fact that it is convenient to draw each node (component) as a block or box in which its name and/or its function can be written. 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 diagram representing a small gate-level logic circuit called an EXCLUSIVE-OR or modulo-2 adder. This circuit has the same general form as the more abstract graph of Fig-ure 2.1.

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 the structure of a system as the abstract graph consisting of its block diagram with no functional information. Thus Figure 2.1 shows the structure of the small system of Figure 2.2. A structural description merely names components and defines their interconnection. 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). We define 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 description for the logic circuit of Figure 2.2. This tabulation of all possible combinations of input-output values is called a truth table. Another description of the same EXCLUSIVE-OR behavior can be written in terms of mathematical equations as follows, noting that $/(a) =/(x,^c2)$:

$/(0,0) = 0$

$/(0,1)=1$

$/(1,0) = 1$

$/(U) = 0$
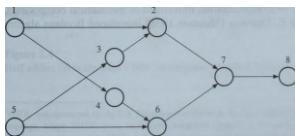
Figure 2.1

A graph with eight nodes and nine edges.

SECTION 2.1 System 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 are independent: neither can be derived from the other. The block diagram of Figure 2.2 serves as both a structural and behavioral description for the logic circuit in question, 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 being arithmetic-logic units or memory circuits. Such functional descriptions do not completely 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 we need a more precise description of system behavior, we generally supply a separate narrative 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 a system's structure and behavior by means of a block diagram—the term schematic diagram is also used—in which we identify the functions of the components. We can 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-level programming 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 Morrison 1961]. Babbage's notation, of which he was very proud, centered around the use of special 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 a2-bit 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 <= x xor y;

carry <= x and v;end behavior;

(a)

sumhalf_adder

Inputs Outputs

X v	sum carry
0 0	0 0
0 1	1 0
1 0	1 0
1 1	0 1

(b)

(c)

Figure 2.4

Half adder: (a) behavioral VHDL description; (b) block symbol; and (c) truth table.

'VHDL was sponsored by the U.S. Department of Defense. Its name stands for VHSIC hardware descriptionlanguage, where VHSIC (very high-speed integrated circuits) is the acronym of another Department ofDefense research program. VHDL is based on the programming language Ada. while Verilog. whose originsare industrial, is based on the C language. Both HDLs are now embodied in fonrul standards sponsored bythe Institute of Electrical and Electronics Engineers (IEEE).

SECTION 2.1

68 distinguished by the keywords in and out, respectively. The size of each 10 port,

meaning the number of signals associated with it, is specified here as 1 bit by the key-word bit. Thus we can conclude from the entity part of Figure 2.4a that half_adder

System esign nas tWQ i_l>it inputs, named x and y, and two 1-bit outputs, named sum and earn-. Fig-

ure 2.4b presents the same information in graphical form.lt is customary in such dia-grams to put inputs on the left and outputs on the right, eliminating the need forarrowheads to indicate the direction of signal flow.

The architecture part of a VHDL description specifies behavior and/or internalstructure. Figure 2.4a defines the half adder's behavior only; we are assuming for themoment that it is a primitive module or "black box," whose internal structure is eithernot known or not of interest. The functions of the half adder's two outputs sum andcarry are specified by two Boolean functions xor and and, which are built into VHDL;that is, they are predefined functions. In VHDL xor stands for the EXCLUSIVE-ORfunction, which we have encountered already—it is defined in Figure 2.3. The ANDfunction denoted by and is another basic logic function, which may be defined as fol-lows: AND(jc,v) = 1 if and only if x = 1 and y = 1. Note that VHDL expresses AND(jc,v)in the equivalent "infix" format x and y. An alternative description of the behavior ofhalf_adder appears in Figure 2.4c in the form of a truth table.

Figure 2.4a illustrates a few of the many notational conventions of VHDL, whichcollectively make the language quite complex. The symbol <= is called signal assign-ment and indicates that the value of the expression on the right of <= is assigned to thesignal on the left. Hence

carry <= x and y (2.1)

means that the signal carry is the AND function of x and y. This notation is equivalentto writing carry = AND(jc, v) in ordinary mathematical notation. The other features ofFigure 2.4a such as the use of begin-end to bracket related items represent minor syn-tactical details borrowed from programming languages.

VHDL is a rich language that can say the same thing in several ways. For exam-ple, we might replace (2.1) by

if xy - '11' then carry <= 1 else carry <= 0;

VHDL can also convey timing or performance information in various ways. For exam-ple, to indicate that it takes 5 ns for the carry signal to change in response to a changein its input signals x and y, we can rewrite statement (2.1) as

carry <= x and y after 5 ns;

If the half adder's internal structure is of interest, we can specify it by means of astructural architecture description, as shown in Figure 2.5a. The same structure isdefined by the block diagram of Figure 2.5b. Again inputs are assumed to be on the leftand outputs on the right. Two internal component types are identified and are describedby VHDL component statements that have much the same form as entity. They namethe component types {xor_circuit and nand_gate in the example) and specify the namesand types of the components' 10 signals. Internal signals (lines or buses) created byconnections between the components are specified by a signal statement, in this case a1-bit internal signal named alpha. Finally, all the copies of each component used in thecircuit are individually named and their 10 connections are specified. This is accom-plished by the part of the architecture description in Figure 2.5a bracketed by begin-end, which may be thought of as a (wiring) network specification or netlist. There isone copy named XOR of xor_circuix and two copies of nand_gate named NAND1 andNAND2. The second line in this netlist

NAND1: nand_gate port map (d => x, e => >,/=> alpha);

entity half_adder is

port (x,y: in bit; sum, carry: out bit);end half_adder;

architecture structure of half_adder is

component xor_circuit port (a,b: in bit; c: out bit); end component;

component nand_gate port (d,e: in bit;/: out bit); end component;

signal alpha: bit;begin

XOR: xor_circuit port map (a => x, b => y, c => sum);

NAND1: nand_gate port map (d => *, e => >',/=> 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 sat-isfy 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
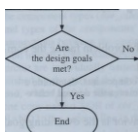
Modify the designto meet the goals

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

Level	Components	It density units	Information	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 5

to *
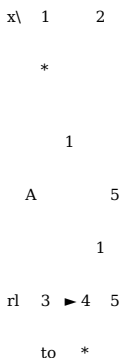
(a)

Figure 2.8

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 hardware and software design. If the foregoing system is to be designed using medium-scale ICs 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 depends on the designers' skill and experience. In the following sections we examine design at the register and processor levels in detail, beginning with the better-understood register level. We assume that the reader is familiar with binary numbers and with gate-level design concepts [Armstrong and Gray 1993; Hayes 1993; Hachtel and Somenzi 1996], which we review in the next section.

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 components are 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 a Boolean function, is a mapping from the set of 2" input combinations of n binary variables onto the output values 0 and 1. Such a function is denoted by r(.v,. v:

SECTION 2.1 System Design

xn) or simply by z. The function z can be defined by a truth table, which specifies for 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, $s_0(x_q, v'o\ c_-)$, and $c_0(x_q, V_q, c_-)$, which are the sum and carry outputs, respectively, of a logic circuit called a full adder. This useful logic circuit computes the numerical sum 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 that the sum of three Is is $CqS0 = 11_2$, that is, the base-2 representation of the number three. When discussing logic circuits, we will normally reserve the plus symbol (+) for the logical OR operation, and write out plus for numerical addition. We will also use a subscript to identify the number base when it is not clear from the con-text; for example, twelve is denoted by 12_{10} in decimal and by 1100_2 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 0	1	0	1	
0 1	0	0	1	
0 1	1	1	0	
1 0	0	0	1	
1 0	1	1	0	
1 11 1	0	1	0	
	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 <
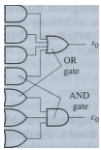
(c)

NOT gate(inverter)
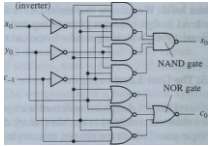
id)

Figure 2.9

Full adder: (a) truth table; (b) realization using half adders: (c) realization using AND andOR gates; (d) realization using NAND, NOR. and NOT gates.

EXCLUSIVE-OR, NOT (inverter), NAND, and NOR. The functions performed byAND, OR, EXCLUSIVE-OR, and NOT gates are denoted by logic expressions ofthe form $x\{x2$, $xx + x2$, $xx © x2$, and xx, respectively, and are defined as follows:

AND: $xxx2 = 1$ if and only if $Xj$and$^$are both 1.OR: $jcj + x2 = 1$ if and only if xx or $x2$ or both are 1.EXCLUSIVE-OR: $x\{ © x2 = 1$ if and only if xx ov$x2$ but not both are 1.NOT: $xx = 1$ if and only if $xl = 0$.

The function performed by a NOT gate is known as inversion. The NAND orNOR functions are obtained by inverting AND and OR, respectively. NAND isdenoted by $xxx2$ and NOR by $xx + x2$. The preceding definitions (except that ofNOT) can be extended to gates with any number of inputs k, but practical consid-erations limit k, which is called the gate's fan-in, to a maximum value of 10 or so.Note that the NOT gate or inverter can be regarded as a one-input version ofNAND or NOR.

A set G of gate types is said to be (functionally) complete if any logic functioncan be realized by a circuit that contains gates from G only. Examples of completesets of gates are {AND, OR, NOT}, {AND, NOT}, {NAND}, and {NOR}.NANDs and NORs are particularly important in logic design because they are eas-ily manufactured using most IC technologies and are the only standard gate typesthat are functionally complete by themselves. With any complete set of logic oper-ations, the set of all logic functions of up to n variables forms a Boolean algebra,named after George Boole (1815-1864), a contemporary of Babbage's [Brown1990]. Boolean algebra allows the function realized by a combinational circuit tobe described in a form that resembles the circuit's structure. It is similar to ordinary(numerical) algebra in many respects, and both numerical and Boolean algebra areembedded in the syntax of a typical HDL.

Figure 2.9b shows a possible gate-level realization of a full adder that employstwo copies of the half adder defined in Figures 2.4 and 2.5 along with a single ORgate. Here we use standard, distinctively shaped symbols for the various gate typesinstead of the generic box symbols of Figure 2.5b. Observe that the two NANDs ineach half adder, one of which is used as an inverter, can be replaced by a single,functionally equivalent AND gate. This equivalence is seen from the fact that theinversions associated with the two NANDs cancel; in algebraic terms, $ab = ab$.

Two alternative gate-level designs for the full adder appear in Figures 2.9cand 2.9d. The AND-OR circuit of Figure 2.9c is defined by the logic (Boolean)equations

(2.3)

(24)

$so = -*o>'oc-i + Vo^'-i + *o>'oc-i + *o?oC-ic0 = (*0 + c_\{)(xQ + y0)(\'() + c\_i)$

whose structure also corresponds closely to that of the circuit. By analogy withordinary algebra, (2.3) and (2.4) are referred to as sum-of-prochuts (SOP) andproduct-of-sums (POS) expressions, respectively. The circuit of Figure 2.9c iscalled a two-level or depth-two logic circuit because there are only two gates, oneAND and one OR, along each path from this adder's external or primary inputs v,.,$y0$, $c\_$, to its primary outputs ,$v0$. $c()$, assuming each primary input variable is avail-able in both true and inverted (complemented) form. The number of logic levels isdefined by the number of gates along the circuit's longest 10 path. Because each

75

CHAPTER 2

Design

Methodology

76 gate imposes some delay (typically 1 ns or so) on every signal that propagates

through it, the fewer the logic levels, the faster the circuit.

s D The half-adder-based circuit of Figure 2.9b has 10 paths containing up to four

gates and so is considered to have four levels of logic. If all gates have the samepropagation delay, then the two-level adder (Figure 2.9c) is twice as fast as thefour-level design (Figure 2.9b). However, the two-level adder has more gates andso has a higher hardware cost. A basic task in logic design is to synthesize a gate-level circuit realization of a given set of combinational functions that achieves asatisfactory balance between hardware cost as measured by the number of gates,and operating speed as measured by the number of logic levels used. Often thetypes of gates that may be used are restricted by IC technology considerations, forexample, to NAND gates with five or fewer inputs per gate. The design of Figure2.9d, which has essentially the same structure as that of Figure 2.9c, uses NANDand NOR gates instead of ANDs and ORs. In this particular case the primaryinputs are provided in true (noninverted) form $jc0$, $y0$, $c\_$, only; hence inverters areintroduced to generate the inverted inputs $x0$, $y0$, c_x.

Computer-aided synthesis tools are available to design circuits like those ofFigure 2.9 automatically. The input to such a logic synthesizer is a specification ofthe desired

function, such as a truth table like Figure 2.9a, or a set of logic equa-tions like (2.3) or (2.4); these are often embedded in a behavioral HDL description.Also given to the synthesizer are such design constraints as the gate types to useand restrictions on the circuit's interconnection structure. One such restriction is anupper bound on the number of inputs (fan-in) of a gate G. Another is an upperbound on the number of inputs of other gates to which G's output line may con-nect; this is called the (maximum) fan-out of G. The output of the synthesizer is astructural description of a logic circuit that implements the desired function andmeets the specified constraints as closely as possible.

Exact methods for designing two-level circuits like that of Figure 2.9c (or Fig-ure 2.9<i with its inverters removed) using the minimum number of gates have longbeen known. They are computationally complex, however—gate minimizationfalls into the class of intractable problems discussed in section 1.1.2—so they areonly practical for designing small circuits. However, practical heuristic methodsfor synthesizing two-level and multilevel logic circuits that are often nearly opti-mal are known and implemented in CAD programs (see Example 2.2). Once agood design of a useful function is known, it can be placed in a library for futureuse. A full adder, for instance, can be used to build a multibit, multilevel adder, asshown in Figure 2.10a.[2] This circuit adds two 4-bit numbers $X = (x_{-i}, x_2, x_x, x_Q)$ and $Y = (>'3, >, 2'>'i'>'o)$ and computes their sum $S = (s_3, s_2, S\setminus, s_0 y$, it also accepts an inputcarry signal c_-, and produces an output carry c_3. 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

[2]This 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
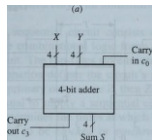
(b)

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   \ /

0 1   1   1   1   1 /

0   II

0   Time ►■
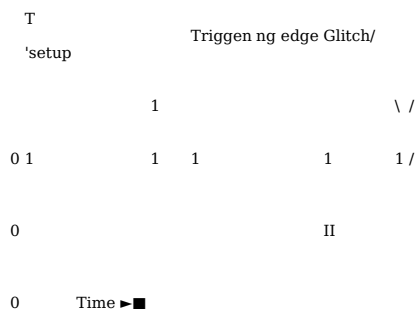
Clock CKDataD

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, $D(3) = 0$ during the critical setup phase of cycle 3, thus ensuring that $y(4) = 0$. Observe that the 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 signals appearing 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 S,(y=l)	s0.i	S,,0		5,.0 5,.l

Next Presentstate output

(a)

Figure 2.12

(a) State table; (b) logic circuit for a serial adder.
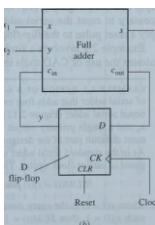
D

nip-Hop

Clock

80 adder of Figure 2.10 is a "parallel" adder, which, ignoring its internal-signal propa-

gation 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

$c(i)z(i) = x^\wedge Oplus x2(i)plus c(i - 1)$ (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 two-state state table of Figure 2.12a. An entry in row 5(0 and column $x^\wedge x^\wedge i)$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 S_0 and the present primary input combination is $x_1(i)x_2(i)$; $z(i)$ is the corresponding primary output signal that must be generated.

Because the serial adder has only two internal states, its memory consists of a single flip-flop storing a state variable y. There are only two possible ways to assign 0s and 1s to y. We select the "natural" state assignment that has $y = 0$ for S_0 and $y = 1$ for S_x, since this equates $y(i)$ with the stored carry signal $c(i - 1)$. Assume that we use an edge-triggered D flip-flop (Figure 2.11) to store y. The combinational logic C then must generate two signals: the primary output $z(i)$ and a secondary output signal $D(i)$ that is applied to the D flip-flop's data input. The flip-flop's behavior is defined by its characteristic equation (2.5); that is, $y(i + 1) = D(i)$. Hence we have

$$D(i) = c(i)$$

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 carry output is D; see Figure 2.12b. Before entering two new numbers to be added, it is necessary to reset the serial adder to the S_0 state. The easiest way to do so is to apply a reset pulse to the flip-flop's asynchronous clear (CLR) input.

Example 2.2 involves a similar, but more complex sequential circuit and demonstrates the use of CAD tools in its design.

example 2.2 design of a 4-bit-stream serial adder. Consider another type of serial adder that adds four number streams instead of the two handled by a conventional serial adder (Figure 2.12). The new adder has four primary input lines x_1, x_2, x_3, x_4 and a single primary output z. To determine the circuit's state behavior—often the most 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

$$SUM(i) = x_1(i) \text{ plus } x_2(i) \text{ plus } x_3(i) \text{ plus } x_4(i) \text{ plus } c(i-1)$$

where $c(i-1)$ is the carry computed in the preceding clock cycle. If $c(i-1)$ is 0 and each $x(i) = 1$, then $SUM(i) = 1$ plus 1 plus 1 plus 1 plus $0 = 4 = 100_2$, so $c(i) = 10_2$. With $c(i-1) = 10_2$, $SUM(i)$ becomes $6 = 110_2$, making $c(i) = 11_2$. Finally, $c(i-1) = 11_2$ makes $SUM(i) = 111_2$ and $c(i) = 11_2$, which is the maximum possible value of c. The carry data to be stored is a binary number ranging from 00_2 to 11_2, which implies

that the adder needs four states and two flip-flops. We will denote the four states by S_0, S_1, S_2, S_3, where S_i 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 S_k, z is obtained by adding $i2$ and the 4 input bits that determine j to form $SUM(i) = (k_2k_1k_0)_2$. It follows that $k = (k_2k_1)_2$ and $z = k_0$. For example, with present state S_2 and present input 7, $SUM(i) = 0$ plus 1 plus 1 plus 1 plus $10_2 = 101_2$, so $z = 1$ and $k = 10_2 = 2$, making S_2 the next-state. Following this pattern, it is straightforward to construct the adder's state table. With D flip-flops, the next-state values $y_1(i + 1)y_2(i + 1)$ coincide with the flip-flops' data input values $D_1(i)D_2(i)$. The adder thus has the general structure shown in Figure 2.13b.

A truth table for the combinational logic C appears in Figure 2.13c. It is derived directly from Figure 2.13a with the states assigned the four bit patterns of y_1y_2 as follows: $S_0 = 00$, $S_1 = 01$, $S_2 = 10$, and $S_3 = 11$. Suppose we want to design C as a two-level

il

CHAPTER 2

Design

Methodology

		Present inputs $x_1x_2x_3x_4$ (decimal)					
	0 1 2	3	4 5 6 7 8 9 10	11	12 13	14 15	
	$S_0,0$ $S_0,1$ $S_0,1$ $S_0,0$	$S_0,1$ $S_0,0$ $S_0,0$ $S_0,1$	$S_0,1$ $S_1,0$ $S_0,0$	$S_1,1$	$S_1,0$ $S_1,1$	$S_1,1$ $S_2,0$	So
Present	$S_0,1$ $S_0,0$ $S_0,0$	$S_1,1$ $S_1,0$ $S_1,1$ $S_1,1$ $S_2,0$ $S_0,0$ $S_1,1$ $S_1,1$	$S_2,0$ $S_1,1$ $S_2,0$ $S_2,0$ $S_2,1$				S_1
state S->	$S_1,0$ $S_1,1$ $S_1,1$	$S_2,0$ $S_1,1$ $S_2,0$ $S_2,0$ $S_2,1$ $S_1,1$ $S_2,0$ $S_2,0$ $S_2,1$	$S_2,0$ $S_2,1$ $S_2,1$ $S_3,0$				S_2
	$S_1,1$ $S_2,0$ $S_2,0$	$S_2,1$ $S_2,0$ $S_2,1$ $S_2,1$ $S_3,0$ $S_2,0$ $S_2,1$ $S_2,1$ $S_3,0$ $S_2,1$ $S_3,0$ $S_3,0$ $S_3,1$					S_3

(a)

Combinational logic C

CK<

CK<

Dy

Reset Clock

Present inputs					Present state		Secondary outputs		Primary output
x_1	x_2	x_3	x_4	y_1	y_2	D_1 D_2			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 1 0				0 0	0 0			1
59	1 1 1 0				1 1	1 1			0

60 1 1 1 1   0 0   1 0   0

61 1111   0 1   1 0   1

62 1 1 1 1   1 0   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 four-streamadder 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 conver-

Counters.

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
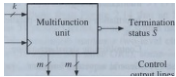
Z, Z2

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
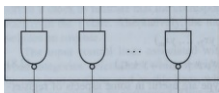
X Y

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 of several sources to a common destination; the source is specified by applying appropriate control (select) signals to the multiplexer. If the maximum number of data sources is k and each 10 data line carries m bits, the multiplexer is referred to as a k-input (or k-way), m-bit multiplexer. It is convenient to make k = 2P, so that data 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 symbol of Figure 2.16; the tapered block symbol shown in Figure 2.19, where the narrow end indicates the data 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-bit multiplexer is therefore defined by m sum-of-product Boolean equations of the form

$Zj= (x0ja0 + xljal+ ••• +x2p\_i ja2p\_i)e$ 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 by connecting 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 associated with every level of the tree and is connected to all multiplexers in that level. Thus each level performs a partial selection of the data line X, to be connected to the output 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 can generate 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_lof MUX, and 2" 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 %2P-

i- 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 2.2 The Register Level

Select s

Enable e
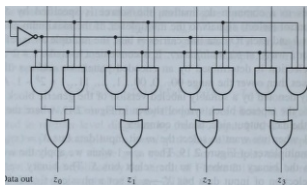
Data out z0

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 input variables v0,

$v^, ..., v_n$. With e= 1, setting $x_t = 1$ (0) if row i of the truth table for z is 1 (0) makes (2.10) into a sum-of-products expression for z. Hence by connecting each input data line to the appropriate logic value 0 or 1, we can realize any of the 2 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 x_0, y_0, and $c_]$ (the carry in) to obtain a 2-bit result consisting of s_0 (the sum bit) and c_0 (the carryout). It is the basic component of a serial adder (Figure 2.12) and has various gate-level realizations 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 might be expected, to the multiplexer's data input buses. Instead constant values 0 or 1 are applied to the data inputs as indicated. Each pattern i of $x^qC^$ selects a specific input data bus X, and routes its 2-bit word to the output bus z = s_0c_0. Observe how this proce-dure effectively maps the truth table for s_0 and c_0 (Figure 2.22a) directly onto M£/X,'s input 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 x_0, y_0 are applied to M£/X2's select lines as before, but we apply one of c_x, c_1? 0, or 1 to each line X_q of data bus X,. Now X_{jj} must realize two rows of the form x$>00 and x^qI in the adder's truth table. If, for example, these rows have the same fixed value a for the output (s_0 or c_0) of interest, then we apply a to $x^$-. If the rows have different values, then either $c_,$ or $c_,$ is applied

89

CHAPTER 2

Design

Methodology

*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)
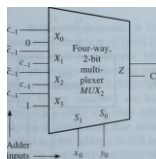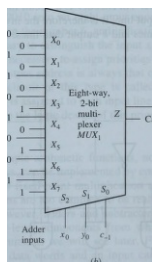
Sum s0Carry c0

Sum sQ— Carry c0

(c)

Figure 2.22

Multiplexer-based full adder: (a) truth table; (b) first version; (c) second version.

90

SECTION 2.2 The Register Level

to X_n, as appropriate. We see from this example that a 2"-input, m-bit multiplexer can realize any (n + 1)-variable, w-output logic function.

Decoders. A l-out-of-2" or 1/2" decoder is a combinational circuit with n input 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 a 1/4 decoder. Several 1/2" decoders can be used to decode more than n lines by connecting them in a tree configuration analogous to the multiplexer tree of Figure 2.21. The main application of decoders is address decoding, where A, is interpreted as an address that selects a specific output line Z; or some circuit attached to z,. For example, decoders are used in RAMs to select storage cells to be read from or written 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 the control input e (enable) of the decoder is viewed as a 1-bit data source to be routed to one of 2" destinations, as determined by the address applied to the decoder. Thus a 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 a four-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 of an active input line; it is therefore the inverse of a decoder. Most encoders have 2 input data lines and k output data lines. For example, when k = 3, entering a data

Enable e
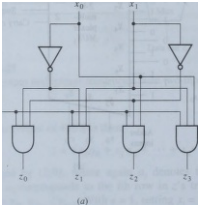
1/4 decoder

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.

CHAPTER 2

Design

Methodology

pattern such as $x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_1 = 00000010$ into an eight-input encoder should produce the response $z_2 Z_1 Z_q = 110$, denoting the number 6, and indicating that $x_6 = 1$. Additional (control) outputs are necessary to distinguish the input x_0 active and no input active states. Moreover, it is also necessary to assign priorities to the input lines and design the encoder so that the output address is always that of the active input 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 x_i has higher priority than x_j if $i > j$. We leave the logic design of this priority encoder as 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 combinational register-level components. Most forms of fixed-point multiplication and division and 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 can derive a variety of other arithmetic circuits, as we will see later. Figure 2.26a shows a component that adds two 4-bit data words and an input carry bit: it is called a 4-bit adder. (A full adder is sometimes called a 1-bit adder.) The adder's carry-in and carry-out lines allow several copies of this component to be chained together to add numbers of arbitrary size; note, however, that the addition time increases with the number size. (See Chapter 4 for coverage of the design of adders and more-complex arithmetic circuits). Another useful arithmetic component is a magnitude comparator, whose function is to compare the magnitudes of* two binary numbers. Figure 2.26b shows the overall structure of a 4-bit comparator.

SECTION 2.2 The 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
1	d rf rf d d				d d 1			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	1 0		
1	</ 1 0 0 0				0 0 0			0	0 1		
1	1 0 0 0 0				0 0 0			0	0 0		

(a)

Figure 2.25

An 8-input priority encoder: (a) truth table; (b) symbol.
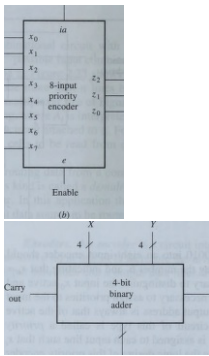
Carry

Sum Z

(a)

X4 ,'

Y

4 /

4-bit magnitude comparator

X<Y X=Y X>Y

(b)

Figure 2.26

Symbols for (a) a 4-bit parallel adder; (b) a 4-bit magnitude comparator.

Magnitude comparators are relatively complex circuits requiring either many gates or many logic levels.

EXAMPLE 2.4 DESIGN OF A 4-BIT MAGNITUDE COMPARATOR. Consider the internal design of the magnitude comparator depicted in Figure 2.26b. It has eight input lines, implying that its truth table has $2^8 = 256$ rows. The comparator is quite difficult to design at the gate level. Furthermore, a two-level (SOP or POS) realization is impractical because of the many gates involved, as well as their large fan-in.

We can design a magnitude comparator for two n-bit numbers X and Y efficiently at the register level by noting that X > Y is equivalent to

$$X - \bar{Y} > 0$$

(2.11)

Now $\bar{Y}$ can be computed by the subtraction step $(2^n - 1) - Y$, where $\bar{Y}$ is the bitwise complement of Y and $2^n - 1$ is a sequence of n 1s. For example, if n = 4 and Y = 1001(9), then $\bar{Y}$ = 0110(6), $2^4 - 1$ = 1111 (15), and $\bar{Y}$= 1111 - 0110 = 1001. Hence inequality (2.11) can be replaced by $X - (2^n - 1 - Y) > 0$, implying

$$X + Y > 2^n - 1 = 11\ldots 1$$ (2.12)

Now suppose we add X and Y using an adder such as that of Figure 2.26a. If the inequality of (2.12) is satisfied, then the adder's carry-out signal cout will be 1, because X + Y will exceed the largest n-bit number $2^n - 1$. In the preceding example with X = 1100(12) and Y = 1001 (9), we have X + Y = 1100 + 0110 = 10010 (18), for which the output carry is 1. We can therefore perform the original magnitude test X > Y as follows:

1. Compute $\bar{Y}$ from Y using an n-bit word inverter.

2. Add X and $\bar{Y}$ via an n-bit adder and use the output-carry signal cout as the primary output. If cout = 1, then X > Y; if cout = 0, then X < Y.

Figure 2.27 shows a direct realization of the above scheme to implement $z_3 = (X > Y)$ for the 4-bit case. By switching X and Y, we can generate $Z_1 = (X < Y)$ in exactly the same manner. We do not need the sum outputs of the two adder modules; hence we can discard them and their associated circuits, thereby reducing the adders to carry-generation circuits.

We have yet to compute the "equals" output denoted $z_2 = (X = Y)$. This calculation requires comparing each bit X_i of X to the corresponding bit Y_i of Y, which can be done by an EXCLUSIVE-NOR gate that produces $X_i \odot Y_i$. Now $z_2 = 1$ when $X_i \odot Y_i = 1$ for all i; that is,

$$z_2 = (x_3 \odot y_3)(x_2 \odot y_2)(x_1 \odot y_1)(x_0 \odot y_0)$$

(2.13)

Figure 2.27 also gives a 4-bit implementation of (2.13) using EXCLUSIVE-NOR and AND word gates. Practical magnitude comparators such as the 74X85 [Texas Instruments 1988] use a similar design that incorporates a fast carry-generation technique (carry lookahead).

93

CHAPTER 2

Design

Methodology
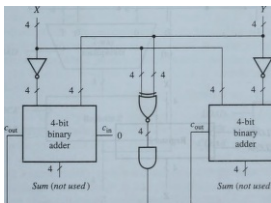
4-bit

binary — 0

adder

Sum (not used)

$Z_1(X<Y)$ $z_2(X = Y)$ $Z_3(X>Y)$

Figure 2.27

Register-level design of a 4-bit magnitude comparator.

94

We turn now to the main sequential components used at the register level.

Registers. An m-bit register is an ordered set of m flip-flops designed to storean /n-bit word (zq,Z\,...,z,,_ \). Each bit of the word is stored in a separate flip-flop,but the flip-flops have common control lines (clock, clear, and so on). Registerscan be constructed from various flip-flop types. Figure 2.28a shows a 4-bit registerconstructed from four D flip-flops, and Figure 2.286 shows a suitable graphic sym-bol for it. The register and its output signal (which denotes the register's state) arefrequently assigned the same name.

The register Z of Figure 2.28 reads in the data word X each time it is clocked.Therefore, to maintain the contents or state of Z at a constant value, it is necessaryto apply that value continuously to Z's input bus. Often we want to load a newvalue of X into Z in a particular clock cycle and subsequently change X withoutchanging Z. To this end, we introduce a control line LOAD, which should cause theregister to read in (load) the current value of X when it is clocked and LOAD hasbeen set to 1. When LOAD = 0, the state of Z should not change when the registeris clocked; it should retain the last value loaded into it. To add this load feature toregister Z of Figure 2.28, we insert a two-input, 4-bit multiplexer MUX into itsinput data bus as shown in Figure 2.29a. The new control line LOAD is connected

CLOCKCLEAR
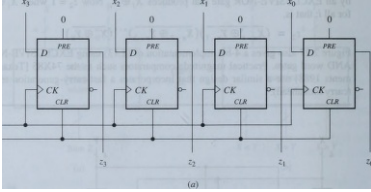
CLOCKCLEAR

X

4 /

Register Z

4 /

Z

(b)

Figure 2.28

A 4-bit D register with parallel 10: (a) logic diagram; (b) symbol.

to MUX's select line s. MUX's data input lines are connected to X and to the regis-ter output Z so that the circuit behaves as follows in each clock cycle. If LOAD = 1,then X is loaded into the register from the input bus: that is. Z := X. If LOAD = 0,then the old value of Z is loaded back into the register; that is, Z := Z.

Registers like those of Figures 2.28 and 2.29 are designed so that external datacan be transferred to or from all its flip-flops simultaneously; this mode of opera-tion is called parallel input-output. In some computer-design situations it is usefulto transfer (shift) the contents of a register in and out 1 bit at a time. A registerdesigned for such operations is a shift register. A right-shift operation changes theregister's state as described by the following register-transfer statement:

(X'Zm-\'Zm-2'---'Z0 '■- (^m-l'Zm-2'---'2l'Zo)

A left shift performs the similar transformation:

(zm-2>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
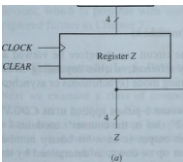
X

4 /

LOADCLOCKCLEAR

> Register Z

Z(*)

Figure 2.29

A 4-bit D register with parallel load: (a) logic diagram; (b) symbol.

96

SHIFTCLEAR
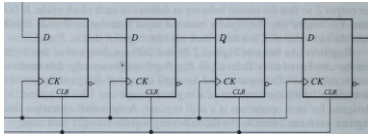
(a)

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 binary numbers, because left- (right-) shifting corresponds to multiplication (division) by two. 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 $S_0, S_1, ..., S_{k-1}$ in response to signals (1 -pulses) on an input line. The k states represent k consecutive numbers, so the state transitions can be described by the statement

$$S_{i+1} := S_i \text{ plus } 1 \text{ (modulo } k)$$

Each 1-input increments the state by one; the circuit can therefore be viewed as counting the input Is. Counters come in many different varieties depending on the number 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^n; that is, the counter's modulus k = 2^n, and it has 2^n states $S_0, S_1, \ldots, S_{2^n-1}$.

The output is an n-bit binary number

COUNT = S_j, and the count sequence is either up or down, as determined by the control line DOWN. In the up-counting mode (DOWN= 0), the counter's behavior is

$$S_{i+1} := S_i \text{ plus } 1 \text{ (modulo } 2^n)$$

COUNT ENABLE CLEAR DOWN

Modulo-2^n up-down counter

COUNT

Figure 2.31

A modulo-2^n up-down counter.

97

whereas in the down-counting mode (DOWN = 1), the behavior becomes $S_{i+1} := S_i$ minus 1 (modulo 2^n)

In some counters modulus-select control lines can alter the modulus; such counters are termed programmable.

Counters have several applications in computer design. They can store the state of a control unit, as in a program counter. Incrementing a counter provides an efficient 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 word from a specified source to a specified destination on the same or a different IC; the source 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 with them, since they require logic circuits to control access to them and, when used over longer distances, signal amplification circuits (drivers and receivers). The pin requirements and gate density of an IC increase rapidly with the number of external buses connected to it. If these buses are long, the cost of the wires or cables used must also be taken into account.

To reduce costs, buses are often shared, especially when they connect many devices. 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 requires more complex bus-control mechanisms. Although shared buses are relatively cheap, they do not permit simultaneous transfers between different pairs of devices, which is possible with unshared or dedicated buses. Bus structures are explored further in Chapter 7.

2.2.2 Programmable Logic Devices

Next we examine a class of components called programmable logic devices or PLDs, a term applied to ICs containing many gates or other general-purpose cells whose interconnections can be configured or "programmed" to implement any desired combinational or sequential function [Alford 1989]. PLDs are relatively easy to design and inexpensive to manufacture. They constitute a key technology for building application-specific integrated circuits (ASICs). Two techniques, are used to program PLDs: mask programming, which requires a few special steps in

98

the IC chip-manufacturing process, and field programming, which is done by designers or end users "in the field" via small, low-cost programming units. Some field-programmable PLDs are erasable, implying that the same IC can be repro-grammed many times. This technology is especially convenient when developing and debugging a prototype design for a new product.

Programmable arrays. The connections leading to and from logic elements in a PLD contain transistor switches that can be programmed to be permanently switched on or switched off. These switches are laid out in two-dimensional arrays so that large gates can be implemented with minimum IC area. The programmable logic 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 of an x means that the corresponding connection has been programmed to the off (dis-connected) state.

The gate structures of Figure 2.32b can be combined in various ways to imple-ment logic functions. The programmable logic array (PLA) shown in Figure 2.33 is 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 form various 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 is possible 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 output D_2. The last row is programmed to generate $x_2 x_3 x_4 y_2$, which is used in computing the output D_2, the last row is programmed to generate $\overline{x_1} x_2 \overline{y}_x$ for output D_1. The inputs to the OR gates are also programma-ble, so each output column can include any subset of the product terms produced by 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

*2

•*3

X4

Data in

D, D2

Data out

Figure 2.33

PL A implementing the combinational part C of the adder of Figure 2.13.

CHAPTER 2

Design

Methodology

Closely related to a PLA is a read-only memory (ROM) that generates all 2^n possible rc-variable product terms (minterms) in its AND plane. This enables each output column of the OR plane to realize any desired function of n or fewer vari-ables in sum-of-minterms form. Unlike a PLA, the AND plane is fixed; the pro-gramming that determines the functions generated by a ROM is confined to the OR plane. A small ROM with three input variables, $2^3 = 8$ rows, and two output col-umns is shown in Figure 2.34/?. It has been programmed to realize the full-adder function defined by Figure 2.34a—compare the multiplexer realizations of the full adder appearing in Figure 2.22. Note the use of dots to denote the fixed connec-tions in the AND plane. This particular ROM can be programmed to realize any two of the 256 Boolean functions of three or fewer variables. Field-programmable ROMs are known as PROMs (programmable ROMs).

PLAs and ROMs are universal function generators capable of realizing a set of logic functions that depend on some maximum number of variables. They are two-level logic circuits in which the lines can have large fan-out and the gates (espe-cially the output gates) can have large fan-in. High fan-in and fan-out tends to make these circuits' propagation delays quite high, however. A ROM is a memory device only in the sense that its OR plane "stores" the 2^n data words that have been programmed into it. A stored word is read out each time the ROM receives a new input combination or address. The AND plane therefore serves as a l-ouf-ol-2^n address decoder.

SECTION 2.2 The Register Level

Inputs			Outputs	
x0	>'o	C-l	s0	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    0  0 1
11
    1    1  1 1
```

(a)
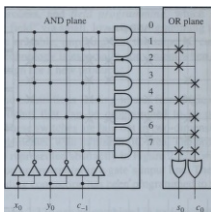
(b)

Figure 2.34

ROM implementation of a full adder: (a) truth table; (b) ROM array.

Comparing Figures 2.34a and 2.34b, we see that a ROM effectively stores the entire truth table of the functions it generates. Consequently, the effort needed to design a ROM is trivial. The process of reading the stored information from a ROM is referred to as table lookup. Read-only memories are suitable for imple-menting circuits whose 10 functions are difficult to specify in logical terms; some code conversion and arithmetic circuits are of this type. The usefulness of ROMs is limited by the fact that their size doubles with each new primary input variable. Unlike a ROM, a PLA stores a condensed (minimized) form of the truth table and so generally occupies much less chip area than an equivalent ROM.

Many variants of the preceding PLD types exist [Alford 1989]. Registered PLAs have flip-flops attached via programmable connections to the outputs of the OR plane, allowing a single IC to implement medium-sized sequential circuits. Programmable array logic (PAL) circuits have an AND plane that is programma-ble, but an OR plane with fixed connections designed to link each output line to a fixed set of AND rows, typically about eight rows. Such a PAL output can realize only a two-level expression containing at most eight terms. A PAL's advantages are ease of use in some applications, as well as higher speed because output fan-out is restricted.

Field-programmable gate arrays. This important class of PLDs was introduced in the mid-1980s. A field-programmable gate array (FPGA) is a two-dimensional array of general-purpose logic circuits, called cells or logic blocks, whose functions are programmable; the cells are linked to one another by programmable buses. The cell types are not restricted to gates. They are small multifunction circuits capable of realizing all Boolean functions of a few variables; a cell may also contain one or two flip-flops. Like all field-programmable devices, FPGAs are suitable for imple-menting prototype designs and for small-scale manufacture.

FPGAs can store the program that determines the circuit to be implemented in a RAM or PROM on the FPGA chip. The pattern of the data in this configuration

memory CM determines the cells' functions and their interconnection wiring. Each bit of CM controls a transistor switch in the target circuit that can select some cell function or make (break) some connection. By replacing the contents of CM, designers can make design changes or correct design errors. This type of FPGA can be reprogrammed repeatedly, which significantly reduces development and manufacturing costs. Some FPGAs employ fuses or antifuses as switches, which means that each FPGA IC can be programmed only once. These one-time pro-grammable FPGAs have other advantages, however, such as higher density, and smaller or more predictable delays.

Two types of logic cells found in FPGAs are those based on multiplexers and those based on PROM table-lookup memories. Figure 2.35a shows a cell type (the C-module) employed by Actel Corp.'s ACT series of multiplexer-based FPGAs [Greene, Hamdy, and Beal 1993; Actel 1994]. This cell is a four-input, 1-bit multi-plexer with an AND and OR gate added. A variant called the S-module has a D flip-flop connected to the primary output; there are also special cells attached to the FPGA's 10 pins. An ACT FPGA contains a large array (many thousands) of such cells organized in rows separated by horizontal wiring channels as illustrated in Figure 2.35b. Vertical wire segments are attached to each cell's 10 terminals. These wires enable connections to be established between the cells and the wiring channels by means of one-time-programmable antifuses positioned where the hori-zontal and vertical wires cross. In addition, long vertical wires run across the entire array to carry primary IO signals, power (logical 1), and ground (logical 0).

Our discussion of multiplexers as function generators implies that the FPGA cell of Figure 2.35a can generate any Boolean function of up to three variables if the inputs are supplied in both true and complemented form. This cell can also generate various useful functions of more than three variables due to the presence

101

CHAPTER 2

Design

Methodology

*2

Four-input,

1-bit multiplexer

xA x5 x6 x-i

{a)
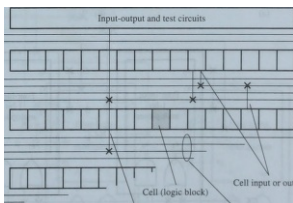
Cell inpul or outputCell (logic block)

Vertical wire Horizontal wiring channel

(*)

Figure 2.35

Actel ACT-series FPGA: (a) basic cell (C-module); (b) chip architecture

102

SECTION 2.2 The Register Level

of the two extra gates. Figures 2.36a, 2.36b. and 2.36c show how this cell imple-ments a functionally complete set of logic gates. Observe how the cell's AND and OR gates help to realize four-input AND and OR functions. Figure 2.36a1 show show the same, basically combinational cell implements an edge-triggered D flip-flop.

EXAMPLE 2.5 FPGA IMPLEMENTATION OF A SERIAL ADDER. We will use

the Actel C-module of Figure 2.35a to realize the serial adder of Figure 2.12. The tar-get circuit contains a combinational part C. which is a full adder defined by the equations

$$c = xxx_2 + xly + x^\wedge y$$

(2.14)

0 ^X

Four-input, 1-bit

multiplexer

>£-*

c d

abed

1 -X-

Four-input.

1-bitmultiplexer

d

a + b + c + d

0 -X

Four-input.

1-bitmultiplexer

X X X X

(a)

(b)

(c)

CK
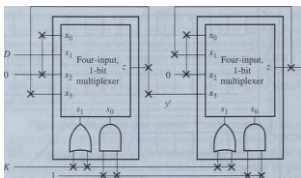
(d)

Figure 2.36

FPGA cell of Figure 2.35a programmed to realize: (a) a four-input AND gate; (») a four-input OR gate; (c) aninverter; (d) a D flip-flop.

Here z is the sum bit and c is the carry bit. A single D flip-flop stores the value of c pro-duced in each clock cycle and applies it to C as y in the next clock cycle. We willassume that if the complements of any of the input variables jc_1, x_2, or y are needed,they must be generated explicitly in the FPGA. We will also try to use as few cells aspossible in the target circuit.

Figure 2.36d shows that two cells are required for the D flip-flop, assuming thatwe don't need the complement of y. It's not immediately clear how many cells areneeded to produce the sum and carry. A little experimentation shows that the carryfunction does indeed have a one-cell realization; see Figure 2.37. Observe that Equa-tion (2.14) can be rewritten as

$$c = y(xx + x_2) + xyx_2$$

which suggests the way we use the Actel cell's AND and OR gates. No amount ofexperimentation yields a one-cell realization of the sum function. The multiplexerrealization of the full adder we gave earlier (Figure 2.22c) requires the data inputs tobe supplied to the sum part in both true and complemented form. We will thereforedevote a third cell to generating y so we can realize z in the manner of Figure 2.22c.

103

CHAPTER 2

Design

Methodology
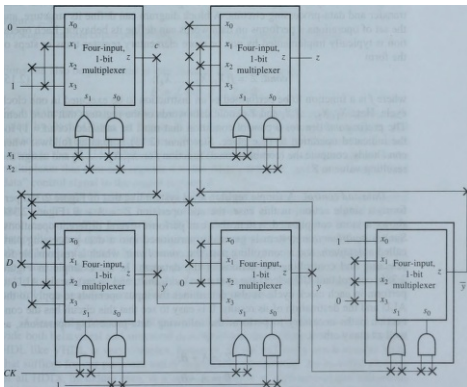
Figure 2.37

FPGA implementation of a serial adder.

104 The resulting design given in Figure 2.37 for the serial adder employs a total of five

cells.SECTION 2.2

The Register Level

FPGAs are very well suited to computer-aided design and manufacture; the process of mapping a new design into one or more FPGA chips can be almost entirely automated. It requires first translating or "compiling" the design specification—a schematic diagram or an HDL description, for example—into a logic (gate and flip-flop) model. Specialized place-and-route CAD software is then employed to assign the logic elements to cells, to determine the switch settings needed to set each cell's function, and to establish the intercell connections. Finally, the design is physically transferred to one or more copies of the FPGA chip via an appropriate programming unit, a process that has been aptly described as "desktop manufacturing."

2.2.3 Register-Level Design

A register-level system consists of a set of registers linked by combinational data-transfer and data-processing circuits. A block diagram can define its structure, and the set of operations it performs on data words can define its behavior. Each operation is typically implemented by one or more elementary register-transfer steps of the form

$$cond: Z := f(X], X2,...,Xk); \quad (2.15)$$

where/is a function to be performed or an instruction to be executed in one clock cycle. Here $X_{,,}$ X->, ..., Xk and Z denote data words or the registers that store them. The prefix cond denotes a control condition that must be satisfied (cond = 1) for the indicated operation to take place. Statement (2.15) is read as follows: when cond holds, compute the (combinational) function/on Xx, X2, ...,Xk and assign the resulting value to Z.

Data and control. A simple register-level system like that of Figure 2.38a performs a single action, in this case, the add operation Z := A + B. Figure 2.386 shows a more complicated system that can perform several different operations. Such a multifunction system is generally partitioned into a data-processing part, called a datapath, and a controlling part, the control unit, which is responsible for selecting and controlling the actions of the datapath. In the example in Figure 2.386, control unit CU selects the operation (add, shift, and so on) for the ALU to perform in each clock cycle. It also determines the input operands to apply to the ALU and the destination of its results. It is easy to see that this circuit has the connection paths necessary to perform the following data-processing operations, as well as many others.

Z := A + B;

B:=A-B;

Less obvious operations that can be performed are the simple data transfer Z := B, which is implemented as Z := 0 + B: the clear operation B := 0, which is implemented as B := B - B; and the negation operation B := 0 - B. A few double opera-

Register A Register B

<   ii

Adder

' r

Register Z

Register^

—1 ° I

\ «*"«- L.

\ plexer /

Register B

MultifunctionALU

Register Z

Controlunit CU

(a)

(b)

Figure 2.38

(a) Single-function circuit performing Z:= A + B; (b) a multifunction circuit.

105

CHAPTER 2

Design

Methodology

tions can also be performed in one clock cycle, for example,

B :=Z+B,Z:=Z+B;

Each of the foregoing operations requires CU to send specific control signals, indicated by dashed lines in Figure 2.38b, to various places in the datapath. For instance, to execute the subtraction Z := A - B, the controller CU must send select signals to the ALU to select its subtract function; it must send select signals to the multiplexer that connects register A to the ALU's left port; and it must send a "loaddata" control signal to the output register Z.

An example of a large multifunction system is a computer's CPU. Its control unit, which is responsible for the interpretation of instructions, is called the program control unit or I-unit. The CPU's datapath is also called the E-unit. Further datapath/control subdivisions are possible in complex systems, yielding a hierarchy of levels of control. In relatively simple machines such as that of Figure 2.38/?, the control unit can be a special-purpose hard-wired sequential circuit designed using standard gate-level techniques. In more complex cases, both the datapath and control units may have to be treated at the register level.

A description language. HDLs, which were introduced in section 2.1.1, provide both behavioral and structural descriptions at the register level. A full-fledged HDL like VHDL is very complex, however, so we will use a much smaller HDL that suffices for our purposes and is largely self-explanatory. An essential element of all HDLs, including ours, is a state assignment or register-transfer statement. which has the general form of (2.15), and specifies a conditional state transition that takes place in a single clock cycle. An alternative notation for (2.15) is '

if cond = 1 then Z :=/(*,. X2 Xk);

106 There is often a close correspondence between the elements of an HDL

, ., description and hardware components and signals in the system being described.

The Register Level ^or examPle-tne statement Z := A + B describes the circuit of Figure 2.38a. In this interpretation, + represents the adder. The input connections to the adder from registers A and B are inferred from the fact that A and B are the arguments of +, while the output connection from the adder to register Z is inferred from Z : =. An exact correspondence between hardware structures and HDL constructs can be hard to specify without considerable verbosity. To keep our HDL concise, we use it primarily for behavioral descriptions and supplement it with block diagrams to describe structure.

Figure 2.39 illustrates the use of our HDL to describe the behavior of a complete system at the register level. This 8-bit multiplication circuit, named multiplier8, computes the product Z = Y x X. where the numbers are 8-bit binary fractions in sign-magnitude form. (The actual design of this multiplier, which implements a binary version of "long" multiplication based on repeated addition and shifting, is examined later in Example 2.7.) Two 8-bit buses INBUS and OUT-BUS form multiplier8's input and output ports, respectively, and link it to the outside world. The circuit contains three 8-bit data registers A, A/, and Q, as well as a 3-bit control register COUNT that counts the number of add-and-shift steps to decide when multiplication is complete. The A and Q registers can be merged into a single 16-bit shift register denoted A.Q. The operands X (the multiplier) and F (the multiplicand) are initially transferred from INBUS into the Q and M registers, respectively. The product is computed by multiplying F by 1 bit of X at a time and adding the result to A. After each addition step, the contents of A.Q are shifted 1 bit to the right so that the next multiplier bit required is always

in <2[7], the right-mostbit in the Q register. (Consequently, the multiplier F is eventually shifted out of Qand lost.) After seven iterations to multiply the magnitude parts of X and F, the signof the product is computed and placed in the left-most position of A, that is, in A[0].

multiplier8   (in: INBUS: out: OUTBUS):

              register A[0:1]. M[0:7], £[0:7]. COUNT[0:2];

              bus INBUS[0J], OUTBUS10J);

BEGIN: A := 0. COUNT := 0, M := INBUS:

              Q := INBUS:

ADD: A[0:7] := A[l:7] + M[l:7] x Q[l];

SHIFT: A[0] :=0,A[1:7]. Q:=A.Q[0:6],

TEST: COUNT := COUNT + 1;

              if COUNT* 1 then go to ADD.

FINISH: A[0] := M[0] xor Q[l], Q[l] := 0:

OUTPUT: OUTBUS := Q:

              OUTBUS := A:

end multiplier8:

Figure 2.39

Formal language description of an 8-bit binary multiplier.

The final product ends up in A.Q, from which it is transferred 8 bits at a time toOUTBUS.

The description of the multiplier consists mostly of register-transfer opera-tions. The registers are defined by the initial register statement, which gives theirnames, their sizes, and the order in which their bits are indexed. For example,

register M[0:7];

means that M is a register composed of eight flip-flops individually identified asM[i], where i runs from 0 to 7 from left to right. Equivalently, we could write

M = M[0].M[\].M[2].M[3].M[4].M[5].M[6].M[1];

Buses are used in much the same way as registers and are defined similarly. Regis-ter-transfer operations that take place simultaneously, that is, during the same clockcycle, are separated by commas, while a semicolon separates sets of operations thatmust occur in successive clock cycles. Thus the statement

A := 0, COUNT := 0, M := INBUS;

appearing on the line labeled BEGIN in Figure 2.39, specifies three distinct actionsto take place in the same clock period: clear the A register (transfer the all-0 oper-and to it), clear the COUNT register, and transfer the data on INBUS to register M.Note that a register can be read from and written into in the same clock cycle, ashappens to Q in the statement

A[0] := M(0) xor Q[l], Q[l] := 0-

The order in which a list of statements terminating in semicolons are written isthe sequence in which the actions they define should occur. Deviations from thissequence are specified by control statements and by the use of statement labels. Weuse the if ... then control statement to make an action sequence depend on somecircuit condition. For example, the conditional branch statement

if COUNT * 7 then go to ADD,

(2.16)

in Figure 2.39 means the following: Test the state of the 3-bit COUNT register. IfCOUNT is not equal to 7, that is, 1112, then the next action to be taken is specifiedby the statement labeled ADD. If COUNT = 7, then the next action is specified bythe statement FINISH.

107

CHAPTER 2

Design

Methodology

Design techniques. The design problem for register-level systems is as fol-lows. Given a set of operations to be executed, design a circuit using a specified setof register-level components that implement the desired functions while satisfyingcertain cost and performance criteria. As noted already, it is difficult to impose use-ful mathematical structures on register-level behavior or structure correspondingto, say, Boolean algebra and the two-level constraint in gate-level design. Lackingsuch mathematical tools, register-level design methods tend to be heuristic anddepend heavily on the designer's expertise. We can, however, state the followinggeneral approach to the design problem.

1. Define the desired behavior by a set of sequences of register-transfer operations,such that each operation can be implemented directly using the available designcomponents. This constitutes an algorithm AL to be executed.

108 2. Analyze AL to determine the types of components and the number of each type

section 2 2 required for the datapath DP.

The Register Level ^- Construct a block diagram for DP using the components identified in step 2.Make the connections between the components so that all data paths implied byAL are present and the given performance-cost constraints are met.

4. Analyze AL and DP to identify the control signals needed. Introduce into DP thelogic or control points necessary to apply these signals.

5. Design a control unit CU for DP that meets all the requirements of AL.

6. Verify, typically by computer simulation, that the final design operates correctlyand meets all performance-cost goals.

Algorithm design (step 1) involves a creative design process analogous to writ-ing a computer program and depends heavily on the skill and experience of thedesigner. The identification of the data-processing components in step 2 is straight-forward, but complications arise when the possibility of sharing components exists.For example,

c: A:=A + B,C:=C + D;

defines two addition operations. Since these additions do not involve the sameoperands, they can be done in parallel if two independent adders are provided.However,

costs can be lowered by sharing a single adder and performing the two additions sequentially, thus:

$c(t0)$: A := A + B;

$c(tQ+1)$: C := C + D;

This example illustrates a fundamental cost-performance trade-off. The identifica-tion of the parallelism inherent in a multistep algorithm can be exceedingly diffi-cult.

A typical datapath unit DP has a regular and relatively simple structure designed for processing data of some fixed word size w. Its main components are registers, buses, and combinational circuits, all oriented toward w-bit words. The design of DP (step 3 above) requires defining an interconnection structure that links the components needed by the various parts of AL. The specification and design of the control unit CU (steps 4 and 5) is a relatively independent process. Unlike DP, the control unit often has a small number of states that interact in an irregular fashion, making it suitable for gate-level, sequential circuit design (sec-tion 2.1.3). Specialized methods such as microprogramming are used to design large control units, a topic we consider in Chapter 5.

Design verification (step 6) plays a crucial role in the development process because mistakes, often of a subtle kind, are unavoidable in the design of a com-plex system. Simulation via CAD tools is used to identify and correct functional errors before the new design is committed to hardware. CAD tools are also used to predict or measure the system's operating speed. If a particular design does not meet some specification —an algorithm step is executed too slowly, or component costs are exceeded—it is necessary to return, sometimes repeatedly, to steps 1 through 5 and modify AL, DP, or CU.

We now present two examples of sequential circuits designed at the register level. The first revisits the 4-bit-stream adder, whose behavior and gate-level design

are covered in Example 2.2. It illustrates some advantages of a high-level, func-tional approach to design, as well as the important design technique of pipelining.

EXAMPLE 2.6 DESIGN OF A PIPELINED 4-BIT-STREAM SERIAL ADDER. Con-sider again the design of a circuit to add four unsigned binary numbers presented seri-ally (least significant bits first) to produce their arithmetic sum, also in serial form. This adder has four input lines x1,x2,x3,x4 and a single output line z. Our first, gate-level design (Example 2.2) started with the construction of a (4 x 16)-entry state table (Fig-ure 2.13a), and culminated in a circuit (Figure 2.13Z?) containing two D flip-flops and a large (eight-input, three-output) combinational circuit.

This time we will start with the observation that we can add the four bit streams in pairs using a basic register-level component, the serial adder (Figure 2.12). We can add streams x1 and x2 using one serial adder SA] and, at the same time, add streams x3 and x4 using a second serial adder SA2. The outputs of SAX and SA2 are then combined by a third serial adder SA3 to obtain the desired output z. This process leads to the circuit 4ADDX in Figure 2.40a, which contains three D flip-flops and three full adders. Because the full adders are relatively simple—several representative logic realizations appear in Figure 2.9—4ADDX contains far fewer gates than the design of Figure 2.13.

SA3's combinational logic (a full adder) receives signals directly from the corre-sponding full adders in SAX and SA2. Hence 4ADDX has more levels of combinational logic than a simple serial adder. Consequently, for 4ADDX to operate properly, it must be clocked at a frequency /' < /. where / is the maximum permissible frequency of a serial adder. We can, however, operate the 4-bit-stream adder at the higher frequency /, if we insert a pair of flip-flops as buffers between SAX:SA2 and 5A3, as illustrated in Figure 2.40b. Now the inputs to SA3 in clock cycle / consist only of the signals com-puted by SAX and SA2 in cycle /' - 1 and stored in the buffer flip-flops of the new design 4ADD2. This, however, means that each result bit produced by 4ADD2 is delayed by one clock cycle. It might therefore be thought that 4ADD2 is significantly slower than 4ADDX. This is not the case, however, because in both circuits a new final result bit z is generated in every clock cycle. Although it takes two clock cycles to calculate each sum bit, 4ADD2 overlaps the computation of two successive sum bits so that, once it is in full operation, it also produces one result bit per cycle. Breaking a computation into a sequence of simpler subcomputations that can be overlapped is called pipelining and is an important technique in computer design.

In the final circuit 4ADD3 (Figure 2.40c), we have introduced a flip-flop to store the output z of SA3; we have also regrouped the internal (carry) flip-flops of the serial adders to make them part of the buffer registers—recall that their role is to store carry bits generated in clock cycle /' - 1 and used in clock cycle i. 4ADDi has a circuit struc-ture called a pipeline. It is composed of two stages, each of which consists of some combinational logic followed by a buffer register. Suppose the first four data bits enter stage 1 at time (clock cycle) 1. Their partial sum bits z, and z2 are computed and passed on to stage 2. The first result bit z = zx plus z2 is then computed by stage 2 dur-ing clock cycle 2. At the same time a second set of four data bits can be entered into and processed by stage 1. In clock cycle 3, the result sum is computed by stage 2 while stage 1 handles a third set of input data, and so on. Clearly if a steady stream of data enters the pipeline, then a new result bit emerges every clock cycle, beginning with clock cycle 2.

Modern computers often employ pipelines of this sort for complex arithmetic operations such as floating-point addition, as we will see in Chapter 4. They also pro-cess instructions by means of a special multifunction pipeline composed of as many as a dozen stages (Chapter 5).

CHAPTER 2

Design

Methodology

SECTION 2.2 The Register Level

Serial adder SA |

Serial adder SA3

1 1 CLK CLR

Serial adder SA2

1 1 CLK CLR

CLK CLR

(a)

Serial adder SA

Serial adder SA1

^_

CLK CLR

D Q

D Q

CLK CLR

CLK CLR CLK CLR

GO

Stage 1

Full adder FA,

Full adder FA -,

d o\—>

D Q

d e-.

CLK CLR

Stage 2

Fulladder FA

d a

d e-i

(c)

Figure 2.40

Four-bit-stream serial adder: (a) basic design 4ADD]; (b) buffered design 4ADD2; (c) two-stage pipeline design 4ADDi.

Next we examine a bigger register-level design problem, a sequential circuitthat multiplies two binary numbers. This circuit is too complex to design at the gatelevel; it also has well-defined data-processing and control parts.

EXAMPLE 2.7 DESIGN OF A FIXED-POINT BINARY MULTIPLIER. Fixed-point

multiplication is often implemented in computers by a binary version of the manualmultiplication algorithm for decimal numbers based on repeated addition and shifting.Consider the task of multiplying two 8-bit binary fractions X = XqX^x^x^x^ andY = y0Vj\sv;.y4.v5y6_VT to form the product P = XxY. Each number is assumed to be insign-magnitude form, where the left-most bit (with subscript 0) of the number denotesits sign: 0 for positive and 1 for negative. The remaining seven bits represent the num-ber's magnitude. Note that for fractions, it is convenient to index the numbers from leftto right, so that bit x_t has weight 2"'. Hence when x0 = 0, X = XqX^XjXjX^^x^j denotes

Q[7] iMultiplier register

n M[0]

Multiplicand register

Externalcontrol 'signals

111
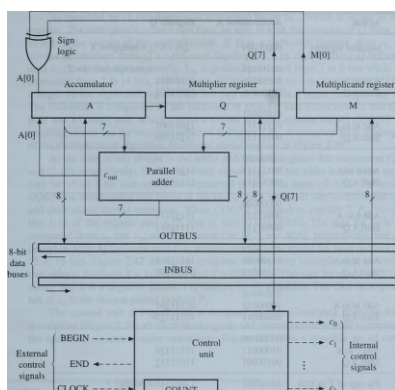
Figure 2.41

Block diagram of an 8-bit binary multiplier multiplier^.

the positive number N given by

N = .v,2-' + x22~2 + jc32"3 + V"* + x52'5 + V-6 + V7

When x0 - 1, X denotes -N.

The multiplication algorithm that we will implement first multiplies the magnitudeparts XM and YM of A" and Y thus:

Pm'^m^m (217)

where PM - p^p2... pu is the magnitude of the product P. It computes the sign/?0 of Pvia the simple operation p0 := .v0 xor y„. The final result P - PoPiP? ■■■ P\A is '^ b'tslong. The magnitude multiplication (2.17) is clearly the central design problem. Theunsigned product PM is computed in seven add-and-shift steps defined as follows:

P;:=PI + .v7 ,x,M; (2.18)

Pl+]:=2-sP- (2.19)

112

SECTION 2.2The Register Level

Step	Action	Accumulator A		Register Q
0	Initialize registers	00000000	10110011 = multiplier X	
1	Add M to AShift A. Q		01010101010101010100101010 = multiplicand M = Y1011001*111011001	
2	Add M to AShift A. 2		01010101011111110011111 U01100111101100	
3	Add 0 to AShift A. Q	00000000000111111000111111	11101100 nnono	
4	Add 0 to AShift A. Q	00000000000001111100001111	11110110 union	

Add A/ to AShift A. Q

5		01010101011001000011001 0 in nonomnoi
6	Add A/ to AShift A.Q	010101011000011101000011 oinnoi101 n no
7	Add 0 to AShift A.Q	loin no 000000000100001100100001 11011111
8	Put sign of Pin A [0]and set Q[l] to 0 10100001	11011110 = product/'

Figure 2.42

Illustration of the binary multiplication algorithm.

where P0 = 0, P7 - PM, and / goes from 1 to 7. The quantities P0,/>,,,...,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 OUT-BUS 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 theadd-shift 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 Q's is written into flip-flop 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

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.

SECTION 2.3

The Processor Level
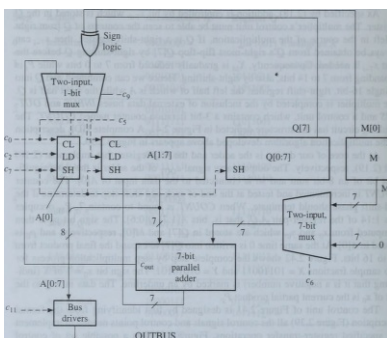
M[l:7]

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

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.

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.

System bus

Li

j_   i\

*!   Cache
If

/i   /i

IF   —   i

Programcounter PC   Instructionregister IR   i

Arithmetic-logic unit   Registerfile

t   i   ■■ 1

Addressgeneration |#—T Instructiondecoding

> 1 1 1
i_   s
J i

Prun ogram ccit (I-unit   >n)   tro   Control signal Datapath(E-unit)
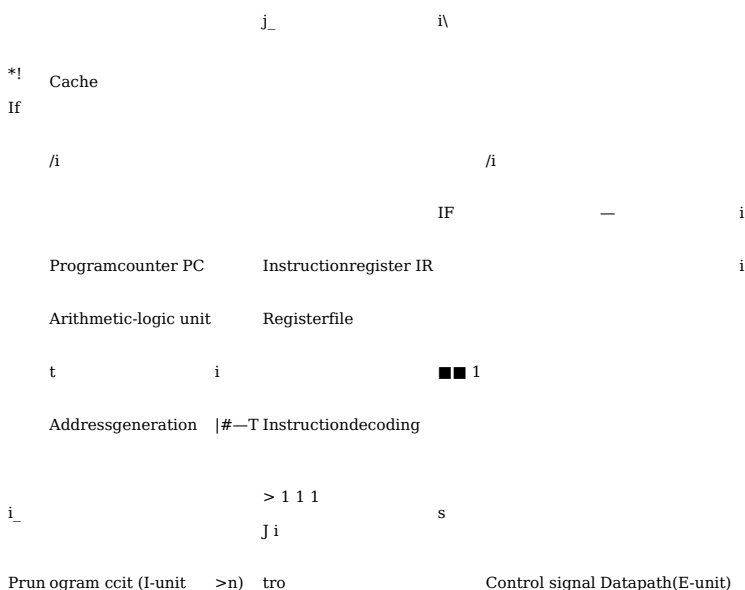
Figure 2.46

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

IO device	Type Input	Output	Medium to/from which IO device 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		Images on paper
Dot-matrix display panel		X	Images on screen
Keyboard/keypad	X		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 modify it where necessary to accommodate new technologies or meet new performance requirements. 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 new 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 understanding of 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 a copy of the machine under working conditions. Reflecting its limited theoretical basis, 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 major steps: First select a prototype design and adapt it to satisfy the given performance constraints. Then determine the performance of the proposed system. If unsatisfac-tory, modify the design and repeat this step; continue until an acceptable design is obtained. This conservative approach to computer design has been widely followed and accounts in part for the relatively slow evolution of computer architecture. It is rare to find a successful computer structure that deviates substantially from the norm. The need to remain compatible with existing hardware and software stan-dards also influences the adherence to proven designs. Computer owners are understandably reluctant to spend money retraining users and programmers, or replacing well-tested software.

The systems of interest here are general-purpose computers, which differ from one another primarily in the number of components used and their autonomy. The variety of interconnection or communication structures used is fairly small. We will 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 of special-purpose 10 processors typical of the second and subsequent generations is

119

CHAPTER 2

Design

Methodology

Centralocessing unit CPU M   Main memory

Systembus   ICN

IO   Figure 2.48

          D,   D2 D*

devices   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) network that controls memory-processor communication. Figure 2.50 shows a prototype structure employing two CPUs; it is therefore a multiprocessor. The uniprocessor systems of Figures 2.48 and 2.49 are special cases of this structure. Even more complex structures such as computer networks can be obtained by linking several copies of the foregoing prototype structures.

Performance measurement. Many performance figures for computers are derived 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,   IOP2   IOP„

D,   D2   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

$t_E = T/N = CPI/f$ 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 I_1, I_2, ..., $I_n$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

Instruction type	Probability ol occurrence	
	Program A	Program B
	(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 archi-

tecture 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

Performance measure	DEC 3000 Model 400	DEC 4000 Model 610	DEC 10000 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   1 r   Serviced

Queue   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

$Pl(t) = l-e\sim h$

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

$$lQ = p/(1-P) \quad (2.21)$$

• 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 we conclude that $/Q = Xt_Q$, in other words,

$$t_Q = I_Q/X \quad (2.22)$$

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

$$t_Q = l/(p - X) \quad (2.23)$$

The quantities $/Q$ and t_Q refer to tasks that are either waiting for access to the server or are actually being served. The mean number of tasks waiting in the queue excluding those being served is denoted by $/w$, while r_w denotes the mean time spent waiting in the queue, excluding service time. (The subscript W stands for "waiting.") The mean utilization of the server in an M/M/l system, that is, the mean number of tasks being serviced, is $X/\i$; hence subtracting this from $/Q$ yields $/w$:

^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 $f_w = l_w/X$; therefore, Little's equation holds for both the Q and the W subscripts.

To illustrate the use of the foregoing formulas, consider a server computer that is processing jobs in a way that can be approximated by the M/M/l model. Arriving jobs are queued in main memory until they are fully executed in one step by the CPU, which therefore is the server. New jobs arrive at an average rate of 10 per minute, and the computer is, on the average, idle 25 percent of the time. We ask two questions: What is the average time T that each job spends in the computer? What is the 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 follows that 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 $T = t_Q = 1/(40/3 - 10) = 0.3$ min. From Little's equa-tion, $N = l_Q = Xt_Q = 3$; 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 engineering staff. An average of 10 engineers use the terminal during an eight-hour work day, and each user occupies the terminal for an average of 30 minutes, mostly for simple and routine 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, complain that 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/l queueing 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 of eight 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 $f_w = 50$ mm, which confirms the users' estimate of their average waiting time for terminal access.

The manager is now convinced that the company needs additional terminals and agrees to buy enough to reduce r_w from 50 to 10 min. The question then arises: How many 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 $t_w < 10$ or, equivalents, $t_n < 40$. The SECTION 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 indicated above, the computer's CPU is lightly utilized, then a few additional terminals should not 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 to form three separate queues for three terminals or to maintain the independence of the queues 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.

2.4 SUMMARY

The central problem facing the digital system designer is to a devise a structure (a circuit, network, or system) from given components that exhibits a specified behavior or performs a specified range of operations at minimum cost. Various methods exist for describing structure and behavior, including block diagrams (for structure), truth and state tables (for behavior), and HDLs (for behavior and struc-ture). Computer systems can be viewed at several levels of abstraction, where each level is determined by its primitive components and information units. Three levels have been presented here: the gate, register, and processor levels, whose compo-nents process bits, words, and blocks of words, respectively. Design at all levels is a complex process and depends heavily on CAD tools.

The gate level employs logic gates as components and has a well-developed theory based on Boolean algebra. A combinational circuit implements logic or Boolean functions of the form $z\{x_x, x_2, ..., x_n)$, where z and the x,'s assume the val-ues 0 and 1. The circuit can be constructed from any functionally complete set of gate types such as {AND, OR, NOT} or {NAND}. Every logic function can be realized by a two-level circuit that can be obtained using exact or heuristic minimi-zation techniques. Sequential circuits implement logic functions that depend on time; unlike combinational circuits, sequential circuits have memory. They are built from gates and 1-bit storage elements (flip-flops) that store the circuit's state and 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 exists for the design and analysis of register-level circuits. They are often described by HDLs whose fundamental construct is the register-transfer statement

cond: $Z := F,(X_1,X_2,...,X_{it})$;

denoting the conditional transfer of data from registers $X_l,X_2,...,X_k$ to register Z via a combinational processing circuit $F\{$. Register-level circuits often consist of a datapath 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 signals needed to control the datapath are then identified. Finally, a control unit is designed that 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 using average or worst-case behavior. Processor-level design is heavily based on the use of prototype structures. A prototype design is selected and modified to meet the given performance specifications. The actual performance of the system is then evaluated, 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 performance measurements on an actual system are used extensively.

127

CHAPTER 2

Design

Methodology

2.5 PROBLEMS

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 of the full-adder circuit of Figure 2.9b. (b) Following Figure 2.5, construct a structural VHDL 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	0 1	1 1	
	0	1 0	1 1	
	0	1 1	0 1	
	1	0 0	1 0	
	1	0 1	0 0	
11	11	0	0 0	
		1	1 1	

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

0 0 11 Next state

State 0>'(') 1

ioio y(' + D

129
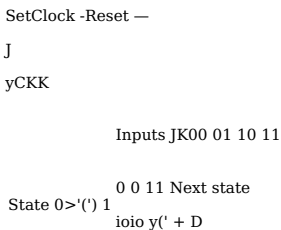
CHAPTER 2

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
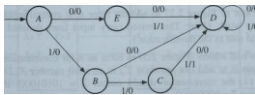
Figure 2.57

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 as-serted 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 ex-ample, 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 clear-

2.28.

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

Figure 2.58

Expansion of a 4-bit magnitude comparator to form a 16-bit comparator.
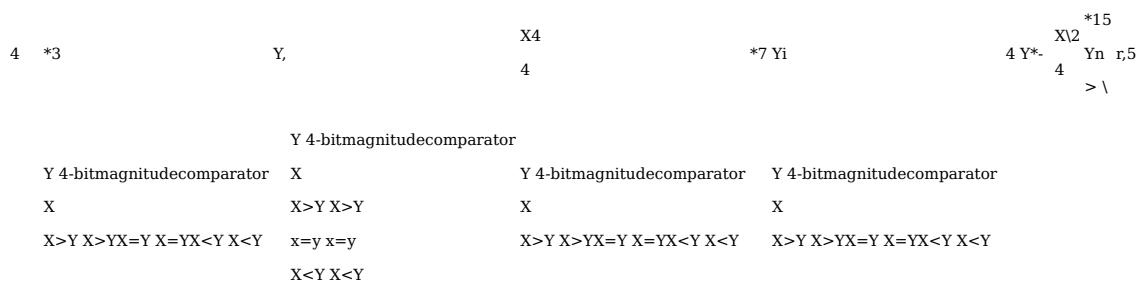
u

1

o —

X>YX= YX< Y

132

SECTION 2.5 Problems

x] Four-input.

1-bitX2 multiplexer

> CK

CLR

Figure 2.59

S-module from the Actel FPGA series.

2.34. Reconsider the FPGA implementation of the serial adder given in Figure 2.37. Sup-pose that it can now be implemented using two cell types: the original Actel C-module and the more recent sequential S-module defined in Figure 2.59. Construct a new version of the adder in the style of Figure 2.37 using as few modules as you can.

2.35. The 4-bit-stream serial adder 4ADDX of Figure 2.40a contains three flip-flops, one in each serial adder, so it can have up to eight internal states. However, according to the analysis in Example 2.2, only four states are needed for 4-bit-stream serial addi-tion. Does this imply that one flip-flop can be removed from 4ADDX and, if so, which one? Explain your reasoning clearly.

2.36. Consider the operation of the serial adder pipeline 4ADD3 shown in Figure 2.40c. It is reset to the all-0 state in clock cycle 0. and the following data is entered into the pipeline at the indicated times:

Clock cycle: 0 1 2 3 4 5 6 7 g

xv   0 1 0 1 1 0 0 0 0

x2:   0 1 1 1 1 0 0 0 0

x3:   0 0 0 1 1 0 0 0 0

x4:   0 1 1 1 0 0 0 0 0

":

Determine the value of z for each clock cycle in the above table.

2.37. Suppose that the pipelined serial adder of Figure 2.40c is reset in clock cycle 0. The least significant bits of four serial numbers (integers) Nx, N2, N3, N4 to be added are applied to the adder in clock cycle 1, and four new data bits are applied in each sub-sequent clock cycle. If each number consists of thirty-two 1-bits and therefore repre-sents $2^{32} - 1$, in what clock cycle will the most significant bit of the sum $N_1 + N_2 + 7V_3 + N_4$ be loaded into the output z flip-flop?

2.38. Construct a pipelined adder in the style of Figure 2.40c that can add six instead of four separate bit streams.

2.39. Design at the register level a modulo-16 binary counter CAT/?. The counter has two function control input lines: LOAD, which loads the counter with an initial value from a 4-bit external bus BUS, and COUNT, which increments the counter by one. The available component types (use as many of each as you need) for building CNTR are the 4-bit D register of Figure 2.28; the 4-bit adder of Figure 2.26a; the two-input, 4-bit multiplexer of Figure 2.20; and the two-input, m-bit NAND word gate of Figure 2.17 with m = 1,2, and 4.

2.40. Consider the counter described in the preceding problem. Suppose that there is an-other control input DOWN which, when set to 1, causes the counter to count down (decrement) instead of up. When DOWN = 0, CNTR behaves like an up-counter, as in the original design. In each case a suitable pulse applied to the COUNT line in-crements or decrements the counter. Using the same set of register-level component types, design this modulo-16 up-down counter.

2.41. Figure 2.60 is an HDL description of an algorithm for multiplication in low-speed digital systems. It is implemented by three up-down counters CQ, CM, and CP which store the multiplier, multiplicand, and product, respectively, and the product P is formed by incrementing the counter CP a total of P times. Although this multipli-cation method is slow, it requires a simple logic circuit and can easily accommodate complicated number codes. Suppose that the numbers to be multiplied are four-digit integers in sign-magnitude BCD code. For example, the number -1709 is represent-ed by the bit sequence 1 0001 0111 0000 1001. CQ, CM, and CP are to be con-structed from modulo-10 up-down counters with parallel input-output capability. Carry out the logic design of this multiplier at the register level.

2.42. Devise a counting algorithm similar to that of Figure 2.60 to perform integer divi-sion on unsigned four-digit BCD integers. The inputs are a dividend Y and a divisor X; the outputs are a quotient Q and a remainder R, which must satisfy the following equation:

Y=QxX + R, with 0 < R < X

multiplierbc(IOBUS[16:0]);

register <2[15:0]. C015:O], CA/[15:0], CP[3\:0], QS, MS;BEGIN: Q := IOBUS[\5:0], QS := IOBUS[\6];

CM := IOBUS[15:0], MS := IOBUS[\6), CQ := Q, CP := 0;TEST!: if CM := 0 or CQ = 0 then go to DONE,

ADD: CQ:=CQ-l,CP:=CP+\

TEST2: \fCQ*0 then go to ADD,

SUB: CM =CM-\,CQ.= Q,

TEST3: if CM * 0 then go to ADD,

DONE: lOBUS[\6] := QSxorMS, IOBUS[15:0) := CP[31:16];

IOBUS[\5:0] := CP[\5:0];

Figure 2.60

A multiplication algorithm using counters.

134 Describe your algorithm formally by means of our HDL. Carry out the register-level

logic design of a machine that performs division on four-digit BCD integers using theSECTION 2.5 counting approach.

Problems

2.43. (a) Name the various types or levels of memory found in a typical computer. Whyis more than one memory type needed? (b) Identify all the places in a computerwhere instructions are stored at various times, (c) Explain why secondary-memoryunits such as hard-disk drives are part of the 10 system, whereas main memory isnot.

2.44. Let P be a processor that operates at a clock frequency of 100 MHz. Suppose, fur-ther, that advances in VLSI technology allow P to be replaced by a new CPU Fwhose architecture and organization are identical to those of P, but whose clock rateis 125 MHz. How does replacing P by P1 in the execution of a set of benchmarkprograms Q affect (a) the value of its CPI and (b) the total CPU time required to ex-ecute Ql

2.45. A possible measure of the performance of a CPU P that employs instruction-levelparallelism is the average number of instructions per cycle or IPC needed to executea benchmark program set Q. Suppose that a total of N instructions are executed inthe processing of Q by P. Further suppose that P has a clock cycle time of 7"clock,and T is the total CPU time required for P to execute Q. Obtain an expression forIPC in terms of N, T, and Tdock.

2.46. Consider the instruction mixes appearing in Figure 2.51. Suppose that the system'sclock frequency is 100 MHz, and all instructions except floating-point instructionshave an average execution time of 10 ns. (a) What is the average execution time offloating-point instructions, if the overall average execution time per instruction forprogram B is 18.1 ns? (b) What is the CPI for program B?

2.47. Suppose that the instructions listed in Figure 2.51 have the following average exe-cution characteristics: load, store, and floating-point instructions require four clockcycles each; fixed-point instructions require two clock cycles; all others require oneclock cycle. If both programs involve the execution of 2.5 million instructions,which of the two completes execution sooner?

2.48. The MIPS performance measure is often considered useful only when used to com-pare members of the one processor family from the same manufacturer, as in Figure2.52. Give some reasons why this is generally true. (Misuse of this measure has ledto the suggestion that MIPS really means "meaningless information from pushysalesmen!")

2.49. What happens in a single-server queue like that of Figure 2.53 if X > |i?

2.50. Suppose that CPU behavior in a multiprogramming system can be analyzed usingthe M/M/l queueing model. Programs are sent to the CPU for execution at a meanrate of eight programs per minute and are executed on a first-come first-served ba-sis. The average program requires six seconds of CPU execution time, (a) What isthe mean time between program arrivals at the CPU? (b) What is the mean numberof programs waiting for CPU execution to be completed? (c) What is the mean timea program must wait for its execution to be completed?

2.51. Suppose that people arrive at a public telephone booth at an average rate of 10 perhour. The lengths of the calls made from the booth are found to have a negative ex-ponential distribution with a mean length of 2.5 minutes, (a) What is the probabilitythat someone arriving at the telephone booth will find it occupied? (b) The tele-phone company will install a second booth if a customer must wait an average of

Queuelength 5

m
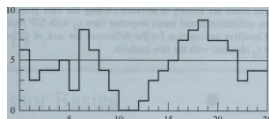
Time t

Figure 2.61

Observed queue lengths in a single-server queueing system.

four minutes or more to gain access to the first telephone. By how much must theflow of customers to the first telephone increase in order for the telephone companyto install the second phone?

2.52. A certain computer system executes a stream of tasks in a manner that can be accu-rately modeled by an M/M/l queueing system. The computer is busy 75 percent ofthe time, and the average job spends four minutes in the computer, (a) How manyjobs are in the computer on average? (b) What is the maximum rate at which jobsmay arrive at the system before it becomes overloaded? State clearly your definitionof overloaded.

2.53. Figure 2.61 shows the queue lengths observed in a single-server queueing systemover a "typical" operating period of 25 time units. Each value of /(?) represents theobserved queue length, including the item being served, at time t. Stating your as-sumptions, answer the following questions about this system, (a) What is the meanqueue length /Q? (b) What is the mean utilization of the server?

2.54. This problem involves manual simulation of a computer system that is executing astream of jobs. The jobs arrive randomly, are queued until selected for execution,and depart immediately after execution is completed. The arrival and executiontimes for a particular job stream are given by the following table:

Job number

1

S

10 11 12

Arrival time:

Execution time (min):

Departure time:

System response time (min)

9:00 9:05 9:08 9:09 9:16 9:21 9:24 9:26 9:32 9:39 9:40 9:43 AM258165824137

Assuming that jobs are executed on a first-come first-served basis, find the mean response time t_Q of the system by completing the above table. What is the computer's utilization factor p from 9:00 am until the last job departs?

2.55. Consider the computer job stream in the preceding problem. Suppose the FCFS queueing discipline is replaced by shortest job first (SJF). in which the next job se-lected for execution is the one in the queue with the shortest execution time. (Assume

136 that all execution times are known in advance.) Using the data given above, deter

mine the system utilization p and mean response time t_Q with SJF replacing FCFS

References

Provide a brief intuitive explanation for the difference (or lack of difference) in the values of p and t_Q obtained with the two methods.

2.6 REFERENCES

1. Actel Corp. FPGA Data Book and Design Guide. Sunnyvale, CA, 1994.

2. Alford, R. C. Programmable Logic Designer's Guide. Indianapolis: Howard W. Sams, 1989.

3. Allen, A. O. "Queueing Models of Computer Systems." IEEE Computer, vol. 13, (April 1980) pp. 13-24.

4. Armstrong, J. R. and F. G. Gray. Structured Logic Design with VHDL. Englewood Cliffs, NJ: Prentice-Hall, 1993.

5. Brayton, R. K. et al. Logic Minimization Algorithms for VLSI Synthesis. Boston: Kluwer, 1984.

6. Brown, F. M. Boolean Reasoning. Boston: Kluwer, 1990.

7. Greene, J., E. Hamdy, and S. Beal. "Antifuse Field Programmable Gate Arrays." Pro-ceedings of the IEEE, vol. 81 (July 1993) pp. 1042-56. [Reprinted in Ref. 1, pp. 4-29 to 4-43].

8. Hachtel, G. D. and F. Somenzi. Logic Synthesis and Verification Algorithms. Boston: Kluwer, 1996.

9. Hayes, J. P. Introduction to Digital Logic Design. Reading, MA: Addison-Wesley, 1993.

10. Kant, K. Introduction to Computer System Performance Evaluation. New York: McGraw-Hill, 1992.

11. McGrory, J. J., A. Carlton, and B. J. Askins. "Transaction Processing Performance on PA-RISC Commercial Unix Systems." Digest of Papers: COMPCON Spring 1992, San Francisco, February 1992, pp. 199-206.

12. McLellan, E. "The Alpha AXP Architecture and 21064 Processor." IEEE Micro, vol. 13 (June 1993) pp. 36-47.

13. Morrison, P. and E. Morrison, eds. Charles Babbage and His Calculating Engines. New York: Dover, 1961.

14. Navabi, Z. VHDL Modeling and Analysis of Digital Systems. New York: McGraw-Hill, 1993.

15. Price, W. J. "Benchmark Tutorial." IEEE Micro, vol. 9 (October 1989) pp. 28^3.

16. Robertazzi, T. G. Computer Networks and Systems: Queueing Theory and Perfor-mance Evaluation. 2nd ed. New York: Springer-Verlag, 1994.

17. Shannon, C. E.: "A Symbolic Analysis of Relay and Switching Circuits." Trans. AIEE, vol. 57 (1938) pp. 713-23. [Reprinted in N. J. A. Sloane and A. D. Wyner, eds. Claude Elwood Shannon Collected Papers. New York: IEEE Press, 1993, pp. 471-95.]

18. Siewiorek, D. P., C. G. Bell, and A. Newell. Computer Structures: Readings and Ex-amples. New York: McGraw-Hill, 1982.

19. Simon, H. A. "The Architecture of Complexity." Proc. Amer. Phil. Soc, vol. 106 (De-cember 1962) pp. 467-82. [Reprinted with revisions in H. A. Simon. The Sciences of the Artificial. 3rd ed. Cambridge, MA: MIT Press, 1996, pp. 183-216.]

20. Smith, D. J. HDL Chip Design. Madison, AL: Doone Publications, 1996.

21. Texas Instruments. TTL Logic Data Book. Dallas, 1988.

22. Thomas, D. E. and P. R. Moorby. The Verilog Hardware Description Language. 3rd ed. Boston: Kluwer, 1996.

CHAPTER 3

Processor Basics

This chapter considers the overall design of instruction-set processors as exempli-fied by the central processing unit (CPU) of a computer. The fundamentals of CPU organization and operation are examined, along with the selection and formats of instruction and data types. Various representative microprocessors of both the RISC and CISC types are presented and discussed.

3.1

CPU ORGANIZATION

We begin by considering the organization of the central processor (microproces-sor) of a computer and the methods used to represent the information it is intended to process.

3.1.1 Fundamentals

The primary function of the CPU and other instruction-set processors is to execute sequences of instructions, that is, programs, which are stored in an external main memory. Program execution is therefore carried out as follows:

1. The CPU transfers instructions and, when necessary, their input data (operands) from main memory to registers in the CPU.

2. The CPU executes the instructions in their stored sequence except when the exe-cution sequence is explicitly altered by a branch instruction.

3. When necessary, the CPU transfers output data (results) from the CPU registers to main memory.

137

138

SECTION 3.1 CPU Organization

Main

CPU   Instructions memory

M

Data

(a)

Cache   Main

CPU Instructions   memory   Instructions memory

CM   MM
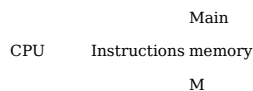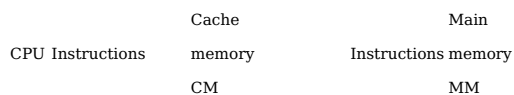
Data   Data

External memory M

(b)

Figure 3.1

Processor-memory communication: (a) without a cache and (b) with a cache.

Consequently, streams of instructions and data flow between the external memoryand the set of registers that forms the CPU's internal memory. The efficient management of these instruction and data streams is a basic function of the CPU.

External communication. If, as in Figure 3.1a, no cache memory is present,the CPU communicates directly with the main memory M, which is typically ahigh-capacity multichip random-access memory (RAM). The CPU is significantlyfaster than M: that is. it can read from or write to the CPU's registers perhaps 5 to10 times faster than it can read from or write to M. VLSI technology, especially thesingle-chip microprocessor, has tended to increase the processor/main-memoryspeed disparity.

To remedy this situation, many computers have a cache memory CM positioned between the CPU and main memory. The cache CM is smaller and fasterthan main memory and may reside, wholly or in part, on the same chip as the CPU.It typically permits the CPU to perform a memory load or store operation in a single clock cycle, whereas a memory access that bypasses the cache and is handledby main memory takes many clock cycles. The cache is designed to be transparentto the CPU's instructions, which "see" the cache and main memory as forming asingle, seamless memory space consisting of 2^m addressable storage locationsM(0), M(l), ..., M(2m-1). In this chapter we will take this viewpoint and use M torefer to the external memory, whether or not a cache is present. A specific memorylocation in M with address adr is referred to as M(adr) or simply as adr. Whennecessary, we will use MM to distinguish the main memory from the cache memory CM, as in Figure 3.1fr. The structure of caches and their interactions with mainmemory are further studied in Chapter 6.

The CPU communicates w ith IO devices in much the same way as it communi-cates with external memory. The IO devices are associated with addressable regis-ters called IO ports to which the CPU can store a word (an output operation) or fromwhich it can load a word (an input operation). In some computers there are no IO

instructions per se; all 10 data transfers are implemented by memory-referencing 139instructions, an approach called memory-mapped 10. This approach requires thatmemory locations and 10 ports share the same set of addresses, so an address bitpattern that is assigned to memory cannot also be assigned to an 10 port, and viceversa. Other computers employ 10 instructions that are distinct from memory-refer-encing instructions. These instructions produce control signals to which 10 ports,but not memory locations, respond. This second approach is sometimes called 10-mapped 10.

User and supervisor modes. The programs executed by a general-purpose com-puter fall into two broad groups: user programs and supervisor programs. A user orapplication program handles a specific application, such as word processing, ofinterest to the computer's users. A supervisor program, on the other hand, managesvarious routine aspects of the computer system on behalf of its users; it is typicallypart of the computer's operating system. Examples of supervisory functions arecontrolling a graphics interface and transferring data between secondary and mainmemory. In normal operation the CPU continually switches back and forth betweenuser and supervisor programs. For example, while executing a user program, theneed often arises for information that is available only on some hard disk unit in thecomputer's IO system. This condition causes the supervisor to temporarily suspendexecution of the user program, execute a routine that initiates the required 10 data-transfer operation, and then resume execution of the user program.

It is generally useful to design a CPU so that it can receive requests for super-visor services directly from secondary memory units and other 10 devices. Such arequest is called an interrupt. In the event of an interrupt, the CPU suspends execu-tion of the program that it is currently executing and transfers to an appropriateinterrupt-handling program. As interrupts, particularly from IO devices, require arapid response from the CPU, it checks frequently for the presence of interruptrequests.

CPU operation. The flowchart in Figure 3.2 summarizes the main functions ofa CPU. The sequence of operations performed by the CPU in processing aninstruction constitutes an instruction cycle. While the details of the instructioncycle vary with the type of instruction, all instructions require two major steps: afetch step during which a new instruction is read from the external memory M andan execute step during which the operations specified by the instruction are exe-cuted. A check for pending interrupt requests is also usually included in theinstruction cycle, as shown in Figure 3.2.

The actions of the CPU during an instruction cycle are defined by a sequenceof microoperations, each of which typically involves a register-transfer operation.The time required for the shortest well-defined CPU microoperation is the CPUcycle time or clock period Tdock and is a basic unit of time for measuring CPUactions. Recall that/, the CPU's clock frequency (in MHz) is related to Tdodt (infis) by rclock = 1//. As we will see, the number of CPU cycles required to process aninstruction varies with the instruction type and the extent to which the processingof individual instructions can be overlapped. For the moment we will assume thateach instruction is fetched from M in one CPU clock cycle (this is usually truewhen M is a cache) and can be executed in another CPU cycle.

SECTION 3.1CPU Organization

(Begin J

^^^ Are^^ there instructions ^>-^ waiting? ^^
No   No   4
I Yes

\   i

Fetch the next

instruction

1!

Execute theinstruction

^s^ Are ^*^^

^^ there interrupts ^>N. waiting? ^

Yes   Transfer to interrupt-

handling program

Figure 3.2

Overview of CPU behavior.

Accumulator-based CPU. Despite the improvements in IC technology overthe years, CPU design continues to be based on the premise that the CPU shouldbe as fast as the available technology and overall design requirements allow.Since cost generally increases with circuit complexity, the number of compo-nents in the CPU must be kept relatively small. The CPU organization proposedby von Neumann and his colleagues for the IAS computer (section 1.2.2) is thebasis for most subsequent designs. It comprises a small set of registers and thecircuits needed to execute a functionally complete set of instructions. In manyearly designs, one of the CPU registers, the accumulator,l played a central role,being used to store an input or output operand (result) in the execution of manyinstructions.

Figure 3.3 shows at the register level the essential structure of a small accu-mulator-oriented CPU. This organization is typical of first-generation computers(compare Figure 1.12) and low-cost microcontrollers. Assume for simplicity thatinstructions and data have some fixed word size n bits and that instructions can beadequately expressed by means of register-transfer operations in our HDL. Instruc-tions are fetched by the program control unit PCU, whose main register is the pro-

'The term accumulator originally meant a device that combined the functions of number storage and addi-tion. Any quantity transferred to an accumulator was automatically added to its previous contents. Accumula-tor is still often used in this restricted sense.

ToM and10 devices

Instructiondecoder   ». Control
                          •••^ signals

Xr-^

IR | | AR | | PC |

1Program   i i   L   U' i

controlunitPCU / *•

              ii

c   System bus

1   i

                              i   r

DR   AC

t ,   r   i i

Arithmetic-logic unit

Dat.   i processing unit DPU
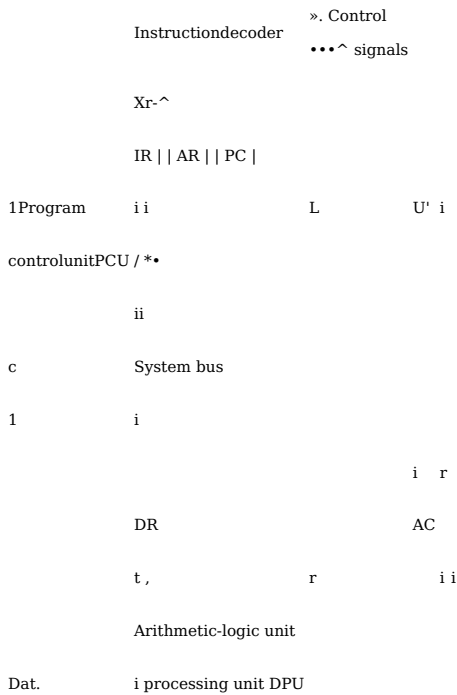
Legend

Program control unit PCUAR: Address registerIR: Instruction registerPC: Program counter

Data processing unit DPUAC: Accumulator registerDR: Data register

141

CHAPTER 3Processor Basics

Figure 33

A small accumulator-based CPU.

gram counter PC. They are executed in the data processing unit DPU. whichcontains an n-bit arithmetic-logic unit (ALU) and two data registers AC and DR.Most instructions perform operations of the form

xi:=y;.(xi,x2)

where XI and X2 denote a CPU register (AC, DR, or PC) or an external memorylocation M(adr). The operations fl performed by the ALU are limited to fixed-point (integer) addition and subtraction, shifting, and logical (word-gate) opera-tions.

Some insti actions have an operand in an external memory location M(adr).and must therefore include the address part adr. Memory addresses are stored intwo address registers in the PCU: the program counter PC, which stores instructionaddresses only, and the general-purpose (data) address register AR. An instruction/ that refers to a data word in M contains two parts, an opcode op and a memoryaddress adr, and may be written as / = op.adr. Each instruction cycle begins withthe instruction fetch operation

IR.AR := M(PC);

(3.1)

which transfers the instruction word / from M to the CPU. The opcode op is loadedinto the PCU's instruction register IR, and the address adr is loaded into addressregister AR. Hence (3.1) is equivalent to

IR := op, AR := adr,

142

SECTION 3.1CPU Organization

Instructions that do not reference M do not use AR; their opcode part specifies theCPU registers to use, as well as the operation/; to be carried out. Once it has placedthe opcode of / in IR, the CPU proceeds to decode and execute it. Note that, at thispoint, the CPU can increment PC in order to obtain the address of the next instruc-tion.

The two essential memory-addressing instructions are called load and store.The load instruction for our sample CPU is

AC := M(adr);

which transfers a word from the memory location with address adr to the accumu-lator. It is often written in assembly-language programs as LD adr. The corre-sponding store instruction is

M(adr) := AC;

which transfers a word from AC to M and may be written as ST adr. Note how the accumulator AC serves as an implicit source or destination register for data words.

Programming considerations. Data-processing operations normally require upto three operands. For example, the addition

Z:=X+Y

(3.2)

has three distinct operands X, Y, and Z. The accumulator-based CPU of Figure 3.3 supports only single-address instructions, that is, instructions with one explicit memory address. However, AC and DR can serve as implicit operand locations so that multioperand operations can be implemented by executing several instructions in sequence. For example, a program to implement (3.2), assuming that X, Y, and Z all refer to data words in M, can take the following form:

HDL format	Assembly-language format	Narrative format (comment)
AC := M(X);	LDX	Load X from M into accumulator AC.
DR := AC;	MOV DR, AC	Move contents of AC to DR.
AC:=M(10;	LD Y	Load Y into accumulator AC.
AC:=AC + DR;	ADD	Add DR to AC.
M(Z):=AC;	ST Z	Store contents of AC in M.

The preceding program fragment uses only the load and store instructions to access memory, a feature called load/store architecture. It is common (but as we will see, not always desirable) to allow other instructions to specify operands in memory. A CPU like that of Figure 3.3 can be designed to implement memory-referencing instructions of the form

AC :=y;(AC, M(adr))

whose execution requires two steps: one to move M{adr) to or from DR and one to perform the designated operation fr With an add instruction of this form, we can reduce the foregoing program from five to three instructions.

143

CHAPTER 3 Processor Basics

HDL

Format

Assembly-language format

Narrative format (comment)

AC := M(X); LD X

AC:=AC + M(K); ADD YM(Z) := AC; ST Z

Load X from M into accumulator AC. Load Y into DR and add to AC. Store contents of AC in M.

The memory-referencing ADD Y instruction can be expected to take longer to exe-cute than the original ADD instruction that references only CPU registers. Memory references also complicate the instruction-decoding logic in the PCU. However, overall execution time should be reduced because we have eliminated an LD and a MOV instruction completely. As we will see later, the cost-performance impact of replacing a simple instruction with a more complex one has subtle implications that lie at the heart of the RISC-CISC debate.

Instruction set. Figure 3.4 gives a possible instruction set for our simple accumulator-based CPU, assuming a load/store architecture. These 10 instruc-tions have the flavor of the instruction sets of some recent RISC machines, which demonstrate that small instruction sets can be both complete and efficient. We are, however, ignoring some important practical implementation issues in the interest of simplicity. We have not, for instance, specified the precise instruction or data formats to be used, and we do not consider such problems as numerical over-flow—this condition occurs when an arithmetic instruction produces a result that is too big to fit in its destination register.

Type	Instruction	HDL format	Assembly-language format	Narrative format (comment)
Data transfer	Load	AC := M(X)	LDX	Load X from M into AC.
	Store	M(X) := AC	STX	Store contents of AC in M asX.Copy contents of AC to DR.
	Move register	DR := AC	MOV DR. AC	
	Move register	AC := DR	MOV AC, DR	Copy contents of DR to AC.
Data processing	Add	AC := AC + DR	ADD	Add DR to AC.
	Subtract	AC := AC - DR	SUB	Suhtract DR from AC.
	And	AC := AC and DR	AND	And hitwise DR to AC.
	Not	AC := not AC	NOT	Complement contents of U
Program control	Branch	PC := M(adr)	BRA adr	Jump to instruction «ith address adr.
	Branch zero	if AC = 0 then PC :=M(adr)	BZadr	Jump to instruction adr itAC = 0.

Figure 3.4

Instruction set for the CPU of Figure 3.3.   -

SECTION 3.1 CPU Organization

The load and store instructions obviously suffice for transferring data between the CPU and main memory. We know from Boolean algebra that the AND and NOT operations are functionally complete, implying that the instruction set enables any logical operation to be programmed. We also know that addition and subtraction suffice for implementing most arithmetic operations. Consider, for example, the arithmetic operation negation, for which many CPUs have a single instruction of the type AC := - AC. We can easily implement negation by a three-instruction sequence as follows:

HDL format	Assembly-language format	Narrative format (comment)
DR := AC;	MOV DR, AC	Copy contents X of AC to DR.
AC := AC - DR; SUB		Compute AC = X - X = 0.
AC := AC - DR; SUB		Compute AC = 0 - X = -X.

Figure 3.4 also gives a small set of program control instructions: an unconditional branch instruction BRA and a conditional branch-on-zero instruction BZ that tests the contents of AC. Observe that these instructions load a new address into the pro-gram counter PC, thus altering the instruction execution sequence. The BZ instruc-tion allows more powerful program control operations such as procedure call and return to be implemented; it also facilitates complex operations such as multiplica-tion, as we demonstrate in Example 3.1.

example 3.1 a multiplication program. Suppose we want to use the tiny instruction set of Figure 3.4 to program the multiplication operation

AC := AC x N

where the multiplicand is the initial contents of the accumulator AC and the multiplier N is a variable stored in memory. We will assume that the multiplier and multiplicand are both unsigned numbers and that they are sufficiently small that the product will fit in a single word. We can construct the desired program along the following lines. We will execute the basic ADD instruction N times to implement AC x N in the form AC + AC + ... + AC. We will treat the memory location storing N as a count register and, after each addition step, decrement it by one until it reaches zero. We will test for N = 0 by means of the BZ instruction, and so we will have to transfer N to AC in order to per-form this test. We will also have to use some memory locations as temporary registers for storing intermediate results and some other quantities, such as the initial value Y of AC. In particular, we will use memory locations one, mult, ac, and prod to store the constant 1, N, Y, and the partial product P, respectively. Here one, mult, ac, and prod are symbolic names for certain memory addresses that we have arbitrarily assigned. They are translated into numerical memory addresses by an assembler program prior to execution.

An assembly-language program implementing this plan appears in Figure 3.5. Its main body (lines 5 to 17) is traversed N times in the course of a multiplication. At the end the result P is in memory location prod. The first two instructions (lines 5 and 6) of the program check the value of N by reading it into AC and testing it with the BZ instruction. If the initial value of N is zero, the program exits immediately with the cor-rect result P = 0. If N is nonzero, the instructions in lines 7 to 11 load it from mult into AC, subtract one from it, and then return the new, decremented value of N to mult. The

Line	Location	Instruction or data	Comment	
0	one	00... 001	The constant one.	145 CHAPTER 3
1	mult	N	The multiplier.	Processor Basics
2	ac	00... 000	Location for initial value Y of AC.	
3	prod	00. .000	Location for (partial) product P.	
4		ST ac	Save initial value y of AC.	
5	loop	LD mult	Load N into AC to test for termination.	
6		BZ exit	Exit if N = 0: otherwise continue.	
7		LD one	Load 1 into AC.	
8		MOV DR. AC	Move 1 from AC to DR.	
9		LD mult	Load N into AC to decrement it.	
10		SUB	Subtract 1 from N.	
11		ST mult	Store decremented N.	
12		LD ac	Load initial value y of AC.	
13		MOV DR. AC	Move K from AC to DR.	
14		LD prod	Load current partial product P.	
15		ADD	Add Y to P.	
16		ST prod	Store the new partial product P.	
17		BRA loop	Branch to loop.	
18	exit			

Figure 3.5

A program for the multiplication operation AC := AC x N.

main step of adding Y to the accumulating partial product, that is, P :- P + Y. is imple-mented in straightforward fashion by lines 12 to 16 of the program. Finally, a return ismade to loop via the unconditional branch BRA (line 17).

This program uses most of the available instruction types and illustrates severalweaknesses of an accumulator-based CPU. Because there are only a few data registersin the CPU, a considerable amount of time is spent shuttling the same information backand forth between the CPU and memory. Indeed, most of the instructions in this pro-gram are of the data-transfer type (ST, LD. and MOV), which do bookkeeping for thefew instructions that actually compute the product P. It would both shorten the pro-gram and speed up its execution if we could store the quantities 1. /V, Y. and P in theirown CPU registers, as they are repeatedly required by the CPU.

Program execution. We now examine the execution process for the multipli-cation program of Figure 3.5. Of course, the program must be translated into exe-cutable object code prior to execution, but we can treat the assembly-languageprogram as a symbolic representation of the object code. Recall that we areassuming that every instruction is one word long and can be fetched from M in asingle CPU clock cycle. We further assume that every instruction is also exe-cuted in a single clock cycle. Hence each instruction requires two CPU clockcycles—one to fetch the instruction from M and one to execute it. At the entl of

-————^————^—————^——^—— Clock Instruction

cycle cycle PC AR PCU actions DPU actions

i ST ac 1004 IR.AR := M(PC), PC := PC + 1

2 1002 M(AR) := AC

3 LD mult 1005 IR.AR := M(PC), PC := PC + 1

4 1001 AC := M(AR)

5 BZ exit 1006 IR.AR := M(PC). PC := PC + 1

6 1001 Test A; no further action if A * 0 None

7 LD one 1007 IR.AR := M(PC). PC := PC + 1

8 1000 AC := M(AR)

9 MOV DR, AC 1008 IR.AR := M(PC). PC := PC + 1

10 dddd DR := AC

11 LD mult 1009 IR.AR := M(PC), PC := PC + 1

12 1001 AC:=M(AR)

13 SUB 1010 IR.AR := M(PC), PC := PC + 1

14 dddd AC:=AC-DR

15 ST mult 1011 IR.AR := M(PC), PC := PC + 1

16 1001 M(AR):=AC

17 LD ac 1012 IR.AR := M(PC), PC := PC + 1

18 1002 AC:=M(AR)

19 MOVDR, AC 1013 IR.AR := M(PC). PC := PC + 1

20 dddd DR := AC

21 LD prod 1014 IR.AR := M(PC), PC := PC + 1

22 1003 AC := M(AR)

23 ADD 1015 IR.AR := M(PC), PC := PC + 1

24 dddd AC := AC + DR

25 ST prod 1016 IR.AR := M(PC). PC := PC + 1

26 1003 M(AR) := AC

27 BRA loop 1017 IR.AR := M(PC), PC := PC + 1

28 1005 PC:=AR None

29 LD mult 1005 IR.AR := M(PC), PC := PC + 1

30 1001 AC:=M(AR)

31 BZ exit 1006 IR.AR := M(PC). PC := PC + 1

32 1018 Test A: PC := AR if A = 0 None

33 1018

Figure 3.6

Cycle-by-cycle execution trace of the multiplication program of Figure 3.5.

the fetch step, the PCU decodes the instruction's opcode to determine what oper- 147ation to perform during the execution stage. It can also increment PC in prepara-tion for the next instruction fetch. Recall that an edge-triggered register can beboth read from and written into in the same clock cycle so that the new data isready for use at the beginning of the next clock cycle. Hence every fetch cycleincludes the following pair of register-transfer operations:

IR.AR := M(PC), PC := PC + 1 (3.3)

The subsequent execution cycle depends on the instruction opcode placed in IR.

Figure 3.6 depicts all the main actions taken by the CPU, including the mem-ory addresses it generates, during execution of the program of Figure 3.5. Data ofthis type is referred to as an execution trace and is often obtained by simulation ofthe target CPU. (In effect, Figure 3.6 is a hand simulation of the multiplication pro-gram.) Execution traces are useful for analyzing program behavior and executionspeed. In this example the program's data and instructions have been assigned to aconsecutive sequence of memory locations 1000, 1001, 1002, . . . , where 1001 isthe location named one in Figure 3.5. The first executable instruction is ST ac,which is in location 1004, so execution begins when PC is set to 1004. Observehow the contents of the program counter PC are incremented steadily until a branchinstruction is encountered, at which point the branch address contained in thebranch instruction may replace the incremented contents of PC.

3.1.2 Additional Features

Next we examine some more advanced features of CPUs and look at representativecommercial microprocessors of the RISC and CISC types.

Architecture extensions. There are many ways in which the basic design ofFigure 3.3 can be improved. Most recent CPUs contain the following extensions,which significantly improve their performance and ease of programming.

• Multipurpose register set for storing data and addresses: These replace the accumu-lator AC and the auxiliary registers DR and AR of our basic CPU. The resulting CPUis sometimes said to have the general register organization exemplified by the third-generation IBM System/360-370 (Figure 1.17), which has 32 such registers. The setof general registers is now usually referred to as a register file.

• Additional data, instruction, and address types: Most CPUs have instructions to han-dle data and addresses with several different word sizes and formats. Although

somemicroprocessors have only add and subtract instructions in the arithmetic category,relatively little extra circuitry is required for (fixed-point) multiply and divideinstructions, which simplify many programming tasks. Call and return instructionsalso simplify program design.

• Register to indicate computation status: A status register (also called a conditioncode or flag register) indicates infrequent or exceptional conditions resulting fromthe instruction execution. Examples are the appearance of an all-zero result or aninvalid instruction like divide by zero. A status register can also indicate the user andsupervisor states. Conditional branch instructions can test the status register, whichsimplifies the programming of conditional actions.

• Program control stack: Various special registers and instructions facilitate the trans-fer of control among programs due to procedure calling or external interrupts. ManyCPUs use a flexible scheme for program-control transfer, which employs part of theexternal memory M as a push-down stack (see also Example 1.5). The stack memoryis intended for saving key information about an interrupted program via push opera-tions so that the saved information can be retrieved later via pop operations. A CPUaddress register called a stack pointer automatically keeps track of the stack's entrypoint.

Figure 3.7 shows the organization of a processor with the foregoing features. Ithas a register file in the DPU for data and/or address storage. The ALU obtainsmost of its operands from the register file and also stores most of its results there. Astatus register monitors the output of the ALU and other key points. The principalspecial-purpose address registers are the program counter and the stack pointer.Special circuits are included for address computation, although the main ALU canalso be used for this purpose. The control circuits in the PCU derive their inputsfrom the instruction register, which stores the opcode of the current instruction, and

Data processing unit DHL*

To M andIO system

Registerfile

Arithmetic-logic unit

Dataregister

Statusregister

System bus

Programcontrolunit PCU

Addressregister

Programcounter

Stackpointer

Instructionregister

Address-generationlogic

Controlcircuits

VInternal control signals

Figure 3.7

A typical CPU with the general register organization.

the status register. Communication with the outside world is via a system bus thattransmits address, data, and control information among the CPU, M, and the 10system. Various nonprogrammable "buffer" registers serve as temporary storagepoints between the system bus and the CPU.

Pipelining. As discussed in Chapter 1, modern CPUs employ a variety ofspeedup techniques, including cache memories, and several forms of instruction-level parallelism. Such parallelism may be present in the internal organization ofthe DPU or in the overlapping of the operations carried out by the DPU and PCU.These features add to the CPU's complexity and will be explored in depth later inthis book.

The considerable potential for parallel processing at the instruction level is evi-dent even in the simple CPU of Figure 3.3. We see from the execution trace of Fig-ure 3.6 that the main PCU and DPU activities take place in different clock cycles.If these activities do not share a resource such as the system bus, they can be car-ried out at the same time. In other words, while the current instruction is being exe-cuted in the DPU, the next instruction can be fetched by the PCU. For example, thethree-instruction negation routine we gave earlier to change AC to -AC would beexecuted as follows in the style of Figure 3.6:

Clockcycle

Instructioncycle

PC

PCU actions

DPU actions

1 MOV DR, AC 2000 IR.AR := M(PC), PC := PC + 1

2 2001 DR:=AC

3 SUB 2001 IR.AR := M(PC), PC := PC + 1

4 2002 AC := AC - DR

5 SUB 2002 IR.AR := M(PC), PC := PC + 1

6 2003 AC := AC - DR

By merging the execution part of each instruction cycle with the fetch part of thefollowing instruction cycle, we can reduce the overall execution time from sixclock cycles to four, as shown below. (We use subscripts to distinguish the first andsecond SUB instructions.)

Clock cycle	Instruction cycle	PC	PCU actions	DPU actions
1	MOV	2000	IR.AR := M(PC), PC = PC+ 1	
2	MOV/SUB,	2001	IR.AR := M(PC), PC = PC+ 1	DR := AC
3	SUB,/SUB2	2002	IR.AR := M(PC), PC = PC+ 1	AC := AC - DR
4	SUB2	2003		AC := AC - DR

This overlapping of instruction fetching and execution is an example of instruction pipelining, which is an important speedup feature of RISC processors. Figure 3.8 illustrates graphically the type of two-stage pipelining discussed abpve. Each instruction can be thought of as passing through two consecutive stages of

Fetch

Execute

Fetch

Execute

Fetch

Execute

Instruction /,Instruction 72Instruction 73 (branch)Instruction 74

Clock cycle 12 3 4

Figure 3.8

Overlapping instructions in a two-stage instruction pipeline.

Fetch

Execute

processing: a fetch stage implemented mainly by the PCU and an execution stage implemented mainly by the DPU. Hence two instructions can be processed simultaneously in every CPU clock cycle, with one completing its fetch phase and the other completing its execute phase. A two-stage pipeline can therefore double the CPU's performance from one instruction every two clock cycles to one instruction every clock cycle.

A problem arises when a branch instruction is encountered, such as the BRAloop instruction stored in address (line) 17 of the multiplication program (Figure3.5). Immediately before this instruction is fetched in some clock cycle i the pro-gram counter PC stores the address 17. PC is then incremented to 18 in preparationfor clock cycle i + 1. Clearly in clock cycle i +1, the CPU should not fetch theinstruction stored at address 18—that instruction is not even in the multiplicationprogram. In clock cycle i + 1, BRA is executed, which causes loop = 5 to be loadedinto PC, implying that the next instruction should be taken from location 5. Thefetching of this instruction can't begin until cycle i + 2, however, as illustrated inFigure 3.8 with i = 4. It follows that we cannot overlap the branch instruction andthe instruction that follows it (73 and 74 in the case of Figure 3.8).

Thus we see that branch instructions reduce the efficiency of instruction pipe-lining, although we will see later that steps can be taken to reduce this problem. Wewill also see that instruction processing is usually broken into more than two stagesto increase the level of the parallelism attainable.

EXAMPLE 3.2 THE ARM6 MICROPROCESSOR [VAN SOMEREN AND ATACK

1994]. We now examine in some detail the architecture of a microprocessor familythat embodies the RISC design philosophy in a relatively direct and elegant form. TheARM has its origins in the Acorn RISC Machine, a microprocessor developed in theUnited Kingdom in the 1980s to serve as the CPU of a personal computer. Subse-quently, the family name was changed—without changing its acronym, however—toAdvanced RISC Machine. The ARM family is primarily aimed at low-cost, low-powerapplications such as portable computers and games. For example, the Newton, a hand-held "personal digital assistant" introduced by Apple Corp. in 1993 employs theARM6 microprocessor, whose main features are described below.

The ARM6 is a 32-bit processor in that both its data words and its address wordsare 32 bits (4 bytes) long. It has a load/store architecture, so only its load and storeinstructions can address external memory M. As in most computers since the IBM Sys-tem/360, main memory is organized as an array of individually addressable bytes. Thus

Processor Basics

the maximum memory size of an ARM6 computer is 232 bytes, also referred to as 4 151

gigabytes (4G bytes). The ARM6 employs an instruction pipeline to meet the goal of

one instruction executed per CPU clock cycle. Note that it shares all these features with CHAPTa more powerful (and more expensive) RISC microprocessor, the PowerPC (Example1.7). The ARM6's instruction set is much smaller than the PowerPC's, however—ithas no floating-point instructions, for example.

The internal organization of the ARM's CPU is shown in Figure 3.9. It has a 32-bitALU and a file of 32-bit general-purpose registers. To permit direct interaction betweendata and control registers, the ARM has the unusual feature of placing its PC and statusregisters in the register file; conceptually, we will continue to view these registers aspart of the PCU. There are several modes of operation, including the normal user andsupervisor modes, and four special modes associated with interrupt handling. In usermode the register file appears to contain sixteen 32-bit registers designated R0:R15.where R15 is also the program counter PC, as well as a current program status registerdesignated CPSR. (Additional registers, which we will not discuss here, are used whenthe CPU is in other operating modes; they are "invisible" in user mode.) The ALU isdesigned to perform basic arithmetic operations on 32-bit integers. It employs combina-tional logic for addition and subtraction and a sequential shift-and-add method similarto that described in Example 2.7 for multiplication. A combinational shift circuit isattached to the ALU to support multiplication and other operations. A separate address-incrementer circuit implements address-manipulation operations such as PC := PC + 1independently of the ALU. Access to external memory M (a cache or main memory) isstraightforward. The address of the desired location in M is placed in the PCU's addressregister. In the case of a store instruction, the data to be stored is also placed in theDPU's write data register. A load instruction causes a data word to be fetched frommemory and placed in the read data register. Several internal buses transfer data effi-ciently among the DPU's registers and data processing circuits.

All ARM6 instructions are 32 bits long, and they have a variety of formats andaddressing modes. There are about 25 main instruction types, which are listed in Fig-ure 3.10. (We have omitted block move and coprocessor instructions.) This number isdeceptively small, however, as instructions have options that substantially increase thenumber of operations they can perform. Most instructions can be applied either to 32-bit operands (words) or to 8-bit operands (bytes). Operands and addresses are usuallystored in registers that can be referred to by short. 4-bit names, allowing a singleARM6 instruction to specify as many as four operands. The available address spaceis shared between memory and 10 devices (memory-mapped IO). Consequently, theload/store instructions used for CPU-memory transfers are also used for 10 opera-tions.

Any instruction can be conditionally executed, meaning that execution may ormay not occur depending on the value of designated status bits (flags) in the CPSR.The status flags are set by a previous instruction and include a negative flag N (the pre-vious result R computed by the ALU was a negative number), a zero flag Z (/? waszero), a carry flag C (R generated an output carry), and an overflow flag V (/? generateda sign overflow). Hence every ARM6 instruction is effectively combined with a condi-tional branch instruction. The basic unconditional move instruction MOV RO. Rl canhave any of 15 conditions attached to it to determine if it is to be executed (see problem3.8). Some examples:

MOVCC RO, Rl ;IfflagC = 0, then RO := R 1

MOVCS RO. Rl ;IfflagC= l.thenRO:=Rl

MOVHI RO. R1 ; If nag C = 1 and flag Z = 0. then RO := R1

Data

processingunitDPU

ToM

and I/O

Register file

Status registers

Program counter PC

A bus H~l

Bbus

Shifter

Arithmetic-logic unit

ALU bus

Write data register

Read data register 1

System bus

Address register

Instruction register

Address incrementer

Program control unit PCU

Controlcircuits

Figure 3.9

Overall organization of the AJIM6.

An ARM6 instruction can also include a shift or rotation operation that is appliedto one of its operands. For instance:

MOV RO, R1,LSL#2

R0:=R1 x4

(3.4)

means logically left shift (LSL) the contents of Rl by 2 bits and move the result to RO.This shift is tantamount to multiplying R1 by four before the move.

The opcode suffix S specifies whether or not an instruction affects the status flags.If S is present, appropriate flags are changed; otherwise, the flags are not affected. Forexample, the ARM6's move instructions affect the N, Z, and C flags, so appending S

153

CHAPTER 3Processor Basics

Type

Instruction

HDL

format

Assembly-language format

Narrativeformat (comment)

Data Move register

transfer Move register

Move invertedLoad

Store

Data Add

processing Add with carry

SubtractSubtract with

carryReverse subtractReverse subtract

with carryMultiply

R3 := R9R0:=12

R7:= ROR5 := M(adr)

M(adr) := R8

R3 := R5 + 25R3 := R5 + R6 + C

R3 :=R5-9R3 := R5 - 9 - C

R3 := 9 - R5R3 := 9 - R5 - C

Rl :=R3xR2

MOVMOV

R3.R9R0,#12

Multiply and add R1 := (R3 x R2) + R4

AndOr

Exclusive-orBit clear

Program Branchcontrol Branch and link

R4:=Rlla/i</25l6R4:=R11 or25l6R4:=Rllxor25,6R4:=R11 a 25I6

PC := PC + adrR14:=PC.PC := PC + adr

Flags :=R1 - 14

Software interrupt

Compare

Compare inverted Flags := Rl + 14

Logical compare Flags := Rl xor 14

Compare inverted Flags := Rl or 14

MVN R7.R0LDR R5, adr

STR R8,adr

ADD R3,R5,#25ADC R3,R5,R6

SUB R3,R5,#9SBC R3,R5,#9

RSB R3.R5,#9RSC R3,R5,#9

MUL R1,R2,R3MLA R1,R2,R3,R4

AND R4.R 11.0x25

ORR R4,R 11.0x25

EOR R4.R 11,0x25

BIC R4,R11,#25

BadrBL adr

SWI

CMP rl.#14CMN rl.#14TEQ rl.#14TST rl.#14

Copy contents of register R9 to register R3Copy operand (decimal number 12) to reg-ister RO.Copy bitwise inverted contents of RO to R7Load R5 with contents of memory location

adr.Store contents of R8 in memory locationadr.

Add 25 to R5; place sum in R3.

Add R6 and carry bit C to R5: place sum in

R3.Subtract 9 from R5; place difference in R3.Subtract 9 and borrow bit from R5; place

difference in R3.Subtract R5 from 9; place difference in R3.Subtract R5 and borrow bit from 9; place

difference in R3.Multiply R3 by R2; place result in Rl.Multiply R3 by R2: add R4; place result in

Rl.Bitwise AND Rl 1 and 25,6; place result in

R4.Bitwise OR Rl 1 and 2516; place result in

R4.Bitwise XOR Rl 1 and 2516; place result in

R4.Bitwise invert 25; AND it to R11. place

result in R4.

Jump to designated instruction.

Save old PC in "link" register R14; then

jump to designated instruction.Enter supervisor mode.Subtract 14 from R1 and set (lags.Add 14 to Rl and set flagsXOR 14 to Rl and set flags.AND 14 to R1 and set flags.

Figure 3.10

Core instruction set of the ARM6.

154 to, say, MOVCS, yields MOVCSS. which checks the moved data item D. It sets N = 1

(0) if D,, = 1 (0), it sets Z = 1 (0) if D is zero (nonzero), and it sets C to the shifter's

SECTION 3.1 output value.

CPU Organization Like Qther rjsCs, the arm6 has an instruction pipeline that permits the various

stages of instruction processing to be overlapped. The pipeline has three stages: fetch,decode, and execute; in effect, the ARM6 breaks the first stage of the two-stage pipe-line of Figure 3.8 in two. This structure permits the CPU to check every instruction'scondition code in stage 2 to determine whether the instruction should be executed instage 3. Some instructions such as multiply require more than one cycle for execution,but most require only one. Note that inclusion of an operand shift in an instruction as in(3.4) does not require an additional cycle, thanks to the fast (combinational) shifter.

A CISC machine. We turn next to a widely used CPU family, the Motorola680X0 family, which was introduced in 1979 with the 68000 microprocessor. Thisexample of an older CISC architecture is more streamlined and "RISC-like" thanother CISCs. Later members of the family such as the 68060 [Circello et al. 1995]have speedup features such as instruction pipelining, floating-point executionunits, and superscalar instruction issue. We examine an intermediate member ofthe series, the 68020, a 32-bit machine whose design broadly resembles that of athird-generation mainframe computer [Motorola 1989].

The 68020 is a one-chip microprocessor introduced in 1985 to serve as theCPU of a general-purpose computer such as a personal computer or workstation.Figure 3.11 outlines the organization of the 68020. It is designed to handle 32-bitwords (termed long words in 680X0 literature) efficiently, but instructions are alsoprovided to handle operands of 1, 8, 16, and 64 bits. As in the ARM6, memoryaddresses are 32 bits long, permitting a total of 232 different memory locations,each storing 1 byte. Memory-mapped IO is also used in the 680X0 series. The data-processing unit has a register file containing sixteen 32-bit registers, half of whichare data registers designated D0:D7 and half are address registers designatedA0:A7. The ALU can execute a large set of fixed-point (but not floating-point)instructions. Instruction interpretation and other control functions of the CPU areimplemented by a microprogrammed control unit.

The 68020 has about 70 distinct instruction types (or around 200 if all opcodevariants are distinguished), which are summarized in Figure 3.12. A given instruc-tion such as MOVE can be defined with several different types of operands, andthe operands can be addressed in various ways. For example, the following move-register instruction written in 680X0 assembly-language format

MOVE.L DLA6 (3.5)

causes the entire contents (a long word as indicated by the opcode suffix .L) of dataregister Dl to be copied to address register A6. In other words, (3.5) implementsthe register transfer A6 := Dl. If .L is replaced by .B, then the resulting instruction

MOVE.B D1,A6

causes only the byte stored in the low-order position (bits 0:7) of Dl to be copiedto the corresponding part of A6.

Besides the direct addressing mode illustrated by the preceding example, the68020 has several other addressing modes that give the programmer considerable

Program control unit PCU

Control memory(microrom)

Control memory 2(nanorom)

Addresssequencer

Instructionqueue

Instructioncache

155

CHAPTER 3Processor Basics

Main control signals

Data

processingunitDPU

Arithmetic-logic

Data registers

Address registers

DODlD2

D3D4D5DbD7

AOAlA2A3A4A5A6

User-programmableregisters

A7 | User stack pointer |

PC [Program counterUser status register(condition code) | CC |

A7' System stack pointer

System status registerSupervisor registers

Buscontrol

circuits

ToMand 10

r=C

System bus

Figure 3.11

Organization of the 68020.

flexibility in accessing data. Most instructions can address memory as well as CPUregisters. For example, if (3.5) is replaced by

MOVE.L D1.(A6) (3.6)

the resulting operation is M(A6) := D1, that is, a store operation with A6 serving asthe memory-address register. This is an instance of indirect addressing. Note thatwhile (3.5) takes 4 clock cycles to execute, (3.6) takes 12 cycles because of thetime required to access external memory. The 68020's data-processing instructionscan also access M directly, so the 68020 does not have the load/store architecture

156

Type

Opcode Description

SECTION 3.1CPU Organization

Data transfer EXGMOVEMOVEAMOVECMOVEMMOVEPMOVEQMOVES

Dataprocessing

SWAP

ABCD

ADD

ADDA

ADDI

ADDQ

ADDX

AND*

AS*

CLR

DIVx

EORjc

EXT

LSjc

MULx

NBCD

NEG

NEGX

NOT

OR*

PACK*

ROx

ROXx

SBCD

SUB

SUBA

SUBI

SUBQ

SUBX

UNPK*

Exchange (swap) contents of two registers.

Move (copy) data unchanged from source to destination in CPU or M.Copy data to address register.

Copy data to or from control register (privileged instruction).Copy multiple data items to or from specified list of registers.Copy data between register and alternate bytes of memory.Copy "quick" (8-bit) immediate data to register.Copy data using address space specified by a control register (privi-leged instruction).Swap left and right halves of register.

Add decimal (BCD) numbers with carry (extend) flag.

Add binary (twos-complement) numbers.

Add to address register (unsigned binary addition).

Add immediate binary operand.

Add "quick" (3-bit) immediate binary operand.

Add binary with carry (extension) flag.

Bitwise logical AND (x = I denotes immediate operand).

Arithmetic left (x = L) or right (x = R) shift with extension.

Clear operand by resetting all bits to 0.

Divide signed (x = S) or unsigned (x = U) binary numbers.

Bitwise logical EXCLUSIVE OR (x = I denotes immediate operand).

Extend the sign bit of subword to fill register.

Logical (simple) left (x = L) or right (x = R) shift.

Multiply signed (x = S) or unsigned (x = U) binary numbers.

Negate decimal number (subtract with carry from zero).

Negate binary number (subtract from zero).

Negate binary number (subtract with carry from zero).

Bitwise logical complement.

Bitwise logical OR (x = I denotes immediate operand).

Convert number from unpacked to packed BCD format.

Rotate (circular shift) left (x = L) or right (x = R).

Rotate left (x = L) or right (x = R) including the X (extend) flag.

Subtract decimal (BCD) numbers.

Subtract binary (twos-complement) numbers.

Subtract from address register (unsigned binary subtraction).

Subtract immediate binary operand.

Subtract "quick" (3-bit) immediate binary operand.

Subtract binary with borrow (extend) flag.

Convert number from packed to unpacked BCD format.

Figure 3.12

Instruction set of the 68020.

characteristic of a RISC. For example:

ADD (A0), DOspecifies the memory-to-register add operation DO := M(A0) + DO.

EXAMPLE 3.3 680X0 PROGRAM FOR VECTOR ADDITION. Figure 3.13 gives

an example of 680X0 assembly-language code that illustrates several of its basicinstruction types and addressing methods. This program adds two 1000-element vec-tors A and B to produce a third vector C. Each vector is assumed to be a decimal

Type Opcode Description

Program Bcc Branch relative to PC if specified condition code cc is set.

control Bxcx Test, modify, and/or transfer (depending on xxx) a specified bit; set Z flag to

indicate old bit value.BExxt* Test, modify, and/or transfer (depending on xxx) a specified bit field; set flags

to indicate old bit-field value.BKPT* Execute a breakpoint trap (used for debugging).BRA Branch unconditionally relative to PC.

BSR Call (branch to) subroutine at address relative to PC; save old PC in stack.

CALLM* Call subroutine (program module) saving specified control information in stack.CASx* Compare specified operands and update register.CHKx Check register against specified values (address bounds); trap if bounds are

exceeded.CMP* Compare two operand values; set flags based on result; x indicates operand

type.DBcc Loop instruction: Test condition cc and perform no operation if condition is

met; otherwise, decrement specified register and branch to specified address.ILLEGAL* Perform trap operation corresponding to an illegal opcode.JMP Branch unconditionally to specified (nonrelative) address.

JSR Call (jump to) subroutine at specified (nonrelative) address; save old PC in

stack.LEA Compute effective address and load into address register.

LINK Allocate local data and parameter region in the stack.

NOP No operation (except increment PC); instruction execution continues.

PEA Compute effective address and push into stack.

RTD Return from subroutine and deallocate stack parameter region.

RTE Return from exception (privileged instruction).

RTM* Return and restore control (module state) information.RTR Return and restore condition codes.

RTS Return from subroutine.

Sec Set operand to Is (Os) if condition code cc is true (false).

STOP Load status register and halt (privileged instruction).

TRAP Begin exception processing at specified address.

TRAPcc If condition cc is true, then begin exception processing.TST Test an operand by comparing it to zero and setting flags.

UNLK Deallocate local data and parameter area in the stack.

External cpxwr* If condition holds, then branch with external coprocessor as specified by xxx.synchro- RESET Reset or restart external device (privileged instruction),nization TAS Test operand and set one of its bits to 1 using an indivisible memory-access

cycle.

♦Instruction not in the original 68000 instruction set.

Figure 3.12

(continued).

CHAPTER 3Processor Basics

number composed of 1000 two-digit bytes. Each vector is stored in a fixed block ofmain memory whose location is known. For example, vector A is stored in memorylocations 1001,1002,1003, ...,1999,2000.

The desired addition is accomplished by executing the ABCD (add using the BCDnumber format) instruction 1000 times. The address registers A0, Al, and A2 are usedas pointers to the current 1-byte operands, and they are initialized to the required start-ing values using the first three MOVE instructions. These instructions use immediateaddressing denoted by the prefix # to specify instruction fields that contain 'actualaddress values, while a register name such as A0 indicates that the desired operand is

SECTION 3.1CPU Organization

Location Instruction

Comment

MOVE.L #2001, A0

MOVE.L #3001, A1

MOVE.L #4001. A2

START ABCD -<A0),-(A1)

MOVE.B (A1),-(A2)

TEST CMPA #1001, A0

BNE

START

Load address 2001 into register A0 (pointer to vector A).

Load address 3001 into register A1 (pointer to vector B).

Load address 4001 into Kgister A2 (pointer to vector C).

Decrement contents of A0 and Al by 1, then add M(A0) toM(A1) using 1-byte decimal addition.

Decrement A2 and then store the 1-byte sum M(A1) inlocation M(A2) of vector C.

Compare 1001 to address in A0. If equal, set the Z flag(condition code) to 1; otherwise, reset Z to 0.

Branch to START if Z is not equal to 1.

Figure 3.13

680X0 assembly-language program for vector addition.

the contents of the named register—this is direct addressing. The ABCD and MOVE.B(move byte) instructions use indirect addressing, indicated by parentheses. In this casethe data specified by (A0) is the content of the memory location whose address isstored in A0. that is, the data in M(A0). Finally the minus prefix in the operand -(A0)means that A0 is decremented by one before it is used to access main memory, a modeof addressing called autoindexing.

The program of Figure 3.13 loads three starting addresses into the selected addressregisters. Since the ABCD and MOVE.B instructions begin by automatically decre-menting these registers, their initial values are made one bigger than the biggestaddress assigned to the corresponding vector. The ABCD instruction performs the fol-lowing set of operations:

A0:=A0- 1,A1 :=A1-1;M(A1):=M(A1) + M(A0); set flags

which are relatively slow because of the memory access required. The MOVE.Binstruction implements the memory-to-memory move operation with autoindexing

A2:=A2- 1; M(A2):=M(A1); set flags

The compare-address instruction CMPA checks for program termination by comparingthe current address in A0 to 1001, the lowest address assigned to vector A. It actuallysubtracts its first operand (1001 in this case) from its second and sets the status flags(condition code) based on the result. Hence if A0 > 1001, then A0 - 1001 > 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

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, loga-

rithms, 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 num-bers. Two main number formats have evolved: fixed-point and floating-point. Thebinary fixed-point format takes the form $bAb^\wedge)c.. .bK$, 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 $M \times B^E$, 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 <^ ^-""^ " Decimal Processor Basics

Numbers -

^\. / Binary

Floating-point <^

Nonnumerical data \ Decimal

Figure 3.15

The basic information types.

the chapter. Nonnumerical data usually take the form of variable-length characterstrings encoded in one of several standard codes, such as ASCII (American Stan-dards Committee on Information Exchange) code.

Word length. Information is represented in a digital computer by means ofbinary words, where a word is a unit of information of some fixed length n. An n-bit word allows up to 2^n different items to be represented. For example, with n =4, we can encode the 10 decimal digits as follows:

0 = 0000 1=0001 2 = 0010 3 = 0011 4 = 0100

5 = 0101 6 = 0110 7 = 0111 8=1000 9=1001 (3'7)

To encode alphanumeric symbols or characters, 8-bit words called bytes are com-monly used. As well as being able to encode all the standard keyboard symbols, abyte allows efficient representation of decimal numbers that are encoded in binaryaccording to (3.7). A byte can store two decimal digits with no wasted space. Mostcomputers have the 8-bit byte as the smallest addressable unit of information intheir main memories. The CPU also has a standard word size for the data it pro-cesses. Word size is

typically a multiple of 8, common CPU word sizes being 8,16, 32, and 64 bits.

No single word length is suitable for representing every kind of information encountered in a typical computer. Even within a single domain such as a com-puter's instruction set, we often find several different word sizes. For example, instructions such as load and store that reference memory need long address fields. Instructions whose operands are all in the CPU need not contain memory addresses and so can be shorter. The precision of a number word is determined by its length; it is common therefore to have numbers of various sizes. Figure 3.16 gives a sam-pling of data sizes used by the Motorola 680X0. As here, the term word is often restricted to mean a 32-bit (4 byte) word. (680X0 literature refers to 32-bit words with the nonstandard term long word.) Fixed-point numbers come in lengths of 1,2, 4, or more bytes. Floating-point numbers also come in several lengths, the short-est (single precision) number being one word (32 bits) long.

The circuits of a CPU must be carefully designed to permit various informa-tion formats to coexist smoothly. For example, if instruction length varies, as is the case in many CISC microprocessors, the program control unit must be designed to determine an instruction's length from its opcode and to fetch a variable number of instruction bytes from memory. It must also increment the program counter by a

162

Bits Name

Illustration

Typical uses

1 Bit

8 Byte

16 Halfword

32 Word

D

64 Double word I 1 f

Status flag. Logic variable.

Smallest addressable memory item.

Binary-coded decimal digit pair.

<

Short fixed-point number. Short address (offset).Short instruction.

Fixed- or floating-point number. Memoryaddress. Instruction.

'

1 1 1 1 Long instruction. Double-precision

""l!l I.i.imiIjmmiiI n . . *

floating-point number.

Figure 3.16

Some information formats of the Motorola 680X0 microprocessor series.

variable amount to obtain the address of the next consecutive instruction. Thus while the ARM6 has instructions of length 4 bytes only, the 68020's instructions range in length from 2 to 10 bytes.

Instruction sets commonly have features to make it easy to apply instructions to nonstandard-length operands. An example is the add-with-carry (ADC) instruc-tion and its counterpart subtract with carry, which enable add and subtract instruc-tions to apply to long fixed-point numbers by adding them in short segments and propagating carries from segment to segment. Suppose, for example, that we want to add two unsigned 64-bit (double word) binary integers A and B using the ARM6 instruction set (Figure 3.10), which is designed to add 32-bit words. Let A be placed in registers R0 and R1, with the right (least significant) half of A in R0. Similarly, let B be placed in registers R2 and R3, with its right half in R2. We first apply the ADD instruction with inputs R0 and R2 and place the resulting sum in R4. We also instruct ADD to activate the status flags, which requires an 5 suf-fix to the ARM6 opcode, changing it to ADDS. (In most other computers the flags are set automatically by all data-processing instructions.) ADDS results in the carry flag C assuming the value of the carry-out bit produced by the addition R0 + R2. Then we apply the ADC (add with carry) instruction with inputs R1 and R3 to compute the sum Rl + R3. In the following ARM6 code, the final sum A + B is placed in R4 and R5.

HDL format

ARM6

assembly-languageformat

Narrative format (comment)

C.R4 := R0 + R2 ADDS R4.R0.R2

R5: = R1+R3+C ADC R5,R1,R3

Add right words and store carry signal C.Add left words plus C.

Storage order. A small but important aspect of data representation is the way in which the bits of a word are indexed. We will usually follow the convention illustrated in Figure 3.17, where the right-most bit is assigned the index 0 and the bits are labeled in increasing order from right to left. The advantage of this conven-tion is that when the word is interpreted as an unsigned binary integer, the low-order indexes correspond to the numerically less significant bits and the high-order indexes correspond to the numerically more significant bits. Similarly, we label the

Byte 3 Byte 2

i i i i i i i I i i i i i i i

Byte 1

i i i i i i

ByteO

'

163

H 23 15 ? Figure 3.17 Processor Basics

Most Least b

significant significant Indexing convention for the bits

bit bit and bytes of a word.

bytes of a word from right to left, with index 0 assigned to the numerically least significant byte. Figure 3.17 therefore shows the format used to store a 4-byte word in a one-word register.

Since words are stored as individually addressable bytes in memory M, a ques-tion arises as to the storage order in M of the bytes within each word. Suppose that a

sequence W0,Wl,...,Wmof m 4-byte number words is to be stored. Suppose fur-ther that we write W(as Bl3,Bj2,BiA,BlQ, where as in Figure 3.17, we place the leastsignificant byte BiQ on the right and assign it the lowest index 0. Now the entiresequence can be rewritten as

W0,Wx,...,Wm = BQ3,B02,B0l,B00,Bl3,Bl2,BluBl0,...,

Bm,3>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].

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 indistin-guishability 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 binaryhalf-word, 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 $X = (x_0, x_u .., *_{n-1})$ to form the(n + l)-bit word $X^* = (x_0, *_1, . . . , *_{n-1}, %)$; 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

$C_n = X_n © X_1 © ... © X$

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 $X' = (x'_0, x_1, . . . , x_n vc'_0)$. B then determines the parity of the receivedword by recomputing the parity bit according to (3.9) thus:

CHAPTER 3Processor Basics

C*n = x'n © X ,

© •*'„-!

SECTION 3.2Data Representation
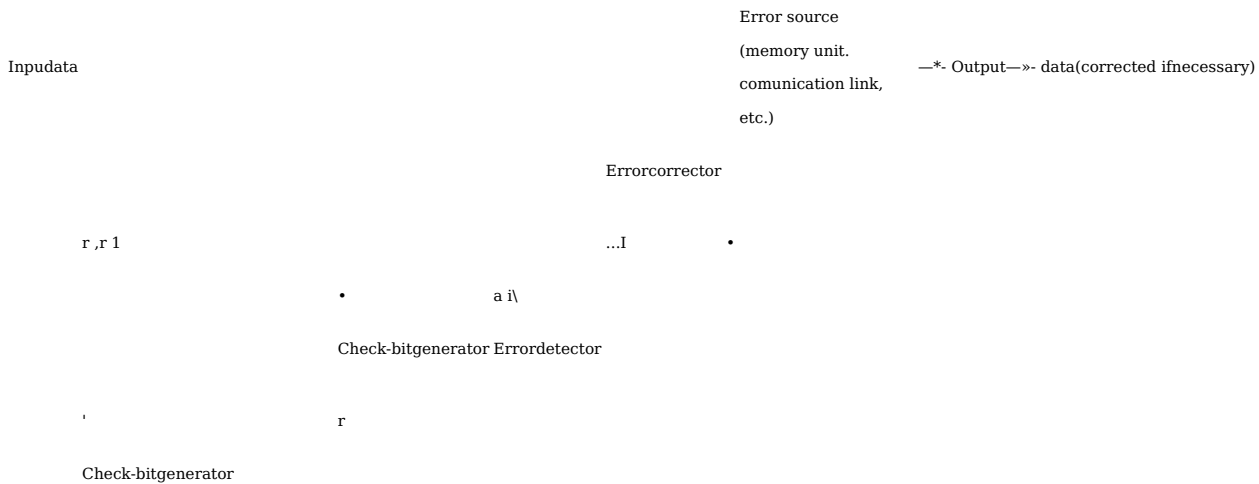
Erro r detection and correction logic.

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

$2^C > n + c + 1$

(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

$1 \times 10^2 + 9 \times 10^1 + 2 \times 10^0 + 7 \times 10^{"1} + 3 \times 1(T^2$

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

leads to the format

SECTION 3.2 xn-\xn-2xn-2 ■ ■ ■ *2*1*0 (3-12)

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 <\N\ < 2" - 1. The other most widely used fixed-point format treats (3.12) as a frac-tion with the binary point lying between xn_x and xn_2. The fraction format denotesnumbers with magnitudes in the range 0 < IM < 1 - 2~n.

Signed numbers. Suppose that both positive and negative binary numbers areto be represented by an n-bit word X = x^x^yX^.. .x2xlx0. The standard formatfor positive numbers is given by (3.12) with a sign bit of 0 on the left and the mag-nitude to the right in the usual positional notation. This means that each magnitudebit xh 0 < / < n - 2, has a fixed weight of the form 2k+l, where k depends on theposition of the binary point. A natural way to represent negative numbers is toemploy the same positional notation for the magnitude and simply change the signbit xn_\ to 1 to indicate minus. Thus with n = 8, +75 = 01001011, while -75 =11001011. This number code is called sign magnitude. Note that humans normallyuse decimal versions of sign-magnitude code. Nevertheless, operations like sub-traction are costly to implement by logic circuits when sign-magnitude codes areused. However, multiplication and division of sign-magnitude numbers is almostas easy as the corresponding operation for unsigned numbers, as Example 2.7 (sec-tion 2.3.3) shows.

Several number codes have been devised that use the same representation forpositive numbers as the sign-magnitude code but represent negative numbers indifferent ways. For example, in the ones-complement code, -X is denoted by X, thebitwise logical complement of X. In this code we again have +75 = 01001011, butnow -75 = 10110100. In the twos-complement code, -X is formed by adding 1 tothe least significant bit of X and ignoring any carry bit generated from the mostsignificant (sign) position. If X = xn_xxn_2. ■ .x0 is an n-bit binary fraction, -X can beexpressed as follows:

-X= xn_x .xn_2xn_i...xlxQ+0.00 ...0\ (modulo2) (3.13)

I • I

Implicit binary point Implicit binary point

where the use of modulo-2 addition corresponds to ignoring carries from the signposition. If X is an integer, then (3.13) becomes

-X=xn_xxn_2xn_3...xxx0.+000...0\. (modulo2") (3.14)

I I

Implicit binary point Implicit binary point

For example, in twos-complement code +75 = 01001011 and -75 = 10110101.Note that in both complement codes x„_, retains its role as the sign bit, but theremaining bits no longer form a simple positional code when the number is nega-tive.

The primary advantage of the complement codes is that subtraction can be per-formed by logical complementation and addition only. Consider the twos-complement code. To subtract X from Y, just add -X to Y, where -X is obtained bylogical complementation and addition of a 1 bit, as in (3.13) and (3.14). As we willsee later, the sign bits do not require special treatment; consequently, twos-complement addition and subtraction can be implemented by a simple adderdesigned for unsigned numbers. Multiplication and division are more difficult toimplement if twos-complement code is used instead of sign magnitude. The addi-tion of ones-complement numbers is complicated by the fact that a carry bit fromthe most significant magnitude bit xn_2 must be added to the least significant bitposition x0. Otherwise ones-complement codes are quite similar to twos-comple-ment codes and so will not be considered further.

Figure 3.21 illustrates how integers are represented using all three codeswhen n = 4. These codes are all referred to as binary codes to distinguish themfrom the so-called decimal codes discussed below. Observe that in all cases, 0000represents zero. Only in the case of twos-complement code, however, is the nega-

169

CHAPTER 3Processor Basics

Decimal representation	Binary code		
	Sign magnitude	Ones complement	Twos complement
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+ 1	0001	0001	0001
+0	0000	0000	0000
-0	1000	mi	0000
-1	1001	1110	nn
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	mi	1000	1001

Figure 3.21

Comparison of three 4-bit codes for signed binary numbers.

170 tive (numerical complement) of 0000 also 0000. This unique representation of

,, zero is a significant advantage, for example, in implementing instructions like

Data Representation BNE in Figure 3.13 that test for zero. Consequently, twos-complement code is byfar the most popular code for representing signed binary numbers in computers.

i

Exceptional conditions. If the result of an arithmetic operation involving n-bitnumbers is too large (small) to be represented by n bits, overflow (underflow) issaid to occur. It is generally necessary to detect overflow and underflow, since theymay indicate bad data or a programming error. Consider, for example, the additionoperation

$Zn\text{-}\backslash Zn\text{-}2\text{-} \blacksquare\blacksquare^q'\text{-}= Xn\text{-}\backslash Xn\text{-}2\text{-} \blacksquare \text{-}X0 + Vn\text{-}l>'/i\text{-}2'' \blacksquare \text{-}^O$

using «-bit twos-complement operands. Assume that bitwise addition is performedwith a carry bit c, generated by the addition of xt, y,, and c,_,. The output bits z, andCj can be computed according to the full-adder logic equations

$c, = xft + x,\text{-}cM + yf^$

Let v be a binary variable indicating overflow when v = 1. Figure 3.22 shows howthe sign bit $z_{,,}$, and v are determined as functions of the sign bits xn_x, yn_i and thecarry bit $c_{,,}2$. The overflow indicator v is therefore defined by the logic equation

$v = \bullet X/i\text{-}l>'/i\text{-}lC/i\text{-}2 + xn\text{-}l>n\text{-}l\ Cn\text{-}2$

If the combinations {.xn_x,yn_x,cn_i) = (0,0,1) and (1,1,0), which make v = 1, areremoved from the truth table of Figure 3.22, then zn_x is defined correctly for all theremaining combinations by the equation

$z_{,,}i = *_{,,}i © y_{,,}i © c_{,,}2$

Consequently, during twos-complement addition the sign bits of the operands canbe treated in the same way as the remaining (magnitude) bits.

A related issue in computer arithmetic is round-off error, which results fromthe fact that every number must be represented by a limited number of bits. An

Inputs			Outputs	
Xn-\	>Vi	Cn-1	Zn-l	V
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	0
	0	1	0	0
111	1	0	1	1
	1	1	1	0

Figure 3.22

Computation of the sign bit $;_{,,}$, and the overflow

indicator v in twos-complement addition.

operation involving n-bit numbers frequently produces a result of more than n bits. 171For example, the product of two Ai-bit numbers contains up to In bits, all but n ofwhich must normally be discarded. Retaining the n most significant bits of theresult without modification is called truncation. Clearly the resulting number is inerror by the amount of the discarded digits. This error can be reduced by a processcalled rounding. One way of rounding is to add r;/2 to the number before trunca-tion, where r7 is the weight of the least significant retained digit. For instance, toround 0.346712 to three decimal places, add 0.0005 to obtain 0.347212 and thentake the three most significant digits 0.347. Simple truncation yields the less accu-rate value 0.346. Successive computations can cause round-off errors to build upunless countermeasures are taken. The number formats provided in a computershould have sufficient precision that round-off errors are of no consequence tomost users. It is also desirable to provide facilities for performing arithmetic to ahigher degree of precision if required. Such high precision is usually achieved byusing several words to represent a single number and writing special subroutines toperform multiword, or multiple-precision, arithmetic.

Decimal numbers. Since humans use decimal arithmetic, numbers beingentered into a computer must first be converted from decimal to some binary rep-resentation. Similarly, binary-to-decimal conversion is a normal part of the com-puter's output processes. In certain applications the number of decimal-binaryconversions forms a large fraction of the total number of elementary operationsperformed by the computer. In such cases, number conversion should be carriedout rapidly. The various binary number codes discussed above do not lend them-selves to rapid conversion. For example, converting an unsigned binary numberxn-ixn-2---xot0 decimal requires a polynomial of the form

/i-i

Jfc+i

L*,2'

to be evaluated.

Several number codes exist that facilitate rapid binary-decimal conversion byencoding each decimal digit separately by a sequence of bits. Codes of this kind arecalled decimal codes. The most widely used decimal code is the BCD {binary-coded decimal) code. In BCD format each digit di of a decimal number is denotedby its 4-bit equivalent bi3bi2biAbj0 in standard form, as in (3.7). Thus theBCD number representing 971 is 100101110001. BCD is a weighted (positional)number code, since bLj has the weight 10'27. Signed BCD numbers employ decimalversions of the sign-magnitude or complement formats. The 8-bit ASCII code rep-resents the 10 decimal digits by a 4-bit BCD field; the remaining 4 bits of theASCII code word have no numerical significance.

Two other decimal codes of moderate importance are shown in Figure 3.23.The excess-three code can be formed by adding 00112 to the corresponding BCDnumber— hence its name. The advantage of the excess-three code is that it ma\ beprocessed using the same logic used for binary codes. If two excess-three num-bers are added like binary numbers, the required decimal carry is automaticallygenerated from the high-order bits. The sum must be corrected by adding +3. For

CHAPTER 3Processor Basics

Decimaldigit		Decimal code			
SECTION 3.2Data Representation	BCD	ASCII	Excess-three		Two-out-of-five
0	0000	0011 0000	0011		11000

1	0001	0011 0001	0100	ooo it
2	0010	0011 0010	0101	00101
3	0011	00110011	0110	00110
4	0100	0011 0100	0111	01001
5	0101	0011 0101	1000	01010
6	0110	00110110	1001	01100
7	0111	0011 0111	1010	10001
8	1000	0011 1000	1011	10010
9	1001	0011 1001	1100	10100

Figure 3.23

Some important decimal number codes.

example, consider the addition 5 + 9 = 14 using excess-three code.

1000 = 5+ 1100 = 9Carry 1 <— 0100 Binary sum

+ 0011 Correction

0111 =4 Excess-three sum

Binary addition of the BCD representations of 5 and 9 results in 1110 and no carrygeneration. (The binary sum of two BCD numbers can also be corrected to give theproper BCD sum as described later.) Some arithmetic operations are difficult toimplement using excess-three code, mainly because it is a nonweighted code; thatis, each bit position in an excess-three number does not have a fixed weight.

The final decimal code illustrated by Figure 3.23 is the two-out-of-five code.Each decimal digit is represented by a 5-bit sequence containing two Is and three0s; there are exactly 10 distinct sequences of this type. The particular merit of thetwo-out-of-five code is that it is single-error detecting, since changing any one bitresults in a sequence that does not correspond to a valid code word. Its drawbacksare that it is a nonweighted code and uses 5 rather than 4 bits per decimal digit.

The main advantage of the decimal codes is ease of conversion between theinternal computer representation that allows only the symbols 0, 1 and externalrepresentations using the 10 decimal symbols 0, 1, 2,..., 9. Decimal codes havetwo disadvantages.

1. They use more bits to represent a number than the binary codes. Decimal codestherefore require more memory space. An n-bit word can represent 2" numbersusing binary codes; approximately 10"/4 = 20830" numbers can be represented ifa 4-bit decimal code such as BCD or excess-three is used.

2. The circuitry required to perform arithmetic using decimal operands is morecomplex than that needed for binary arithmetic. For example, in adding BCD

numbers bit by bit, a uniform method of propagating carries between adjacent 173positions is not possible, since the weights of adjacent bits do not differ by aconstant factor.

CHAPTER 3Processor Basics

Hexadecimal numbers. One or two other numerical codes are encountered inthe design or use of computers. Of particular importance is hexadecimal (hex)code, which is characterized by a base r = 16 and the use of 16 digits, consisting ofthe decimal digits 0,1,...,9 augmented by the six digits A,B,C,D,E, and F, whichhave the numerical values 10, 11, 12, 13, 14, and 15, respectively. The unsignedhexadecimal integer 2FA0C has the interpretation

$2 \times 16^4 + F \times 16^3 + A \times 16^2 + 0 \times 16^1 + C \times 16^0$

$= 2 \times 65,536 + 15 \times 4,096 + 10 \times 256 + 0 \times 16 + 12 \times 1 = 195,084$

Hence $2FA0C_{16} = 195,084_{10}$.

Hexadecimal code is useful for representing long binary numbers, a conse-quence of the fact that the base 16 is a power of two. A hexadecimal number isconverted to binary simply by replacing each hex digit by the equivalent 4-bitbinary form. For example, we can convert $2FA0C_{16}$ to binary by replacing the firstdigit 2 by 0010, the second digit F by 1111, the third digit A by 1010, and so on,yielding

$2FA0C_{16} = 0010111110100000011002$

Conversely, we can convert a binary number to hex form by replacing each four-digit group by the corresponding hex digit. Clearly hexadecimal-binary numberconversion is very similar to BCD-binary conversion. By treating any binary wordas an unsigned integer, we can easily convert the word to hex form as indicatedabove. Hex code provides a very convenient shorthand for binary information.

3.2.3 Floating-Point Numbers

The range of numbers that can be represented by a fixed-point number code isinsufficient for many applications, particularly scientific computations where verylarge and very small numbers are encountered. Scientific notation permits us torepresent such numbers using relatively few digits. For example, it is easier towrite a quintillion as

1.0×10^{18} (3.15)

than as the 19-bit, fixed-point integer 1 000 000 000 000 000 000. The floating-point codes used in computers are binary (or binary-coded) versions of (3.15).

Basic formats. Three numbers are associated with a floating-point number: amantissa M, an exponent E, and a base B. The mantissa M is also referred to as thesignificand or fraction in the literature. These three components together representthe real number $M \times B^E$. For example, in (3.15) 1.0 is the mantissa, 18 is the expo-nent, and 10 is the base. For machine implementation the mantissa and exponentare encoded as fixed-point numbers with a base r that is usually 1 or 10. The base B

174

SECTION 3.2Data Representation

is also r, or some power of r, for reasons that will become obvious. Since B is aconstant, it need not be included in the number code; it is simply built into the cir-cuits that process the numbers. A floating-point number is therefore stored as aword (M,E) consisting of a pair of signed fixed-point numbers: a mantissa M,which is usually a fraction or an integer, and an exponent E, which is an integer.The number of digits in M determines the precision of (M,£); B and E determine itsrange. With a word size of n bits, 2" is the most real numbers that (M,E) can repre-sent. Increasing B increases the range of the representable real numbers but resultsin a sparser distribution of numbers over that range.

As a small example, suppose that M and E are both 3-bit, sign-magnitude inte-gers and B = 2. Then M and E can each assume the values ±0, ±1, ±2, and ±3. Allbinary words of the form (M,E) = (.x00, xxx) represent zero, where x denotes either0 or 1. The smallest nonzero positive number is (001,111), denoting $1 \times 2^{-3} = 0.125$; (101,111) denotes -0.125. The largest representable positive number is(011,011), which denotes $3 \times 2^3 = 24$, while (111,011) denotes the largest negativenumber -24. Observe that the left-most bit, which is the sign of the mantissa, isalso the sign of the floating-point number. Figure 3.24 illustrates the real numbersrepresentable by this 6-bit, floating-point format. As the figure shows, they aresparsely and nonuniformly distributed over the range ±24.

The floating-point representation of most real numbers is only approximate.For instance, the 6-bit format of Figure 3.24 cannot represent the number 1.25; it isapproximated by (011,101), representing 1.5, or by either (001,000) or (001,100),representing 1.0. Moreover, the results of most calculations with floating-pointarithmetic only approximate the correct result. For example, in the system of Fig-ure 3.24, the exact result (18) of the addition (011,001) + (011,010), which imple-ments 6 + 12, is not representable. The closest representable number, that is, thebest approximation to 18, is (010,011) = 16. Overflow occurs in this small systemwhen a result's magnitude exceeds 24, and underflow occurs when a nonzero resulthas a magnitude less than 0.125. In practice, floating-point numbers must havelong mantissas

(at least 20 bits), and the results of floating-point operations mustbe carefully rounded off to minimize the errors inherent in floating-point represen-tation. It is common practice for floating-point processing circuits to include a fewextra mantissa digits termed guard digits to reduce approximation errors; the guarddigits are removed automatically from the final results.

-24
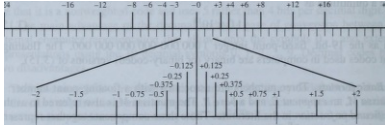
♦24

Figure 3.24

The real numbers representable by a hypothetical 6-bit, floating-point format.

Normalization and biasing. Floating-point representation is redundant in the 175sense that the same number can be represented in more than one way. For example,1.0 x 1018, 0.1 x 1019, 1000000 x 1012, and 0.000001 x 1024 are possible represen-tations of a quintillion. It is generally desirable to have a unique or normal form foreach representable number in a floating-point system. Consider the common casewhere the mantissa is a sign-magnitude fraction and a base of r is used. The man-tissa is said to be normalized if the digit to the right of the radix point is not zero,that is, no leading zeros appear in the magnitude part of the number. Thus, forexample, 0.1 x 1019 is the unique normal form of a quintillion using base 10, a dec-imal mantissa, and a decimal exponent. A binary fraction in twos-complementcode is normalized when the sign bit differs from the bit to its right. This impliesthat no leading Is appear in the magnitude part of negative numbers. Normaliza-tion restricts the magnitude \M\ of a fractional binary mantissa to the range

1/2 < IMI < 1

Normal forms can be defined similarly for other floating-point codes. An unnor-malized number is normalized by shifting the mantissa to the right or left andappropriately incrementing or decrementing the exponent to compensate for themantissa shift.

The representation of zero poses some special problems. The mantissa must, ofcourse, be zero, but the exponent can have any value, since $0 \times B^E = 0$ for all valuesof E. Often in attempting to compute zero, round-off errors result in a mantissa thatis nearly, but not exactly, zero. For the entire floating-point number to be close tozero, its exponent must be a very large negative number -K. This requirement sug-gests that the exponent used for representing zero should be the negative numberwith the largest magnitude that can be contained in the exponent field of the num-ber format. If k bits are allowed for the exponent including its sign, then 2k expo-nent bit patterns are available to represent signed integers, which can range eitherfrom -2*"1 to 2k~x - 1 or from -2k~x + 1 to 2k~x, so that K is 2k~x or 2k~x - 1.

A second complication arises from the desirability of representing zero by asequence of 0-bits only. This convention gives zero the same representation in bothfixed- and floating-point formats, which facilitates the implementation of instruc-tions that test for zero. These considerations suggest that floating-point exponentsshould be encoded in excess-/^ code similar to the excess-three code of Figure3.23, where the exponent field E contains an integer that is the desired exponentvalue plus K. The quantity K is called the bias, and an exponent encoded in thisway is called a biased exponent or characteristic. Figure 3.25 shows the possiblevalues of an 8-bit exponent with bias 127 and 128.

Standards. Until the 1980s floating-point number formats varied from onecomputer family to the next, making it difficult to transport programs between dif-ferent computers without encountering small but significant differences in suchareas as round-off errors. To deal with this problem, the Institute of Electrical andElectronics Engineers (IEEE) sponsored a standard format for 32-bit and largerfloating-point numbers, known as the IEEE 754 standard [IEEE 1985]. which hasbeen widely adopted by computer manufacturers. Besides specifying the permissi-ble formats for M, £, and B, the IEEE standard prescribes methods for handlinground-off errors, overflow, underflow, and other exceptional conditions

Exponent bitpattern E

111111

000000

11

10

100... 01100. . .00Oil ... 11Oil ... 10

0100

Unsignedvalue

Number represented

255254

129128127126

1

0

Bias = 127 Bias = 128

	Bias = 127	Bias = 128
	+ 128	+ 127
	+ 127	+126
	+2	+ 1
	+1	0
	0	-1
	-1	-2
	-126	-127
	-127	-128

Figure 3.25

Eight-bit biased exponents with bias = 127 (excess-127

code) and bias = 128 (excess-128 code).

EXAMPLE 3.4 THE IEEE 754 FLOATING-POINT NUMBER FORMAT [IEEE

1985; Goldberg 1991]. This standard format for 32-bit numbers is illustrated inFigure 3.26. It comprises a 23-bit mantissa field M, an 8-bit exponent field E, and assign bit 5. The base B is two. As in all signed binary number formats, both fixed-pointand floating-point, S occupies the left-most bit position. M is a fraction that with 5forms a sign-magnitude binary number. For the reasons discussed earlier, floating-point numbers are usually normalized, meaning that the magnitude field should con-tain no insignificant leading bits. Hence the magnitude part of a normalized sign-mag-nitude number always has 1 as its most significant digit. There is no need to actuallystore this leading 1 in floating-point numbers, since it can always be inserted by thearithmetic circuits that process the numbers. Consequently, in the IEEE 754 format thecomplete mantissa (called the significand in the standard) is actually l.Af, where the 1to the left of the binary point is an implicit or hidden leading bit that is not stored withthe number. Use of the hidden 1 means that the precision of a normalized number iseffectively increased by 1 bit. The exponent representation is the 8-bit excess-127 codeof Figure 3.25; hence the actual exponent value is computed as E- 127. The base B ofthe floating-point number is 2, so that a 1-bit left (right) shift of M corresponds toincrementing (decrementing) E by one.

Consequently, a 32-bit floating-point number conforming to the IEEE 754 stan-dard represents the real number N given by the formula

$N=(-l)s2^{E'n}\1.M)$

(3.16)

Sign 5

£'

i i i i ii

8-bit exponent

(excess-127binary integer)

23-bit mantissa

(fraction part of sign-magnitude

binary significand with hidden bit)

Figure 3.26

IEEE 754 standard 32-bit floating-pointnumber format.

provided $0 <E < 255$. For example, the number N = -1.5 is represented by 177

1 01111111 10000000000000000000000 CHAPTER 3

where S = 1, E = 127, and M = 0.5, since from (3.16) we have $W = (-1)'2^{127-127}(1.5)$ = Processor Basics-1.5. Nonzero floating-point numbers in this format have magnitudes ranging from$2"^{126}(1.0)$ to $2+127(2 - 2~23)$, that is, from $1.18 \times 10~38$ to 3.40×10^{38} approximately.In contrast, 32-bit, fixed-point binary formats for integers can only represent nonzeronumbers with magnitudes from 1 to $2^{31} - 1$ (approximately 2.15×10^9). The 64-bitversion of the IEEE 754 standard is a straightforward extension of the 32-bit case. Itemploys an 11-bit exponent E and a 52-bit mantissa M and defines the number

$7V=(-1)^5 2^{£-1023}(1.M)$ (3.17)

where $0 < E < 2047$.

The IEEE floating-point standard addresses a number of subtle problems encoun-tered in floating-point arithmetic. Well-defined formats are specified for the results ofoverflow, underflow, and other exceptional conditions, which often yield unpredictableand unusable numbers in computers employing other floating-point formats. The IEEEstandard's exception formats are intended to set flags in the host processor, which sub-sequent instructions can use for error control, in many cases with little or no loss ofaccuracy. If the result of a floating-point operation is not a valid floating-point number,then a special code referred to as not a number (NaN) is used. Examples of operationsthat result in NaNs are dividing zero by zero and taking the square root of a negativenumber. NaN formats are identified in the standard by M ^ 0, and E = 255 (32-bit for-mat) or E = 2047 (64-bit format).

When overflow occurs, meaning that a number has been produced whose magni-tude is too big to represent by the usual format, the result is referred to as infinity, oroo, and is identified by M = 0, and E = 255 (32-bit format) or E = 2047 (64-bit format).The 754 standard stipulates that operations using the floating-point infinities ±°°should follow certain properties of infinity in real-number theory, such as -<» + N = <»and -oo < N < +°° for any finite N. If underflow occurs, implying that a result is non-zero, but too small to represent as a normalized number, it is encoded in a denonnal-izecP form characterized by E = 0 and a significand 0.M having a leading 0 instead ofthe usual leading 1. Denormalization reduces the effect of underflow to a systematicloss of precision equivalent to a small round-off error. Finally, floating-point zero isidentified by an all-0 exponent and significand, but the sign 5 may be 0 or 1. Note thatas the tiny denormalized numbers are diminished, they eventually reach zero.

In summary, the number N represented by a 32-bit IEEE-standard, floating-pointnumber has the following set of interpretations.

If E = 255 and M * 0, then N = NaN.If E = 255 and M = 0, then N = (-1)VIf 0 <E < 255, then N = {-\)S2^{E-X2}\1.M).If E = 0 and M * 0, then N = (-1)S2^{£"126}(0.M).If E = 0 and M = 0, then N = (-1)s0.

The interpretation of 64-bit and larger floating-point numbers is similar.

3The term unnormalized applies to numbers with any value of E and a leading 0 instead of a leading 1 associ-ated with their mantissas. Such numbers are encountered only as intermediate results during floating-pointcomputations and are not relevant to the standard.

SECTION 3.3Instruction Sets

178 Typical of other floating-point number formats still in use is that of the IBM Sys-

tem/360-370. It consists of a sign bit S, a 7-bit exponent field E, and a mantissa fieldM containing 24,56, or 112 bits. M is treated as a fraction, which with S forms a sign-magnitude number; there is no hidden leading 1. E is an integer in excess-64 code,corresponding to an exponent bias of 64. Unlike the IEEE 754 format where the baseB of the representation is two, the System/360-370 Has B = 16. Consequently, M isinterpreted as a hexadecimal (base 16) number with every hexadecimal digit corre-sponding to 4 bits, and the exponent is treated as a power of 16. The value of a float-ing-point number in the normalized System/360-370 format is therefore given by

$N=(-1)^5 16^{£-64}(0.M)$

where M is a 6-, 14-, or 28-digit hexadecimal number. For example, the number0.125×16^5 is encoded as

0 1000101 00100000... 0000

Note that the left-most four bits 0010 of the mantissa represent the nonzero hexa-decimal digit 2; hence the above number is normalized. The number zero is alwaysrepresented by the all-0 word, making the floating-point representation of zeroidentical to the System/360-370 fixed-point (twos-complement) representation.There are no equivalents of the IEEE 754 standard's NaN, infinity, and denormal-ized results. While most floating-point instructions are performed with automaticnormalization of the results, a few may be specified without normalization, thusproviding some of the advantages of denormalization. Due to the larger value of Bbeing used, the System/360-370 32-bit format can represent numbers with magni-tudes ranging from 5.40×10^{-79} to 7.24×10^{75} approximately.

3.3

INSTRUCTION SETS

Next we turn to the representation, selection, and application of instruction sets.This topic embraces opcode and operand formats, the design of the instructiontypes to include in a processor's instruction set, and the use of instructions in exe-cutable programs.

3.3.1 Instruction Formats

The purpose of an instruction is to specify both an operation to be carried out by aCPU or other processor and the set of operands or data to be used in the operation.The operands include the input data or arguments of the operation and the resultsthat are produced.

Introduction. Most instructions specify a register-transfer operation of the form$X_x:=op\{X\{,X_2,...,X_n)$

which applies the operation op to n operands $X_x, X_2,..., X_n$, where n ranges from zeroto four or so. We can write the same instruction in the assembly-language notation

op $X_,,X_2,...,X_,,$ (3.18) 179

which defines the operation and its operands by specific "fields" within the instruc- chapter 3tion word (3.18). The operation op is specified by a field called the Processoropcode (operation code). The n $X_,, X_2,..., X_n$ fields are referred to as addresses. An Basicsaddress X, typically names a register or a memory location that stores

an operandvalue. In some instances X, itself is the desired value, in which case it is called animmediate address.

To reduce instruction size and thereby reduce program storage space, it iscommon to specify only m < n operands explicitly in the instruction; the remainingoperands are implicit. The explicit address fields refer to general-purpose CPU reg-isters or memory locations, while the implicit ones refer to special-purpose regis-ters. If m is the normal maximum number of explicit main-memory addressesallowed in any processor instruction, the processor is called an m-addressmachine. Implicit input operands must be placed in known locations before theinstruction that refers to them is executed.

Inside the computer, instructions are stored as binary words. There can be sev-eral different sizes and formats, depending on the instruction type. RISCs tend tohave few instruction formats, while CISCs tend to have many to accommodatemore opcode types and operand addressing methods. The Motorola 680X0 (Exam-ple 3.3) is a CISC microprocessor series with many different instruction formatsand sizes, a sampling of which appear in Figure 3.27 [Motorola 1989]. Instructionlength in the 680X0 varies from 2 to 10 bytes. The 2-byte opcode field of the680X0 is often used to hold one or two 3-bit register addresses, blurring the dis-tinction between opcode and operand.

In the 680X0 family, simple instructions are assigned short formats. For exam-ple, the add-register instruction

ADD.L D1.D2 (3.19)

denotes register-to-register addition of 32-bit (long word) operands, that is,

D2:=D2 + D1

where Dl and D2 are two of the 680X0's data registers (Figure 3.11). This instruc-tion fits in the third 2-byte format F3 of Figure 3.27, which accommodates tworegister-address fields. A variant of the same two-address instruction can also referto an operand in memory:

ADD.L ADR1, D2 (3.20)

This instruction specifies the memory-to-register addition operation

D2:=D2 + M(ADR1)

and so combines the load and add operations. It uses the 6-byte format F6 to con-tain the 4-byte immediate address field ADR 1. It also requires a memory access toobtain one of its input operands, the 4-byte long word with start address ADR1.Note that the binary (machine language) opcodes corresponding to (3.19) and(3.20) have to be different to distinguish their operand types.

The longest (10 byte) format F8 of the 680X0 is employed by such memory-to-memory move instructions as

MOVE.B ADR1,ADR2

180

SECTION 3.3Instruction Sets

15   0

Opcode
1
 ii i

Opcode Rl
2   Opcode
 i ' i   i i

Opcode Rl
3   Opcode R2
 i i i   i i

31   15 0

Opcode/registers   Immediate operand IMM (short)
F4
 i i i i i i i i i i i i i i i i i i i i i

Opcode/registers
F5   Memory address ADR1 (short)i i i i i i i i i i i i i j i
 i i i i i i i i r i i i i i i i

47   31 0

Opcode/registers Immediate operand IMM (long)
F6
 i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i i

Opcode/registers
F7   Memory address ADR1i i i i i i i i i i i i i i i i i i i i i i i i i i i
 i i i i i i i i i

79   53 32

Opcode/registers Memory address ADR1
F8
 i i i i   iii i i i i i i i i i i i i i i i i

Memory address ADR2

1 ' ' ' ' ' ' ' ■ ' ' ' ' ' ' ' ' i ' ' '

Figure 3.27

A selection of instruction formats of the Motorola 680X0.

which copies (via the CPU) the byte stored in memory location ADR1 to memorylocation ADR2, that is,

M(ADR2):=M(ADR1)

RISC formats. The instruction formats of the 680X0 accommodate a widevariety of operations and addressing modes. They also try to reduce object-programsize by encoding the more common instructions in short formats and the less fre-quent and more complex instructions in longer formats. Since such instructions areoften primitives in high-level programming languages, they serve to reduce bothprogram length and what has been called the semantic gap between the user and thecomputer languages.

Complex instructions lead to several difficulties, which RISCs with theirsmaller and streamlined instruction sets attempt to minimize.

• The many instruction types and formats of a CISC complicate the program-control unit that decodes instruction opcodes and issues the control signals that

govern their execution. The 68020 employs a large, two-level micropro- 181grammed PCU (Figure 3.11), whereas the ARM6 has a smaller hardwired cir-cuit as its PCU.• Fast, single-cycle instruction execution is harder to achieve with a complexinstruction set, and it is more difficult for a compiler to optimize object-code per-formance.

A typical RISC employs instructions of fixed length. Memory addressing isrestricted to load and store instructions, so the operands of most instructions are reg-ister addresses, which are short and easy to accommodate in a one-word format.Figure 3.28 shows the single 32-bit format used by instructions in the RISC 1 com-puter, a prototype RISC machine designed by David A. Patterson and his colleaguesat the University of California, Berkeley, around 1980 [Patterson and Sequin 1982].Most of the 31 instruction types defined for the RISC 1 perform register-to-registeroperations of the form

Rd := F(Rs,S2)

(3.21)

where Rd is the destination register, Rs is the first source register, and the right-most 5 bits of S2 define a second-source register. If bit 13 of the instruction is set toone, then S2 is interpreted as an immediate address, that is, as a 13-bit constant.The instructions of the ARM6 microprocessor (Example 3.2), like those of theRISC 1, are all 32 bits long, but they come in a large and CISC-like number of for-mats [Furber 1989].

Operand extension. A CPU is designed primarily to process data words andaddresses of one specific length—a 32-bit word in the case of the ARM6 andRISC 1—although some instructions handle longer or shorter operands. Numeri-cal operands can be unsigned binary number words, such as memory addresses, orsigned data words that employ twos-complement code. (Recall from section 3.2.1that the same arithmetic circuits can be used with unsigned and twos-complementnumbers.) Instructions often contain operand fields that are shorter than the stan-dard word size, for example, the 13-bit immediate address field S2 in the RISC 1format of Figure 3.28. This problem is unavoidable in RISC instruction sets wherethe instruction length and the standard word size are the same. Consequently, asystematic method is needed to extend short operand values to full-size, signed orunsigned numbers.

When a short w-bit, twos-complement number is used in an n-bit arithmeticoperation where n > m, a technique called sign extension is employed. This tech-niques replicates the left-most bit s of the short operand N, which corresponds to itssign bit, n - m times and attaches s"~m = ss...s to the left side of N. Sign extension

CHAPTER 3

Processor

Basics

31

Set condition code23

Set immediate address12

Opcode

J I I I I L

Source Rs

J__l I L

Destination Rd

_i I I I

Source S2

i I I ' ' ' ' I I I 1 L

Figure 3.28

Instruction format of the Berkeley RISC 1

182 changes a 13-bit operand

SECTION3.3 #=1010101010101 (3.22)

Instruction Sets

in the S2 field of Figure 3.28 to the 32-bit word

^sign-extended =1111111111111111 1 1 110101 01010101 (3.23)

In this case s = 1 and n -m = 19. If 5 were 0, then sign extension would precede Nby 19 leading 0s. The point of sign extension is that it does not change the numeri-cal value of a twos-complement number. For instance, both (3.22) and (3.23) repre-sent the same negative integer, namely, -2,64610, in twos-complement code, as canreadily be verified. Sign extension maintains a number's correct sign and magni-tude because it introduces only numerically insignificant leading 0s (positive num-bers) or insignificant leading Is (negative numbers). If N is to be treated as anunsigned binary number, then it is always extended by leading 0s, independent ofthe value of 5. This technique has been called zero extension. Applying zero exten-sion to (3.22) yields

^zero-extended = 00000000 00000000 00010101 01010101

Next we ask: How is an n-bit memory address, which is a long (typically 32-bit) unsigned integer, constructed from a short m-bit address field, when n > 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	25	0
J-typeformat	Opcodei i i i 1 i : i	Branch address ADR i i i i i i i i i i i i i i i i i i	

	31	25	20	15	0
I-typeformat	Opcode	Rs	Rt	Immediate operand IMM	

vRegister addresses (2)

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 a 2^{26}-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 2^{30} 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 (2^{26}= 6.71 X 10^7 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 2^5 = 32 general-purpose registers in itsregister file, so register addresses can be fully specified vuth no difficult). The second

(I type) format is used by ALU-immediate instructions such as

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^k 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 full-lengthaddress 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 than operands 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 addressing mode: The address field X itself can be V(X), it can specify directly the location of 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 an address. The advantage of indirection, as we will see, is increased programming flexibility. We can achieve further flexibility by providing addresses that are automatically 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 is immediate addressing. By implication X is a constant, since it is very undesirable to modify instruction fields during execution.[4] More often than not, X is a variable in the usual mathematical sense, and the corresponding address field identifies the storage location that contains the required value V(X). Thus X corresponds to a variable, and its value V(X) can be varied without modifying the instruction address field. Operand specification of this type is called direct addressing.

The addressing modes of the operands appearing in a machine-language instruction, which can vary from operand to operand, are defined in the instruc-tion's opcode. Some assembly languages allow addressing modes to be similarly defined by distinct opcodes. For example, the assembly language of the Intel 8085 series has the opcode MOV (move) to specify data transfers involving direct addressing only. Therefore, the register-to-register transfer A := B, for instance, is specified by

MOV A,B

(3.26)

The A and B operands of (3.26) are considered to be directly addressed, since the contents 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 (*move immediate*) must be used. Note that (3.27) uses both the 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 the Intel convention.) In (3.28) the prefix # indicates that the immediate addressing mode is to be used for the operand in question. Deleting the # from (3.28) causes

[4]Self-modifying programs like the IAS code shown in Figure 1.15 (section 1.2.2) reflect the made qithe 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), which would be an instance of direct memory addressing.

It is sometimes useful to change the location (as opposed to the value) of X without changing the address fields of any instructions that refer to X. This may be accomplished by indirect addressing, whereby the instruction contains the address W of a storage location, which in turn contains the address X of the desired operand value V(X). By changing the contents of W, the address of the operand value required by the instruction is effectively changed. While direct addressing requires only one fetch operation to obtain an operand value, indirect addressing requires two. Figure 3.30 illustrates these different ways of specifying operands in the case of 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 with all opcodes of an instruction set or assembly language is a desirable feature termed orthogonality. Orthogonal instruction sets simplify programming both by reducing the number of distinct opcodes needed and by simplifying the rules for operand address 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 vary from 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 direct address formation, requires the complete operand address to appear in the instruction 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 desired data 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 the operand 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 most cases of interest, each operand is associated with a single address register R from a set 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 addresses referred to by a block of instructions B. This address modification permits the processor to move (relocate) the entire block B from one region of main memory to another without invalidating the addresses in B. When used in this way, R may 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. In this role R is called an index register. The indexed items X(0), X(\),...,X(k) are stored in consecutive addresses in memory. The instruction-address field D contains the address of the first item X(0), while the index register R contains the index i. The address of item X(i) is D + R. By changing the contents of the index register, a single instruction can be made to refer to any item X(i) in the given data list.

The main drawbacks of relative addressing are the extra logic circuits and processing time needed to compute addresses.

So far we have assumed that each operand is a single memory word and can therefore be specified by a single address. If an instruction must process variable-length data consisting of many words, each operand specification is divided into two parts: an address field that points to the location of the first word of the operand and a length field L that indicates the number of words in the operand. The CPU automatically increments the instruction address field as successive words of the operand are accessed. The access is complete when L words have been accessed.

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) or X(k-l) stored in location A + 1 or A -1. respectively. To facilitate stepping through a sequence of items in this manner, addressing modes that automatically increment or decrement an address can be defined; the resulting address-modification process

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 auto-

matically before the instruction is executed; this process is called predecrementing. Similarly, (A3)+ specifies that A3 should be incremented automatically after the current instruction has been executed ipostincrementing). In each case the amount of the address increment or decrement is the length in bytes of the indexed operands.

Most processors have only a few, simple addressing modes for CPU registers, principally direct and immediate addressing. Immediate addresses represent data values that come with the instruction fetch and are placed in the instruction register IR. In register direct addressing, the address (name) R of the register containing the 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 addressing for 99 and register direct (or simply direct) addressing for Dl.

The term register indirect addressing refers to indirect addressing with a register R name in the address field. It is often used to access memory, in which case R becomes a memory address register. For example,

MOVE.B (A0),D1

uses parentheses to indicate that (A0) is an indirect address involving the 680X0's A0 addresss register. This move-byte instruction—the opcodes's .B suffix specifies a 1-byte operand—corresponds to

D1[7:0]:=M(A0)

and copies the byte addressed by A0 into the low-order byte position of data register Dl. (The other three bytes of Dl are unchanged.) An extension of this addressing mode is register indirect with offset, which can also be viewed as a type of base or indexed addressing. This mode is the only memory addressing mode employed by the MIPS RX000 series (Example 3.5). The RXOOO's store-word instruction, for example, 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 effective address Rs + OFFSET. The PowerPC has two addressing modes: register indirect with offset as described above (but called register indirect with immediate index) and a second mode (called register indirect with index) in which the effective address is Rs + Ri, where Ri is a register name.

The Motorola 680X0, like other CISC-style architectures, has many addressing modes, including the following: immediate, register direct, register indirect, register indirect with postincrement, register indirect with predecrement, register indirect 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- 189 lowing example.

EXAMPLE 3.6 STACK CONTROL IN THE MOTOROLA 680X0 [GILL. CORWIN

and logar 1987; motorola 1989]. A stack is a sequence of storage locations that are accessible from only one end referred to as the top of the stack. A write operation addressed to a stack, termed a push operation, stores a new item at the top of the stack, while a read operation, termed a pop operation, removes the item stored at the top of the stack. Push or pop changes the position of the stack top by an amount that depends on the length of the operand pushed or popped. A stack is controlled by an address register called the stack pointer SP. This register stores the address of the last operand 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 and hardware for handling stacks that are intended as communication areas for program-control instructions like call and return. A few early computers such as the Burroughs B6500/7500 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, as we now show, its various addressing modes make it easy to treat any contiguous region of its external memory M as a stack.

Suppose that the programmer designates the address register A2 of the 680X0 to be a stack pointer and that the stack grows toward the low addresses of M. To push the contents 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 address register 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 postincrement addressing mode.

Number of addresses. Some computers, notably CISCs like the 680X0, have instructions of several different lengths containing various numbers of addresses. A source of controversy in the early days was the question of how many explicit operand addresses to include in instructions. Clearly the fewer the addresses, the shorter the instruction format needed. However, limiting the number of addresses also limits the range of operations that an instruction can perform. Roughly speaking, fewer addresses mean more primitive instructions and therefore longer

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<0,1) B(0,0)

A2 = stack pointer register

OF FF 78 50

1 word -

M
Address

0FFF7858*0FFF78570FFF78560FFF78550FFF78540FFF78530FFF78520FFF78510FFF78500FFF784F

t
Stack1 Top
rof
stack

B(2.3)

B(1.0)

B(l,l)

Bd.2i

B(l,3)

FF

FF

FF

FF

FF

1 byte   —

M
Address

0FFF78580FFF78570FFF78560FFF78550FFF78540FFF78530FFF78520FFF7851OFFF78500FFF784F

a

B(2,3)

B(1.0)

Bfl.lj

B(1.2)   Stack

B(1.3)   1 TopVofstack

B(O.O)

B(0,1)

B(0,2)

Bi0.3)

FF

1 byte   —

Figure 3.31

State of the Motorola 680X0 {a) immediately before and (b) immediately afterexecution of the push instruction MOVE.L D6,-(A2).

programs to perform a given task. While the storage requirements of shorterinstructions and longer programs tend to balance, larger programs require longerexecution times. On the other hand, long instructions with multiple addressesrequire more complex decoding and processing circuits. RISC instructions, withthe exception of load and store, contain short register addresses only, so two orthree addresses can be accommodated within a short and fixed-length instructionword.

Most instructions require no more than three distinct operands. For example,the fundamental arithmetic operations—addition, subtraction, multiplication, anddivision—require three operands: two input operands and one output operand. Athree-address instruction can therefore specify all needed operands. For example,the three-address add instruction

ADD Z, X, Y

means add the contents of memory locations X and Y and place the result in loca-tion Z; that is, $Z := X + Y$. A one-address add instruction has the format

ADD X

The unspecified operands are assumed to be stored in fixed locations such as theaccumulator AC. in which case the instruction specifies the operation $AC := AC + X$. In the case of a two-address instruction, the accumulator is used to store theresult (the sum) only.

ADD X, Y

has the typical interpretation $AC := X + Y$. Another possibility is to use oneaddress, say, X, to store both the addend X and the sum as follows: $X := X + Y$. Inthe latter case the addition operation destroys the X operand. Figures 3.32a, b, andc show how processors that employ one-address, two-address, and three-addressinstructions, respectively, might implement the operation

$X := A X B + C X C$

(3.32)

where the four operands A, B, C, and X are assumed to be stored in external mem-ory.

A few computers have been designed so that most instructions contain noexplicit addresses; they can be called zero-address machines; see also Example1.5. Addresses are eliminated by storing operands in a push-down stack. All oper-ands used by a zero-address instruction are required to be in the top locations in thestack. For example, the addition $X + Y$ is invoked by an instruction such as

ADD

that causes the top two operands, which should be X and Y, to be removed from thestack and added. The resulting sum $X + Y$ is then placed at the top of the stack. Astack pointer automatically keeps track of the stack top. Push and pop instructionsare needed to transfer data to and from the stack. PUSH X causes the contents of Xto be placed at the top of the stack. POP X causes the top word in the stack to betransferred to location X. Note that PUSH and POP are not themselves zero-address instructions; as implemented by (3.30) and (3.31), for instance, they aretwo-address instructions. Figure 3.33 shows how a program for (3.32) might beconstructed for a zero-address, stack machine.

191

CHAPTER 3

Processor

Basics

3.3.2 Instruction Types

We now turn to the question: What types of instructions shou'd be included in ageneral-purpose processor's instruction set? We are concerned with the instructions

iyi	Instruction	Comments
SECTION 3.3Instruction Sets	LOAD AMULTIPLY B	Transfer A to accumulator AC.AC := AC x B
	STORET	Transfer AC to memory location T. '
	LOADC	Transfer C to accumulator AC.
	MULTIPLY C	AC := AC x C
	ADDT	AC := AC + T
	STORE X	Transfer result to memory location X.
	(a) One-address machine	

Instruction	Comments
MOVE T.A	T:=A
MULTIPLY RE-	T := T x B
MOVE X,C	X:=C
MULTIPLY X,C	X := X x C
ADD X,T	X:=X + T

Instruction	Comments
MULTIPLY T,A,B	T := A x B
MULTIPLY X,C,C	X := C x C
ADD X,X,T	X := X + T

(c) Three-address machine

Figure 3.32

Programs to execute the operation X := Ax B + Cx C in one-address,two-address, and three-address processors.

that are in the processor's machine language. All processors have a well-definedmachine language, and some implement a lower-level "micromachine" languagespecified by microinstructions. A typical machine instruction defines one or tworegister transfer (micro) operations, and a sequence of such instructions is neededto implement a statement in a high-level programming language such as C. Becauseof the complexity of the operations, data types, and syntax of high-level languages,few attempts have been made to construct computers whose machine languagedirectly corresponds to a high-level language. As noted earlier, there is a semanticgap between problem-specification languages and the machine instruction set thatimplements them, a gap that language-translation programs such as compilers andassemblers must bridge.

The requirements to be satisfied by an instruction set can be stated in the fol-lowing general, but rather imprecise, terms:

Instruction	Comments
PUSH A	Transfer A to top of stack.
PUSHB	Transfer B to top of stack.
MULTIPLY	Remove A,B from stack and replace by A x B.
PUSHC	Transfer C to top of stack.
PUSHC	Transfer second copy of C to top of stack.
MULTIPLY	Remove C,Cfrom stack and replace byCxC
ADD	Remove CxC,AxB from stack and replace by their sum.
POPX	Transfer result from top of stack to X.

Figure 3.33

Program to execute X

193

CHAPTER 3

Processor

Basics

= AxB+CxC in a zero-address, stack processor.

• It should be complete in the sense that we should be able to construct a machine-language program to evaluate any function that is computable using a reasonableamount of memory space.

• It should be efficient in that frequently required functions can be performed rap-idly using relatively few instructions.

• It should be regular in that the instruction set should contain expected opcodesand addressing modes; for example, if there is a left shift, there should be a rightshift.

• To reduce both hardware and software design costs, the instructions may berequired to be compatible with those of existing machines—previous membersof the same computer family, for instance.

Because of the wide variation in CPU architectures between different computerfamilies, standard machine or assembly languages do not exist. There are, never-theless, broad similarities between all instruction sets, which go back to the IAScomputer and other early machines.

Completeness. A function fix) is said to be computable if it can be evaluated ina finite number of steps by a Turing machine (see section 1.1.1). While real com-puters differ from Turing machines in having only a finite amount of memory, theycan, in practice, evaluate any computable function to a reasonable degree ofapproximation. When viewed as instruction-set processors, Turing machines havea very simple instruction set. In our discussion of Turing machines, we definedfour instruction types: write, move tape one square to the left, move tape onesquare to the right, and halt, all of which are conditional on the control processor'sstate. It follows that complete instruction sets can be constructed for finite-statemachines using equally simple instruction types. In fact, computers have been pro-posed that employ only a single type of instruction; see problem 3.44. While verysmall instruction sets require simple, and therefore inexpensive, logic circuits toimplement them, they lead to excessively complex programs. There is therefore afundamental trade-off between processor simplicity and programming complexity.

194 Instructions are conveniently divided into the following five types:

section 3.3 1. Data-transfer instructions, which copy information from one location to another

instruction Sets either in the processor's internal register set or in the external main memory.

2. Arithmetic instructions, which perform operations on numerical data.

3. Logical instructions, which include Boolean and other nonnumerical operations.

4. Program-control instructions, such as branch instructions, which change thesequence in which programs are executed.

5. Input-output (IO) instructions, which cause information to be transferredbetween the processor or its main memory and external IO devices.

These types are not mutually exclusive. For example, the arithmetic instructionA := B + C implements the data transfer A := B when C is set to zero.

Figure 3.34 lists representative instructions from the five types defined above,which have been culled from the instruction sets of various computers. The data-transfer

instructions, particularly load and store, are the most frequently used instructions in computer programs, despite the fact that they involve no explicit computation. The arithmetic instructions cover a wide range of operations and are sometimes used as a rough measure of the complexity of an instruction set. The logical instructions include the word-based Boolean operations, as well as opera-tions that have no obvious numerical interpretation. The major branch instructions are jump (un)conditionally and the call and return instructions used for subroutine linkage. The simplest IO instructions are data-transfer instructions addressed to IO ports, which transfer one or more words between an IO port and either the CPU or M. If the CPU delegates control of IO operations to an IO processor (IOP), the CPU needs instructions that enable it to supervise the execution of IO programs by the IOP. Instructions that are specific to particular IO devices, such as REWIND TAPE, PRINT LINE, and SCAN KEYBOARD, are treated as data by the CPU and IOP and are interpreted as instructions only by the IO devices to which they are transferred.

The completeness of an instruction set can be demonstrated informally by showing that it can program certain key operations in each of the five instruction groups. It must be possible to transfer a word between the processor and any mem-ory location. It must be possible to add two numbers, so an add instruction is included in most instruction sets. Other arithmetic operations can readily be pro-grammed using addition. As noted in section 3.2.2, subtraction of twos-complement numbers requires addition and logical complementation (NOT) only. More com-plex arithmetic operations such as multiplication, division, and exponentiation can be programmed using addition, subtraction, and shifting, as in Example 2.7. If a logically complete set of Boolean operations such as {AND,NOT} is in the instruc-tion set. then any other Boolean operation can be programmed. Branching requires at least one conditional branch instruction that tests some stored quantity and alters the instruction execution sequence based on the test outcome. An unconditional branch can easily be realized by a conditional branch instruction.

RISC versus CISC. While an instruction set that is limited to two or three instructions is impractical, there is no agreement about the appropriate size or membership of a general-purpose instruction set. Early computers like the IAS had a small and simple instruction set forced by the need to minimize the amount of

Type	Operation name(s)	Description
Data	MOVE	
transfer	LOAD	
	STORE	
	SWAP (EXCHANGE)	
	CLEAR	
	SET	
	PUSH	
	POP	
Arithmetic	ADD	
	ADD WITH CARRY	
	SUBTRACT	
	MULTIPLY	
	DIVIDE	
	MULITPLY AND ADD	
	ABSOLUTE	
	NEGATE	
	INCREMENT	
	DECREMENT	
	ARITHMETIC SHIFT	
Logical	AND "i	
	OR	
	NOT	
	EXCLUSIVE-OR	
	LOGICAL SHIFT	
	ROTATE	
	CONVERT (EDIT)	
Program	JUMP (BRANCH)	
control	JUMP CONDITIONAL	
	JUMP TO SUBROUTINE (BRANCH-AND-LINK)	
	RETURN	

EXECUTE

SKIP CONDITIONAL

TRAP (SOFTWARE

INTERRUPT)TESTCOMPARE

Copy word or block from source to destination.Copy word from memory to processor register.Copy word from processor register to memory.Swap contents of source and destination.Transfer word of Os to destination.Transfer word of Is to destination.Transfer word from source to top of stack.Transfer word from top of stack to destination.

Compute sum of two operands.

Compute sum of two operands and a carry bit.

Compute difference of two operands.

Compute product of two operands.

Compute quotient (and remainder) of two operands.

Compute product of two operands; add it to a third

operand.Replace operand by its absolute value.Change sign of operand.Add 1 to operand.Subtract 1 from operand.Shift operand left (right) with sign extension.

Perform the specified logical operation bitwise.

Shift operand left (right) introducing Os at end.Left- (right-) shift operand around closed path.Change data format, for example, from binary to decimal.

Unconditional transfer: load PC with specified address.Test specified conditions; if true, load PC with specified

address.Place current program control information including PC in

known location, for example, top of stack; jump to

specified address.Restore current program control information including PC

from known location, for example, from top of stack.Fetch operand from specified location and execute as

instruction; note that PC is not modified.Test specified condition; if true, increment PC to skip next

instruction.Enter supervisor mode.

Test specified condition; set flag(s) based on outcome.Make logical or arithmetic comparison of two or moreoperands; set flag(s) based on outcome.

195

CHAPTER 3

Processor

Basics

Figure 3.34

List of common instruction types.

196

Type

Operation name(s) Description

Programcontrol

SET CONTROLVARIABLES

WAIT (HOLD)

NO OPERATIONInput-output INPUT (READ)

OUTPUT (WRITE)START IOTEST 10

HALTIO

Large class of instructions to set controls for protection pur-poses, interrupt handling, timer control, and so forth (oftenprivileged). '

Stop program execution; test a specified condition continu-ously; when the condition is satisified, resume instructionexecution.

No operation is performed, but program execution continues.

Copy data from specified 10 port to destination, for example,output contents of a memory location or processor register.

Copy data from specified source to 10 port.

Transfer instuctions to IOP to initiate an 10 operation.

Transfer status information from IO system to specified desti-nation.

Transfer instructions to IOP to terminate an 10 operation.

Figure 3.34

(continued).

CPU hardware. These instruction sets included only the most frequently used oper-ations such as load a register from memory, store a result in memory, and add twofixed-point numbers. As hardware became cheaper, instructions tended to increaseboth in number and complexity so that by 1980 a typical computer had dozens ofinstruction types, with versions to handle several data types and addressing modes.These large instruction sets contain infrequently used but hard-to-program opera-tions like floating-point divide. Since such operations are primitives in program-ming languages, they serve to reduce the semantic gap between the user's languageand the computer's. However, complex instructions lead to a number of complica-tions in both hardware and software design, which we now consider.

Suppose that a particular operation F can be implemented either by a singlecomplex instruction IF or by a multiinstruction routine PF composed of simpleinstructions. Execution of PF will generally be slower than execution of IF becausethe processor must spend more time fetching the instructions of PF and, dependingon the nature of F, handling the intermediate data that links the instructions. A fur-ther drawback of PF is that it occupies more memory space than IF occupies. Anobvious disadvantage of lF is that it adds to the complexity of a processor's controlunit, thereby increasing both the size of the processor and the time required todesign it.

Clearly a program involving F is simplified by using IF in place of PF. Whenthe program is written in a high-level language, however, as most programs are, theexecution speedup that justifies a complex instruction like /Fmay not be fully real-izable. A compiler will typically translate F into the corresponding machineinstruction IF, if available, which uses fixed CPU registers and has a fixed execu-tion time. On the other hand, if IF is not available, an efficient or optimizing com-piler may be able to generate object code QF corresponding to PF that exploitsinformation known at compilation time to reduce F's execution time. Suppose, forinstance, that F is fixed-point multiplication and is implemented by both IF and QFvia a shift-and-add algorithm of the kind described in Example 2.7. If one of F's

operands is a small constant or zero, then the compiler can easily generate a shorterform of PF that is faster than the generic n-step multiply instruction IF. The speedgap between IF and PF can also be narrowed by designing the small instruction setrequired for PF to reduce the instruction fetch and execute cycle times as far aspossible, preferably to one CPU clock cycle each. Another speed advantage of PFover IF is that PF can be interrupted in midoperation at an appropriate instructionboundary, whereas IF must proceed to termination before the CPU can respond toan interrupt.

Motivated by considerations of the foregoing sort, a number of computerdesigners advocated machines with relatively small and simple instruction sets,which have been dubbed RISCs for reduced instruction-set computers. RISC archi-tecture is contrasted with the complex instruction-set computer (CISC) architecturefound in most pre-1980 designs such as the IBM System/360-370 and the Motorola680X0. The major attributes of RISCs have been defined as follows [Colwell et al.1985]:

• Relatively few instruction types and addressing modes.

• Fixed and easily decoded instruction formats.

• Fast, single-cycle instruction execution.

• Hardwired rather than microprogrammed control.

• Memory access limited mainly to load and store instructions.

• Use of compilers to optimize object-code performance.

Several of these RISC attributes are closely related. For example, the small sizeand regularity of the instruction set simplifies the design of a hardwired programcontrol unit, which in turn facilitates the achievement of fast single-cycle execu-tion. The stress placed on efficient compilation requires the machine architects andcompiler writers to cooperate closely in the design process.

RISC architectures restrict the instructions that access memory to load andstore. Consequently, most RISC instructions involve only register-to-register oper-ations that are internal to the CPU. To support them, a larger-than-usual number ofregisters may be placed in the CPU. This design facilitates single-cycle executionand minimizes the CPU cycle time. Pipelining the instruction execution processalso supports single-cycle execution. Since complex instructions are not in theinstruction set, they must be implemented by multiinstruction routines, whichprompts the attention to efficient compilation. Machine code compiled for a RISCcomputer is likely to have more instructions than the corresponding CISC code butcan execute more efficiently, especially if only fixed-point (integer) instructionsare involved. However, if the frequency of complex operations is high, then theperformance of the CISC machine may be better than that of the RISC machine.

197

CHAPTER 3

Processor

Basics

EXAMPLE 3.7 INSTRUCTION SET OF THE MIPS RX000 [KANE AND HEIN-

RI c H 1992]. The RX000 microprocessor series and its instruction formats were intro-duced in Example 3.5 (section 3.3.1). A microprocessor in this family is implementedby a single IC and has the major components indicated in Figure 3.35. These include afile of 32 general-purpose 32-bit registers and the processing logic to perform the basicfixed-point ALU functions: add, subtract, multiply, divide and logical operations using32-bit operands. Numerical operands are treated as unsigned or signed integers intwos-complement code. One register R0 in the register file permanently stores the con-stant zero. Some special-purpose arithmetic circuits perform address computation. The

198

SECTION 3.3Instruction Sets

Localcontrol logic

Register filegeneral purpose

32-bit registers)

«

System controlcoprocessor(Control registers,memory manage-ment unit)

iLIF

Processing logic(ALU, shifter,

multiplier/divider,address logic)

iL

if

I 1
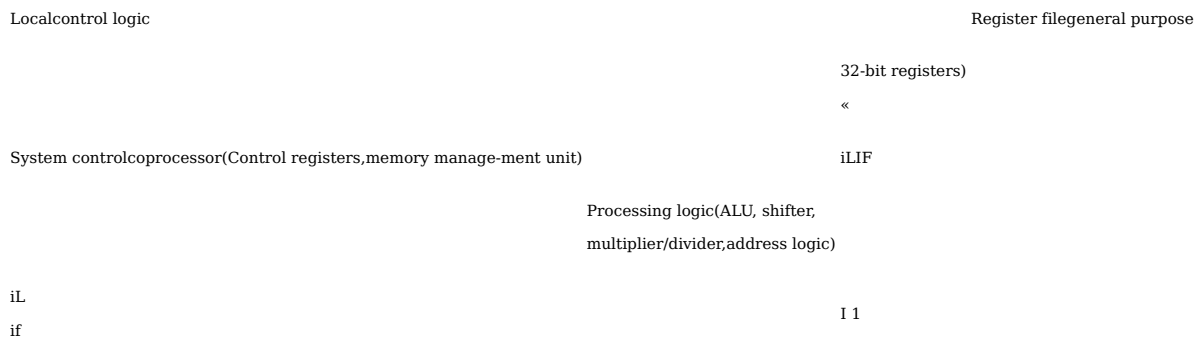
System bus

To M and10 system

Figure 3.35

Overall organization of the MIPS RXOOO.

overall organization of the RXOOO E-unit is similar to that of the ARM6 (Figure 3.9).As in the ARM6 case, the E-unit of the RXOOO is pipelined to support the goal of exe-cuting instructions at a peak rate of one instruction per clock cycle. Floating-pointoperations meeting the requirements of the IEEE 754 standard are supported by an on-chip or off-chip floating-point unit (FPU).

In addition to the control logic needed for instruction execution, the RXOOO con-tains a unit referred to as the system control coprocessor whose functions include com-munication with external memory (caches and main memory) and the automaticaddress translation logic needed to support a virtual memory system. The virtual mem-ory feature uncouples the address space seen by the programmer from the computer'sphysical address space, making it possible, for example, to run a large program in asmall amount of physical memory. The system control coprocessor is essentially invis-ible to the applications programmer. The RXOOO can have several additional coproces-sors implemented on additional ICs.

We now consider in detail the RXOOO's basic (MIPS I) instruction set, which issummarized in Figure 3.36. There are 74 types, divided almost equally between data-transfer, data-processing, and program-control instructions. All are 32 bits long and useone of the I, J, and R formats illustrated in Figure 3.29. The smallest addressable itemin external memory M is, as usual, an 8-bit byte, which requires a 32-bit address tospecify its location. Smaller address fields such as the 26-bit branch address field of J-type instructions are automatically extended to 32 bits before loading into the programcounter PC. Note that to increment PC to point to the next sequential instruction of aprogram requires the step PC := PC + 4. The 16-bit (half-word) IMM field of I-typeinstructions serves either as an immediate data operand or else as an address offset. Ineither case it is also extended to 32 bits either by zero extension or by sign extension.During initialization, the microprocessor can be reset to store data according to eitherthe big-endian or the little-endian convention.

Following the basic RISC philosophy, communication between the CPU andexternal memory M is via load and store instructions only, using the I-type format (Fig-ure 3.29). The RXOOO has instructions to load and store data in bytes and half-words (2bytes), as well as full, 4-byte words. If a byte or half-word is to be loaded into a CPUregister, then the loaded item is expanded to a full word by sign extension, unless the"unsigned" version of the load instruction is specified, in which case zero extension is

Type

Instruction

Assembly-language format

Narrativeformat (comment)

199

Datatransfer

Load byte

Load byte unsigned

Load half-word

LB Rt, Source

Dataprocessing

Load register Rt with sign-extended

memory byte.Load register Rt with zero-extended

memory half-word.Load register Rt with sign-extended

memory half-word.Load register Rt with zero-extended

memory half-word.Load register Rt with memory word.Load left side of register Rt with 1 to

3 memory bytes.Load right side of register Rt with 1 to

3 memory bytes.Store least significant byte of register

Rt in memory.Store least significant half-word of

register Rt in memory.Store register Rt in memory.Store left 1 to 3 bytes of register Rt in

memory.Store right 1 to 3 bytes of register Rt in

memory.Move immediate operand IMM.O16 into

register Rt.

(Four special register-move instructions for use with multiplication and division)

(Eight special data-transfer instructions for use with coprocessors, including the systemcontrol coprocessor)

Add ADD Rd.Rs.Rt

Load half-word

unsignedLoad wordLoad word left

Load word right

Store byte

Store half-word

Store wordStore word left

Store word right

Load upper immediate

LBU Rt.Source

LH Rt,Source

LHU Rt.Source

LW Rt.SourceLWL Rt.Source

LWR Rt,Source

SB Rt.Dest

SH Rt,Dest

SW Rt.DestSWL Rt.Dest

SWR Rt.Dest

LUI Rt,IMM

Add unsignedAdd immediate

ADDU Rd.Rs.RtADDI Rt.Rs.IMM

Add immediate unsigned ADDIU

Subtract

SUB Rd.Rs.Rt

Subtract unsigned SUBU Rd.Rs.Rt

AND	AND Rd.Rs.Rt
AND immediate	ANDI Rt.Rs.IMM
NOR	NOR Rd.Rs.Rt
OR	OR Rd.Rs.Rt
OR immediate	ORI Rt.Rs.IMM

XOR

XOR Rd.Rs.Rt

Add Rs to Rt; put result in Rd (trap on

overflow).Add Rs to Rt: put result in Rd.Add sign-extended IMM to Rs; put

result in Rt (trap on overflow).Rt.Rs.IMM Add sign-extended IMM to Rs; put

result in Rt.Subtract Rt from Rs; put result in Rd

(trap on overflow).Subtract Rt from Rs: put result in Rd.Bitwise AND Rt and Rs; put result

inRd.Bitwise AND zero-extended IMM and

Rs; put results in Rt.Bitu ise NOR Rt and Rs; put result

inRd.Bitwise OR Rt and Rs; put result in Rd.Bitwise OR zero-extended IMM and Rs;

put result in Rt.Bitwise XOR Rt and Rs: put result

inRd.

Figure 3.36

Instruction set of the MIPS RXOOO.

Type Instruction

Assembly-language format

Narrativeformat (comment)

XOR immediateSet on less than

Set on less

than unsignedSet on less

than immediate

Set on less thanimmediate unsi

gned

XORI Rt,Rs,IMM Bitwise XOR zero-extended IMM

and Rs; put result in Rt.SLT Rd,Rs,Rt Compare Rt with Rs as signed integers;

if Rs < Rt, then Rd := 1, else Rd := 0.SLTU Rd,Rs,Rt Compare Rt with Rs as unsigned integers;

if Rs < Rt, then Rd := 1, else Rd := 0.SLTI Rt,Rs,IMM Compare sign-extended IMM with Rs as

signed integers; if IMM < Rs, then

Rt := I.elseRt := 0.SLTIU Rt,Rs,IMM Compare sign-extended IMM with Rs as

unsigned integers; if IMM < Rs, then

Rt := l,elseRt:=0.

(Two multiply and two divide instructions)(Six logical and arithmetic shift instructions)

Program Jumpcontrol Jump and link

J ADRJAL ADR

Jump and link register JALR Rd.Rs

Rs.Rt.IMM

Rs,Rt,IMM

Rs,IMM

Rs,IMM

Rs,IMM

Branch on equal BEQ

Branch on not equal BNE

Branch on less than 0 BLTZ

Branch on greater than 0 BGTZBranch on less than or BLEZ

equal to 0Branch on greater than

or equal to 0Branch on less than 0

and linkBranch on greater than or

equal to 0 and linkSystem call SYSCALL

Break BREAK

(10 miscellaneous coprocessor instructions)

Jump unconditionally to address ADR.Place PC + 8 in R31 and jump

unconditionally to address ADR.Place PC + 8 in Rd and jump

unconditionally to address in Rs.If Rs = Rt, then jump to PC + 8 + IMM.If Rs * Rt, then jump to PC + 8 + IMM.If Rs < 0, then jump to PC + 8 + IMM.If RS > 0, then jump to PC + 8 + IMM.If Rs < 0, then jump to PC + 8 + IMM.

BGEZ RsJMM If Rs > 0, then jump to PC + 8 + IMM.

BLTZAL RsJMM

BGEZAL RsJMM

Place PC + 8 in R31; if Rs < 0, then

jump to PC + 8 + IMM.Place PC + 8 in R31; if Rs > 0, then

jump to PC + 8 + IMM.Jump unconditionally to the exception

handler.Jump unconditionally to the exception

handler.

Figure 3.36

(continued)

used. For example, if M(Source) = 10101111, then the load byte instructionLB Rt,Source transfers

11111111 11111111 11111111 10101111

to the destination register Rt, whereas LBU Rt,Source transfers

00000000 00000000 00000000 10101111

to Rt. While most load and store instructions assume that full words are aligned onmemory word boundaries, that is, their addresses terminate with 00, the RX000 pro-

vides four special instructions LWL. LWR. SWL. and SWR to load and store mis-aligned words.

The RXOOO's data-processing instructions include a typical set of arithmetic andlogical operations. They employ two instruction types implying two different address-ing modes: I type, in which case the instruction contains a 16-bit immediate operand inits EMM field, and R type, in which case all operands are stored in registers. For exam-ple, the logical OR instruction

OR Rd.Rs.Rt

implements the word-OR operation Rd := Rs or Rt. whereas the corresponding ORimmediate instruction

201

ORI Rt.Rs.EMM

implements Rt := Rs or EMM, with EMM zero-extended to 32 bits.

The RX000 does not employ the usual set of status flags (zero, carry, overflow,and so on) to indicate special properties of results. The only exceptional condition thatis automatically detected is twos-complement overflow in the case of ADD, ADDI. andSUB. When that happens, an automatic trap occurs, accompanied by a switch fromuser to supervisor state. To avoid such traps, "unsigned" versions of the precedinginstructions are provided. ADDU, for example, is identical to ADD except that nooverflow trap occurs under any circumstances.

Four compare or "set" instructions test register values and place the binary testoutcome in a register Rd, effectively using Rd as a flag. For example, if Rt containszero, then the "set on less than" instruction

SLT Rd.Rs,Rt

determines whether Rs contains a negative number. If Rs is less than Rt. then SLT setsRd to 1; otherwise, it resets Rd to 0. While it seems a waste of hardware to use an entire32-bit register to store a binary flag, such exception-indicating registers are more easilyaccessed by exception-handling software than individual flag bits. However, certainother common operations are complicated; see problem 3.41.

For simplicity, we will not discuss the RXOOO's shift instruction, which has nounusual features. We will also not discuss the multiply and divide instructions, whichare unusual in that they require many cycles to execute and are handled by a specialarithmetic unit within the CPU. Once execution of a multiply or divide instructionbegins, other instructions may execute in parallel in the RX000*s main arithmetic-logiccircuitry.

In the program-control category, the RXOOO has unconditional "jump" instruc-tions, which employ absolute addressing with the J-type format, and conditional"branch" instructions, which employ PC-relative addressing and have the R format.The conditions tested by branch instructions are all determined by examining the con-tents of registers, which as noted above, serve as flags in this architecture. Consider, forexample, the branch on less than or equal to zero instruction

BLEZ Rs,EMM

It is executed in two clock cycles / and t +1. In the first cycle f, a target address TAR-GET is determined as follows. The address offset IMM has 2 bits appended to its rightend and the sign 5 of IMM (bit 15 of the instruction BLEZ) is extended by 14 bits toform a full 32-bit address. In other words, the branch address is given by

TARGET :=.v14.EMM.00

SECTION 3.3Instruction Sets

202 In the second clock cycle t + 1, the CPU checks for the branch condition by examining

the contents of the specified general register Rs. If Rs contains zero or if its sign bit is1, indicating a negative number, then the operation PC := PC + TARGET is performed.Since PC is automatically incremented by four at the start of each clock cycle, we haveeffectively added TARGET plus eight to the contents oT PC present at the start of cyclef; for brevity, this is indicated by PC + 8 + IMM in Figure 3.36.

The various branch instructions have "link" versions that unconditionally save thePC contents in a designated register. These are useful for implementing procedure callsand interrupts.

The design and control of instruction-processing logic are examined in Chap-ters 4 and 5.

3.3.3 Programming Considerations

To design programs using the instruction sets discussed in the preceding sections, asymbolic format called assembly language can be used. This section discusses thebasic features of assembly language and their relationship both to the computerorganization and to the machine-language programs that are actually executed bythe host processor. Most computer programming is now done using higher-levellanguages such as C, which, like assembly language, must be translated (compiled)into machine language prior to execution.

Assembly language. Machine-language programs (object programs) are lists ofinstructions, each of which has the general form

opcode operand,operand,...,operand

For example, the machine-language version of the instruction for the Motorola680X0 microprocessor series "Load the (immediate) decimal operand 2001 intoaddress register A0," which is used in the program of Figure 3.13, has the 32-bitbinary format

00100000 01111000 00000111 11010001 (3.33)

It may also be written more compactly in hexadecimal code thus:

2078 07D1 (3.34)

Here 2078 is the opcode word indicating "move long (32-bit) operand to registerA0," while the operand field 07D1 is the hexadecimal equivalent of the decimalnumber 2001. Assembly-language versions of this instruction are

MOVE.L #2001,A0 (3.35)

and MOVE.L #$07D1,A0 (3.36)

where the opcode and the operand A0 are represented in symbolic form. The prefix# denotes an immediate operand in the Motorola convention, while $ indicates thatbase 16 rather than base 10 is being used. Before they can be executed, assembly-language instructions like (3.35) and (3.36) must be translated into the equivalent

machine-language form represented by (3.33) and (3.34). The translation or assem-bly process is carried out by a system program known as an assembler, which isanalogous to a compiler that translates a high-level language program into machinecode.

In addition to using symbolic names for opcodes and registers, assembly lan-guages allow symbolic names to be assigned to user-defined constants and vari-ables, such as the immediate operand appearing in (3.35) and (3.36). For example,many assembly languages use the statement

A EQU 2001

(3.37)

to indicate that the symbol A is to be equivalent (EQU) to the decimal number2001. If statement (3.37) is present in a program for the 680X0 microprocessor,then (3.35) and (3.36) can be replaced by

MOVE.L A,A0

which is assembled into exactly the same machine code as before. This instruction also corresponds to the register-transfer operation denoted symbolically by A0 := A. Statement (3.37) is considered an assembly-language instruction but, unlike the MOVE instructions, does not translate into an executable instruction in machine language. Rather it is an instruction that tells the assembler how to treat the symbol A during the program-translation process. This type of nonexecutable assembly-language instruction is called a directive or pseudoinstruction.

The memory location to be assigned to an instruction can be indicated symbol-ically by means of a label at the beginning of an assembly-language statement. For example, the label LI in

LI MOVE.L A,A0 ; Load initial value into A0

(3.38)

is assigned to a physical memory address by the assembler, normally to the one immediately following the address assigned to the preceding instruction. Labels are generally used in an assembly-language instruction only when another instruc-tion needs to refer to the first one. For example, the 680X0 instruction

IMP LI ; Branch unconditionally to instruction labeled LI

(3.39)

causes a branch to instruction (3.38), which has the label LI; JMP LI is the assembly-language equivalent of the high-level language statement go to LI. All assembly languages allow the programmer to introduce comments, which have no effect on the assembly process but are useful for documenting a program to improve its readability. As illustrated by (3.38) and (3.39), 680X0 assembly lan-guage uses a semicolon as a prefix to mark comments.

Assemblers also allow the programmer to assign a symbolic name to a sequence of instructions, permitting those instructions to be treated as a single instruction like entity termed a macroinstruction, or simply a macro. Assembly lan-guages often have built-in macros that appear to the programmer to augment the machine's instruction set. For example, the MIPS RX000 machine language lacks the logical NOT instruction found in other computers. However, a NOT instruction 3f the form

203

CHAPTER 3

Processor

Basics

NOT Rd, Rs ; Form bitwise logical complement Rs of Rs and place in -Rd

204

SECTION 3.3 Instruction Sets

is easily synthesized from the RXOOO's NOR instruction, as follows:

NOR Rd, Rs, 0 ; Compute the NOR function Rs + 0 = Rs and place in Rd

Thus we conclude that assembly-language instructions have the following gen-eral format:

label opcode operand,operand,...,operand comments

where the opcode can be an executable command corresponding to a machine-language opcode, a directive, or a macro. Like machine languages, assembly lan-guages vary from computer to computer and are usually defined (not always consistently) by a computer's primary manufacturer.

Assembly process. The input to the assembler program is a source program written in assembly language. The output is an object program in machine lan-guage and an optional assembly listing that shows both the assembly-language and machine-language versions of the program and the correspondence between them. The object code can be combined with other machine-language programs to pro-duce a final composite executable program. A system program called a linker per-forms the task of combining different programs in this fashion. The use of symbolic names for shared data and labels plays an important role in allowing the linker to merge different assembly-language programs, or perhaps to merge the work of different programmers.

Nonexecutable assembly-language instructions such as the EQU statement (3.37) are known as directives. They are used to define the values of program parameters, to assign programs and data to specific physical or symbolic memory locations, and to control the output of the assembly process. In the case of macro-assemblers, directives are also used to define macros. Figure 3.37 lists a represen-tative set of the directives found in most assembly languages. The EQU directive tells the assembler to equate two different names for the same thing. In (3.37) EQU

Type

Opcode Description

Symbol definition Memory assigment

Macro definition

Miscellaneous

EQU Equate symbolic name (in label position) to operand value.

ORG Origin: use operand value as starting address for subsequent

instructions. DS Define storage: reserve the specified number of consecutive

locations (bytes) in memory. DC Define constant: store the operand values as constants.

MACRO Start of macro definition. ENDM End of macro definition.

END End of program(s) to be assembled.

TITLE Use operand as title on each page of assembly listing.

IF Start of conditional block of instructions to be assembled only

if a specified condition is met. ENDIF End of conditional block.

Figure 3.37

List of representative assembly-language directives.

assigns a symbolic name to a constant; it can also be used to equate two symbolic names for variables, as in

ALPHA EQU BETA

which defines a new variable ALPHA that must always have the same value as a previously defined parameter BETA. The ORG (origin) directive tells the assem-bler which memory address to assign for storing the subsequent executable code or data. For example, in

LI

ORG 100 MOVEL A, A0

the ORG directive states that the MOVE instruction is to be assigned to memory location 100, which equates the symbolic address or label LI to the physical address 100. The assembler needs this address value to translate into machine code the address fields of any instructions that refer to LI. Once the start address of a block of code has been established, the assembler automatically keeps track of the memory locations to be assigned to all items in the block.

Sometimes it is useful to reserve a block of memory for future use, for exam-ple, as a buffer storage area for IO data, without specifying its contents. The DS (define storage) instruction is provided for this purpose. Thus the directive

L2 DS 500

states that a block of 500 memory bytes should be reserved, beginning at the cur-rent location L2. If it is desired to actually define data to be placed in a program,the DC (define constant) directive is used. DS and DC typically exist in severalversions depending on the word size to be used. For example, the 680X0 directive

L3 DC.B 1,2,3,4,5,6,7

causes the seven specified operand values to be placed (in binary form) in sevenconsecutive 1-byte memory locations starting with L3. If the same data is to bestored in the ASCII character code, then the format

L3 DC.B '1234567'

is used. We now turn to an example that illustrates the directives discussed so far.

EXAMPLE 3.8 ASSEMBLY OF VECTOR ADDITION PROGRAM FOR THE

motorola 680X0 . This particular programming task, which was considered ear-lier for the IAS computer (Example 1.4), the PowerPC (Example 1.7), as well as the680X0 (Example 3.3), is to add two 1000-element vectors A and B creating a sum vec-tor C. We assume again that the vectors are 1000-byte decimal (BCD) numbers. The680X0 series has a 1-byte add instruction ABCD (add BCD), which is placed in a pro-gram loop and executed 1000 times to accomplish the desired vector addition. The pro-gram can be described abstractly in the following high-level language format:

forI= 1 to 1000 do

C[i] := A[i] + B[i] + carry; (3.40^

We assume that A, B, and C are stored in three consecutive 1000-byte blocks of mem-ory as depicted in Figure 3.38.

Hexadecimaladdress

0000

0100

01040108010C

0114

03E9

07D10BB9OFAO

- MOVE.L#2001. AO -

- MOVE.L#3001. Al -

_ 0256 •>

J 02600264

MOVE.L#4001. A2 I

ABCD-(AO), -(Al) - 0268

BXE SF6

VectorA

VectorB

VectorC

Decimaladdress

0000

> Program

0276

1001

2001

3001

4000

> Data

Figure 3.38Memory allocation forthe 680X0 vector additionprogram.

To determine how best to implement (3.40) in assembly language, the availableinstruction types and addressing modes must be examined carefully. The ABCDinstruction, besides being limited to byte operands, allows only two operand addressingmodes: direct register addressing and indirect register addressing with predecrement-ing. As explained earlier, the latter mode causes the contents of the designated addressregister to be automatically decremented just before the add operation is carried out.This approach is convenient for stepping through lists, in this case the elements of avector, and hence it is selected here. Two of the address registers A0 and Al are chosento address or point to the current elements of A and B, respectively. Thus the basicaddition step is implemented by the instruction

ABCD -(AOMAl)

(3.41)

which is equivalent to

A0:=A0-1, Al :=A1-1;M(A1) := M(A0) + M(A1) + carry:

A third address register A2 is used to point to vector C, and the result computed by(3.41) is stored in the C region by the 1-byte data transfer instruction

MOVE.B (A1),-(A2)

(3.42)

MOVE.L #2001, AO

MOVE.L #3001, A1

MOVE.L #4001,A2

ABCD   -(A0),-(A1)

MOVE.B (A1),-(A2)

CMPA   #1001,A0

START

Because addresses are predecremented, AO, Al, and A2 must be initialized to valuesthat are one greater than the highest addresses assigned to A, B, and C, respectively.The foregoing instructions (3.41) and (3.42) are executed 1000 times, that is, until thelowest address (1001 in the case of vector A) is reached. This point can be detected bythe CMPA (compare address) instruction

CMPA #1001, A0

which sets the zero-status flag Z to 1 if A0 = 1001 and to 0 otherwise. When Z ^ 1, abranch is made back to (3.41) using the BNE (branch if not equal to 1) instruction. Theresulting code, which also appears with comments in Figure 3.13, is as follows:

START

BNE

Figure 3.39 shows an assembly listing of the foregoing code with various direc-tives added for both illustrative purposes and to complete the program. The assembly-language source code appears on the right side of Figure 3.39, while the assembledobject program appears on the left in hexadecimal code. The left-most column containsthe memory addresses assigned by the assembler to the machine-language instructionsand data, which are then listed to the right of these memory addresses. The first ORGdirective causes the assembler to fix the start of the program at the hexadecimaladdress 0100. The symbolic names A, B, and C are assigned by EQU directives to theaddresses of the first elements of the three corresponding vectors. The subsequentMOVE.L (move long) instructions contain arithmetic expressions that are evaluatedduring assembly and replaced by the corresponding numerical value. For example, theexpression A + 1000 appearing in the first MOVE.L instruction is replaced by 1001 +1000 = 2001. In general, assembly languages allow arithmetic-logic expressions to beused as operands, provided the assembler can translate them to the form needed for theobject program. The statement MOVE.L #2001, A0 is thus the first executable state-ment of the program, and its machine-language equivalent 2078 07D1 is loaded intomemory locations 0100:0103 (hex), as indicated in Figures 3.38 and 3.39. The remain-der of the short program is translated to machine code and allocated to memory in sim-ilar fashion.

Many 680X0 branch instructions use relative addressing, which means that thebranch address is computed relative to the current address stored in the program counterPC. Consider, for instance, the conditional branch instruction BNE START, the lastexecutable instruction in the vector-addition program. As shown by Figure 3.39. thecorresponding machine-language instruction is 66F6 in which 66 is the opcode BNEand F6 is an 8-bit relative address derived from the operand START. Now F6)6 =111101102, which when interpreted as a twos-complement number is -10l0 or -0A,6.After BNE START has been fetched from memory locations 011416 and 011516, PC isautomatically incremented to point to the next consecutive memory location 0116,6.Hence at this point PC = 00000116,6. Now when the CPU executes the branch instruc-tion BNE, it computes the branch address as PC + (-0A) = 0000010C!6. which, asrequired, is the physical address of the instruction (ABCD) with the symbolic addressSTART.

The remainder of the vector-addition program illustrates the assembly-languagedirectives that define data regions. ORG is used again to establish a start address for thedata region; in this case the start address is 10011(, = 03E9I6. The DS.B (define storage

207

CHAPTER 3

Processor

Basics

208

SECTION 3.3Instruction Sets

Machine language

Location Code/Data

Assembly language

68000/68020 program for vector addition

The vectors are composed of a thousand 1-byte (two digit) decimalnumbers. The starting (decimal) addresses of A, B, and C are1001, 2001, and 3001, respectively.

0100 2078 07D1

0104 2278 0BB9

0108 2478 0FA1

010C C308

010E 1511

0110 B0F8 03E9

0114 66 F6

; Define origin of program at hex address 1000100 ORG $100

; Define symbolic vector start addresses03E9 A EQU 1001

07D1 B EQU 2001

0BB9 C EQU 3001

; Begin executable code

MOVE.L A+1000,AO ; Set pointer beyond end of AMOVE.L B+1000,A1 ; Set pointer beyond end of BMOVE.L C+1000,A2 ; Set pointer beyond end of CSTART ABCD -(A0), -(Al); Decrement pointers & addMOVE.B (A1),-(A2) ; Store result in CCMPA A,A0 ; Test for termination

BNE START : Branch to START if Z * 1

; End executable code

03E9

03E9

07D1 010101

07D4 16 16 16

Begin data definition

ORG A

DS.B 1000

DC.B 1.1.1

DC.B 22,22,22

END

; Define start of vector A: Reserve 1000 bytes for A; Initialize elements 1:3 of B: Initialize elements 4:6 of B; End program

Figure 3.39

Assembly listing of the 680X0 program for vector addition.

in bytes) directive reserves a region of 1000 bytes. This directive merely causes the assembler's memory location counter, which it uses to keep track of memory addresses, to be incremented by the specified number of bytes. As indicated by Figure 3.38, this action makes the location counter point to the start of the region storing vec-tor B. The two DC. B (define constant in bytes) commands initialize six elements of B to the specified constant values. Finally the END directive indicates the end of the assembly-language program.

Macros and subroutines. Two useful tools for simplifying program design by allowing groups of instructions to be treated as single entities are macros and sub-routines. A macro is defined by placing a portion of assembly-language code between appropriate directives as follows:

name MACRO operand,..., operand

Body of macro

ENDM

The macro is subsequently invoked by treating the user-defined macro name, which appears in the label field of the MACRO directive, as the opcode of a new (macro) instruction. Each time the macro opcode appears in a program, the assem-bler replaces it by a copy of the corresponding macro body. If the macro has oper-ands, then the assembler modifies each copy of the macro body that it generates by inserting the operands included in the current macro instruction. Macros thus allow an assembly language to be augmented by new opcodes for all types of operations; they can also indirectly introduce new data types and addressing modes. A macro is typically used to replace a short sequence of instructions that occur frequently in a program. Note that although macros shorten the source code, they do not shorten the object code assembled from it.

Suppose, for example, that the following two-instruction sequence occurs in a program for the Intel 8085 [Intel 1979]:

LDHL ADR ;Load M(ADR) into address register HLMOV A,M ;Load M(HL) into accumulator register A

This code implements the operation A := M(M(ADR)), which loads register A treating ADR as an indirect memory address. We can define it as a macro named LDAI (load accumulator indirect) as follows:

LDAI

MACROLDHLMOVENDM

ADRADR

A,M

;Load M(ADR) into address register HL;Load M(HL) into register A

With this macro definition present in an 8085 program, LDAI becomes a new assembly-language instruction for the programmer to use. The subsequent occur-rence of a statement such as

LDAI 1000H (3.43)

in the same program causes the assembler to replace it by the macro body

LDHL 1000HMOV A,M

with the immediate address 100016 from (3.43) replacing the macro's dummy input parameter ADR. Note that the macro definition itself is not part of the object pro-gram.

A subroutine or procedure is also a sequence of instructions that can be invoked by name, much like a single (macro) instruction. Unlike a macro, how-ever, a subroutine definition is assembled into object code. It is subsequently used.

SECTION 3.3 Instruction Sets

not by replicating the body of the subroutine during assembly, but rather during program execution by establishing dynamic links between the subroutine object code and the points in the program where the subroutine is needed. The necessary links are established by means of two executable instructions named CALL or JUMP TO SUBROUTINE, and RETURN. Consider, for example, the following code segment:

CALL SUB 1

NEXT

SUB1

Main (calling) program

Subroutine SUB 1

RETURN

After CALL SUB1 has been fetched, the program counter PC contains the address NEXT of the instruction immediately following CALL; this return address must be saved to allow control to be returned later to the main program. Thus a call instruction first saves the contents of PC in a designated save area. It then transfers the address that forms the operand of the call statement, SUB1 in this case, into PC. SUB 1 is the address of the first executable instruction in the subroutine and also serves as the subroutine's name. The processor then begins execution of the sub-routine. Control is returned to the original program from the subroutine by execut-ing RETURN, which simply retrieves the previously saved return address and restores it to PC.

CALL and RETURN may use specific CPU registers or main-memory loca-tions to store return addresses. The RX000, for instance, uses a CPU register from its register file to save a return address on executing any of its jump/branch-and-link-register instructions, which serve as call instructions; see Figure 3.36. Many computers use a memory stack for this purpose. CALL then pushes the return address into the stack, from which it is subsequently retrieved by RETURN. The stack pointer SP automatically keeps track of the top of the stack, where the last return address was pushed by CALL and from which it will be popped by RETURN.

Figure 3.40 illustrates the actions taken by the CALL instruction in a stack realization. For simplicity, we assume that opcodes and the addresses are all one memory word long. The instruction CALL SUB1 is stored in memory locations 1000 and 1001, and we assume that the assembler has replaced SUB1 with the physical address 2000. Immediately before the CALL instruction cycle begins, the program counter PC contains the address 1000, as shown in Figure 3.40a. The CALL opcode is fetched and decoded, and PC is incremented to 1001. On identify-ing the instruction as a subroutine call, the CPU fetches the address part 2000 of the instruction and stores it in the (buffer) address register AR; again PC is incre-mented to 1002. At this point the system state is as shown in Figure 3.40b, and PC contains the return address to the main program. Next the contents of PC are pushed into the stack. Then the contents of AR are transferred to PC, and the stack

IR | MOV

AR 17893PC

1000

SP I 3500

CALL

2000

Main- program

Subroutine" SUB1

RETURN

1034

*

Stack y»

1000

2000

3500

IR |CALL

AR |2000

PC | 1002 •-{-

SP I 3500
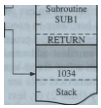
CALL

2000 _

Main

program

1000

2000

3500

(a)

(b)

IR | CALL |

AR | 2000

2000

3499 «f-[

_CALL_2000

Mainprogram

SubroutineSUB1

RETURN

1002

1000

2000

3500

(c)

Figure 3.40

Processor and memory state during execution of a CALL instruction: (a) initial state,(b) state immediately after fetching the instruction, and (c) final state.

pointer SP is decremented by one. The resulting state of the system is depicted inFigure 3.40c.

3.4SUMMARY

The main task of a CPU is to fetch instructions from an external memory M andexecute them. This task requires a program counter PC to keep track of the activeinstruction, and registers to store the instructions and data as they are processed.The simplest CPUs employ a central data register called an accumulator, alongwith an ALU capable of addition, subtraction, and word-oriented logic operations.In most CPUs a register file containing 32 or more general-purpose registers

SECTION 3.5Problems

212 replaces the accumulator. RISC processors such as the ARM and the MIPS RXOOO

allow only load and store instructions to access M, and use small instruction setsand techniques such as pipelining to improve performance. CISC processors suchas the Motorola 680X0 have larger instruction sets and some more powerfulinstructions that improve performance in some applications but reduce it in others.The arithmetic capabilities of simpler processors are limited to the fixed-point(integer) instructions unless auxiliary coprocessors are used. More powerful CPUshave built-in hardware to execute floating-point instructions.

Computers store and process information in various formats. The basic unit ofstorage (the smallest addressable unit) is the 8-bit byte. The CPU is designed tohandle data in a few fixed-word sizes, 32-bit words being typical. The two majorformats for numerical data are fixed-point and floating-point. Fixed-point numberscan be binary (base 2) or, less frequently, decimal, meaning a binary code such asBCD that preserves the decimal weights found in ordinary (base 10) decimal num-bers. The most common binary number codes are sign magnitude and twos com-plement. Each code simplifies the implementation of some arithmetic operations;twos complement, for example, simplifies the implementation of addition and sub-traction and so is generally preferred. A floating-point number comprises a pair offixed-point numbers, a mantissa M, and an exponent E and represents numbers ofthe form M X BE where B is an implicit base. Floating-point numbers greatlyincrease the numerical range obtainable using a given word size but require muchmore complex arithmetic circuits than fixed-point numbers require. The IEEE 754standard for floating-point numbers is widely used.

The functions performed by a CPU are defined by its instruction set. Aninstruction consists of an opcode and a set of operand or address fields. Varioustechniques called addressing modes are used to specify operands. An instruction'soperands can be in the instruction itself (immediate addressing), in CPU registers,or in external memory M. Operands in registers can be accessed more rapidly thanthose in M. An instruction set should be complete, efficient, and easy to use insome broad sense. Instructions can be grouped into several major types: data trans-fer (load, store, move register, and input-output instructions), data processing(arithmetic and logical instructions), and program control (conditional and uncon-ditional branches). All practical computers contain at least a few instructions ofeach type, although in theory one or two instruction types suffice to perform allcomputations. RISCs are characterized by streamlined instruction sets that are sup-ported by fast hardware implementations and efficient software compilers. WhileCISCs have larger and more complex instruction sets, they simplify the program-ming of complex functions such as division. The use of subroutines (procedures)and macroinstructions can simplify assembly-language programming in all typesof processors.

3.5PROBLEMS

3.1. Show how to use the 10-member instruction set of Figure 3.4 to implement the follow-ing operations that correspond to single instructions in many computers; use as few in-structions as you can. (a) Copy the contents of memory location X to memory locationY. (b) Increment the accumulator AC. (c) Branch to a specified address adr if AC ^ 0.

3.2. Use the instruction set of Figure 3.4 to implement the following two operations as-suming that sign-magnitude code is used, (a) AC := -M(X). (b) Test the right-most bitb of the word stored in a designated memory location X. If b = 1, clear AC; otherwise,leave AC unchanged. [Hint: Use an AND instruction to mask out certain bits of aword.]

3.3. Consider the possibility of overlapping instruction fetch and execute operations whenexecuting the multiplication program of Figure 3.5. (a) Assuming only one word canbe transferred over the system bus at a time, determine which instructions can be over-lapped with neighboring instructions, (b) Suppose that the CPU-memory interface isredesigned to allow one instruction fetch and one data load or store to occur during thesame clock cycle. Now determine which instructions, if any, in the multiplication can-not be overlapped with neighboring instructions.

3.4. Write a brief note discussing one advantage and one disadvantage of each of the fol-lowing two unusual features of the ARM6: (a) the inclusion of the program counter PCin the general register file; (b) the fact that execution of every instruction is conditional.

3.5. Use HDL notation and ordinary English to describe the actions performed by eachof the following ARM6 instructions: (a) MOV R6,#0; (b) MVN R6,#0; (c)ADD R6,R6,R6; (d) EOR R6,R6,R6.

3.6. Suppose the ARM6 has the following initial register contents (all given in hex code):

Rl = 11110000; R2 = 0000FFFF; R3 = 12345678; NZCV = 0000

Identify the new contents of every register or flag that is changed by execution of thefollowing instructions. Assume each is executed separately with the foregoing initialstate, (a) MOV R1,R2; (b) MOVCS R1.R2; (c) MVNCS R2.R1; (d) MOV R3,#0;(e)MOV R3,R4, LSL#4.

3.7. Suppose the ARM6 has the following initial register and memory contents (all givenin hex code):

Rl = 00000000; R2 = 87654321; R3 = A05B77F9; NZCV = 0000

Identify the new contents of every register or flag that is changed by executionof the following instructions. Assume each is executed separately with the forego-ing initial state, (a) ADD R1,R2,R3; (b) ADDS R1.R3.R3; (c) SUBS R2,R1,#1;(</)ANDS R3,R2,Rl;(e)EORCSS R1,R2,R3.

3.8. Use the instruction set for the ARM6 given in Figure 3.10 to write short code segmentsto perform the tasks given below. Note that an opcode can be followed by two optionalsuffixes, a two-character condition code to determine branching and S to activate thestatus flags. Figure 3.41 lists all possible condition fields. The required tasks are:(a) Replace the contents of register Rl by its absolute value, (b) Perform the 64-bitsubtraction R5.R4 := R1.R0 - R3.R2, where the even-numbered registers contain theright (less significant) half of each operand.

3.9. Write the shortest ARM6 program that you can to implement the following conditionalstatement:

while (x ^ y) do x := \ - 1:Assume that x and y are stored in CPU registers R1 and R2, respectively.

213

CHAPTER 3

Processor

Basics

214   Code   Mnemonic Flag test Usual interpretation

	Code	Mnemonic		Flag test	Usual interpretation
SECTION 3.5Problems	00000001	EQNE		Z= 1 Z = 0	Result equal to zero.Result not equal to zero.
	0010	CS or	HS	C=l	Unsigned overflow: result higher or same.
	0011	CC or	LO	C = 0	No unsigned overflow: result lower.
	0100	MI		N=l	Result negative.
	0101	PL		N = 0	Result positive or zero.
	0110	VS		V= 1	Signed overflow.

0111	VC	V = 0	No signed overflow.
1000	HI	C = 1 and Z = 0	Unsigned result higher.
1001	LS	C = 0 or Z = 1	Unsigned result lower or same.
1010	GE	N = V	Signed result greater or equal.
1011	LT	N =V	Signed result less than.
1100	GT	Z = 0 and N = V	Signed result greater than.
1101	LE	Z = 1 or N = V	Signed result less than or equal.
1110	AL	None	Always (unconditional branch).
mi	NV	None	Never (no branching)

Figure 3.41

Condition codes of the ARM6 and their interpretation.

3.10. Identify five major differences between the instruction sets of the ARM6 and the680X0 and comment on their impact on the CPU cost and performance.

3.11. Use HDL notation and ordinary English to write the actions performed by each ofthe following 680X0 instructions: (a) MOVE (A5)+,D5; (b) ADD.B $2A10,D0;(c) SUBI #10,(A0); (d) AND.L #SFF,D0.

3.12. The 680X0 has two types of unconditional branch instructions BRA (branch always)and JMP (jump). Therefore, branch to statement L can be implemented either by BRAL or JMP L. What is the difference between these two instructions? Under what cir-cumstances is each type of branch instruction preferred?

3.13. Write a program for the 680X0 that replaces the word DATA stored in memory loca-tion ADR by its bitwise logical complement DATA if and only if DATA ^ 0.

3.14. Modify the vector addition program of Figure 3.13 (Example 3.3) to compute the sumC := A + B for 100 instead of 1000 one-byte decimal numbers. Assume that the loca-tions of the A and B operands are unchanged, but the result C is now required to replace(overwrite) B.

3.15. Suppose that the hex contents of two CPU registers in a 32-bit processor are as follows:

R0 = 01237654: Rl = 7654EDCB

The following store-word instructions are executed to transfer the contents of theseregisters to main memory M.

STORE R0,ADR 215

STORE Rl.ADR+4 CHAPTER 3

Assuming that M is byte-addressable, give the contents of all memory locationsaffected by the above code (a) if the computer is big-endian and (b) if the computer islittle-endian.

3.16. Suppose that a 680X0-based computer C,, which is big-endian, is communicatingwith another computer C2, which is similar to Cx except that it is little-endian. C2stores 4-byte (long) words from its register file into a common memory M, which Cxsubsequently loads into its data registers. Outline an efficient way to program C,'sload operations so that data words always appear in the correct form in its register file.

3.17. The usual objection to tagged architecture is that the presence of tags in stored data in-creases memory size and cost. It has been argued, however, that tags can actually re-duce storage requirements by decreasing program size. Analyze the validity of thisargument.

3.18. Figure 3.42 lists all the 16 code words of a code known as a Hamming code [Ham-ming 1986], which is designed to check 4-bit words using three check bits. Prove thatall single-bit errors can be corrected and all double-bit errors can be detected by thiscode.

3.19. Consider the small Hamming code defined in Figure 3.42. Show that each check bit c,can be expressed in the form c, = axdx © a2d2 © a3d3 © a4d4, where ay = 0 or 1 and djis an information (data) bit. Hence the check bits for this (and other) Hamming codescan be generated by a set of EXCLUSIVE-OR (parity) circuits.

Information bits Check bits

0 0 0 0 0 0 0

0 0 0 1 111

0 0 10 110

0 0 11 0 0 1

0 10 0 10 1

0 10 1 0 10

0 110 0 11

0 111 10 0

10 0 0 0 11

10 0 1 10 0

10 10 10 1

10 11 0 10

110 0 110

110 1 0 0 1

1110 0 0 0

1111 111

ProcessorBasics

Figure 3.42

Hamming SECDED code for 4-bit words.

216

SECTION 3.5Problems

3.20. Convert the following three 2'-bit words to standard decimal form assuming they rep-resent (a) sign-magnitude and (b) twos-complement integers: FFFF16; FEDCBA9816;7EDCBA916.

3.21. The following binary word W= 10001011101001 is stored in a 14-bit register. What is the decimal number represented by W if it is interpreted as an integer in each of the fol-lowing codes: (a) unsigned binary; (b) sign-magnitude; (c) twos-complement?

3.22. Using 32-bit integer formats, give the sign-magnitude, twos-complement, and BCD representation of each of the following decimal numbers: +999, -999, +1000, -1000, zero. State your assumptions concerning sign representation.

3.23. (a) What are the decimal equivalents of the largest fixed-point binary numbers that can be represented in 32-, 64-, and 128-bit words? (b) Convert the following sign-magnitude words to decimal: 10111011, 01010101, 1011101010111010. (c) Repeat part (b) assuming this time that the numbers are in twos-complement code.

3.24. Figure 3.43 shows the single-precision number format used in the B6500/7500 and other early Burroughs computers. This format is used for both fixed- and floating-point numbers—an unusual feature. The total length of a number is 47 bits, including the ex-ponent, mantissa, and two sign bits. The implicit number base B - 8. Fixed-point num-bers are treated as a special case of floating point where the exponent E is always zero (encoded as 0000002). The exponent and mantissa are treated as sign-magnitude inte-gers and biasing is not used. Write a note listing the advantages and disadvantages of combining fixed- and floating-point representation in this way.

3.25. Consider again the B6500/7500 single-precision number format described in the pre-ceding problem, (a) Give in decimal form the largest and the smallest nonzero numbers that can be represented, when no normalization is used, (b) Again calculate the largest and the smallest nonzero numbers, this time assuming that the numbers are normalized according to the following definition: a B6500/7500 number is normal if there are no leading-zero digits in the mantissa.

3.26. A floating-point processor is being designed with a number format that must meet the following requirements:

• Numbers in the range ±1.0 x 10*64 must be represented.

• The precision required is eight decimal digits; that is, the eight most significant dig-

its of the decimal equivalent of every number in the required range must be repre-sentable.

Unused bitSign of MSign of E

000   E   M

        Mill 1 1 1 [1 1 1 1 1 1 1 1 1 1 1 1 ! 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
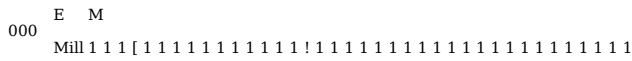
Tag Exponent

E (6 bits)

Mantissa M (39 bits)

Figure 3.43

The B6500/7500 format for single-precision numbers.

CHAPTER 3

• The representation of each number should be unique, with zero represented by a 217

sequence of Os.

• Binary arithmetic is to be used throughout with B = 2. where B is the floating-point

number base. Processor

BasicsDesign a number format that satisfies these requirements and uses as few bits as pos-sible. Indicate clearly the number codes used and why they were chosen.

3.27. Suppose that in the 6-bit floating-point format illustrated by Figure 3.24, B = 2, E is a 3-bit sign-magnitude integer as before, but M is now a 3-bit sign-magnitude fraction.

(a) What are the decimal values of the largest and smallest nonzero real numbers that can be represented by this format? (b) How many different real numbers can be rep-resented?

3.28. Consider the 6-bit floating-point format defined in Figure 3.24. Suppose that E and B are unchanged, but M is a 3-bit sign-magnitude fraction and that all floating-point num-bers are normalized with an excess-A' biased exponent, (a) What is a suitable value for this bias K and why? (b) How many different real numbers can be represented in this normalized format?

3.29. Obtain the (approximate) decimal values that conform to the IEEE 754 floating-point format of the following two numbers:

A= 100101111 100000000000000000000000

5 = 0 10001110 00000000000000000000001

3.30. Derive the correct floating-point representation for the decimal numbers +3.25 and-3.25 using the 32-bit IEEE 754 floating-point standard.

3.31. Consider the 64-bit IEEE floating-point number format defined in section 3.2.3. Deter-mine the largest positive number, the smallest nonzero positive number, and the nega-tive number with the largest magnitude that can be represented in this format. Assume that the three numbers are to be normalized and give your answers in the form of 16-digit hexadecimal strings.

3.32. The floating-point number format used by the IBM System/360-370 series is defined in section 3.2.3. Determine the total number of different normalized numbers that the 32-bit version of this format can represent.

3.33. Consider a 32-bit RISC-style processor P whose only addressing modes for register-to-register instructions are immediate and direct and whose only addressing mode for load/store instructions is register indirect with offset. Assume also that the CPU has 64 general-purpose registers R0:R63 that can serve either as data or address registers. A single 32-bit instruction format contains four fields: an opcode, two register fields, and a 16-bit immediate address field, (a) What is the maximum number of opcode types?

(b) Using an ad hoc but typical assembly-language notation with clear comments, de-scribe how a single instruction of P might perform each of the following three opera-tions: load a word from M; store a byte in M; double the number word stored in a register (there is no multiply opcode).

3.34. Consider the 32-bit RISC-style processor P sketched in the preceding problem. De-scribe how one or more instructions of P might perform each of the following three op-erations, assuming that P has no explicit clear, swap, or push opcodes: clear a register; swap the contents of two registers; push a word into a stack. Again use an ad hoc but

218 typical assembly-language notation with clear explanatory comments. Use as few in-

structions as you can.

SECTION 3.5Problems

3.35. Suppose the memory data register DR in a CPU like that of Figure 3.3 transfers 32-bit words to M in a single clock cycle. The data item D t© be stored may be 16 or 32 bits long. If a 16-bit data item D is placed in DR, it is automatically extended to 32 bits as it is transmitted from DR to M. The size of D is given by a flag 5, whose 0 and 1 values denote 16 and 32 bits, respectively. The extension method is given by a second flag E, whose 0 and 1 values denote zero extension and sign extension, respectively. Design a register-level logic circuit to perform the needed extension, making it as simple and as fast as possible.

3.36. A memory data register DR can transfer 32-bit words to M in a single clock cycle. The data items to be stored can be 4, 8, 16, or 32 bits long, and short items are always sign-extended to 32 bits for transmission to M. A 2-bit flag S in the CPU is set to 00, 01, 10, or 11 to indicate a data size of 4, 8, 16, or 32 bits, respectively. Design an efficient logic circuit at the register level to implement the sign extension.

3.37. Consider the instruction formats of the MIPS RX000 defined in Example 3.5. Suppose that the currently executing instruction / in an RX000 CPU is stored at (hexadecimal) memory address FFFFFF0016. (a) If/is not a branch instruction, what is the (hexadec-imal) memory address of the instruction that will be executed immediately after /? (b) Suppose that / is an unconditional jump instruction that contains the 26-bit branch ad-dress field ADR = 2A9FFFF16. Again what is the (hexadecimal) memory address of the instruction that will be executed immediately after /?

3.38. Use a figure similar to Figure 3.31 to show the state of the CPU and M in the Motorola 680X0 immediately before and after execution of the stack-pop instruction MOVE.L (A2)+,D6.

3.39. The stack shown in Figure 3.31 for a 680X0-based computer grows toward the low-address end of M. Suppose that the stack is required to grow in the opposite direction, that is, toward the high-address end of M. Construct the push and pop instructions needed for this case.

3.40. The 680X0 instruction JSR SUB pushes the contents of the program counter PC onto a stack using stack pointer register SP and then causes a jump to the instruction at mem-ory location SUB. Its operation may be described as follows:

(-SP) := PC; PC := SUB

(a) Show how to use the 680X0 MOVE instructions to simulate JSR, assuming that JSR can have SP and PC as operands, (b) The last instruction executed by a subroutine should be return from subroutine (RTS) which restores to PC the address saved earlier by JSR; this instruction should also update SP. Again use the 680X0 MOVE instruc-tions to simulate RTS, again assuming that SP and PC can be operands of MOVE.

3.41. The MIPS RX000 has no status flag C to indicate whether an arithmetic instruction ap-plied to an unsigned number generates a carry, that is, overflows a 32-bit register. In fact, the RX000 add unsigned instruction that computes Rd := Rs + Rt

ADDU Rd,Rs,Rt

sets no status flags under any circumstances. Using standard instructions (but no flags), devise a short program that will determine whether the foregoing instruction

causes overflow. A useful RX000 instruction for this purpose is the compare instruc- 219

tion

CHAPTER 3SLTU Rd,Rs,Rt Processor

which compares the contents of Rs and Rt, treating both as 32-bit unsigned numbers. BasicsIf Rs < Rt, then Rd := 1 = 0311; otherwise, Rd := 0 - 032. The RX000 also has a typicalset of conditional branch instructions that test the contents of a register for zero.

3.42. An arithmetic right shift (ARS) instruction—arithmetic left shifts are uncommon—shifts an operand D k bits to the right and fills the vacated positions by sign extension. The bits shifted out from the right end of D are discarded. It is often stated that a fc-bitARS implements division by 2k when applied to a twos-complement integer D; that is,the shifted result SD is the integer quotient Q on dividing D by 2k. The discarded bitsrepresent the integer remainder R. (a) Show that this division-by-2* interpretation isvalid when D is positive, (b) Show that the division-by-2* interpretation of ARS is in-valid for negative D by considering operands of length 4 bits and finding a specificcounterexample.

3.43. As noted in the preceding problem, ARS instructions cannot be used directly to im-plement division of twos-complement integers by 2k. Some computers provide a spe-cial instruction—let us call it SI—such that if we apply SI to the result SD producedby a k-bit ARS, we obtain the correct integer quotient for division by 2k. For this two-instruction combination to work, ARS is designed to set a special flag F when its in-put operand D is negative and the bits shifted out and discarded by ARS include atleast one 1 bit. What is the function performed by SI? Explain informally how itworks with ARS to implement division by 2k.

3.44. Single-instruction computers (SICs) have attracted interest for many years. They areextreme cases of RISCs in which the instruction set has been reduced to the absoluteminimum. One type of SIC is based on a conditional move (CMOVE) instruction. Thisinstruction has the two-address format

CMOVE dest, source

corresponding to if cond then dest := source, where cond is a condition code and allmovable items are w-bit words stored in a common rc-bit address space shared by M,10 devices, and CPU registers (which can be placed in M). CMOVE combines condi-tional load and store instructions of the type found in the ARM—it is a pure load/storearchitecture. The CPU contains the logic needed to fetch instructions (a PC andaddress-generation logic), but it does not contain the usual ALU logic. Instead, special"10 processors" execute all arithmetic and logical operations. For example, A x B isimplemented by moving A, B, and any necessary control words to the input ports of anexternal multiplier MULT and subsequently moving the result from MULTs outputport. The tested conditions cond can include a flag C that is set by the sign bit of thelast word moved. It is also desirable to have an always-true condition to implement anunconditional move. Most proposed CMOVE architectures support a few addressingmodes, including indexing. Write a note analyzing the advantages and disadvantagesof this type of SIC architecture.

3.45. Consider a set of four processors P0, />,, 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 pro-grams 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 assembly-language 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. En-

glewood 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 z_i, c_i of two 1-bit numbers x, and v, can be expressed by the half-adderlogic equations

z, = x, 0 >>,-

where z_t 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 ripple-carry 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,
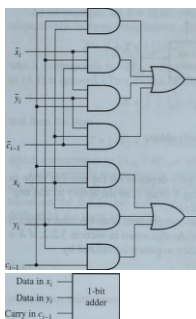
(a)

(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

CHAPTER 4 Datapath Design

Data
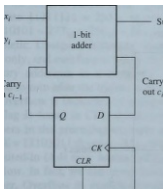
Carry

Sum.

Reset Clock

Figure 4.2

A serial binary adder.

L J

1-bit adder

1-bit adder

c_{n-i}

rr Ti

$-\hat{}n\text{-}l > 'n\text{-}I \ x_{n}\text{-}l \ V_{n}\text{-}l$

1-bit adder

$*0 \ ^{}0$

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 in use. 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 $X = x_{n-1} x_{n-2}...x_0$ is a twos-complement integer, then negation is realized by

$-X =$

$-l*_{n-2}-$

$.. X_n + 1$

(4.2)

where + denotes addition modulo 2^n. An efficient way to obtain the ones-comple-ment portion $\bar{X} = \bar{x}_n \bar{x}_{n-2}... \bar{x}_0$ of -X in (4.2) uses the word-based EXCLUSIVE-OR function $X \oplus s$ with a control variable s. When s = 0, $X \oplus s = X$, but when $s=1, X \oplus s = \bar{X}$. Suppose that Y and $X \oplus s$ are now applied to the inputs of an n-bit adder. The addition of 1 required by (4.2) to change X to -X can be realized by applying s to the carry input line of the adder. In the resulting circuit shown in Fig-ure 4.4, the control line s selects the addition operation Y + X when s = 0 and the subtraction operation $Y — X = Y + \bar{X} + 1$ when s=1. Thus extending a parallel adder to perform twos-complement subtraction as well as addition merely requires connecting n two-input EXCLUSIVE-OR gates to the adder's inputs; these gates are represented by a single rc-bit word gate in Figure 4.4.

$z = \quad Y \pm X$

1 \

n

Carry

Carry $\hat{}$ cn-\ rc-bit parallel adder 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 -21_{10} and 40_{10}, 227

respectively, in twos-complement code. Bit-by-bit addition produces

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

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,+ *,<:,_, + >',<",_x, denoting the carry signal c, tobe sent to stage i + 1, can be rewritten in terms of g, and p,.

ct=g,+Pf,-\ (4-8)

Similarly, c{_x can be expressed in terms of g{_x, /?,_,, and c,_2.

c^_1=ft_1+p^1cl_2 (4.9)

On substituting (4.9) into (4.8) we obtain

Ci = gi+Pig^i+PiPi-iCi-2

Continuing in this way, c, can be expressed as a sum-of-products function of the pand g outputs of all the preceding stages. For example, the carries in a four-stagecarry-lookahead adder are defined as follows:

C\=g\+P\80 + PlP0Cin

C2 = 82 + P28l + P2P180 + P2PlP0cin

c3 = 8i+ P382 + /^Sl + PzP2P\80 + P3P2P\P0cin

(4.10)

c„-i

Carry-lookahead generator

Pn-\

A

Sn-\

1-bitadder

Pn-2

Zn-2

<V2

1-bitadder

rr n

Po

1-bitadder

xn-\ yn-\

cn-2 yn-2

n

f0 >0

Figure 4.5

Overall structure of carry-lookahead adder.

Figure 4.5 shows the general form of a carry-lookahead adder circuit designed inthis way.

We can further simplify the design by noting that the sum equation for stage /

is equivalent to

$z_i = \wedge ev, \copyright c,\_,$

$zf = />, \copyright \& \copyright <:,.\_,$

(4.11)

Combining the pg equations (4.7), the carry-lookahead equations (4.10), and themodified sum equations (4.11) for $0 < i < 3$, we obtain the 4-bit carry-lookaheadadder depicted in Figure 4.6. This design is found in practical adders such as the74283 IC [Texas Instruments 1988]. It has four levels of logic gates, so the adder'smaximum delay is $4d$, where d is the (average) gate delay. This delay is indepen-dent of the number of inputs n as long as carry generation is defined by two-levellogic as in (4.10). However, the number of gates grows in proportion to n^2 as nincreases. In contrast, the number of gates in a two-level adder of the sum-of-products type grows exponentially with n, while the number of gates in a ripple-carry adder grows linearly with n. The complexity of the carry-generation logic inthe carry-lookahead adder, including its gate count, its maximum fan-in, and itsmaximum fan-out, increases steadily with n. Such practical cost considerationslimit n in a single carry-lookahead adder module to four or so.

Adder expansion. The methods of handling carry signals in the two main com-binational adder designs considered so far, namely, ripple-carry propagation (Fig-ure 4.3) and carry-lookahead (Figure 4.5), can be extended to larger adders of thekind needed to execute add instructions in, say, a 64-bit computer. If we replace then 1-bit (full) adder stages in the /7-bit ripple-carry design of Figure 4.3 with n k-bitadders, we obtain an nk-bit adder. Four 4-bit adders such as the 4-bit carry-lookahead circuit of Figure 4.6 can be connected in this way to form the 16-bitadder appearing in Figure 4.7. This design represents a compromise between a 16-stage ripple-carry adder, which is cheap but slow, and a single-stage 16-bit

SECTION 4.1

Fixed-Point

Arithmetic
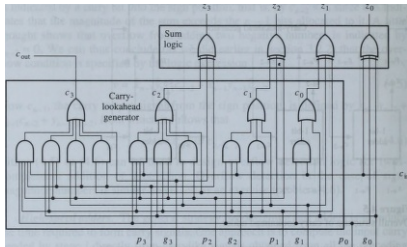
*G9 m fts m

Figure 4.6

A 4-bit carry-lookahead adder.

M5:-12

y-i x2 yi x\ y\ xo >o
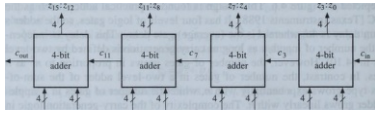
^15^12 yi5:>'l2

*n:*8 >n:>8

x1:x4 >7:y4

xi-xo ys-yo

Figure 4.7

A 16-bit adder composed of 4-bit adders linked by ripple-carry propagation.

carry-lookahead adder, which is fast, expensive, and impractical because of thecomplexity of its carry-generation logic. The circuit of Figure 4.7 effectively com-bines sets of four xiyi inputs into groups that are added via carry lookahead; theresults computed by the various groups are then linked via ripple carries.

Comparing Figures 4.3 and 4.7, we see that we have effectively replaced com-ponents designed for 1-bit addition with similar but larger components intended for4-bit addition. If we apply the same principle to the carry-lookahead circuit of Fig-ure 4.5, we get the expanded design of Figure 4.8. Again we are replacing 1-bit

- c3 4-bit carry-lookahead generator ln ■*—i

P3 #3 c2 Pi 8l c\ P\ 8\ c0 Po So

i		1 i		i i	i i k	i	i i	i 1	\ i	i

P g P g ^7^4 P 8 P B

4-bitadder Cll 4-bitadder c7 4-bitadder ■4 ' 4-bitadder c in
•<

4$ 4$ 4| 4| 4> 4j 4> 4}

xi S:x12 y\s :>'i 2 *11:*S >u :.v ! x?:^ jv^ x3:x0 ft^o
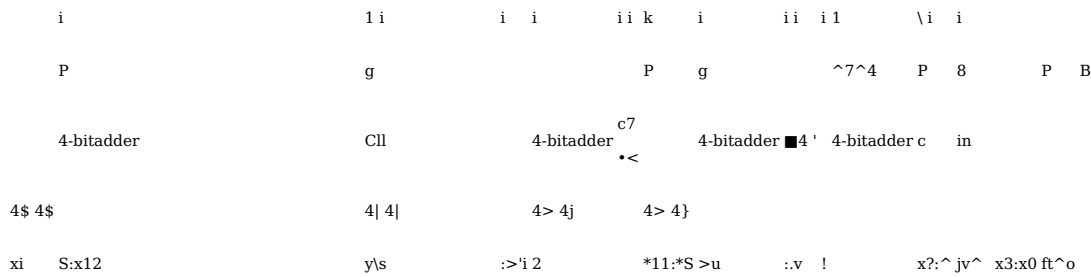
Figure 4.8

A 16-bit adder composed of 4-bit adders linked by carry lookahead.

adders with 4-bit adders, but now each adder stage produces a propagate-generatesignal pair pg instead of cout, and a carry-lookahead generator converts the four setsof pg signals to the carry inputs required by the four stages. The "group" g and psignals produced by each 4-bit stage are defined by

$g = x_i y_i + x_{i\_1} y_{i\_1}(x_i + y_i) + x_{i\_2} y_{i\_2}(x_i + >'),)(*,\_! + yt\_{}$

$+ *t-a)Y-3\&j+y^xi-\ + y_{i-}\)(x_{i-2}+y_{,-2})p = (*,- + y^-i + y,\_i)(x,\_2 + y,\_2)(*,\_3 + y,-\_3)$

(4.12)

which directly extend (4.7). It is not hard to show that the logic to generate thegroup carry signals cout, cn, c7, and c3 in Figure 4.8 is exactly the same as that ofthe carry-lookahead generator of Figure 4.6 and is therefore defined by Equations(4.10).

EXAMPLE 4.1 DESIGN OF A COMPLETE TWOS-COMPLEMENT ADDER-SUBTRACTER. To illustrate the preceding concepts, we will design a twos-comple-ment adder-subtracter that computes the three quantities $X + Y$, $X - Y$, and $Y - X$, aswell as overflow and zero flags. The design goal is to minimize the number of gatesused; operating speed is not of concern. The circuit is required in several versions thathandle different data word sizes, including 4, 8, and 16 bits. We will assume that wehave standard gate-level and 4-bit register-level components available as buildingblocks.

The lowest cost adders employ ripple-carry propagation and can easily provideaccess to the internal signals needed by the flags. Recall that overflow detection usesc_{n_2}, the input carry to the sign position. Zero detection requires access to all the sumoutputs and poses no special problems. Figure 4.9a shows die logic diagram of anappropriate 4-bit ripple-carry adder. The overflow flag is defined by Equation (4.6) as$v = c_3 \& c_2$ and is realized here by an XOR gate. The zero flag is defined by $z = $

232

SECTION 4.1

Fixed-Point

Arithmetic

Over- Zero

flow v z3 z z2

1-bit

adder

1-bitadder

4-bit «adder

1-bitadder

1-bitadder

c3 >3 *2 >'2

ci y\

xo >o

(a)
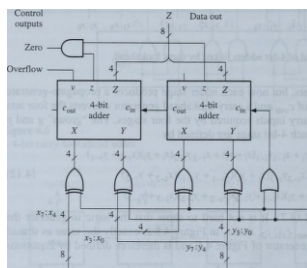
Controlinputs

COMPXCOMPY

Y Data in

(b)

Figure 4.9

Low-cost addition and subtraction of twos-complement numbers: (a) 4-bit adder moduleand (b) 8-bit adder-subtracter.

$z_3 + z_2 + z_j + Z_q$ ^d implemented by a NOR gate. We can use k copies of this adder toproduce a 4/c-bit ripple-carry adder in the usual way. The overflow flag for the entirecircuit is taken from the v output of the left-most (most significant) stage, while the zoutputs of all the stages are ANDed to produce the zero flag.

To extend the adder to an adder-subtracter, the design of Figure 4.4 is a good start-ing point. It uses an XOR word gate to complement the X input, thereby enabling thecircuit to compute $X + Y$ and $Y- X$. To implement the third operation $X - Y$, we could

insert a two-way 4-bit multiplexer into each of the data-in buses so that both X and Y 233

can be applied to each of the adder-subtracter's data inputs. A cheaper solution is to

insert a second XOR word gate into the Ybus, enabling Y to be complemented indepen- CHAPTER 4

dently. We can then compute X - Y in the form X + Y+l. Datapath Design

The complete design of an 8-bit adder-subtracter along the foregoing lines isdepicted in Figure 4.9b. It contains two 4-bit adders of the type in Figure 4.9a linked bytheir carry lines. Two lines COMPX and COMPY control the XOR gates that change Xand Kto X and Y, respectively. The OR gate sets the adder's carry-in line to 1 duringsubtraction. A two-input AND gate combines the two z outputs to produce the zeroflag, which is 1 if and only if the entire 8-bit result Z = 0.

Three of the four signal combinations on COMPX and COMPY control linesimplement the desired three arithmetic functions. The fourth combination 11 imple-ments the sum X + Y + 1, which is an arithmetic function implemented by our designthat has no obvious uses.1 Such superfluous functions are common in the design of dataprocessing circuits.

4.1.2 Multiplication

Fixed-point multiplication requires substantially more hardware than fixed-pointaddition and, as a result, is not included in the instruction sets of some smaller pro-cessors. Multiplication is usually implemented by some form of repeated addition.A simple but slow method to compute X x Y is to add the multiplicand Yto itself Xtimes, where X is the multiplier. (A version of this technique using counters is dis-cussed in problem 2.4.) Often multiplication is implemented by multiplying Y by Xk bits at a time and adding the resulting terms. Figure 4.10 shows this process forunsigned binary numbers in pencil-and-paper calculations with k = 1. The mainoperations involved are shifting and addition. The algorithm of Figure 4.10 is inef-ficient in that the 1-bit products xjZ'Y must be stored until the final addition step iscompleted. In machine implementations it is desirable to add each $jCy2'F$term as it isgenerated to the sum of the preceding terms to form a number Pi+, called a partialproduct. Figure 4.11 shows the calculation in Figure 4.10 implemented in this way.The computation involved in processing one multiplier bit Xj can be described by aregister-transfer statement of the form

PM :=Pl + Xj2'Y (4.13)

1010 Multiplicand Y

1101 Multiplier X = xix2xlx0

1010 x0Y

0000 x{lY

1010 x222Y

1010 x^Y 3 Figure 4.10

10000010 Product P = Z,Xj2YJ Typical pencil-and-paper method for

, = o

multiplication of unsigned binary numbers.

'On the other hand, it has been observed, that "there is no feature of a machine, however pathological, whichcannot be exploited by a programmer." (Kampe I960].

234	1010	Multiplicand YMultiplier X = x}x2xlx0
SECTION 4.1	1101	
Fixed-Point	00000000	P0 = 0
Arithmetic	1010	x0Y
	00001010	P, = P0 + x0Y
	0000	x{lY
	00001010	P2 = Px+x{2Y
	1010	x222YJ
	00110010	P} = P2 + x222Y
	1010	x323Y
	10000010	P4 = P3 + xg?Y-P

Figure 4.11

The multiplication of Figure 4.10 modified formachine implementation.

where 2'Y is equivalent to Y shifted /' positions to the left. In the version of this mul-tiplication algorithm presented in Example 2.7 (section 2.2.3), P, is shifted rightwith respect to a fixed multiplicand Y so that (4.13) is replaced by the equivalenttwo operations

/>,:=/>, + */; PM*Z*ti (4.14)

The multiplication of sign-magnitude numbers requires a straightforwardextension of the unsigned case discussed above. The magnitude part of the productP = X X Y is computed by the unsigned shift-and-add multiplication algorithm, andthe sign ps of P is computed separately from the signs of X and Y as follows: ps :=xs ⊗ ys. The implementation of sign-magnitude multiplication using this sequentialmethod is covered in Example 2.7.

Twos-complement multipliers. The multiplication of twos-complement num-bers presents some difficulties in the case of negative operands. For example, whena negative P, is right-shifted as in (4.14), leading Is rather than leading 0s must beintroduced at the left end of the number. More seriously, the multiplication processmust treat positive and negative operands differently.

A conceptually simple approach to twos-complement multiplication is tonegate all negative operands at the beginning, perform unsigned multiplication onthe resulting (positive) numbers, and then negate the result if necessary. Twos-complement negation for an integer X = xn_lxn_2xn_3...x]x0 is specified by

-X= i„_1x„_2Jc„_3...x1x0 + 000...01 (modulo2") (4.15)

and can easily be implemented by an adder and an EXCLUSIVE-OR word gate, asshown in Figure 4.4. However, up to four extra clock cycles are needed to negate Xand Y and the double-length product P. Several faster schemes have been proposedto handle negative operands. Since these hinge on certain properties of the twos-complement representation, we consider the latter first.

Clearly x, = 1 - jc, (modulo 2), so we can rewrite (4.15) as follows:

(modulo 2") (4.16)

-X = 111... 11

-2An

.3...jc1jc0 + 000...01

Since $2^n = 111...11 + 000...01$, this equation is equivalent to $-X = 2^n - X$, which, incidentally, indicates the origin of the term twos-complement. Now if X is positive

$(x_{n\_1} = 0)$, we can express its value as 235

n-2

x= £2'*, (4-17)

(= 0

If X is negative $(x_{n\_x} = 1)$, then (4.17) does not hold. However, we can rewrite (4.16) as

-X= ll\...ll-(0x„_2xn_y..x:x0+ 100...00) + 000...01

= 2*-l-xn_2xn_3...xlx0 (4.18)

because 2"1 = 111... 11 — 100.. .00 + 000.. .0. Hence for negative X,

X = -2 + x„_2x„_3 ...XxXq

n-2

= -2"-'+ X2'*« (4-19)

i = 0

Finally, we combine (4.17) and (4.19) into a single formula

n-2

Z = -2""Vi+ X2'*, (4-20)

i = 0

which is valid for both positive and negative n-bit integers. For example, suppose that n = 6 and X = 101101. Evaluating X according to (4.20) yields

X = -25 x 1 + 24 xO + 23 x 1 + 22 x 1 + 21 xO + 2° x 1

= -32 + 8 + 4+ 1 =-19

Equation (4.20) implies that we can treat bits xn_2xn_3...x1x0 of a negative twos-complement integer in the same way as the corresponding (magnitude) bits of a positive number; each bit xt has the positive weight 2'. Weight +2""1 is assigned to the sign bit xn_x of a positive number; however, since x„_, = 0, its contribution to the number is zero. In the negative case, the sign xn_x is assigned weight -2"~'; this adds -2"~l to the number, ensuring that it is negative.

If X = xn_lxn_2...xlx0 is a twos-complement fraction instead of an integer, then the negation formula (4.15) remains valid, but because bit /" now has weight 2'""+1 instead of 2', Equation (4.20) is replaced by

n-2

* = -2Vi+ X2"" + \ (4-2D

; = 0

In effect we have multiplied (4.20) by the scaling factor 2"*"-1'. For example, let n - 4 and X = 1011, which represents the fraction -0.62510. Application of (4.21) yields

X = -2°X1 +2_1 X0 + 2_2X1 +2'3Xl

= -1.000 + 0.250 + 0.125 = -0.625

Suppose that X is the multiplier operand in a shift-and-add multiplication algo-rithm to compute P = X x Y for twos-complement numbers. Equations (4.20) and

CHAPTER 4 Datapath Design

236

SECTION 4.1

Fixed-Point

Arithmetic

Sign

logic

Accumulator

Multiplier register

Q[0]

Paralleladder

f

Pis-Pi

Data out

M[7] Multiplicand register

M

<£

Controlunit

PYPoData out

X YData in

Figure 4.12

The datapath of the twos-complement multiplier.

(4.21) suggest that we can use an unsigned multiplication technique like those illustrated in Figures 4.12 and 4.13 with one change: When multiplying by the sign bit, perform subtraction rather than addition in the final step if a minus sign xn_x = 1 is encountered. This observation is the basis of a twos-complement multiplication algorithm developed by James E. Robertson, which has been widely used in com-puter design [Robertson 1955; Cavanagh 1984]. We now show one way to adapt the circuit developed in Example 2.7 for sign-magnitude multiplication to deal with the twos-complement case.

2Cmultiplier   (in: INBUS; out: OUTBUS):

register A[7:0], M[7:0], Q[7:0], COUNT[2:0], F;

```
            bus INBUS[7:0], OUTBUS[7:0]:

BEGIN:        A := 0. COUNT := 0, F := 0.

INPUT:        M:= INBUS;

              Q:= INBUS;

ADD:          A[7:0] := A[7:0] + M[7:0] x Q[0],

              F :=(M[7] and Q[0]) or F;

RSHIFT:       A[7] := F, A[6:0].Q := A.Q[7:1], COUNT := COUNT + 1;

TEST:         if COUNT * 7 then go to ADD;

SUBTRACT:     A[7:0] := A[7:0] - M[7:0] x Q[0], Q[0] := 0;

OUTPUT:       OUTBUS := Q;

              OUTBUS := A;

end 2Cmultipl er;
```

Figure 4.13

HDL description of the multiplier for 8-bit twos-complement fractions.

EXAMPLE 4.2 DESIGN OF A MULTIPLIER FOR TWOS-COMPLEMENT

fractions. Consider again the task of multiplying two 8-bit binary fractions X =x1x(/x5x4xyx2xlx0 and Y = Jvy^iV^^vv'^Vo t0 forrn tne product P = YxX, this time usingtwos-complement code. (Example 2.7 analyzed this problem for the sign-magnitudecase.) Assume that the multiplier will have a register-level structure similar to that inFigure 2.41, with registers A, M, and Q storing the various operands and A.Q forminga right-shift register. Since sign bits will be included in additions and subtractions, weneed an 8-bit adder-subtracter, rather than the 7-bit magnitude-only adder used in theearlier design. Figure 4.12 shows the datapath of the proposed design at the registerlevel.

To develop the required twos-complement multiplication algorithm for thismachine, we consider the four possible cases determined by the signs of X and Y.

1. x-j = y7 = 0; that is, both X and Y are positive. The computation in this case is effec-tively unsigned multiplication with the product P computed in a series of add-and-shift steps of the form

P. := P, + x,Y:

Vi

All partial products P{ are nonnegative, so leading 0s are introduced into A duringthe right-shift operation indicated by the factor 2"1.

2. x7 = 0, y-j = 1; that is, X is positive and Y is negative. The partial product />,- will bezero, and leading 0s should be shifted into A as before, until the first 1 in X isencountered. Multiplication of Y by this 1 and addition of the result to A causes />,to become negative, from which point on leading Is rather than 0s must be shiftedinto A. These rules ensure that a right shift corresponds to division by 2 in twos-complement code.

3. x7 = 1, v7 = 0; that is, X is negative and Y is positive. This follows case 1 for the firstseven add-and-shift steps yielding the partial product

237

CHAPTER 4Datapath Design

For the final step, often referred to as a correction step, the subtraction P := P7 ■performed. The result P is then given by

6 / 6 \

P = -Y+ X2"V = U+ E2'-7JcIJy

yis

which is XxYby (4.21).4. x1 = y1= 1; that is, both X and Kare negative. The procedure used here follows case2, with leading 0s (Is) being introduced into the accumulator whenever its contentsare zero (negative). The correction (subtraction) step of case 3 is also performed,which ensures that the final product in A.Q is nonnegative.

Each addition/subtraction step can be performed in the usual twos-complementfashion by treating the sign bits like any other and ignoring overflow. Care is needed inthe shift step to ensure that the correct new value is placed in the accumulator's signposition A[7]. This value must be a leading 0 if the current partial product in A.Q ispositive or zero, and 1 if it is negative. We introduce a flip-flop F to control the valuesassigned to A[7]. F is initially set to 0, and is subsequently defined by

F := (v7an</-t,-)0r F

Z38		Step	Action	F	Accumulator A	Register Q
SECTION 4.1			Initialize registers 0 00000000			
		0				10110011 = multiplier X
Fixed-Point			11010101			
		1				= multiplicand Y= M10110011
Arithmetic			Add M to A 1 11010101			
			Right-shift F.A.Q		11101010	11011001
		2			11010101	
			Add M to A		[10111111	U011001
			Right-shift F.A.Q		[11011111	11101100
		3			00000000	
			Add zero to A		I 11011111	lnonoo

	Right-shift F.A.Q	11101111	11110110
4		00000000	
	Add zero to A	11101111	nnono
	Right-shift F.A.Q	I 11110111	11111011
5		11010101	
	Add M to A]	[11001100	union
	Right-shift F.A.Q	11100110	oinnoi
6		11010101	
	Add M to A]	10111011	oninoi
	Right-shift F.A.Q	11011101	101 nno
7		00000000	
	Add zero to A	11011101	10111110
	Right-shift F.A.Q	11101110	11011111
8		11010101	
	Subtract M from A	00011001	11011111
	Set Q[0] to 0	00011001	11011110 = product P

Figure 4.14

Illustration of the Robertson multiplication algorithm for twos-complement fractions.

Here y_7 is the sign of the multiplicand stored in M[7], and x, is the current multiplier bit being tested in Q[0]. Thus F is set to 1 if Y is negative and at least one nonzero x_t is encountered. Once set to 1, it remains at that value. A negative Y and a positive or neg-ative X therefore produce a series of negative partial products. This situation is to be expected, since bits x6:x0 of the multiplier X are always treated as if they were positive. A positive Y, or X = 0, causes F to remain permanently at 0. Note that the sign p_{i5} of the product P requires no separate computational step. As in Example 2.7, the least signifi-cant bit p0 of P is set to 0 to make the result exactly 16 bits long.

Figure 4.13 presents an HDL description of the twos-complement multiplication algorithm, which summarizes the foregoing analysis; compare the corresponding sign-magnitude algorithm in Figure 2.39. An application of the present algorithm to the case X = 10110011 and Y = 11010101 appears in Figure 4.14. The sign bit x7 of the multi-plier X is underlined to show its passage through Q. Observe how F becomes 1 in step 1, when the negative multiplicand is first added to the accumulator. F continues to sup-ply leading Is to the A register until step 8. Then because Q[7] = x_n = 1, a subtraction is performed that produces the proper sign p_{i5} = 0 in A(0). Setting Q[0] = p0 to 0 com-pletes the multiplication process.

Booth's algorithm. Another interesting and widely used scheme for twos-complement multiplication was proposed by Andrew D. Booth in the 1950s

[Booth 1951]. Like Robertson's method in Example 4.2, Booth's algorithm 239 employs both addition and subtraction, but it treats positive and negative operands uniformly —no special actions are required for negative numbers. Booth's algo-rithm can also be readily extended in various ways to speed up the multiplication process; see problems 4.16 and 4.17. A version of this algorithm implements the ARM6's multiply instruction.

The multiplication algorithms we have considered so far involve scanning the multiplier X from right to left and using the value of the current multiplier bit x_i to determine which of the following operations to perform: add the multiplicand Y, subtract Y, or add zero, that is, no operation. In Booth's approach two adjacent bits $x_i x_{i\_1}$ are examined in each step. If $x_p x_{j\_1}$ = 01, then Y is added to the current partial product P_j, while if $x-x^\wedge$ = 10, Y is subtracted from Pt. If $x-x^\wedge$ = 00 or 11, then nei-ther addition or subtraction is performed; only the subsequent right shift off, takes place. Thus Booth's algorithm effectively skips over runs of Is and runs of 0s that it encounters in X. This skipping reduces the average number of add-subtract steps and allows faster multipliers to be designed, although at the expense of more com-plex timing and control circuitry.

The validity of Booth's method can be seen as follows. Suppose that X is a positive integer and contains a subsequence X* consisting of a run of k 1 s flanked by two 0s.

$X^* = x_i x_{i\_l} x_{i\_2} \dots x_{j\_k+l} x_i x_{i\_k\_l}$

$= 0 1 1 \dots 1 1 0$

In a direct add-and-shift multiplication algorithm such as Robertson's, Y is multi-plied by each bit of X* in sequence and the results are summed so that X*'s contri-bution to the product P = X x Y is

/-I

$^\wedge y$ (4.22)

j = i-k

Now when Booth's algorithm is applied to X*, it performs an addition when it encounters $x_{j} x_{j\_1}$ = 01, which contributes 2^iY to P. It performs a subtraction at $x_{i\_k} x_{i\_k\_1}$ = $1^\wedge$ which contributes -2^{i_k}Y to P. Thus the net contribution of X* to the product P in this case is

$2^i Y - 2^{i\_k} Y = 2^{i\_k}(2^k\-\)Y$

/,-1

$= 2^{i\}-^* £2^m r$

m = 0

k- 1 m = 0

$2^m + ^\._kY$ (4.23)

Suppose the index m is replaced by j = m + i - k. Then the upper and lower limits of the summation in (4.23) change from k - 1 and 0 to / - 1 and /' - k, respectively, implying that (4.22) and (4.23) are, in fact, the same. It follows that Booth's algo-rithm correctly computes the contribution of X*, and hence of the entire multiplier X, to the product P. Equation (4.20) implies that the contribution of a negative X*

CHAPTER 4 Datapath Design

(in: INBUS; out: OUTBUS);

SECTION 4.1

Fixed-Point		bus INBUS[7:0], OUTBUS[7:0];
Arithmetic	BEGIN:	A := 0, COUNT := 0,
	INPUT:	M := INBUS; Q[7:0]:= INBUS, Q[-1]:=0;
	SCAN:	if Q[l] Q[0] = 01 then A[7:0] := A[7:0] + M[7:0], go toTEST; else if Q[l] Q[0] = 10 then A[7:0] := A[7:0] - M[7:0];
	TEST:	if COUNT = 7 then go to OUTPUT,
	RSHIFT:	A[7] := A[7], A[6:0].Q = A.Q[7:0],
	INCREMENT:	COUNT := COUNT + 1, go to SCAN;
	OUTPUT:	OUTBUS := A, Q[0] := 0;OUTBUS :=Q[7:0];
		end BoothMult;

Figure 4.15

HDL description of an 8-bit multiplier implementing the basicBooth algorithm.

to P can also be expressed in the formats of (4.20) and (4.23); a similar argumentdemonstrates the correctness of the algorithm for negative multipliers. The argu-ment for fractions is essentially the same as that for integers.

The twos-complement multiplication circuit of Figure 4.12 can easily be mod-ified to implement Booth's algorithm. Figure 4.15 describes a straightforwardimplementation of the Booth algorithm using the above approach with n = 8 and acircuit based on Figure 4.12. An extra flip-flop Q[-l] is appended to the right endof the multiplier register Q, and the sign logic for A is reduced to the simple signextension A[7] := A[7]. In each step the two adjacent bits Q[0]Q[-1] of Q areexamined, instead of Q[0] alone as in Robertson's algorithm, to decide the opera-tion (add Y, subtract Y, or no operation) to be performed in that step. For compari-son with Robertson's method in Figure 4.13, the operands are assumed to befractions. The application of this algorithm to the example solved by Robertson'smethod in Figure 4.14 appears in Figure 4.16. where the bits stored in Q[0]Q[-1] ineach step are underlined.

Combinational array multipliers. Advances in VLSI technology have made itpossible to build combinational circuits that perform n x H-bit multiplication forfairly large values of n. An example is the Integrated Device TechnologyIDT721CL multiplier chip, which can multiply two 16-bit numbers in 16 ns [Inte-grated Device Technology 1995]. These multipliers resemble the «-step sequentialmultipliers discussed above but have roughly n times more logic to allow the prod-uct to be computed in one step instead of in n steps. They are composed of arraysof simple combinational elements, each of which implements an add/subtract-and-shift operation for small slices of the multiplication operands.

Suppose that two binary numbers X = xn_]xn_2...xlx0 and Y = y„_iy„_2---)'i)'oare to be multiplied. For simplicity, assume that X and Yare unsigned integers. Theproduct P = X X Ycan therefore be expressed as

Step	Action	Accumulator	Register Q
0	Initialize registers	00000000	10110011 = multiplier X
	SetQ[-l]toO	00000000	101100110
1		11010101	= mulitplicand Y= M
	Subtract M from A	00101011	1011001K)
	Right-shift A.Q	00010101	110110011
2	Skip add/subtract	00010101	1101100U
	Right-shift A.Q	00001010	111011001
3		11010101	
	Add M to A	11011111	11101100J.
	Right-shift A.Q	11101111	111101100
4	Skip add/subtract	11101111	111101100
	Right-shift A.Q	11110111	111110110
5		11010101	
	Subtract M from A	00100010	111110110
	Right-shift A.Q	00010001	011111011
6	Skip add/subtract	00010001	oi ii non
	Right-shift A.Q	00001000	101111101

7		11010101	
	Add M to A	11011101	loiimpi
	Right-shift A.Q	11101110	110111110
8		11010101	
	Subtract M from A	00011001	110111110
	Set Q[0] to 0	00011001	110111100 = product/3

CHAPTER 4 Datapath Design

Figure 4.16

Illustration of the Booth multiplication algorithm.

$$p = X2V$$

(4.24)

$$= o$$

corresponding to the bit-by-bit multiplication style of Figure 4.10. Now (4.24) can be rewritten as

$$P = 12'$$

$$X * ^2"'$$

$$; = 0$$

(4.25)

Each of the $n2$ 1-bit product terms $x \cdot y_i$ appearing in (4.25) can be computed by a two-input AND gate—observe that the arithmetic and logical products coincide in the 1-bit case. Hence an $n \times n$ array of two-input ANDs of the type shown in Figure 4.17 can compute all the x^j terms simultaneously. The terms are summed according to (4.25) by an array of $n(n - 1)$ 1-bit full adders as shown in Figure 4.18; this circuit is a kind of two-dimensional ripple-carry adder. The shifts implied by the $2'$ and $2j$ factors in (4.25) are implemented by the spatial dis-placement of the adders along the x and y dimensions. Note the similarities between the circuit of Figure 4.17 and the multiplication examples of Figures 4.10 and 4.11.

SECTION 4.1

Fixed-Point

Arithmetic

The AND and add functions of the array multiplier can be combined into a sin-gle component (cell) as illustrated in Figure 4.19. This cell realizes the arithmetic expression

$$jrs = a \text{ plus } b \text{ plus } xy$$

(4.26)

An $n \times rt$-bit multiplier can be built using n copies of this cell as the sole compo-nent, although, as in Figure 4.18, some cells on the periphery of the array have inputs set to 0 or 1, effectively reducing their operation from (4.26) a plus b plus xy to a plus b (a half adder). The multiplication time for this multiplier is determined by the worst-case carry propagation and, ignoring the differences between the internal and peripheral cells, is $\{In -\}D$, where D is the delay of the basic cell.

Multiplication algorithms for twos-complement numbers, such as Robertson's and Booth's, can also be realized by arrays of combinational cells as the next example shows.

EXAMPLE 4.3 ARRAY IMPLEMENTATION OF THE BOOTH MULTIPLICA-TION algorithm [KOREN 1993]. Implementing the Booth method by a combi-national array requires a multifunction cell capable of addition, subtraction, and no operation (skip). Such a cell B is shown in Figure 4.20a. Its various functions are selected by a pair of control lines H and D as indicated. It is easily seen that the required functions of B are defined by the following logic equations.

$$Z = a \odot bH \circledR cH$$

$$c0M = (a \oplus D)(b + c) + bc$$

When HD = 10, these equations reduce to die usual full-adder equations (4.1); when HD =11, they reduce to the corresponding full-subtracter equations

$$z = a \oplus b \oplus c$$

$$c0_{,,} = ab + ac + be$$
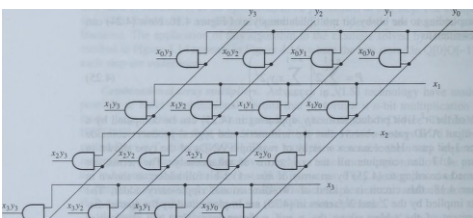
Figure 4.17

AND array for 4 x 4-bit unsigned multiplication.

carryout

Wo
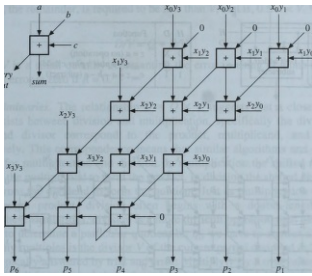
Po

Figure 4.18

Full-adder array for 4 x 4-bit unsigned multiplication.
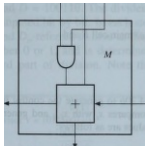
Figure 4.19

Cell M for an unsigned array multiplier.

in which c and cout assume the roles of borrow-in and borrow-out, respectively. When $H = 0$, z becomes a, and the carry lines play no role in the final result.

An H-bit multiplier is constructed from $n^2 + n(n - 1)/2$ copies of the B cell con-nected as shown in Figure 4.20b. The extra cells at the top left change the array's shape from the parallelogram of Figure 4.18 to a trapezium and are employed to sign-extend the multiplicand Y for addition and subtraction. Note how the diagonal lines marked b deliver the sign-extended Y directly to every row of B cells. When Y is positive, it is sign-extended by leading 0s; this is implicit in the array of Figure 4.18. In the present case, when Kis negative, it must be explicitly sign-extended by leading Is.

The operation to be performed by each row ; of B cells is decided by bits x_ix_j of the operand X. To allow each possible ^rv,_, pair to control row operations, we

1 -bit adder/subtracter

H	D	Function
0	X	$z = a$ (no operation)
1	0	coulz = apluf bplus c (add)
1	1	coutz = a - b- c (subtract)

(a)

ZJLZJU'JLVJL^.L/JV-

- s

B ■*-- B

B *

B *

V HT I ^

l-o

J-o

J-o

1-0

5?

Pe

Ps

/>4

Pi Pi

(b)

P\

Po

Figure 4.20

Combinational array implementing Booth's algorithm: (a) main cell B and (b) array multiplier for 4 x 4-bit numbers.

introduce a second cell type denoted C in Figure 4.20b to generate the control input signal H and D required by the B cells. Cell C compares j_c, with $x_j\_]$ and generates the values of HD required by Figure 4.20a; these values are as follows:

4.1.3 Division

In fixed-point division two numbers, a divisor V and a dividend D, are given. The object is to compute a third number Q, the quotient, such that Q X V equals or is very close to D. For example, if unsigned integer formats are being used, Q is com-puted so that

$$D = Q X V + R$$

where R, the remainder, is required to be less than V, that is, $0 < R < V$. We can then 245

Writ£ ^TER4

D/V =Q + R/V (4.27) Datapath Design

Here R/V is a small quantity representing the error in using Q alone to represent D/V; this error is zero if R = 0.

Preliminaries. The relationship $D \sim Q X V$ suggests that a close correspon-dence exists between division and multiplication, specifically the dividend, quo-tient, and divisor correspond to the product, multiplicand, and multiplier, respectively. This correspondence means that similar algorithms and circuits can be used for multiplication and division. In multiplication the shifted multiplier is added to the multiplicand to form the product. In division the shifted divisor is sub-tracted from the dividend to form the quotient. Just as multiplication ends with a double-length product, division often begins with a double-length dividend. Despite these similarities, division is a more difficult operation than multiplication because to determine a particular quotient bit q_i, we have to answer the question: How many multiples is the divisor V of the current partial dividend D(? This ques-tion is typically answered by trial and error: Multiply V by a trial value for q_i sub-tract the result from D,, and check the value of the remainder. Note too that the next quotient bit q_{l+x} cannot be determined until q_i is known. Thus division has an ele-ment of uncertainty not found in multiplication.

One of the simpler binary division methods is a sequential digit-by-digit algorithm similar to that used in pencil-and-paper methods with decimal numbers. Figure 4.21 illustrates this approach for a 3-bit divisor V = 101 and a 6-bit divi-dend D = 100110. The dividend is scanned from left to right, and the quotient is computed bit by bit. In each step divisor V is compared to the current partial divi-dend D_j, referred to here as the partial remainder R,2 The current quotient bit q_t is either 0 or 1, and is determined by comparing V with /?,; this comparison is the hard part of division. Note that decimal division is harder than binary in this

(Jill	Quotient Q = qrftfrfo
Divisor V= 101 100110	Dividend D = R0
000	«iV
100110	R.
101	q22-W
10010	«2
101	<7,2-2V
1000	«3
101	<7o2-3V
011	R4 = remainder R

Figure 4.21

Typical pencil-and-paper method for division of unsigned numbers.

2We use the terms partial dividend and partial remainder interchangeably because the remainder from step i is used as the dividend in step r + 1.

SECTION 4.1

Fixed-Point

Arithmetic

regard because q_i must be selected from 10 possible digit values instead of from two. If the numbers appearing in the division calculation of Figure 4.21 are unsigned binary integers of length six, then (4.27) becomes

100110. / 000101. = 000111. + 000011./000101.

<

corresponding to the decimal division 38/5 = 7 + 3/5. If the numbers are unsigned 6-bit fractions, then Figure 4.21 is interpreted as

.100110/. 101000 = .111000+ .000011/. 101000

corresponding to .59375A625 = .875 + .046875/.625.

In integer arithmetic Q and R are always integers of the standard word size. If fraction formats are used, however, the number of bits of Q is not necessarily bounded. For example, .2OOO/.30O0 = .66666..., a repeating fraction. It is neces-sary, therefore, to limit the number of quotient bits generated by the division pro-cess. Division of .2000 by .3000 might be required to yield a four-digit quotient Q with truncation or rounding determining the final digit of Q. Several other difficul-ties occur in division. If D is too large relative to V, then Q will not fit in the stan-dard word size, resulting in quotient overflow. For instance, the four-digit fraction division .2000/.0100 produces a nonfraction six-digit result 20.0000. When V = 0, the quotient Q is treated as undefined or infinity and a divide-by-zero error is said to occur. Special circuits are employed to check for, and flag, quotient overflow and zero divisors before division begins.

Basic algorithms. Suppose that the divisor V and dividend D are unsigned integers and the quotient $Q = q_n\_{x}q_{n-2}q\_{n-y}$.. is to be computed one bit at a time. At each step i, $2^{-i}V$, which represents the divisor shifted i bits to the right, is compared with the current partial remainder $/?$,-. The quotient bit q_i is set to 1 (0) if $2^{-i}V$ is less (greater) than $/?$,-, and a new partial remainder R_{i+}, is computed according to the relation

$$*_{,+1} := /?_{,}-4,2^{-i}V$$

(4.28)

In machine implementations it is more convenient to shift the partial remainder to the left relative to a fixed divisor, in which case (4.28) is replaced by

$$R_{i+l} := 2R_l - q_i V$$

Figure 4.22 shows the calculation of Figure 4.21 modified in this way. The final partial remainder R4 is now the overall remainder R shifted three bits to the left, so that $R = 2^{-3}R4$.

As observed above, the central problem in division is finding the quotient digit q_i. If radix-r numbers are being represented, then q_i must be chosen from among r possible values. When r = 2, q_i can be generated by comparing V and $2/?$, in the rth step, as is done implicitly in Figure 4.22. If $V > 2/?$,, then $q_i = 0$; otherwise, $q_i = 1$. If V is long, a combinational magnitude comparator circuit may be impractical, in which case q_i is usually determined by subtracting V from $2R_t$ and examining the sign of $2R_j - V$. If $2/?$, - V is negative, $q_i = 0$; otherwise, $q_i = 1$.

The circuit used for multiplication in Example 4.2 (Figure 4.12) is easily mod-ified to perform division, as shown in Figure 4.23. The 2«-bit shift register A.Qstores the partial remainders. Initially the dividend (which can contain up to 2n

Divisor V   Quotient Q

Dividend D =<73V

*i

2/?,

101   100110000   q2v   --2R0 0

R22R2

*3

2fl3q0VR4 = 23/?

1001101001100101 01

1001001001000101 Oil

1000001000000101 0111

011000
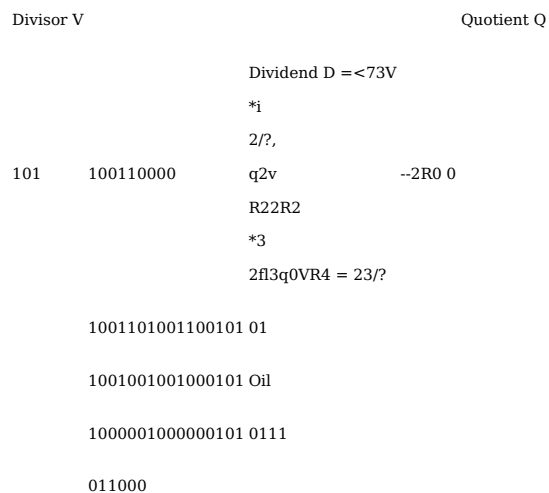
CHAPTER 4Datapath Design

Figure 4.22

The division of Figure 4.21 modified for machine implementation.

bits) is placed in A.Q. The divisor V is placed in the M register where it remainsthroughout the division process. In each step A.Q is shifted to the left. The posi-tions vacated at the right-most end of the Q register can be used to store the quo-tient bits as they are generated. When the division process terminates, Q containsthe quotient, while A contains the (shifted) remainder.

As noted already, the quotient bit qi can be determined by a trial subtraction ofthe form 2Ri - V. This subtraction also yields the new partial remainder Ri+1 when27?, - V is positive; that is, when qi■,= 1. Clearly, the process of determining q{ andRi+l can be integrated. Two major division algorithms are distinguished by the waythey combine the computation of q(and Ri+l. If q{ = 0, then the result of the trial

Accumulator

Quotient (multiplier)register

Divisor (multiplicand)register

A *— Q   M

 i   n x   A'i i i

    n   ' n'

 '   '   r i

     '   r

    Paralleladder-subtracter

n.   In- '

  • Controlunit

     n   n'

                      '

 '   '   '
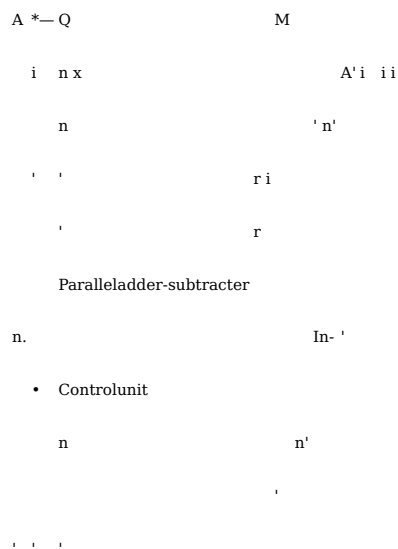
Remainder R Quotient Q

Figure 4.23

The datapath of a sequential n-bit binary divider.

248

SECTION 4.1Fixed-PointArithmetic

subtraction is 2Rt - V; however, the required new partial remainder Rl+, is 2/?,-. Thepartial remainder Ri+l can be obtained by adding V back to the result of the trialsubtraction. This straightforward technique is called restoring division. In everystep the operation

/?,+1:=2/?,-V (4.29)

is performed. When the result of the subtraction is negative, a restoring addition isperformed as follows:

R

i+i

:=tfl+l + V

If the probability of qt = 1 is 1/2, then this algorithm requires n subtractions and anaverage of nil additions.

The restoration step of the preceding algorithm is eliminated in a slightly dif-ferent technique called nonrestoring division. This method is based on the observa-tion that a

restoration of the form

/?, := R; + V

(4.30)

is followed in the next step by the subtraction (4.29). Operations (4.29) and (4.30)can be merged into the single operation

Ri+l:=2R,+ V

(4.31)

Thus when qi = 1, which is indicated by a positive value of Rt, /?/+1 is computedusing (4.29). When qt = 0, Ri+l is computed using (4.31). The calculation of eachquotient bit involves either an addition or a subtraction, but not both. Nonrestoringdivision therefore requires n additions or subtractions, whereas restoring divisionrequires an average of 3«/2 additions and subtractions.

Figure 4.24 presents a nonrestoring division algorithm designed for the circuitof Figure 4.23 with unsigned integers. The divisor V and quotient Q are n bits long(with leading 0s if necessary), while the dividend D is up to In - 1 bits long, whichis the maximum length of the product of two «-bit integers. The flip-flop S isappended to the accumulator A to record the sign of the result of an addition orsubtraction and to determine the quotient bit. Each new quotient bit is placed inQ[0], and the final values of the quotient Q and the remainder R are in the Q and Aregisters, respectively. An application of this algorithm when n = 4 appears in Fig-ure 4.25 with D = 11000012 = 9710 and V= 10102 = 1010.

The restoring and nonrestoring division techniques can be extended to signednumbers in much the same way as multiplication. Sign-magnitude numbers presentfew difficulties; the magnitudes of the quotient and remainder can be computed asin the unsigned number case, while their signs are determined separately. Asremarked in [Cavanagh 1984], there are no simple division algorithms for handlingnegative numbers directly in twos-complement code because of the difficulty ofselecting the quotient bits so that the quotient has the correct positive or negativerepresentation. The most direct approach to signed division is to negate any nega-tive operands, perform division on the resulting positive numbers, and then negatethe results, as needed. A fast division algorithm for twos-complement numbersbased on the nonrestoring approach was devised independently in 1958 by DuraW. Sweeney, James E. Robertson, and Keith D. Tocher and is called the SRTmethod in their honor; see [Cavanagh 1984; Koren 1993] for details.

NRdivider

BEGIN:INPUT:

SUBTRACT.TEST:

CORRECTION:OUTPUT:

end NRdivider;

(in: INBUS; out: OUTBUS);

register S, A[n-1:0], M[n-1:0], Q[n-1:0], COUNT[Tlog2nl:0];

bus INBUS[/i-l:0]. OUTBUS[n-1:0];

COUNT := 0, S := 0,

A := INBUS; {Input the left half of the dividend D)

Q := INBUS; {Input the right half of the dividend D)

M := INBUS; {Input the divisor V}

S.A := S.A-M; {S is the sign of the result}

if S = 0 then

begin Q[0] := 1;

if COUNT = n - 1 then go to CORRECTION; else

begin COUNT := COUNT + 1, S.A.Q[/i-l:l] := A.Q; end

S.A := S.A - M, go to TEST; endelse {if S = l}

begin Q[0] := 0;

if COUNT = n - 1 then go to CORRECTION; else

begin COUNT := COUNT + 1, S.A.Q[n-l:l] := A.Q; endS.A := S.A + M, go to TEST; endif S = 1 then S.A := S.A + M;OUTBUS := Q; {Output the quotient Q)OUTBUS := A; {Output the remainder/?}

249

CHAPTER 4Datapath Design

Figure 4.24

Nonrestoring division algorithm for unsigned integers.

Step	Action	S A	Q	
0	Initialize registers	0 1100	0010 =	= dividend D
1		1010		= divisor V= M
	Subtract M from A	0 0010	0010	
	Reset Q[0]	0 0010	0011	
	Left shift S.A.Q	0 0100	0110	
2		1010		
	Subtract M from A	1 1010	0110	
	Set Q[0]	1 1010	0110	
	Left shift S.A.Q	1 0100	1100	
3		1010		
	Add M to A	1 1110	1100	

| Set Q[0] | 1 1110 1100 |
| Left shift S.A.Q | 1 1101 1000 |

4   1010

| Add M to A | 0 0111 1000 |
| Reset Q[3] | 0 0111 10011001 = : quotient Q |

0111   ■ remainder R

Figure 4.25

Illustration of the nonrestoring division algorithm for unsigned integers.

250

Borrow out u ■<—Control line a —» D   *— Borrow in t—*■ a

Fixed-Point   a   Function

Arithmetic   01

uz = x minus v minus t

z = x

I   1

Figure 4.26

A cell D for array implementation of restoring division.

Combinational array dividers. Combinational array circuits can be used for division as well as for multiplication. Figure 4.26 shows a cell D suitable for imple-menting a version of the restoring division algorithm. This cell is basically a full subtracter with t and u being the borrow-in and borrow-out bits, respectively. The main output z is controlled by input a. When $a = 1$, z is the difference bit defined by the arithmetic equation

$$z = x \text{ minus } v \text{ minus } t$$

When $a = 0$, $z = x$. Thus the behavior of the cell D is given by the logic equations

$$z = x \oplus a \{ y \oplus t)$$

$$u = xy + xt + yt$$

Figure 4.27 shows an array of D cells to divide 3-bit unsigned integers and gen-erate a 4-bit quotient. Each row of the array subtracts the divisor V from the shifted partial remainder $2R_i$ generated by the row above. The sign of the result, and there-fore of the quotient bit, is indicated by the borrow-out signal from the left-most cell in the row. This signal «, is connected to the control inputs a of all cells in the same row. If $U_j = 0$, then the output from the row is $2/?$, - V and $qt= t_{ij}= 1$. If $u_t = 1$, then the output from the row is restored to $2/?$,-, and again $q_i = U_j$ - 0. Thus the output of each row is initially $2R_i$ - V, but it is restored to $2/?$,- when required. Restoration is achieved by overriding the subtraction performed by the row rather than by explic-itly adding back the divisor.

Let d and d' be the carry (borrow) propagation and restore times of a cell, respectively. Let the divisor and dividend be n bits long. Each row of the divider array functions as an n-bit ripple-borrow subtracter, so the maximum time required to compute one quotient bit is $nd + d'$. The time required to compute an m-bit quo-tient and the corresponding remainder is therefore $m(nd + d')$, and the number of cells needed is $m(n + 1) - 1$.

Division by repeated multiplication. In systems containing a high-speed multi-plier, division can be performed efficiently and at low cost using repeated multipli-cation. In each iteration a factor F, is generated and used to multiply both the divisor V and the dividend D. Therefore

Divisor V Dividend D d5

Remainder R

Figure 4.27

A divider array for 3-bit unsigned numbers using the cell D of Figure 4.26.

$Q =$

DxF0xF,xF2x...VxF0xF: xF2x ...

F, is chosen so that the sequence V X F0X F{ X F2 ... converges rapidly toward one. Hence DX F0X F, X F2 ... must converge toward the desired quotient.

The convergence of the method depends on the selection of the F,'s. For sim-plicity, assume that D and V are positive normalized fractions so that $V = 1 — x$,

251

CHAPTER 4 Datapath Design

VXF0 = (1 -x){\ +x)=\-x2

Clearly V X F0 is closer to one than to V. Next set F, = 1 + x2. Hence

$V \times F0 \times F, = (1 - j^{\wedge}Xl + x2) = 1 - x4$

and so on. Let V, denote $VX F0X F^{\wedge}X \ldots X F$,. The multiplication factor at eachstage is computed as $F, = 2 - V,\_,$, which is simply the twos-complement of $V,\_,$.Hence

$F, = 1 + x2$ and $V, = 1$

$2' +$

As i increases, V, converges quickly toward one. The process terminates when V, $=0.11\ldots 11$, the number closest to one for the given word size.

4.2

ARITHMETIC-LOGIC UNITS

The various circuits used to execute data-processing instructions are usually com-bined in a single circuit called an arithmetic-logic unit or ALU. The complexity ofan ALU is determined by the way in which its arithmetic instructions are realized.Simple ALUs that perform fixed-point addition and subtraction, as well as word-based logical operations, can be realized by combinational circuits. ALUs that alsoperform multiplication and division can be constructed around the circuits devel-oped for these operations in the preceding section. Much more extensive data-processing and control logic is necessary to implement floating-point arithmetic inhardware, as we will see later. Some processors having fixed-point ALUs employspecial-purpose auxiliary units called arithmetic (co)processors to performfloating-point and other complex numerical functions.

4.2.1 Combinational ALUs

The simplest ALUs combine the functions of a twos-complement adder-subtracterwith those of a circuit that generates word-based logic functions of the form $J\backslash X,Y)$,for example, AND, XOR, and NOT. They can thus implement most of a CPU'sfixed-point data-processing instructions. Figure 4.28 outlines an ALU that has sep-arate subunits for logical and arithmetic operations. The particular class of opera-tion (logical and arithmetic) to be performed is determined by a "mode" controlline M attached to a two-way multiplexer that channels the required result to the

x —*—t *■

y —^-r

Data

n-bitlogicunit

n -bitadder-subtracter

Two-wayn -bit

multiplexer

,' > Z Data out

kr, „ Flags (cout, p, g,overflow, etc.)

Select 5 Carry in cin

Figure 4.28

A basic n-bit arithmetic-logic unit (ALU).

Mode M

output bus Z The specific operation performed by the desired subunit is deter-mined by a "select" control line S as shown. The ALU's logical operations are per-formed bitwise; that is, the same operation / is applied to every pair of data lines$x^{\wedge}j$. The maximum number of distinct logical operations of the form$/(*,,\} \bullet,)$ is 16,which is the number of distinct truth tables of two Boolean variables. Hence theselect bus S needs to be of size 4 at most, as in Figure 4.28. 5 can also be used toselect up to 16 different arithmetic operations such as $X + Y, X - Y, Y - X, X + 1$(increment), X- 1 (decrement), and so on, as needed.

The logical operations in Figure 4.28 can be obtained by generating all fourminterms of$/(*,,)',)$, namely,

$m3 = xy\{ \ m2 = xft \ m, = *,y,- \ m0 = y\{y.f$

for every pair $x\$t$ of data bits and by using the control lines $S = S3S-,SXS0$ to selectdesired subsets of the minterms to be ORed together. In particular, if we constructthe sum-of-products expression

$f(xity,) = m3S3 + m2S2 + mlSl + m^S0$

(4.32)

then we see that every combination of $S^{\wedge}iS^{\wedge}q$ produces a different function. Forexample, $5 = 0110$ makes$/(x,,y,) = x(y(+ x-y\{$, which is EXCLUSIVE-OR. Becauseof the bitwise nature of the logic operations, we can replace jc(and y, in (4.32) withthe n-bit words X and Y.

$f(X,Y) = XYS3 + XYS2 + XYSX + XYS0$

(4.33)

We can now implement the logic unit directly from Equation (4.33), using severalH-bit word gates as in Figure 4.29. The adder-subtracter can be designed by any ofthe techniques presented earlier, with appropriate additional connections to X, Y,and 5.

Despite its conceptual simplicity, the ALU of Figure 4.28 is more expensiveand slower than necessary. For n = 4, the logic subunit employs about 25 gates andinverters. If the arithmetic subunit is designed with carry lookahead in the style ofFigure 4.6, around 60 gates are needed, depending on the variants of add and sub-tract that are implemented. The multiplexer in Figure 4.28 also requires additional

Y —*<-i>

Data

D-

3Z>

O-i

Dataout

53 S2 5[S0 Select 5

Figure 4.29

An n-bit logic unit that realizesall 16 two-variable functidns.

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)
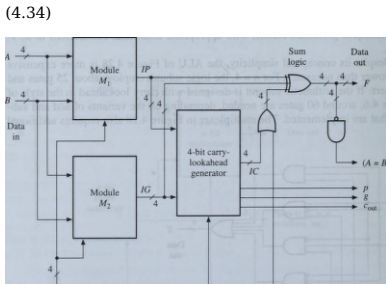
Select 5

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 arith-metic 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 all its input signals. In other words, the overall (A = B) output signal is 1 if and only if each 74181 slice produces (A = B) = 1. This type of technology-specific connection is called a 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 implemented by combinational logic, it is generally impractical to merge these operations with addition and subtraction into a single, combinational ALU. The reason is twofold. Combinational multipliers and dividers are costly in terms of hardware. They are also much slower than addition and subtraction circuits, a consequence of their many 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 division at least n times slower than addition or subtraction. The number of gates in the multiply-divide logic is also greater by a factor of about n. Hence except when n is very small, complete ALUs are usually constructed from low-cost sequential cir-cuits where add and subtract each take one clock cycle, while multiplication and division are multicycle operations.

Basic design. Figure 4.32 shows a widely used sequential ALU design that aims at minimizing hardware costs. This ALU organization is found in the IAS computer (Figure 1.11) and in many computers built after IAS. It is intended to

(A = B)

(A = B)

F><:F,

(A = B)

741814-bitALU

t f

4,- A,-

l15:-A12 B15:512

Fn-F,4 J (A = B)

741814-bitALU

Cl

4>

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

implement multiplication and division using one of the sequential digit-by-digit shift-and-add/subtract algorithms discussed earlier. Three one-word registers are used 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.MQ capable of left- and right-shifting. Additional data processing is provided by a combinational ALU capable of addition, subtraction, and logical operations; we will refer to this unit as the add-subtract unit. This unit derives its inputs from AC and DR and places its results in AC. The MQ register is so-called because it stores the multiplier during multiplication and the quotient during division. DR stores the multiplicand or divisor, while the result (product or quotient and remainder) is stored in the register-pair AC.MQ. The role of these registers is defined concisely as follows:

Addition

Subtraction

Multiplication

Division

AND

OR

EXCLUSIVE-OR

NOT

AC := AC + DR AC := AC - DR AC.MQ := DR x MQ AC.MQ := MQ/DR AC := AC and DR AC := AC or DR AC := AC xor DR AC := not(AC)

DR can serve as a memory data register to store data addressed by an instruction address field ADR. Then DR can be replaced by M(ADR) in the above list of ALU operations, 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 the data 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—its address is the subscript /—so that arithmetic-logic instructions can take the generic two- and 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, easily accessed 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, is often 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 to read R, and R2, and write to R2 in the same clock cycle. RF then needs several access 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 RAM has just one access port with an associated address bus ADR and data bus D. This port can be used to read or write the data word in the single word location we denote by M(ADR).

To build a multiport register file requires a set of registers of the appropriate size and several multiplexers and demultiplexers that allow data words to be steered from any desired registers to the various output ports (read operations) or from the various input ports to registers (write operations). Of course, we don't want several devices writing to the same register R, simultaneously, although they may read from several R/s simultaneously. Figure 4.33 shows a three-port register file that supports simultaneous reads from two ports A and 5, while writing can take place via a third port C. This file contains four 16-bit registers and meets the data access requirements of (4.40) and (4.41). In the two-address case (4.40), the address 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 and fixed-point operations; it is often referred to as an integer or fixed-point unit. It contains a register file RF and a (combinational) ALU capable at least of addition and subtraction. Often specialized circuitry is added for multiplication and divi-sion because the longer delay of these operations and their use of double-length operands make it difficult to include their registers in RF. Also shown are links that 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 such as a floating-point unit.

ALU expansion. It is quite feasible to manufacture an entire sequential ALU for fixed-point w-bit numbers on a single IC chip. Moreover, the ALU can easily be designed for expansion to handle operands of size n = km, or indeed any word size 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 words directly. The resulting array-like circuit is said to be bit sliced because each component ALU concurrently processes a separate "slice" of m bits from each km-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.

CHAPTER 4Datapath Design

2. Temporal expansion: Use one copy of the m-bit ALU chip in the manner of a serial 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 each operand. 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 IC is an example of a bit-sliced combinational ALU. The hardware cost of a bit-sliced ALU such as this increases directly with k, the number of slices, but the ALU's performance measured, say, in cycles per instruction (CPI), remains essentially constant. The cycle period does increase slowly with k, however. In a multicycle ALU, on the other hand, the performance decreases directly with k. but the amount of 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.

SECTION 4.2

Arithmetic-Logic

Units

(Micro) program control unit
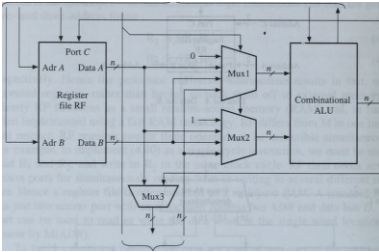
To M and IO system

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-with-carry 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 via ashift-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 multiplier-quotientregister 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

CHAPTER 4Datapath Design

Data

16,

Shift *■signals _

Carry outand flags

Slice [15:12]

Register file—I-

CombinationalALU

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.

SECTION 4.2

Arithmetic-Logic

Units

RAM3 >*-

Q3-

4

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 bit-sliced 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 <•

Data in D

2902

carry-lookahead

generator
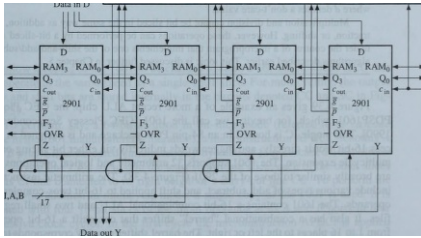
Figure 4.37

A 16-bit 4-slice array of 2901s employing carry lookahead.

Is	Inputs		If		Function Id		Outputs		
	R	S	Y	R(B)	Q				
000	R(A)	Q	000		R + S + C,n	000 F	-	F	
001	R(A)	R(B)	001		S-R-Cjn	oo'i F	-	-	
010	0	Q	010		R-S-Cin	010 R(A)	F	-	
on	0	R(B)	011		RorS	011 F	F	-	
100	0	R(A)	100		RandS	100 F	r'F	2'Q	
101	D	R(A)	101		RandS	101 F	2"'F	-	
110	D	Q	110		RxorS	110 F	2F	2Q	
111	D	0	111		R xnorS	111 F	2F	-	

Figure 4.38 Microoperatiom performed by the 2901.

is specified by the (partial) microinstruction

A,B,Is,IF,ID,Cin = 01! !'0110,001,010,011,0

This microinstruction applies the contents of registers R[7] and R[6] to the R and Sinputs, respectively, of C and selects the ALU function R - S - C,n (subtract with borrow); it also causes the result that appears on F to be stored back into R[6]. Althoughno data-transfer operations are explicitly specified in Figure 4.38, they are easilyobtained from the specified functions. For instance, the operation

Q:=D

loads register Q from an external data source: it can be realized via the logical ORoperation Q := D or 0 as follows:

A,B,Is,IF,ID,Cin = = dddd.ddddA 11,011,000, J (4.42)

where d denotes a don't-care value.

Multiplication and division cannot be bit sliced in the same way as addition, subtraction, or shifting. However, these operations can be performed by a bit-sliced ALUunder the control of a microprogram that implements one of the shift-and-add/subtractalgorithms described earlier. This topic is discussed further in Chapter 5.

Figure 4.39 gives an example of a more recent ALU chip, the GEC PlesseyPDSP1601, which, for brevity, we call the 1601 [GEC Plessey Semiconductors1990]. This single IC is housed in an 84-pin PGA package and is designed to pro-cess 16-bit words directly, and bigger words indirectly via either bit slicing or viamulticycle expansion. The 1601 supports 32 arithmetic and logical operations thatare broadly similar to those of the 2901 (Figure 4.38). The arithmetic instructionsinclude various types of add, subtract, and shift applied to 16-bit twos-complementoperands. The 1601 contains a 16-bit combinational ALU and two small registerfiles. It also has a combinational "barrel" shifter that can shift a 16-bit operandfrom 1 to 16 places to the left or right. The barrel shifter roughly corresponds to the290l's Q shifter but is much more powerful. Shifters of this sort are useful whenimplementing the shifts associated with multiplication, division, and floating-pointoperations. For extension via bit slicing, the 1601 provides carry and shift IO lines

A16-'

T

Register A

B

16-'

II

Register B

f

Carry out CO ■*-Carry in CI —•

CC

Mux A

J \ Mux B /

16-bit

combinational

ALU

2 x 16-bit

ALUregister file

\ Mux S I

16-bitbarrelshifter

/ » Shift out 50

SC

Shift in SI

2x 16-bit

shifterregister file

\ MuxC /

16-'

Figure 4.39

Organization of GEC Plessey 1601 ALU and barrel shifter.

that allow k copies of the 1601 to be chained to form a 16/c-bit bit-sliced ALU thatcan operate at the same speed as a single 1601 slice. For multicycling, the outputcarry and shift bits are stored internally in the circuits denoted CC and SC in Fig-ure 4.39.

To perform, say, a 64-bit addition in bit-slice mode (referred to as cascademode in the 1601 manufacturer's literature), a microinstruction APBCI, denoting Aplus B plus CI, is executed simultaneously by each of four cascaded 1601 slices.The carry-in line CI is set to 0 in the least significant slice; each of the other sliceshas its CI line connected to its right neighbor's carry-out line CO. To perform thesame 64-bit addition in multicycle mode, a single copy of the 1601 is used. It issupplied with four 16-bit slices of the input operands at its A and B ports in foursuccessive clock cycles. In the first cycle the microinstruction APBCI is appliedwith CI = 0. In the remaining three cycles the microinstruction APBCO, denotingA plus B plus CO, is executed, which includes in the sum the output carry bit gen-erated in the preceding clock cycle and stored in CC.

266 4.3

^N4 3 ADVANCED TOPICS

Advanced Topics

This section studies several additional aspects of datapath design. First we discussthe implementation of floating-point operations. Therf we examine the use of pipe-lining to increase the throughput of a datapath unit.

4.3.1 Floating-Point Arithmetic

Let (XM, XE) be the floating-point representation of a number X, which thereforehas the numerical value $XM \times BXe$. Recall from section 3.2.3 that the mantissa (sig-nificand) XM and the exponent XE are fixed-point numbers and that the base B is thesame as the base (radix) of XM. To simplify the discussion, we make the followingrealistic assumptions:

1. XM is an «M-bit binary (twos-complement or sign-magnitude) fraction.

2. XE is an nE-bit integer in excess-$2E$ code, implying an exponent biasof $2^{\wedge"'}$.

3. B-2.

We also assume that the floating-point numbers are stored in normal form only;hence the final result of each floating-point arithmetic operation should be normal-ized.

Basic operations. General formulas for floating-point addition, subtraction,multiplication, and division are given in Figure 4.40. Multiplication and divisionare relatively simple because the mantissas and exponents can be processed inde-pendently. Floating-point multiplication requires a fixed-point multiplication of themantissas and a fixed-point addition of the exponents. For example, if $X = 1.32400111 \times 1017$ and $Y = 1.04799245 \times 1021$, the product $X \times Y$ is given by$(1.32400111 \times 1.04799245) \times 10(17 + 21) = 1.38758607 \times 1038$. Floating-point divi-sion requires a fixed-point division involving the mantissas and a fixed-point sub-traction involving the exponents. Thus multiplication and division are not muchharder to implement than the corresponding fixed-point operations.

Floating-point addition and subtraction are complicated by the fact that theexponents of the two input operands must be made equal before the correspondingmantissas can be added or subtracted. As suggested by Figure 4.40, this exponentequalization can be done by right-shifting the mantissa XM associated with thesmaller exponent XE a total of $YE - XE$ digit positions to form a new mantissa

Addition $X + Y = (XM2*E \sim Ye + Ym) \times 2Ye$ } where $XE < K$,

Subtraction $X- Y= (XM2*E \text{ " } Ye - Ym) \times 2ke$

Multiplication $X \times Y = (XM \times YM) \times 2*E + YfL$

Division $XJY=(XMI \ YM) \times 2*E \sim Ye$

'E

Figure 4.40

The four basic arithmetic operations for floating-point numbers.

$XM2*E \ Ye$, which can then be combined directly with YM. Thus floating-point addi-tion and subtraction have three main steps:

1. Compute $YE-XE$, a fixed-point subtraction.

2. Shift XM by $YE - XE$ places to the right to form $XM \ 2*e \text{ " } Y*$.

3. Compute $XM \ 2*E \sim Ye \pm YM$, a fixed-point addition or subtraction.

For example, to add the decimal floating-point numbers $X = 1.32400111 \times 1017$and $Y= 1.04799245 \times 1021$, we first compute $YE -XE = 21 - 17 = 4$, identifying XEas the smaller exponent. We then right-shift XM by four places to obtain $XM2^{\wedge} =0.00013240$. Finally, we perform the mantissa addition $XM2r" +YM = 0.00013240 +1.04799245 = 1.04812485$, so the final result has mantissa 1.04812485 and expo-nent 21.

Each floating-point arithmetic operation needs an extra step in order to nor-malize the result. A number $X = (XM, XE)$ is normalized by left-shifting (right-shifting) XM and decrementing (incrementing) XE by 1 to compensate for eachone-digit shift of XM. As noted earlier, a twos-complement fraction is normalizedwhen the sign bit $xn \_$, differs from the bit xn_2 on its right, a fact used to terminatethe normalization process. A sign-magnitude fraction is normalized by left-shiftingthe magnitude part until there are no leading 0s, that is, until $xn\_2 = 1$. (The nor-malization rules are different if the base B is not two.) The left-most bit of themantissa may be hidden, since normalization fixes its value; see the discussion ofthe IEEE 754 floating-point standard in Example 3.4.

Difficulties. Several minor problems are associated with exponent biasing. If biased exponents are added or subtracted using fixed-point arithmetic in the course of a floating-point calculation, the resulting exponent is doubly biased and must be corrected by subtracting the bias. For example, let the exponent length be 4, and let the bias be $2^{4-1} = 8$. Suppose that exponents XE = 1111 and YE = 0101 denoting +7 and -3, respectively, are to be added. If ordinary binary addition is used, we obtain the sum XE + YE = 10100, which denotes 12 = 4 + 8 in excess-8 code. The sum 10100 is now corrected by subtracting the bias 1000 to produce 1100, which is the correct biased representation of XE + YE = 4.

Another problem arises from the all-0 representation usually required of zero. If X x Y is computed as (XM x YM) x 2 E + E and either XM or YM is zero, the resulting product has an all-0 mantissa but may not have an all-0 exponent. A special step is then needed to make the exponent bits 0.

A floating-point operation causes overflow or underflow if the result is too large or too small to be represented. Overflow or underflow resulting from mantissa operations can usually be corrected by shifting the mantissa of the result and modifying its exponent; this is done automatically during floating-point processing. For instance, adding the normalized decimal numbers X = 5.1049 x 10⁷ and Y = 7.9379 x 10⁷ produces the sum 13.0428 x 10⁷. which is normalized to 1.3043 x 10⁸ by shifting XM one digit to the right (and rounding off the result) and incrementing the exponent by one. If, however, the exponent overflows or underflows, an error signal indicating floating-point overflow or underflow is generated. A floating-point result that has overflowed may sometimes be retained in "denormalized" form, as discussed in Example 3.4.

To preserve accuracy during floating-point calculations, one or more^ extra bits called guard bits are temporarily attached to the right end of the mantissa

267

CHAPTER 4

Datapath

Design

268

xn-ixn-2---xixo- F°r example, a guard bit jc_, is needed when results are to be rounded rather than truncated to n bits. Rounding is accomplished by adding 1 to xQ and truncating the result to n bits. When a mantissa is right-shifted during the alignment step of addition or subtraction, the bits shifted from the right end can be retained as guard bits. In the case of floati'ng-point multiplication, bits from the right half of the 2/i-bit result of multiplying two Ai-bit (unsigned) mantissas serve as guard bits. Suppose, for instance, that XM = 0.1... and YM = 0.1... are normalized positive mantissas (fractions). Multiplying them by a standard fixed-point multiplication algorithm yields an unnormalized double-length result of the form

PM = XMxYM = 0M... (4.43)

which contains a leading 0. If PM is now truncated or rounded to n bits, then the precision of the result is only n - 1 bits. It is clearly desirable to retain an additional bit from the double-length product so that when (4.43) is normalized by a left shift, the result contains n significant bits. We therefore employ two guard bits in this case, one to maintain precision during normalization and one for rounding purposes.

Floating-point units. Floating-point arithmetic can be implemented by two loosely connected fixed-point datapath circuits, an exponent unit and a mantissa unit. The mantissa unit performs all four basic operations on the mantissas; hence a generic fixed-point arithmetic circuit such as that of Figure 4.32 can be used. A simpler circuit capable of only adding, subtracting, and comparing exponents suffices for the exponent unit. Exponent comparison can be done by a comparator or by subtracting the exponents. Figure 4.41 outlines the structure of a floating-point unit employing the foregoing approach. The exponents of the input operands are put in registers El and E2, which are connected to an adder that computes El + E2. The exponent comparison required for addition and subtraction is made by computing El - E2 and placing it in a counter register E. The larger exponent is then determined from the sign of E. The shifting of one mantissa required before the
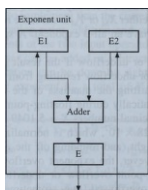
Mantissa unit

AC

MQ

Adder

DR

Data bus

Figure 4.41

Datapath of a floating-point arithmetic unit.

CHAPTER 4

mantissa addition or subtraction can occur is controlled by E. The magnitude of E 269 is sequentially decremented to zero. After each decrement, the appropriate mantissa (whose location in the mantissa unit varies with the operation being performed) is shifted one digit position. Once the mantissas have been aligned, they a a pare processed in the usual manner. The exponent of the result is also computed and placed in E.

All computers with floating-point instructions also have fixed-point instructions, so it is sometimes desirable to design a single ALU to execute both fixed-point and floating-point instructions. This design takes the form of a fixed-point arithmetic unit in which the registers and the adder can be partitioned into exponent and mantissa parts as in Figure 4.41 when floating-point operations are being performed. In recent years it has become more common to implement fixed-point and floating-point instruction in separate units, a fixed-point or integer unit FXU and a floating-point unit FPU. This separation makes it possible for fixed-point and floating-point instructions to be executed in parallel.

Addition. We now consider the implementation of floating-point addition in more detail. Figure 4.42 presents an addition algorithm intended for use with the floating-point unit of Figure 4.41; with minor modifications it can also be used for floating-point subtraction. The mantissa is assumed to be a binary fraction, and the exponent a biased integer; the base B is 2. The first step of the algorithm is equalization of the exponents, which is done by subtracting them and aligning the mantissas by shifting one of them until the difference between the exponents has been reduced to zero. Next the aligned mantissas are added. Finally the result is normalized, if necessary, by again shifting the mantissa and making a compensating change in the exponent. The mantissa and exponent of the final result are placed in the AC and E registers, respectively. Tests are also performed for floating-point overflow and underflow; if either occurs, a flag ERROR is set to 1. A separate test is made for a zero result which, if indicated by AC = 0, causes E to be set to 0 also.

Several improvements can be made to this algorithm: these are left as an exercise (problem 4.29). We can save time by checking to see whether one of the input operands X or Y is zero at the start and simply making the nonzero operand the result. If both X and Y are zero, either operand may be used as the result. If the difference between exponents is very large (IEI > nM), then the shifting process to align one of the mantissas, say, XM in AC, will result in AC = 0 after nM steps. Continued shifting to make E = 0 will not affect the result, which in this case will be YM. Note also that it is more efficient to terminate the shifting after nM steps instead of IEI steps, as is done in Figure 4.42.

Figure 4.43 shows the step-by-step application of the addition algorithm of Figure 4.42 to two 32-bit floating-point numbers. The numbers have the 32-bit format of the IEEE Standard 754 described in Example 3.4. In this format each number N has a 23-bit fractional mantissa M with a hidden bit. an 8-bit exponent E in excess-127 code, and a base B = 2. The value of /V is therefore given by the formula

The numbers to be added in this instance are

AT = 0 01111111 10000000000000000000000 Y = 0 10000111 00101011010000000000000

270

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

Exponent registers Mantissa registers

Step	El	AC	E2	E	Mantissa	
	El		E2	E		
	AC UK					
LOAD	01111111	10000111	00000000	11000000000000... 00	10010101101000... 00	
	= *E		= ^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	10010110011000... 00= AC + DR
Result	10000111= (X+50e	10010110011000... 00 $= i.(X+y)M$
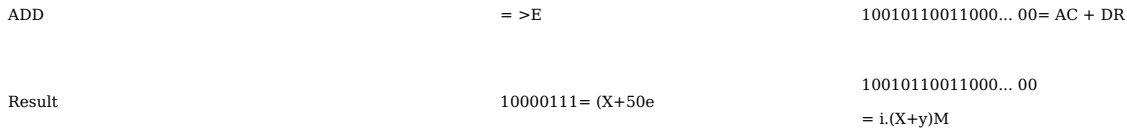
Figure 4.43

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 floating-pointformat 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

1

i   r   System bus

1
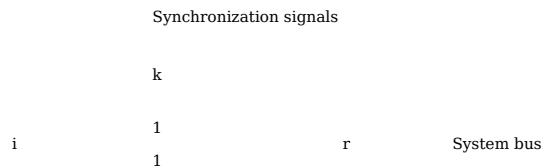
. To main memoryJ and 10 devices

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.

EXAMPLE 4.7 THE MOTOROLA 68882 FLOATING-POINT COPROCESSOR

[motorola 1989]. The Motorola 68882 coprocessor extends 680X0-series CPUs

274	Type	Opcode	Operation specified
	Data transfer	FMOVE	Move word to/from coprocessor data or control register
		FMOVECR	Move word to/from ROM storing constants (0.0,7t, e, etc.)
		FMOVEM	Move multiple words to/from coprocessor
	Data processing	FADD	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 68882 and 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 decimalfloating-point 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

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   'p i r   ■r ii 'i

)ata R   C. R- c. "*..."* R   C

in

Dataout

—v V

Stage S| Stage S2

VStage S„
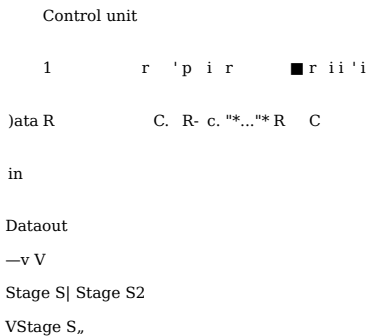
Figure 4.47

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

$S(4) =$

$\dfrac{4N}{N+3}$

$x - (xM. xE)$

$>' = (>'m.Ve)$

Data

			Exponent
Ri Exponentadder *: c2	Mantissashifter R3	Mantissaadder c3	adder and R> mantissa shifter Q
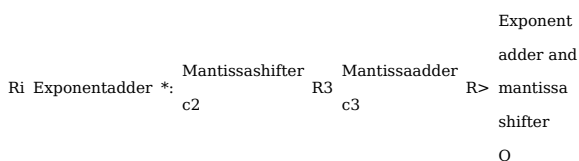
Stage 5,(Exponentcomparison)

Stage S2

(Mantissaalignment)

'VStage S3(Mantissaaddition)

Stage 54(Normalization)

Figure 4.48

Four-stage floating-point adder pipeline.

Dataout

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->'3>

t   t   ♦

C*5. >s)

1   *   *

♦   ♦   •

♦   •   ♦

(*e- ye)

*   *   •

(*5->'5> <*6- >6)

♦   ♦   1

(x4,y4)

•*1+>1 -<2 + >'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

i

El E2 Ml   M2 LHAF1EK 4Datapath

_ i

II

I   Design

Adder

1

1

1 ,

1, u   f II   ji   s2

E4 E5 M4   M5

1 T   1

1

Shifter 1

i   --?"

"' "1

u \   1   i   *3
r t   v *

E6   11   M6 ii M7

1 .

Adder
iii
2

ji   ■   r   54

E7   1   R

Zero-d   er

w   '   check

Adder3   i   r

Shifter 2

u   1   I

E3   M3

_|_

out
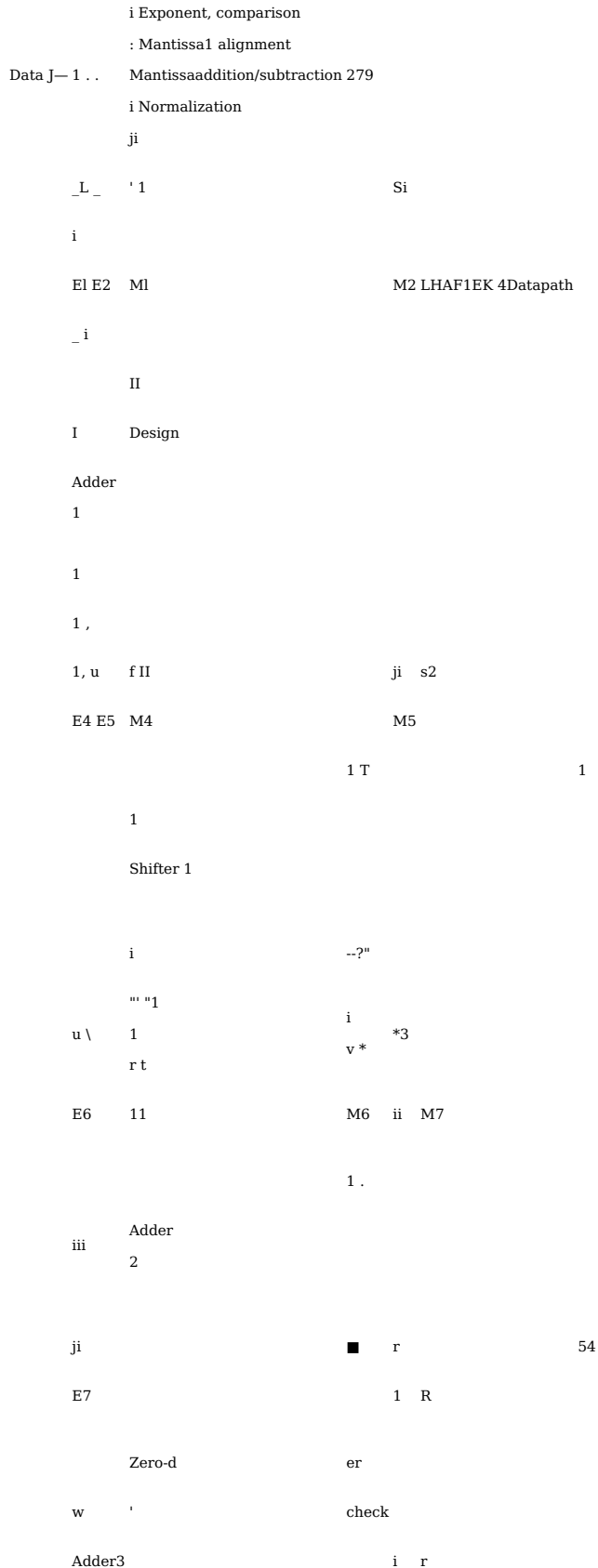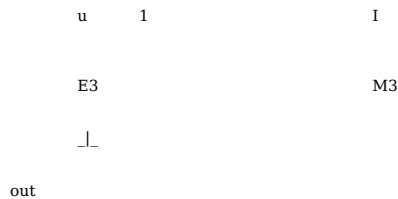
Figure 4.50

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 multifunc-

tion 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

$$T_c = \max\{\tau_i\} + T_R \quad \text{for } i = 1,2,\ldots,m \quad (4.44)$$

The throughput of Pm is $1/T_C = l/(\max\{\tau_i\} + \tau_R)$. A nonpipelined implementationPx of F has a delay of $\sum_{i=j} T_i$ or, equivalently, a throughput of $1/(\sum_{j=1} T_i)$. 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 modi-fied 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   Input X

4 i V 1   1 i r '   r

\   Multiplexer r* - -\ Multiplexer /-   1

I

Stage 5,   «--- -4

3   1

1   Stage 52   ♦ -- -4

'   r

Stage S3   ■« -4

i   '

Stage 54   «--- -4

1   r

Register /?   ^
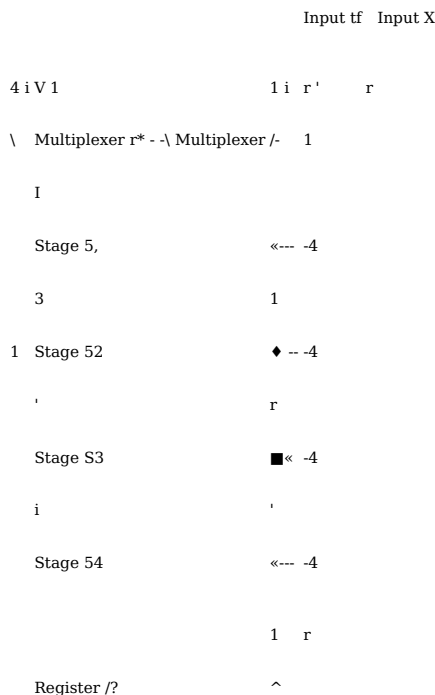
Control

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 con-tinuous 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

$bi + b1 + bu+b^ + ...$

$fc4 +\&\blacksquare + \&,, + fe.fi + ...$

(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)

        1 1

(CAJ

1

(0,6,)

1

J L

(0, bA)

JL_

(0, b3)

(0. fc,)

HZ

(0J>.i

r = 4

/ = 5

'   r I

(*i.*s)

1

(0, 64)

*

(0. fej)

I

(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)

<l ►■

When the last input operand bN has entered the pipeline, the feedback structure is again altered to allow the four partial sums in (4.45) to be added together to produce the desired result bx + b2 + ... + bx. The necessary modification to the feedback struc-ture is shown in Figure 4.52 for the case N = 8. At t = 9, the external inputs to the pipe-line are disabled by setting them to zero, and the first of the four partial sums bx + b5 at the output of stage 54 is stored in register R. Then at t = 10. the new result b2 + b6 from S4 is fed back to the pipeline inputs, along with the previous result bx + b5 obtained from R. Thus computation of bx + b5 + b2 + b6. which is the sum of half of the input operands, begins at this point. After a further delay of one time period, computation of the other half-sum bi + b-, + bA + b% begins. When b, + b5 + b2 + bb emerges from S4 at t = 14, it is stored in R until i»4 + £>„ + b3 + b7 emerges from 54 at t - 16. At this point the outputs of 54 and R are fed back to Sv The final result is produced four time periods later—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)

'   r

b} + b5 + b2 + b6

f= 14

(0,0)

4

(bl + /?5. b-> + />6)

41

(0.0)

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

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

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 $X = x_n \, x_{n-2} \, .. \, .x_0$ and $Y = y_{n-1} y_{n-2} \cdot \cdot \cdot v_0$. 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 stage Si, $0 < i < n - 1$, 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

i-

Register R2

M

4-

M

^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 $P_{n\_x} = XY$ being computed by the last stage. In addition to storing the partial products in the buffer registers denoted /?,, the multiplicand Y and all hitherto unused multiplier bits must also be stored in /?,-.

An /i-stage multiplier pipeline of this type can overlap the computation of n separate 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 M cells 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 disjoint full adders. Its input is three /7-bit numbers to be added, while the output consists of the n sum bits forming a word 5 and the n carry bits forming a word C. Unlike the adders 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 shown in Figure 4.54, they can be added to a third n-bit number W. Observe that the carry connections are shifted to the left to correspond to normal carry propagation. In general, 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 by a conventional adder with carry propagation.

Multiplication can be performed using a multistage carry-save adder circuit of the 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 bit weighted by the appropriate power of 2. Suppose that Mi is In bits long and that a full double-length product is required. The desired product P is Z£Tq M;. This sum 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   y2   Y\   >'o

'   z   3   '   z 2   1   z i   z x y z
    1'   I
                                II i i 1

CS adder

/ wl / "0   w
w3   / w2   /   /   i   C   S
        /   /   1

T V   T "   T   T   CS adder

r   r   r   ▼ «

/   /   /   i

1 1   T   T   » C 5'
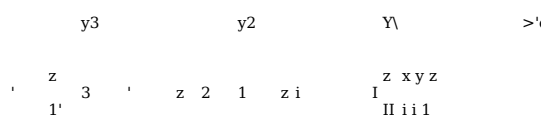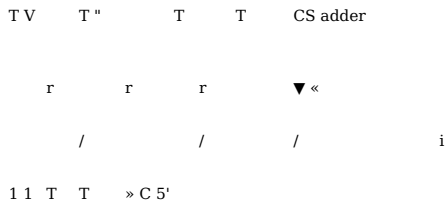
$3 $2 *1 -^O

c'-i ci c\

Figure 4.54

At   WO -sta ge ca irn,   -Si ive i iddei

286

Advanced Topics Multiplier decodingand multiplicand gating <

M5   M4   M3   M2\ Mx A/n i
                                                   r II

CS adder   CS adder

C   5   C   S

                                     i'   i   l

                                     CS adder

                                     C   .5
                                              i

                                     i   ii

                                     CS adder

                                     C   S

                                     Carry-
                                     lookahead
                                     adder
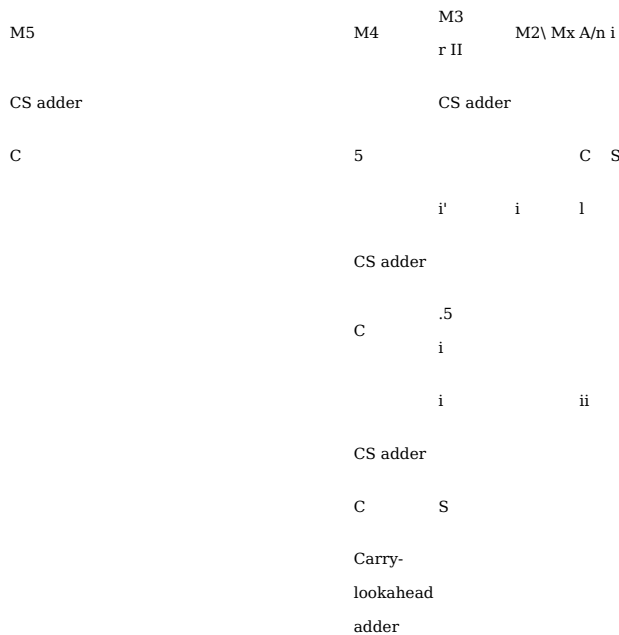
                                              i   Figure 4.55

                                              P   A carry-save (Wallace tree) multiplier

2«-bit carry word. The final carry assimilation is performed by a fast adder—acarry-lookahead adder, for instance—with normal internal carry propagation.

The strictly combinational multiplier of Figure 4.55 is practical for moderatevalues of n, depending on the level of circuit integration used. For large n, the num-ber of carry-save adders required can be excessive. Carry-save techniques can stillbe used, however, if the multiplier is partitioned into k m-bit segments. Only mterms M, are generated and added via the carry-save adder circuits. The process isrepeated k times, and the resulting sums are accumulated. The product is thereforeobtained after k iterations.

Carry-save multiplication is well suited to pipelined implementation. Figure4.56 shows a four-stage pipelined version of the carry-save multiplier of Figure4.55. The first stage decodes the multiplier and transfers appropriately shifted cop-ies of the multiplicand into the carry-save adders. The output of the first stage is aset of numbers (partial products) that are then summed by the carry-save addertree. The carry-save logic has been subdivided into two stages by the insertion ofbuffer registers (denoted R in the figure). The fourth and final stage contains acarry-lookahead adder to assimilate the carries. This type of multiplier is easilymodified to handle floating-point numbers. The input mantissas are processed in afixed-point multiplier pipeline. The exponents are combined by a separate fixed-point adder, and a normalization circuit is also introduced.

The next example describes the pipelined floating-point unit of the Motorola68040 microprocessor, which integrates the functions of the 68020 microprocessor(section 3.1.2 and Examples 3.3, 3.6, and 3.8) and its 68882 floating-point copro-cessor (Example 4.7) in a single IC containing more than 1.2 million transistors.

Inputbus

287

CHAPTER 4

Datapath

Design

Multiplier decodingand multiplicand gating

CS adder

CS adder

C

CS adder

C

CS adder

Carry-

lookahead

adder

CS adder

CS adder

CS adder

Output ^_bus

Figure 4.56

A pipelined carry-save multiplier.

This floating-point unit also implements many of the design techniques covered inthis section.

EXAMPLE 4.9 THE PIPELINED FLOATING-POINT UNIT OF THE MOTOR-OLA 68040 [EDENfield et al. 1990). This member of the 680X0 series of one-chip 32-bit microprocessors was introduced in 1990. It executes the combined instruc-tion sets of the 68020 CPU and the 68882, which are listed in Figures 3.12 and 4.46,respectively, and is about four times as fast as the 68020 for a fixed clock rate. The68040 contains two pipelined arithmetic processors: an integer unit (IU), which handlesinteger instructions, logical instructions, and address calculations, and a floating-pointunit (FPU), which we now examine. A key design goal of the FPU is compatibility withobject code written for the 68020 and 68882, as well as compatibility with the IEEE754 floating-point standard. Only the subset of the 68882's instructions listed in Figure4.57, including the four basic arithmetic operations and square root, are actually real-ized in hardware. Also included is a small set of data-transfer and program-control

Type	Opcode	Operation specified
Data transfer	FMOVE	Move word to/from coprocessor data or control register
	FMOVEM	Move multiple words to/from coprocessor
Data processing	FADD	Add
	FCMP	Compare
	FDIV	Divide
	FMUL	Multiply
	FSUB	Subtract
	FABS	Absolute value
	FNEG	Negate
	FSQRT	Square root
Program control	FBcc	Branch if condition code (status) cc is 1
	FDBcc	Test, decrement count, and branch on cc
	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 4.57

Subset of the 68882 floating-point instruction set implemented by the 68040.

instructions to support floating-point operations. The remaining 68882 instructionsmust be simulated by software, for which the 68040 provides some hardware support.

The 68040's FPU has the three-stage pipeline organization shown in Figure 4.58.It is designed to handle floating-point number sizes of 32, 64, and 80 bits. The FPU isdivided into two largely independent subunits: one for 64-bit mantissas (which expandto 67 bits when guard digits are included) and the other for 16-bit exponents. The FPUobtains its operands from and sends its results to the IU in a way that mimics the68882's communication with its host CPU. The pipeline's first stage 5, (referred to asthe floating-point conversion unit) reformats input and output operands to meet IEEE754 requirements, and is the only stage that communicates with the IU. Stage 5, alsohas an ALU for comparing input exponents, as required in floating-point addition orsubtraction. The second stage 52 (the floating-point execution unit) contains a large(67 bit) ALU, a fast barrel shifter, and an array multiplier; this stage is responsible forexecuting all major operations on mantissas. The final stage S3 (the floating-point nor-malization unit) rounds off and normalizes results; it also deals with exceptional cases.Various buses shown in simplified form in Figure 4.58 provide bypass and feedbackpaths through the pipeline. The clocking of the pipeline is complicated by the need touse several cycles to transfer long operands so that the minimum delay of each stage istwo cycles. The delay of a floating-point operation can vary from 2 clock cycles tomore than 100 in the case of the FSQRT instruction.

Certain instructions such as FABS, FMOVE, and FNEG are executed entirelywithin stage S{ and thus have a delay of two clock cycles. The add and subtract instruc-tions FADD and FSUB use all three stages of the FPU and have a delay of three. Theseinstructions see a pipeline whose organization resembles that of Figure 4.50, with thelatter's middle stages S2 and 53 merged into the 68040's second (execution) stage S2-FMUL is executed primarily by the 64 x 8-bit, fixed-point multiplier in S2. The multi-plication of two mantissas requires several passes through the multiplier circuit, which

implements the carry-save multiplication method discussed earlier. Two passes can bemade per clock cycle of the pipeline, so that the final set of sum-carry pairs is gener-ated in four clock cycles. An additional cycle through SVs ALU assimilates the carriesand yields the final product. The floating-point division instruction FDIV is imple-mented by a shift-and-subtract algorithm of the non-restoring type, which requires nospecial division hardware but takes up to 38 clock cycles.

Datapath

Design

Integer unit

Integer unit

16

Exponentunit

Register

±

17-bit ALU

Register file

64 ,

Mantissaunit

Register

3

Incrementer

Register file

Stage 5,

(Format conversion

and

exponent comparison)

Register

Register

64 x 8-bit

multiplier

Register

67-bit ALU

67-bit shifter

Stage S2(Execution unit)

RegisterI

Register

17-bit ALU

Incrementer

Register

Register

Figure 4.58

Pipelined floating-point unit of the Motorola 68040.

Stage 53

(Normalization,

rounding, and

exception checking)

SECTION 4.3Advanced Topics

Systolic arrays. Closely related conceptually to arithmetic pipelines are thedata-processing circuits called systolic arrays [Johnson, Hurson. and Shirazi1993] formed by interconnecting a set of identical data-processing cells in a uni-form manner. Data words flow synchronously from cell to cell, with each cell per-forming a small step in the overall operation of the array. The data are not fullyprocessed until the end results emerge from the array's boundary cells. A one-dimensional systolic array is therefore a kind of pipeline with identical stages. Atwo-dimensional systolic array has a structure not unlike the divider array in Fig-ure 4.27, but its cells are sequential rather than combinational. In general, a sys-tolic array permits data to flow through the cells in several directions at once. Asin pipelines, buffering must be included within the cells to isolate different sets ofoperands from one another. The name systolic derives from the rhythmic nature ofthe data flow, which can be compared with the rhythmic contraction of the heart(the systole) in pumping blood through the body. Systolic processors have beendesigned to implement various complex arithmetic operations such as convolu-tion (problem 4.34), matrix multiplication, and solution techniques for linearequations. We illustrate the concepts involved by a two-dimensional systolic arraythat performs matrix multiplication.

Let X be an $n \times n$ matrix of fixed-point or floating-point numbers defined by

$$X =$$

4, 1 A1.2 •••-M.nf2, 1 Xl,2 •••JC2, n

Vl

For brevity we write $X = [jc_{;}]$, where x_t is the element in the ith row andjth col-umn of X. The product of X and another $n \times n$ matrix $Y = [v^]$ is the $n \times n$ matrix$Z = [z_{,,}]$ given by

$$z_{i,j} \sim — x_i.$$

XV

k,j

(4.47)

$$k = 1$$

A systolic array for matrix multiplication may be constructed from a cell (Figure4.59a) that executes the following multiply-and-add operation on individual num-bers (scalars):

:=z + x x v

(4.48)

Note that the same type of operation appears in the cell M of the fixed-point arraymultiplier in Figure 4.53, with 1-bit operands replacing the n-bit numbers usedhere. Multiply-and-add is also a basic instruction type in recent CPUs such as thePowerPC.

Each cell C,- - of the matrix multiplier receives its x and y operands from theleft and top, respectively. In addition to computing z, Ci} propagates its x and yinput operands rightward and downward, respectively. The systolic matrix multi-plier is constructed from n(2n - 1) copies of C, , which are connected in the two-dimensional mesh configuration depicted in Figure 4.59b. The n operands formingthe /th row of X flow horizontally from left to right through the /th row of cells asthey might through a one-dimensional pipeline. The n operands forming the y'th

column of Y flow vertically through the _/th column of cells in a similar manner.The x and y operands are carefully ordered and separated by zeros as shown in thefigure so that the specific operand pairs xlhyk} appearing in (4.47) meet at anappropriate cell of the array, where they are multiplied according to (4.48) andadded to a running sum z . The z's emerge from the left side of C(J, so that there isa flow of partial results from right to left through the cell array. Each row of cellseventually issues the corresponding row of the matrix product Z from its left side.

To illustrate the operation of the matrix multiplier, consider the computation ofZ\ i in Figure 4.59/?. Specializing Equation (4.47) for the case where n = 3, we get

$z_{i,i} = *u)'i,i + x\,iyi,\ + -*i,3}!3,i$

291

CHAPTER 4

Datapath

Design

$x(t)z\{t\} = xif)y.y(t) + z\t)$

x(t-\)z\t)

(a)

*3.3 0

7 0

6 0

me ♦ ;

5 v3.,

Time

0 0

0 x2 3 0 x22 0 x2J

4 0

3 0

>3.20

o

-M.lZl.i

o

Z2J

^3.3

0

>2.2

0

-vl.l

0 0

>2.3 0

0 .V,.3

vi; 0

0 0

I I I I I

0 xxi 0 0

*3.A

C,.l   C,.2   C,.3 ^   C..4   C,.s

1   r   1   1   1   1   1   r 'i

Q.1   Cl.2   Q.3   C2.4   cXi

    ■•—   -«—

'   !   1   r   '   '   1   r ''

♦-

*

Time t

(b)

Figure 4.59

Systolic array for matrix multiplication: (a) basic cell and (b) 3 x 5 array.

292 The operand xl, flows rightward through the top row of cells meeting only zero

.r„,.., . . values of v and z until it encounters y, , at cell C\ -, at time r = 3. This cell then com-

SECTION 4.4 ' *»1 I,J

Summary putes z = *t jy,, + 0, which it sends to the cell C, 2 on its left. At the same time C, 3

forwards y, , to the second row of cells for use in computing the second row of theresult matrix Z; it also forwards xu to its right neighbor C, 4. In the next clockcycle (t = 4), xx2 and y2x are applied to Cl2- This cell therefore computes z =.x, 2y2,i + z, where z' = xx ,y, ,. Finally at t = 5, the last pair of operands xx3 and y3,converge at the boundary cell C, ,, which computes z = xl3y3] + z, using the valuez - xi,i3'i,i + x\.iy7.\ supplied by C, 2; z is the desired result z, ,. At time t = 6, Cx ,emits a zero, and at t = 7, it emits the next element zX2 of Z. This process continuesuntil all the elements of the first row of Z have been generated. Concurrently and ina similar way, the remaining rows of cells compute the other rows of Z. Note, how-ever, that xi+x j is produced two cycles later than x/;. The last result z„„ emergesfrom the array at t = An - 3. Thus using 0(n2) cells, this systolic array performsmatrix multiplication in O(n) time, that is, linear time. Roughly speaking, the arraygenerates n elements of the product matrix Z in one step (two clock cycles in thepresent example).

The major characteristics of a systolic array can be deduced from the preced-ing example.

1. It provides a high degree of parallelism by processing many sets of operandsconcurrently.

2. Partially processed data sets flow synchronously through the array in pipelinefashion, but possibly in several directions at once, with complete results eventu-ally appearing at the array boundary.

3. The use of uniform cells and interconnection simplifies implementation, forexample, when using single-chip VLSI technology.

4. The control of the array is simple, since all cells perform the same operations;however, care must be taken to supply the data in the correct sequence for theoperation being implemented.

5. If the X and Y matrices are generated in real time, it is unnecessary to storethem before computing X x Y, as with most sequential or parallel processingtechniques. Thus the use of systolic arrays reduces overall memory require-ments.

6. The amount of hardware needed to implement a systolic array like that of Figure4.59 is relatively large, even taking maximum advantage of VLSI.

Systolic arrays have found successful application in the design of special-purposearithmetic circuits for digital signal processing, where data must be processed inreal time at very high speeds using operations like matrix multiplication.

4.4SUMMARY

The datapath or data-processing part of a CPU is responsible for executing arith-metic and logical (nonnumerical) instructions on various operand types, includingfixed-point and floating-point numbers. The power of an instruction set is oftenmeasured by the arithmetic instructions it contains. The arithmetic functions ofsimpler machines such as RISC processors may be limited to the addition and sub-

traction of fixed-point numbers. More powerful processors incorporate multiplyand divide instructions and in many cases have the hardware needed to processfloating-point instructions as well.

Arithmetic circuit design is a well-developed field. Fixed-point adders andsubtracters are easily constructed from combinational logic. The simplest but slow-est adder circuits employ ripple-carry propagation. High-speed adders reducecarry-propagation delays by techniques such as carry lookahead. Fixed-point mul-tiplication and division can be implemented by shift-and-add/subtract algorithmsthat resemble manual methods. The product or quotient of two km-bit numbers isformed in k sequential steps, where each step involves an m-bit shift and, possibly,a km-bit addition or subtraction. Division is inherently more difficult than multipli-cation due to the problem of determining quotient digits. Both multipliers anddividers can be implemented by combinational logic array circuits but at a substan-tial increase in the amount of hardware required.

The simplest ALU is a combinational circuit that implements fixed-pointaddition and subtraction, typically using the carry-lookahead method; it alsoimplements a set of bitwise (word) logical operations. Multiplication and divisionalgorithms of the shift-and-add/subtract type can be realized by adding a fewoperand registers—an accumulator AC, a multiplier-quotient register MQ, and amultiplicand-dividend register MD—as well as a small control unit. Datapathunits usually contain an addressable register file—in effect, a small, high-speedRAM—to store ALU operands. The register file has several 10 ports to allowoperands in several different registers to be accessed simultaneously. Bit slicing isa useful technique for constructing a large ALU from multiple copies of a smallALU slice. Multicycling allows a small ALU to process large operands at lowerhardware cost but more slowly than bit slicing.

Floating-point and other complex operations can be implemented by an auton-omous execution unit within the CPU or by a program-transparent extension to theCPU called a coprocessor. A floating-point processor is typically composed of apair of fixed-point ALUs—one to process exponents and the other to process man-tissas. Special circuits are needed for normalization and, in the case of floating-point addition and subtraction, exponent comparison and mantissa alignment.

Finally, the throughput of a complex datapath circuit such as a floating-pointprocessor can be substantially increased with low hardware overhead by a tech-nique called pipelining. The operations of interest are broken into a sequence ofsteps, each of which is implemented by a pipeline stage. Buffering between thestages allows an «-stage pipeline to execute up to n separate instructions concur-rently. Hence the pipeline's throughput when executing a long sequence ofinstructions exceeds by a factor of up to /; that of a similar but nonpipelined pro-cessor. Systolic arrays extend the pipeline concept from one to two or more data-processing dimensions.

293

CHAPTER 4

Datapath

Design

4.5PROBLEMS

4.1. Figure 4.60 gives the logic diagram of a small arithmetic circuit found in a commercialIC with the 10 signals renamed to conceal their identities, (a) What is the overall func-tion of this circuit? (b) Identify the purpose of every 10 signal, (c) Why do all input

294
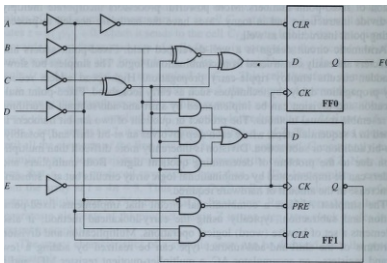
Figure 4.60

Small arithmetic circuit from a commercial IC.

lines contain inverters, which apparently increase the circuit's gate count without con-tributing to its functionality?

4.2. A 1-bit or full subtracter implements the arithmetic equation

where z: and bt denote the difference and borrow functions, respectively, (a) Derive a pair of logic equations defining s, and bt. (b) Design an «-bit subtracter whose opera-tion is analogous to that of a ripple-carry adder.

4.3. Redesign the n-bit twos-complement adder-subtracter of Figure 4.4 so that it can com-pute any of the three operations X + Y, X - Y, or Y - X, as specified by a 2-bit MODE control input.

4.4. Addition and subtraction of sign-magnitude numbers is complicated by the fact that to compute X + Y, the magnitudes \X\ and \Y\ must be compared to determine the operation to perform and the order of the operands. This can be seen from Figure 4.61 which gives a complete procedure for addition of «-bit, sign-magnitude numbers. Design a register-level circuit to compute the three functions X + Y, X - Y, and Y-X. Assume that the word size n is 16 bits and that the standard design components are available, including a 16-bit (unsigned) adder, a 16-bit (unsigned) subtracter, and a 16-bit mag-nitude comparator.

4.5. Suppose that the adder-subtracter circuit of Figure 4.62 has been designed for twos-complement numbers. It computes the sum Z = X + Y when control line SUB = 0 and the difference Z = X - Y when SUB = 1. An overflow flag v is to be added to the circuit, but it is not possible to access internal lines. In other words, only those data and control

1. X and Y both positive: Add X = xn_xxn_2...x0 and Y = >'n_i>n_2—>'o (modulo 2") to form the result Z = z„_iZ„_2—2o- (This is "-bit unsigned addition).

2. X positive; Y negative: Let 1X1 = xn_2xn_y..xn and \Y\ = y„

>fl'

^-3->0

Vr, If LX1 < M, subtract X from y

(modulo 2"). If 1X1 > \Y\, then set v„_, to 0 and subtract Y from X (modulo 2").

3. X negative; Y positive: If in < LX1, subtract Y from X (modulo 2"). If in > 1X1, set.r„_, to 0 and subtract 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 circuit for v.

4.6. Consider again the adder-subtracter of Figure 4.62, assuming now that it has been designed 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 and a 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-bit carry-lookahead generator.

4.8. Show how to extend the 16-bit design of Figure 4.8 to a 64-bit adder using the same two 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 for each of the following for both an n-bit carry-lookahead adder and an n-bit ripple-carry adder: (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. It and 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 assumes that its input carry c,_, = 1. and a second version s?, which assumes that ct_{ = 0. A multiplexer controlled by <?,_, then selects either sj or s® to be sr The advantage of this method is that the sums (and carries) can be generated without waiting for their

296

-<>,

v,,

SECTION 4.5 Problems

CSC

C2 S2 C2 S2

Two-way I2-bit mux fj

CSC

C, 5, C, 5,

Tw

2-bit

o-way /it mux l~~fl

Two-way / c0

3-bit mux /

CSC

o o

c0 50 c0 s0

Two-way2-bit mux

cout - c2

}

S2 5,

*0

Figure 4.63

Three-bit conditional-sumadder.

incoming carries to arrive. Figure 4.63 shows a 3-bit conditional-sum adder fromwhich its general structure and operation can readily be deduced, (a) Construct a gate-level logic circuit for the CS module, (b) Show how to extend the circuit of Figure4.63 from 3 to 7 bits, (c) Briefly compare the conditional-sum and carry-lookaheadtechniques in terms of speed and hardware costs.

4.11. Suppose the combinational array multiplier of Figures 4.17 and 4.18 is given the un-signed integer operands X= 1010 and Y= 1001. Determine the output signals generatedby every adder cell when the array computes X x Y.

4.12. Use the multiplier cell of Figure 4.19 to construct a combinational array multiplier for5-bit unsigned numbers. Draw a logic diagram for the multiplier and show all the sig-nals (including constant signals) applied to every cell.

4.13. Suppose a multiplier MULT)6 for 16-bit unsigned numbers is constructed from ANDand adder arrays as illustrated by Figures 4.17 and 4.18, respectively. Let d denote thepropagation delay of a single gate G and let D = Ad be the delay of a full adder FA.(a) How many copies of FA are needed to build MULT,6? (b) What is the worst-casedelay of MULT16? (c) Observe that the bottom row of full adders in Figure 4.17 is asimple ripple-carry adder ADD. Stating all your assumptions, estimate the speedup inmultiplication that results from replacing ADD in MULT,6 by a carry-lookaheadadder of standard design.

4.14. Suppose the Booth array multiplier of Figure 4.20 is given the signed integer operandsX = 1010 and Y = 1001. Determine the output signals generated by every M cell whenthe array computes X x Y.

4.15. In bit-by-bit multiplication of Kby Xy bit x, = 1 in position i of X causes an addition thatcontributes 2'Y to the solution P. Clearly x{ - 0 contributes nothing to P. In the Boothalgorithm x(= 1 causes either addition or subtraction; in the latter case it contributes

-2'Yto the solution. Thus X x Y is computed in the form (±2']Y ± 2'2Y ± ... ± 2'kY) =(±2'' ± 2'2 ± ... ± 2,k) x Y. Booth's algorithm effectively multiplies by a number X* thathas its digits weighted by -2' as well as the usual +2'. We can make this weighting explicitby "recoding" X into X* using the threedigits 0, 1. and 1. where 1 in position i denotesa weight of-2'. For example, X* = 1 100100 10 is evaluated as +28 - 27 + 24 - 21 =7010 = 01001110-,. A"* is an instance of a signed-digit number, a useful concept in de-signing multipliers and dividers. Using the recoding rules implicit in Booth's algorithm,obtain signed-digit representations of the twos-complement integers A = 011010001and B= 101011110.

297

CHAPTER 4

Datapath

Design

4.16. Booth multiplication skips over runs of zeros and ones, which reduces the number ofadd and subtract steps needed to multiply two n-bit numbers from n to a variable num-ber whose average value is less than n. Some designers argued that this fact can beexploited to reduce the average multiplication time from n to «ave steps, (a) Show that"ave = "/2. (Hint: Assume «ave is known and use it to determine [n + l]ave). (b) Explainwhy practical multipliers are rarely designed to use this speedup technique. (See thefollowing problem for a practical speedup technique for Booth multipliers.)

4.17. A faster version of Booth's multiplication algorithm for twos-complement numbers,known as the modified Booth algorithm (MBA), examines three adjacent bits .r(+1.v,.v_lof the multiplier A" at a time, instead of two. Besides the three basic actions performedby the original Booth algorithm, which can be expressed as add 0. Y, or -Y to A (theaccumulated partial products). MBA performs two more actions: add +2 Yor -2Y to A.These have the effect of increasing the radix from two to four and allow an n-bit mul-tiplication to be done in n/2 clock cycles instead of n (at the usual cost of more hard-ware). Figure 4.64 shows a pencil-and-paper application of MBA to two 8-bit twos-complement integers X and Y. (a) Construct a truth table that defines the basic actionsof MBA as a function of j:/+1jc^c,_,. (b) Give an HDL description of MBA along the linesof Figure 4.15.

4.18. Division circuits usually include logic to detect a dividend-divisor combination thatwill cause the quotient to overflow. Suppose that a divider for «-bit unsigned integershas a double-word (2rc-bit) dividend D and a single-word divisor V. in) What generalcondition must be satisfied for quotient overflow to occur? (b) How would you modify

Operands Values	1	x.+lxrri-l	Action
Multiplicand Y 10101010			
Multiplier X 11001110	0	100	
P0 0000000010101100			Add -2Y to A
P2 00000000000000	2	111	Add 0 to A
P4 111110101010	4	001	Add+KtoA
P6 0001010110	6	110	Add -Y to A
P 0001000011001100			
= Pu+P2 + P4 + Pb			

Figure 4.64

Illustration of the modified (radix-4) Booth method of multiplication.

298 the sequential division circuit of Figure 4.23 to introduce an overflow detector using as

little extra logic as possible?

SECTION 4.5Problems

4.19. Suppose the restoring array divider of Figure 4.27 has the integer operands D = 100110and V- 101. Determine the results Q and /?, as well'as the vertical output

signals gen-erated by every D cell when the array computes D/V.

4.20. Consider the divider array of Figure 4.27 that is designed to handle a word size of n =3 with a double-length (6 bit) dividend D. (a) Why are there four rows of D cells insteadof three? (b) Suppose that dividends are restricted to 3 bits instead of 6. Which cellscan then be deleted from the array?

4.21. Figure 4.65 shows a gate-level logic diagram for the 74181 ALU/function generator.The inputs have been assigned the names used in Figure 4.30, but the eight outputs areabstractly labeled/, :/g. Deduce (without using any outside sources) the correspondencebetween the output signal names in the two figures; that is, identify all the outputs inFigure 4.65 and explain your reasoning.

4.22. (a) What arithmetic and logic functions are computed by the 74181 ALU when 5 =53525,50 = 1100? (b) A useful logic operation of the 74181 is the EXCLUSIVE-ORfunction A © B. What values should 5, M, and cjn have in this case? Briefly explain yourreasoning.

4.23. The 74181 ALU is designed for use as a 4-bit magnitude comparator. For this purposeit must be set to its arithmetic subtract mode (M = 1, S = 0110) with cjn = 1. The rela-tions between the magnitudes of A and B can then be determined from the combinedvalues of the two outputs (A = B) and cout. Identify the specific output values that indi-cate each of the following: A = B, A < B, A < B, A > B, and A > B.

4.24. Show how to connect four copies of the 74181 to form a 16-bit ALU with carry looka-head across all stages.

4.25. Design a register file in the style of Figure 4.33 that stores eight 32-bit numbers andhas one read port A and one write port B.

4.26. Suppose the register file RF16 of Figure 4.33 is to be built out of four identical 4-bitslices denoted RF4. (a) Give a register-level diagram showing the internal structure ofRF4. (b) Show how four copies of RF4 are interconnected to form RF16.

4.27. Design a 16-bit bit-sliced ALU using four copies of the AMD 2901 4-bit slice. Usecarry lookahead and use NAND gates to design the necessary carry-generation logic.Give a block diagram of your design and give a set of Boolean equations that specifythe carry-lookahead function.

4.28. Suppose the 1601 ALU of Figure 4.39 operating at a clock frequency of 20 MHz isused to build an ALU intended to execute a long sequence of 80-bit additions. What isthe maximum throughput in operations per second if the 1601-based ALU is set up toperform 80-bit operations (a) in bit-sliced mode and (b) in multicycling mode.

4.29. Modify the algorithm for floating-point addition in Figure 4.42 to make the followingimprovements: (a) Perform either addition or subtraction as specified by an opcode inthe instruction register IR. (b) Test for zero operands at the start and skip as muchcomputation as possible when X and/or Y is zero, (c) Modify the mantissa assignment

Oi
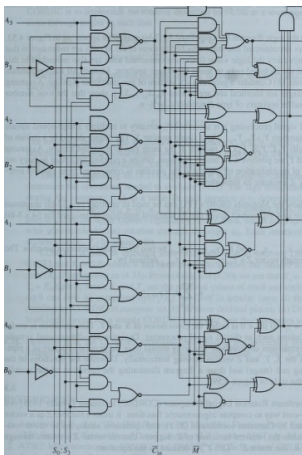
299

CHAPTER 4

Datapath

Design

Figure 4.65

Logic diagram for the 74181 ALU/function generator.

strategy to reduce the amount of shifting when \E\ > nM. (d) Introduce separate flagsOVR_ERROR and UND_ERROR to indicate overflow and underflow, respectively:these flags replace ERROR.

4.30. (a) List the advantages and disadvantages of designing a floating-point processor inthe form of a £-stage pipeline, (b) A floating-point pipeline has five stages 5,, S2. S3,54, and S5 whose delays are 120, 90, 100, 85, and 110 ns, respectively. What is the

300 pipeline's maximum throughput in millions of floating-point operations per second

(MFLOPS)9

SECTION 4.5Problems

4.31. Consider the logic diagram for a pipelined 3 x 3-bit multiplier appearing in Figure 4.53.(a) The six unconnected line stubs attached to some of the M cells are redundant in thatthey always carry the logic value 0. Certain connected lines are also redundant in thissense and are included only to make the stages uniform. Identify all such redundantconnections, (b) Consider a general n x n version of this multiplier pipeline. Assumingthat the stages are identical and are labeled S0, Sl,...,Sn_], show that the total numberof 1-bit buffer registers of type R needed is 3n2 - n.

4.32. In digital signal processing it is sometimes necessary to multiply a high-speed streamof rt-bit numbers Yx, Y2, Y3,... by a single number X. The output should be a stream of«-bit results YyX, Y2X, K3X,... moving at the same rate as the input stream. Assumingthat X and Yt are positive n-bil binary fractions, design a pipeline processor to carry outthis type of multiplication efficiently. If the pipeline is constructed from gates of aver-age delay d, estimate its throughput.

4.33. Outline how the Motorola 68040's FPU can be used to multiply two 32-bit mantissasto produce a 64-bit product, given that its mantissa multiplier is designed for 64 x 8-bitnumbers.

4.34. Let X = x0,x{, ...,*„_ i and Y=y0,yl,...,yn_l be two fixed-point vectors of length n. Thedouble-length vector Z = Zq, zt,..., z2n_2, ^2n-\ defined by

*< = 2**,-

xy,

where Xj = V: = 0 if/ < 0 is called the convolution of X and Y. This operation is usefulin applications such as digital signal processing. Design a one-dimensional systolicarray to implement convolution. The array should have the general structure of a pipe-line with the X, Y, and Z vectors flowing horizontally. Describe the functions of theprocessing cell (stage) and draw a diagram illustrating the operation of the systolicarray in the style of Figure 4.59.

4.35. The Coordinate Rotation Digital Computer (CORDIC) technique [Voider 1959] is afast, low-cost way to compute trigonometric functions. It treats a number Z as a vectorrepresented by Cartesian coordinates (X, Y), and operations analogous to vector rota-tion calculate the required functions of Z. Suppose that the vector Z is rotated throughan angle 9. The result Z' = (X\ Y') is defined by the equations

X' = Xcos9 ±ysin9

r=ycos9TXsine (4.49)

where the upper and lower signs correspond to clockwise and counterclockwise rota-tion, respectively. These equations imply that

X"=X7cos0=X± ytanG

y" = r/cos9 = y ± X tan9 (4.50)

Z" = (X", Y") can be interpreted as the original vector Z after rotation through anangle 9 and a magnitude increase by the factor K = l/cos9. If tanO is a power of 2, then

the multiplication by tan8 in (4.50) can be realized by shifting. The essence ofCORDIC is to implement the rotation described by (4.50) as a sequence of n + 1 rota-tions through angles a, such that

and

9 = ot0± a, ± a2± ...± a„a,- = tan"1 (2"')

(4.51)(4.52)

Then if Z = (X0, Y0), rotation through angle a, is defined by (4.51) and (4r§2) and hasthe form

Yut = Y +X2-

(4.53)

The resulting vector Zn has magnitude Kn\Z0\, where Kn - n"_ 0(cosa,)_1 is a constantdepending on n, which converges toward 1.6468. Observe that the only operations in(4.53) are addition, subtraction, and shifting.

The signs appearing in (4.51) depend on 9 and must be computed in order todetermine the operations needed to evaluate (4.53). The sign computation is done bystoring the constants {a,} in a table. In each iteration it is determined which of +<x,and -a, causes 19 + (oCq ± a, ± ...± a,)l to converge toward zero. If +cc, (-a,) isselected, then the upper (lower) signs in (4.51) are used, which correspond to a clock-wise (counterclockwise) rotation through the angle a,. Each iteration increases theaccuracy of (X,, Y,) by about 1 bit.

CORDIC is used to calculate sin9, cos9, and tan9 as follows: Let X0 = Kn_x ~0.6073 and Y0 = 0.0, where n has been chosen to achieve the desired accuracy. Com-pute (Xn, Y) according to (4.53). From (4.49) and (4.50) we see that Xn = A^cosO andYn = A'^f0sin9; hence Xn and Yn are the required values of cos9 and sin9, respectively,tan9 can now be computed by YJXn. (a) Give in tabular form all the calculationsrequired by CORDIC to compute sin 33° to three decimal places, (b) Draw a register-level logic circuit for a simple CORDIC arithmetic unit that computes sin 9 and cos9.

4.36. Describe how the CORDIC technique presented in the preceding problem can beadapted to compute the inverse trigonometric functions sin""1*, cos"1*, and tan'*.

4.6 REFERENCES

301

CHAPTER 4

Datapath

Design

1. Anderson, S. F. etal. "The IBM System/360 Model 91: Floating-Point Execution Unit."IBM Journal of Research and Development, vol.2 (January 1967) pp. 34-53.

2. AT&T Microelectronics. HS600C and LP600C CMOS Standard Cell Libraries DataBook. Allentown, PA, April 1994.

3. Booth, A. D. "A Signed Binary Multiplication Technique." Quarterly Journal ofMechanics and Applied Mathematics, vol. 4, pt. 2 (1951) pp. 236-40.

4. Cavanagh, J. J. F. Digital Computer Arithmetic. New York: McGraw-Hill, 1984.

5. Edenfield, R. W. et al. "The 68040 Processor: Part I, Design and Implementation."IEEE Micro vol. 10. (February 1990) pp. 66-78.

6. GEC Plessey Semiconductors. Digital Signal Processing IC Handbook. Swindon, UK,1990.

7. Hansen, M. C. and J. P. Hayes. "High-Level Test Generation Using Physically-InducedFaults." Proc. 13th VLSI Test Symp. Princeton, NJ, 1995, pp. 20-28.

8. Hintz, R. G. and D. P. Tate. "Control Data STAR-100 Processor Design." Proc. 6thIEEE Computer Soc. Conf. (Compcon 72), San Francisco, CA, September 1972, ppr 1-4.

SECTION 4.6 References

302 9. Integrated Device Technology Inc. "16 x 16 parallel CMOS multipliers IDT7216L/

IDT7217L." data sheet, Santa Clara, CA, August 1995.

10. Johnson, K. T., A. R. Hurson, and B. Shirazi. "General Purpose Systolic Arrays." IEEEComputer, vol. 26 (November 1993) pp. 20-31.

11. Kampe, T. W. "The Design of a General-Purpose Microprogrammable Computerwith Elementary Structure." IEEE Transactions on Electronic Computers, vol. EC-9 (June 1960), pp. 208-13.

12. Kane, G. and J. Heinrich. MIPS RISC Architecture. Englewood Cliffs, NJ: Prentice-Hall,1992.

13. Kogge, P. M. The Architecture of Pipelined Computers. New York: McGraw-Hill, 1981.

14. Koren, I. Computer Arithmetic Algorithms. Englewood Cliffs, NJ: Prentice-Hall, 1993.

15. Mick, J. and J. Brick. Bit-Slice Microprocessor Design. New York: McGraw-Hill, 1980.

16. Motorola Inc. MC68881/MC68882 Floating-Point Coprocessor User's Manual. Engle-wood Cliffs, NJ: Prentice-Hall, 1989.

17. Robertson, J. E. "Twos Complement Multiplication in Binary Parallel Computers." IRETransactions on Electronic Computers, vol. EC-4 (September 1955) pp. 118-19.

18. Stone, H. S. High-Performance Computer Architecture. 3rd ed. Reading, MA: Addison-Wesley, 1993.

19. Texas Instruments Inc. The TTL Logic Data Book. Dallas, TX, 1988.

20. Voider, J. E. "The CORDIC Trigonometric Computing Technique." IRE Transactionson Electronic Computers, vol. EC-8 (September 1959) pp. 330-34.

21. Wallace, C. S. "A Suggestion for a Fast Multiplier." IEEE Transactions on ElectronicComputers, vol. EC-13 (February 1964) pp. 14-17.

CHAPTER 5

Control Design

In this chapter we study the register-level design of the control part of aninstruction-set processor; the data-processing part was covered in Chapter 4.The two basic approaches to control-unit design—hardwired and micropro-grammed—are discussed in detail. The complex task of controlling pipelinedand superscalar processors is also examined.

5.1

BASIC CONCEPTS

First we discuss the general structure and behavior of control units. Then we exam-ine the design of hardwired controllers, which are characterized by the use of fixed(nonprogrammable) logic circuits.

5.1.1 Introduction

We saw in section 2.1.1 that it is useful to separate a digital system into two parts:a datapath (data processing) unit and a control unit. The datapath is a network offunctional and storage units capable of performing certain (micro) operations ondata words. The purpose of the control unit is to issue control signals to the data-path. These control signals enter the datapath at "control points" where they selectthe functions to be performed at specific times and route the data through theappropriate parts of the datapath unit. In other words, the control unit logicallyreconfigures the datapath to implement some specified instruction or program.

A CPU's datapath contains circuits to perform arithmetic and logical opera-tions on words such as fixed-point or floating-point numbers. The internal sfruc-

SECTION 5.1 Basic Concepts

A Data

Instructions
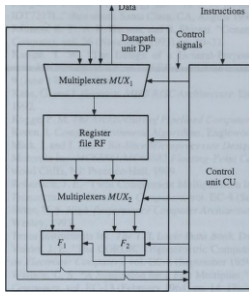
Figure 5.1

Processor composed of a datapath unit DP and a control unit CU.

ture of the datapath circuit DP of a small microprocessor is depicted in Figure 5.1. It contains a register file RF for temporary storage of operands, two functional units Fx and F2 responsible for data processing, and multiplexers to allow the data to be steered through DP. Typical functional units are an ALU performing addi-tion, subtraction, and logical operations; a shifter; or a multiplier. The control unit CU receives external instructions or commands, which it converts into a sequence of control signals that the CU applies to DP to implement a sequence of register-transfer operations.

Figure 5.2 shows the control signals that implement an addition instruction of the form ADD A,B, which we write as

A:=A + B;

(5.1)

in our HDL notation. Assume that this operation can be executed in a single clock cycle, whose timing details are not of concern at this level of abstraction. The input variables A and B are obtained from registers of the same name in RF, and the result is stored back into register A. Observe that the registers of RF permit their contents to be read from and written into in the same clock cycle, a basic property of the (edge-triggered) flip-flops from which such registers are constructed. RF is configured with one input and two output ports to support operations like (5.1) with two or three addresses. Besides selecting the data registers to be used, the con-trol unit CU must also select the operation to be performed on the data, in this case, functional unit Fr's ADD operation. Finally the necessary logical connections for the data to flow through DP must be established by applying appropriate control signals to the multiplexers.

ADDA.B

Active

DP

controlsignals 1

\
AB

1

I

A

Select p-t

Write ARead AReadB

p q r s MUXX .\ \ 1 CU

Select u-wSelect v-x

AddOverflow

A + B

f   RF   ■«—

1 ,

A w   i b   ',

i u v MUX-,

\ W X~^ v ;   r

A B
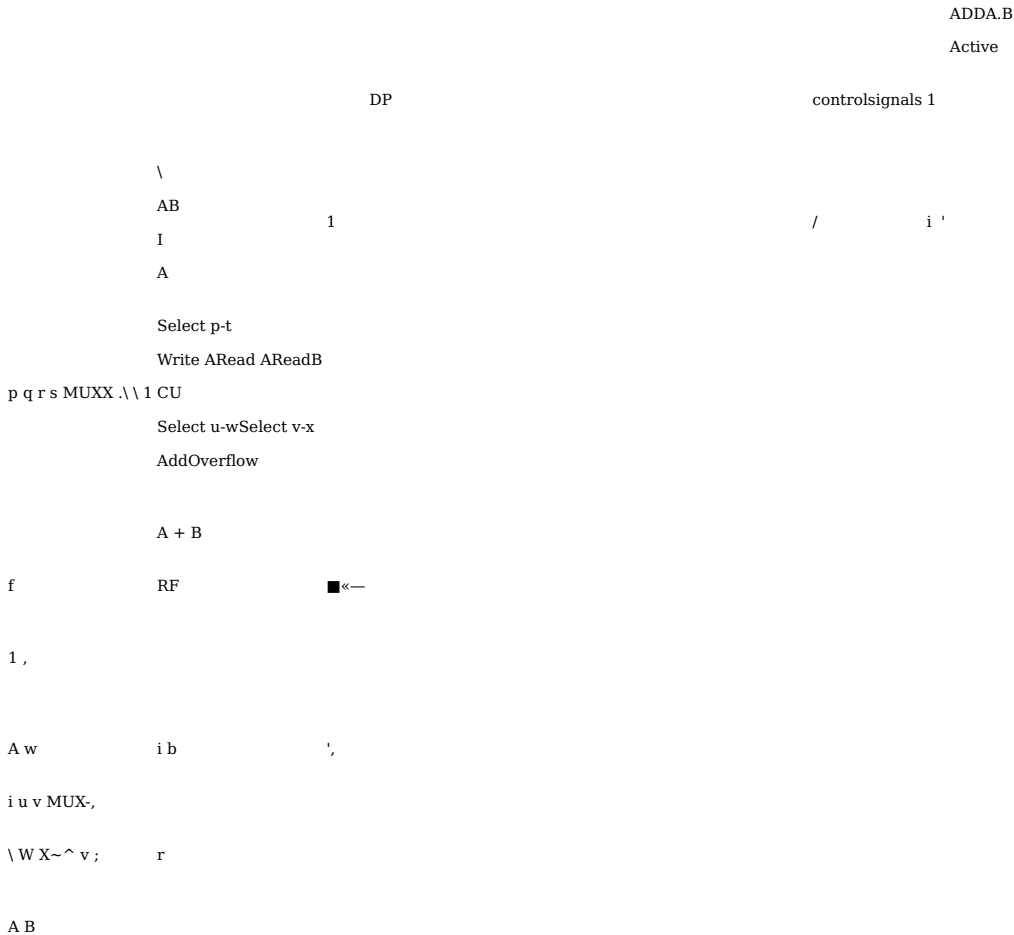
".' 1   F2   Figure 5.2

♦■1   The processor of Figure 5.1 configured to implement the

add operation A := A + B.

CHAPTER 5 Control Design

Thus we see that CU must activate the following three types of control signals during the clock cycle in which the ADD A,B instruction is executed.

• Function select: Add.

• Storage control: Read A, Read B, Write A.

• Data routing: Select p-t, Select u-w, Select v-x

There is usually some feedback of control information from DP to CU to indicate exceptional conditions encountered during instruction execution. In the example of Figure 5.2, the functional unit F, performing the addition sends an overflow signal to CU whenever the sum A + B exceeds the normal word size.

Multicycle operations. Many types of instructions are executed in a single clock cycle—indeed, single-cycle execution is a central goal of RISC design. Some instructions require more than one clock cycle for their execution, however. For example, double-precision addition can be implemented by a two-instruction sequence (program) of the form

ADD ADDC

AL, BL AH, BH

(5.2)

which involves two double-word operands A and B. The first (ADD) instruction in (5.2) adds the low-order half (right word) of B to the low-order half of A, implicitly generating and storing a carry-out signal C. The second (ADDC) instruction adds the high-order half of B to the high-order half of A along with the carry C. thus ensuring that carries are propagated across the full double-length result. This short program is implemented in two consecutive clock cycles by activating the control signals listed below, not all of which appear in Figure 5.2.

SECTION 5.1 Basic Concepts

Cycle

Function select

Add

Storage control

Data routing

Read AL. Read BL. Select p-t. Select u-w, Select v-x Write AL

Add with carry Read AH, Read BH, Select p-t. Select u-w. Select v-x Write AH

(5.3)

This low-level description of the double-precision addition in terms of the control signals to be activated is an example of a microprogram and is contrasted with the higher-level program for the same operation appearing in (5.2). Each line of (5.3) is an example of a microinstruction specifying a set of low-level microoperations. A further complication arises when the execution of a microoperation is condi-tional on the values of certain data or control signals. For example, the various sequential multiplication algorithms covered in the preceding chapters are speci-fied by multistep algorithms that can be viewed as multicycle microprograms. The Booth multiplication algorithm (Figure 4.15), for instance, has statements of the following type:

LOOP:

OUTPUT:

if COND1 = true then ADD A,B else SUB A,B;

if COND2 = true then go to OUTPUT else LOOP;

We can expand the microinstruction format of (5.3) to accommodate condi-tional operations in the following straightforward (but inefficient) way:

Current address

Condition select C

Next address

Function select

C = true C *■ true C = true C * true

Storage control

Data routing

ADR1   COND1 ADR2 ADR2

ADR2 ADR3 COND2 ADR3 ADR1

ADD

SUB

(5.4)

Here we are introducing some new fields to specify a condition C to be tested, as well as alternative control signals to be activated depending on the current value of C. Typically, C corresponds to a status control signal from DP, or to a special sig-nal generated within CU, such as an end-of-loop condition. If, for instance, C = COND1 in the preceding example, then one of the two function-select signals, ADD or SUB, is activated. To vary the order in which the microinstructions are executed, a pair of next-address fields is also provided, one of which is selected by the current value of C. This technique requires attaching an address to every micro-instruction, thus completing the analogy between microinstructions and higher-level (assembly language) formats.

Implementation methods. Historically, two general approaches to control unit design have evolved. One approach views the controller as a sequential logic cir-cuit or finite-state machine that generates specific sequences of control signals in response to externally supplied instructions; see Figure 5.3a. It is designed with the usual goals of minimizing the number of components used and maximizing the speed of operation. Once the unit is constructed, the only way to implement changes in control-unit behavior is by redesigning the entire unit. Such a circuit is therefore said to be hardwired. The format of (5.4) is essentially similar to the state-table format for describing the behavior of a (hardwired) sequential circuit, as illustrated in (5.5).

CHAPTER 5 Control Design

Current state

ADR1 ADR1 ADR2 ADR2 ADR3

Current input

Next state

Current outputs

Function	Storage	Routing
CONDI = 1	ADR2	ADD
CONDI =0	ADR2	SUB
COND2 = 1	ADR3	
COND2 = 0	ADR1	

(5.5)

Microprogramming provides an alternative method of designing program con-trol units. A microprogrammed control unit has the structure shown in Figure 5.3fr.It is built around a storage unit called a control memory, where all the control sig-nals are stored in a programlike format resembling (5.4). The control memorystores a set of microprograms designed to implement or emulate the behavior of thegiven instruction set. Each instruction causes the corresponding microprogram tobe fetched and its control information extracted in a manner that resembles thefetching and execution of a program from the computer's main memory.

Controlsignals   Statussignals Addresslogic —*■ Controlmemory

Sequentiallogiccircuit

Statussignals   i   1 i   I   '   r

Microinstructionregister

Decoder

j   i

1 L_   Control

w   signals

Instructionregister

Instruction

' »o
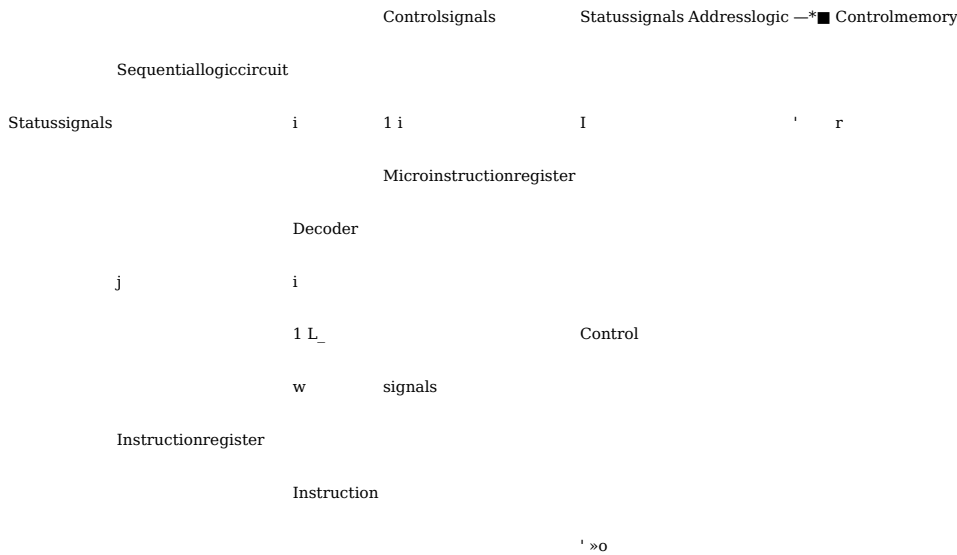
(a)

(b)

Figure 5.3

General structure of (a) a hardwired and (b) a microprogrammed control unit.

308 Microprogramming makes control unit design more systematic by organizing

section 5 control signals into formatted words (microinstructions). Since the control signals

Basic Concepts are embedded in a kind of low-level software—this is referred to as firmware—

design changes can be easily made just by altering the contents of the control mem-ory. On the negative side, microprogrammed control units are more costly to man-ufacture than hardwired units due to the presence of the control memory and itsaccess circuitry. Microprogrammed units also tend to be slower because of theextra time required to fetch microinstructions from the control memory. RISC pro-cessors, with their emphasis on small, fast instruction sets, favor the use of hard-wired control units.

CPU control units, both hardwired and microprogrammed, are often organizedas (micro) instruction pipelines in order to improve their performance. As we sawin section 4.3.2, pipelining is a relatively low-cost way of increasing a processor'sthroughput by decomposing its operation into a sequence of relatively independentsteps. Program control naturally involves a sequence of steps (instruction fetchingand decoding, input operand fetching, operation execution, and result storage) thatcan be carried out concurrently with different instructions. Modern CPUs makeextensive use of pipelines to increase their effective instruction execution rate[Stone 1993].

5.1.2 Hardwired Control

Next we examine the design of control units that use fixed logic circuits to interpretinstructions and generate control signals from them.

Design methods. Control-unit design involves various trade-offs between theamount of hardware used, the speed of operation, and the cost of the design processitself. To illustrate these issues, we consider two systematic approaches to thedesign of hardwired controllers [Hayes 1993: Baranov 1994]. These methods arerepresentative of those used in practice, but by themselves are suitable only forsmall control units such as might be encountered in simple RISC processors orapplication-specific controllers.

• Method 1: The classical method of sequential circuit design, which was dis-cussed briefly in section 2.1.3. It attempts to minimize the amount of hardware,in particular, by using only $|\sim\log2/>\sim|$ flip-flops to realize a F-state circuit.

• Method 2: An approach that uses one flip-flop per state and is known as the one-hot method. While expensive in terms of flip-flops, this method simplifies CUdesign and debugging.

In practice, processor control units are often so complex that no one design methodby itself can yield a satisfactory circuit at an acceptable cost. The most acceptabledesign may consist of several linked, but independently designed, sequential circuits.

State tables. The behavior required of a control unit, like that of any finite-state machine, can be represented by a state table of the general type shown in Fig-ure 5.4a. The rows of the state table correspond to the set of internal states {5,}.These states are determined by the information stored in the machine at discretepoints of time (clock cycles). Let X and Z denote the input and output variables.

Inputs

State h	h	Im
«i su,oU]	512, 0]2 . . .	*l,m« ^l^n
S2 "2,1. ^2,1 ■^2.2' ^2,2		S^ 02.m
Sn Ski. o„,	s& on2 . . .	\.m' OnJn

Inputs

State 'l	h	im	Outputs
S.	S1.1	Sl2	S\jn	0\
S2	^2.1	S2,2	$2*,	02
sn	SnA	Sn.2	•^n,m	0„

309

CHAPTER 5 Control Design

(*)Figure 5.4

State tables for a finite-state machine: (a) Mealy type and (b) Moore type.

The columns correspond to the combinations of the X signals that can be applied to the machine and are denoted here by {/,}. The entry in row 5, and column L has the form SjpOjj, where S,7 is the next state of the machine that results from the applica-tion of input combination /, and 0{ denotes the output signals that appear on Z whenever the machine is in state S, with input / applied. In general, an entry in the state table defines a specific, one-cycle transition between two states.

Control units have a feature that favors a slightly different style of state table: Their output signal values often depend on the current state 5, only and so are inde-pendent of the input combination L. If all outputs are of this type, the circuit is called a Moore machine, in contrast with the more general Mealy machine of Fig-ure 5.4a. (These names honor G. H. Mealy and E. F. Moore who were early researchers into finite-state machine theory [Mealy 1955; Moore 1956].) The state table of Figure 5.4a becomes a Moore machine if for every row i, we have Otj -Oik= 0, for all j, k = 1,2,...,m. In that case we can represent the machine's behavior in the more compact format of Figure 5.4b, where the output signals associated with each row are placed in a separate column.

GCD processor. To illustrate the classical and one-hot approaches to control-unit design, we will apply them to a special-purpose processor that computes the greatest common divisor gcd(X,Y) of two positive integers X and Y\ gcd(X,Y) is defined as the largest integer that divides exactly into both A" and Y. For example, ga/(12,18) = 6, and gcd{\2,\l) = 1. It is customary to assume that gcd(0,0) = 0.

We use a variant of Euclid's algorithm [Cormen, Leisersor., and Rivest 1990] to calculate gcd(X,Y). Figure 5.5 gives an HDL description of this method.

310

SECTION 5.1 Basic Concepts

```
gcd(in: X,Y; out: Z);

register AT?, YR, TEMPR;

XR := X;                {Input the data}

YR := Y;

while XR > 0 do begin                        «

UXR<YR then begin      {Swap AT? and K/?}

TEMPR := YR,

YR := XR.

XR := TEMPR: end

XR := AT? - YR;        {Subtract YR from AT?}

end

Z := YR,end go/;       {Output the result}
```

Figure 5.5

Procedure gcd to compute the greatest common divisor of two numbers.

The basic idea is to subtract the smaller of the two numbers from the other repeatedly—recall that division corresponds to repeated subtraction—until we obtain a number that divides the other. For example, with X = 20 and Y = 12, our gcd algorithm proceeds as follows:

Conditions		Actions
		A7?:=20; YR := 12:
AT?>0:	XR > YR:	XR:=XR-YR = 8;
AT?>0:	XR < YR:	YR := 8; XR := 12; XR = XR-YR = 4,
AT?>0:	XR<YR:	YR := 4: AT? := 8; XR = XR-YR = 4;
AT?>0	XR < YR:	YR := 4; AT? := 4; XR = XR-YR = 0;
AT?<0:		Z:=4;

Hence we conclude that gcd(20,\2) = 4.

Analysis of the gcd procedure suggests that its datapath unit DP should contain a pair of registers XR and YR to store the corresponding variables, one or more functional units to perform subtraction and magnitude comparison, and multiplex-ers for data routing, as indicated in Figure 5.6. We do not need to include a register for the

"temporary" variable TEMPR, as we would in a typical programmed imple-mentation, because we can read from and write to a register in the same clockcycle. The swap operation can therefore be done without conflict in one cycle thus:

X:=Y, Y:=X; (5.6)

The control unit CU generates control signals Load XR and Load YR to loadeach register independently with the input data X and Y. A control signal Select XYroutes X and Kto XR and YR, respectively. Another signal Swap controls the swapoperation defined by (5.6), which requires routing the outputs of the XR and YRregisters to each other's inputs. A final signal Subtract is assumed to control thesubtraction XR := XR - YR by routing the output of the subtracter to XR. The inputsignals to CU are an asynchronous Reset signal, two comparison signals (XR > YR)and (XR > 0) generated by DP, and the usual, implicit clock signal.

We can identify a set of states for CU by examining the behavior defined in theHDL specification (refer to Figure 5.5)—a simple process here, but one that is

X Y

Reset

T T T I T

Multiplexers MUX

Register XR

Register YR

Subtractor

Comparators

Datapath unit DP

Controlunit CU

SubtractSwapSelect XY

LoadXR

LoadYR

(XR > YR)(XR > 0)

Figure 5.6

Hardware needed to implement the gcd procedure.

tedious and error-prone in the case of large control units. A start state 50 is enteredwhen Reset becomes 1; this state also loads X and Y into the DP registers. The sub-sequent actions of the gcd processor are either a swap or a subtraction, for whichwe define the states 5, and S2, respectively. A final state 53 is entered whengcd(X,Y) has been computed. Figure 5.7 gives a Moore-type state table defining theCU's behavior. Each state transition is deduced directly from the HDL description.If the input control signal (XR > 0) = 0, indicating that the while loop should beskipped, a transition is made from S0 to 53; this yields the first next-state entry inthe top row of Figure 5.7. If, on the other hand, (XR > 0) = 1, the while loop isentered, and a transition is made to S, to perform a swap if (XR > YR) = 0; other-wise, the transition is to S2 to perform a subtraction. The latter case defines to thethird entry of the state table, whose input combination is (XR > 0)(XR > YR) =11.

311

CHAPTER 5Control Design

State	Inputs (XR > 0)(XR > YR) 0-	10 11	mmmmtmmmammmummm Outputs Subtract	Swap	Select XY	LoadXR	LoadYR
S0 (Begin)	s3	5, St	0	0	1	1	1
5, (Swap)	s2	Si s2	0	1	0	1	1
S2 (Subtract)	s3	5, S2	1	0	0	1	0
53 (End)	s3	S3 S)	0	0	0	0	0

Figure 5.7

State table defining the control unit of the gcd proce^or.

312 Since a subtraction always follows a swap, all next-state entries in the second row

section 5 are ^2' ^e corresPonding active outputs are the two register-load signals Load XR

Basic Concepts an(* L°ad ^' al°ng witn Swap, which route the outputs of XR and YR to YR and

XR, respectively. The next states for state S2 are the same as those for S0; the activeoutputs are Subtract, which routes the output XR - yR of the subtracter to XR, andLoadXR. The final state S3 is assumed to be a "dead" state that is unaffected by allinputs (except Reset) and produces no active outputs.

Classical method. The major steps of the classical design method are as fol-lows:

1. Construct a P-tow state table that defines the desired input-output behavior.

2. Select the minimum number p of D-type flip-flops and assign a p-bil binarycode to each state.

3. Design a combinational circuit C that generates the primary output signals {z,}and the secondary outputs {D,} that must be applied to the flip-flops.

We now apply this method to the design of the control unit CU for the gcd pro-cessor. We have already constructed the necessary state table (Figure 5.7). Sincethere are four states, we require two flip-flops, whose outputs DXDQ = y^Q defineCU's internal states. We assign the binary patterns to the four states in the follow-ing obvious way:

S0 = 00

S,=0 11 (5.7)

52=1053=1 1

We note in passing that the state assignment pattern affects the complexity of thecircuit in subtle ways.

At this point we can construct a binary version of the state table, the excitationtable, as shown in Figure 5.8. The D flip-flop's characteristic equation D,+ (r + 1) =£>,(?) defines the inputs Dx* and D0+ to the flip-flops. CU's combinational logic Ccan now be derived from the excitation table using any available manual or auto-matic method. Suppose, for instance, that we use two-level sum-of-products (SOP)minimization. It is easily checked that C is defined by the following SOP equa-tions, which lead directly to the design of Figure 5.9. Note that all gates in anAND-OR SOP circuit can be changed to NANDs to produce a NAND-NAND real-ization of the original function.

Dy+ = (XR >0) + (XR> YR) + D0

D0+ = DXDQ+ (XR>XR)Do +(XR>0)-DoSubtract = DlD0 (5.8)

Swap = DlD0Select XY= DXDQLoadXR= D0 + D,LoadYR= D,

Inputs		Present state	Next state				Outputs		Select	Load	Load
(XR> 0)	(XR> YR)	£>i	D0	*V	n	+uo	Subtract	Swap	XY	XR	YR
0	d	0	0	1	1	0	0	1	1		1
0	d	0	1	1	0	0	1	0	0		1
0	d	i	0	1	1	1	0	0	0		0
0	d	l	1	1	1	0	0	0	0	0	0
1	0	0	0	0	1	0	0	1	1		1
1	0	0	1	1	0	0	1	0	0		1
1	0	1	0	0	1	1	0	0	0		0
1	0	1	1	1	1	0	0	0	0	0	0
1	1	0	0	1	0	0	0	1	1		1
1	1	0	1	1	0	0	1	0	0		1
1	1	1	0	1	0	1	0	0	0		0
1	1	1	1	1	1	0	0	0	0	0	0

Figure 5.8

Excitation table for the control unit of the gcd processor.

One-hot method. While the classical design method minimizes a controlunit's memory elements, its effect on the amount of combinational logic C is lessobvious. Furthermore, control units designed by this technique tend to have a com-plicated, "random" structure, which makes design debugging and subsequentmaintenance of the circuit difficult. An alternative approach that simplifies the

Reset

(XR > 0)

(XR > YR)

>

4>

_>

_>

^ry^n>

zv

>CK

CLR b-

D>CK

CLR t>-

Clock

Figure 5.9

All-NAND classical design for the control unit of the gcd processor.

>>

SubtractSwap

LoadXR

Select XYLoadYR

314 design process and gives C a regular and predictable structure, is the one-hot

method, so called because its binary state assignment always contains a single 1—

Basic Concepts tne "not" bit—while all the remaining bits are 0. Thus the state assignment for a

four-state machine like the gcd processor takes the following form:

S0 = 0 0 0 1

(5.9)

52 = 0 1 0 0

53 = 1 0 0 0

In general, P flip-flops are needed to represent P states, so the one-hot method isrestricted to fairly small values of P.

A key feature of this technique is that the next-state and output equations have a simple, systematic form and can be written down directly from the control unit's original symbolic state table. Because the binary pattern assigned to each state is, in effect, fully decoded, we can find out whether the machine is in state S_i merely by inspecting the corresponding hot state variable D_i. The classical method requires us to check all state variables to get this information.

Suppose that state S_i in a one-hot design has the hot variable D_i. Further, suppose that $I_{j1}, I_{j2}, ..., I_{jn}$, denote all input combinations that cause a state transition from S_j to S_i. Then each AND combination of the form D_jI_{jv} must make $D_i = 1$. Hence, considering all such combinations that cause transitions to S_i, we can write

$$D_i{+} = D_1(I_{11} + I_{12} + ... + I_{1n_1}) + D_2(I_{21} + I_{22} + ... + I_{2n_2}) + ... \quad (5.10)$$

This immediately yields the SOP form

$$D_i = D_1I_{11} + D_1I_{12} + ... + D_1I_{1n_1} + D_2I_{21} + D_2I_{22} + ... + D_2I_{2n_2} + ...$$

which is practical to implement by an AND-OR or NAND-NAND circuit, provided that each state transition is determined by relatively few states and input variables, as is common in control-unit behavior. Equation (5.10) can also lead directly to fairly simple factored forms. Consider the state table of Figure 5.7 for the gcd processor's CU. State 5, appears as a next state only for S_0 and S_2, in each case with the input combination $(XR > 0)(XR > XR)$. Hence (5.10) becomes

$$ZV = D_0 \cdot (XR > 0) \cdot (XR > XR) + D_2 \cdot (XR > 0) \cdot (XR > XR)$$

$$= (D_0 + D_2) \cdot (XR > 0) \cdot (XR > XR)$$

The primary output equations are even easier to derive for one-hot designs. If output signal z_k is 1 (active) only in rows k_h for $h = 1, 2, ..., m_k$, then we have

$$z_k = D_{k1} + D_{k2} + ... + D_{km_k} \quad (5.11)$$

De Morgan's law of Boolean algebra allows us to rewrite this OR equation as

$$z_k = \overline{\overline{D_{k1}} \cdot \overline{D_{k2}} \cdots \overline{D_{km_k}}}$$

in which form it can be generated by a single NAND whose inputs are the complemented outputs of the flip-flops. In the gcd processor case, output Load $YR = 1$ in states S_0 and S_1 only; therefore

$$\text{LoadYR} = D_0 + D_1 = \overline{\overline{D_0}\,\overline{D_1}}$$

The entire set of next-state and output equations obtained by applying (5.10) and (5.11) to the gcd processor's CU follows.

CHAPTER 5 Control Design

$$D_1{+} = D_0 \blacksquare (XR > 0) \cdot (XR > XR) + D_2 \cdot (XR > 0) \blacksquare (XR > XR)$$
$$D_2{+} = D_0 \blacksquare (XR > 0) \cdot (XR > XR) + D_1 + D_2 \blacksquare (XR > 0)(XR > XR)$$

$$D_2{+} = D_0(XR{>}0) + D_2(XR{>}0) + D_3$$

Subtract $= D_2$

Swap $= D_1$

Select $XY = D_0$

LoadXR $= D_0 + D_1 + D_2$

Load YR $= D_0 + D_1$

A NAND implementation of these equations appears in Figure 5.10. Note that the asynchronous Reset line must set D_0 to 1 and all other state variables to 0.

The steps of the one-hot design method for a Moore machine can be summarized as follows:

1. Construct a p-row state table that defines the desired input-output behavior.

2. Associate a separate D-type flip-flop D_i with each state S_i, and assign the p-bit one-hot binary code $D_0D_1...D_{i-1}D_iD_{i+1}...D_p = 0,0,...,0,1,0,...,0$ to S_i.

3. Design a combinational circuit C that generates the primary and secondary output signals $\{D_i\}$ and $\{z_k\}$, respectively. $D_i{+}$ is defined by the logic equation

$$D_i{+} = \sum_j D_j (I_{j1} + I_{j2} + \blacksquare \cdot + I_{jn_j})$$

where $I_{j1}, I_{j2}, ..., I_{jn}$ denote all input combinations that cause a transition from S_j to S_j. If $z_k = 1$ (active) only in rows k_h for $h = 1, 2, ..., m_k$ then z_k is defined by

$$z_k = D_{k1} + D_{k2} + \blacksquare + D_{km_k} = \overline{\overline{D_{k1}}\,\overline{D_{k2}}...\overline{D_{km_k}}}$$

We next present an example that illustrates the application of the one-hot method to a computer's IO interface, specifically to a direct-memory access (DMA) controller, which handles data transfers between main memory and high-speed IO devices. (DMA communication is discussed in Chapter 7.)

example 5.1 design OF A DMA CONTROLLER. This problem, which is adapted from [Actel 1994], is representative of control units that link several interacting systems—in this case, main memory and a set of IO devices. The target machine is the control part of a four-channel DMA controller of the kind found in the IO subsystem of most computers. It is a six-state Moore-type machine with four input and five output signals, which are identified as follows:

Inputs. IOREQ Any of four data-transfer request signals

CONT Continue (indicates pending, unprocessed requests)

$(XR > 0)$

$(XR > YR)$
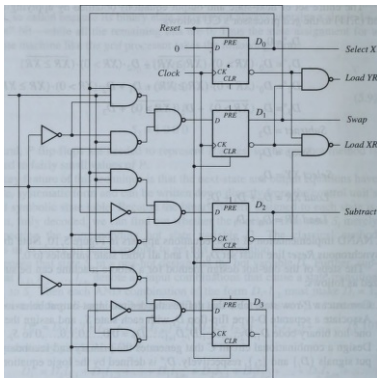
Subtract

Figure 5.10

All-NAND one-hot design for the control unit of the gcd processor.

MACK Memory transfer acknowledgment

PBGNT Processor bus grant (indicates availability of data-transferbus)

Outputs: CE Count enable (bookkeeping function)

CMREQ Channel memory request

CNTLD Counter load (bookkeeping function)

RLD Register load (bookkeeping function)

PBREQ Processor bus request for control of data-transfer bus

The behavior of the DMA controller is given by the state transition diagram of Figure5.11a. Each transition is marked with the corresponding active input control signals.Since every transition is triggered by only one such signal, this notation is quite com-pact. Each state is marked with the (boxed) name of the output control signals that itactivates—the number of such signals ranges from zero to two. A state table in thestyle of Figure 5.4 that is equivalent to Figure 5.1 la is easy to construct, but it is largebecause of the many possible input combinations. Noting that most input signals do notaffect a given state transition, and so are assigned the don't-care value d, we can con-dense the state table into the compact form of Figure 5.1 lb.

IOREQ

MACK

\rld JcONT

1   fs5\
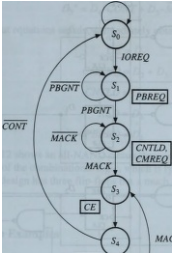
CMREQ |

MACK

(a)

Inputs				Present	Next		Outputs				
IOREQ	COAT	MACK	PBGNT	state	state	PBREQ	CNTLD	CMREQ	RLD	CE	
0	d	d	d		So	So	0	0 0 0 0			
1	d	d	d		So	5,	0	0 0 0 0			
d	d	d	0		5,	Si	1	0 0 0 0			
d	d	d	1		Si	s2	1	0 0 0 0			
d	d	0	d		S2	s2	0	1 1 0 0			
d	d	1	d		s2	S3	0	1 1 0 0			
d	d	d	d		S3	S4	0	0 0 0 1			
d	0	d	d		s*	So	0	0 0 1 0			
d	1	d	d		s*	s,	0	0 0 1 0			
d	d	0	d		Ss	5,	0	0 1 0 0			
d	d	1	d		Ss	S3	0	0 1 0 0			

317

CHAPTER 5Control Design

(b)

Figure 5.11

State behavior of the DMA controller: (a) state transition graph and (b) condensedstate table.

318

SECTION 5.1Basic Concepts

IOREQ

PBGNT 1

MACK

CONT

PBREQ
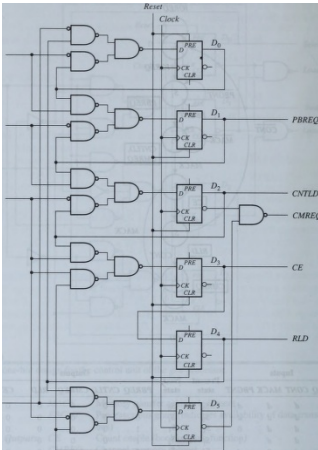
Figure 5.12

One-hot design for the DMA controller.

Now we assign a D flip-flop D, to each state S, and write down the six state-transition equations directly from Figure 5.11.

ZV = D0- IOREQ + D4- CONT

D,+ = DQ- IOREQ + D, • PBGNTD2+ = DrPBGNT+ Dr MACKD{ =D2MACK + D5- MACK

D4+ = D} 319

Ds^DyCONT+DsMACK CHAPTER 5

Control DesignThe output equations are also immediately obtained from Figure 5.11.

CE = D3

CMREQ = D2 + D5

CNTLD = D2

RLD = Dir

PBREQ = D{

Figure 5.12 shows an all-NAND circuit derived from these equations. Note the regularstructure of the combinational logic, which is typical of one-hot designs. An equivalentclassical design has three flip-flops but a much more irregular combinational part.

5.1.3 Design Examples

This section presents some examples to illustrate the foregoing methods for design-ing hardwired control units. We will use these examples again in our discussion ofmicroprogrammed control.

Multiplier control. First consider the design of a control unit CU for the twos-complement (Robertson) multiplier introduced in Example 4.2 (section 4.1.2). Theblock diagram of the multiplier's datapath unit DP (Figure 4.12) is redrawn inexpanded form in Figure 5.13 to show a set of control points, which representabstractly the control signals and associated logic circuits needed to link CU andDP. These control signals are derived from the multiplication algorithm in Figure4.13 and are listed in Figure 5.14. In general, a control point can be associated witheach distinct action (register-transfer operation) opt appearing in the algorithmbeing implemented. Its enabling control signal c, is inserted into the component orinterconnections associated with op,. Operations that take place simultaneouslymay be able to share control signals. (Procedures to eliminate redundant controlsignals are considered later.) The statement labeled BEGIN in Figure 4.13, forinstance, requires the registers A, COUNT, and F to be reset simultaneously to theall-zero state. A single control signal c10 is therefore provided for this purpose. Itcan be connected directly to the CLEAR inputs of the three registers in question, sono additional logic is needed to implement the c,0 control point. Control signals c8and c9 transfer a data word from the input bus INBUS to registers Q and M. respec-tively, and are shown in the corresponding data paths of Figure 5.13; these signalsmay be connected to the registers' (parallel) LOAD inputs. Control signal c5 servesto change the function performed by the parallel adder from addition to subtractionfor the correction step; it also resets Q[0] to 0. The remaining control signals ofFigure 5.14 are defined similarly. Figure 5.13 introduces a control signal calledCOUNT7, which is set to 1 when COUNT = 111, and is set to 0 otherwise.COUNT7, the right-most bit Q[0] of the multiplier register Q and the externalBEGIN signal serve as the primary inputs to CU.

320

SECTION 5.1Basic Concepts

c5— *.

ClO — *

Signlogic

1 ?

r 4 I

Accumulator i ' ' Multiplier register

c6

Q[0]

c3 c4

I I

._J L_.

Paralleladder

c-j *6 <W- -

OUTBUS

8-bitdatabuses

INBL'S

" Q[0]

r BEGIN •

External

control < END ■*

signals

V CLOCK ►

M[7]

Multiplicand register

M

<n

-- eg

COUNT7

_L_

Controlunit

Comparator

1-7

COUNT

■— Cio

6* c9

■>

Cfl

Internal

». c, V control

signals

►■ Mo

Figure 5.13

Twos-complement multiplier with a set of control points.

The multiplication algorithm is reformulated as a flowchart in Figure 5.15 todisplay the control signals from Figure 5.14 and indicate when each one is acti-vated. The flowchart resembles a state transition graph that describes the behaviorof both the control and datapath units. To obtain a state table for the control unitCU, we associate a state S, with every operation block in Figure 5.15. leading to theseven states labeled S^.Sj. An additional state 50 represents the reset or waiting con-dition of the control unit. CU has three primary input signals—BEGIN. Q[0], andCOUNT7; hence there are eight possible input combinations. Figure 5.16 shows aneight-state Moore-type state table in the style of Figure 5.7, which is deriveddirectly from Figure 5.15. This state table is not necessarily the smallest such tabledefining the desired control function. In fact, the 13 output control signals c0:cn,END can be immediately reduced to 8 because several sets are equivalent in thatthey are always activated together, specifically c0 = c^ = cn, c2 = c3 = c4, and c9 =

.__-_-.___ 321

Control

Signal Operation controlled CHAPTER 5

c0 Set sign bit of A to F.

c. Right-shift register-pair A.Q.

c2 Transfer adder output to A.

c3 Transfer A to left input of adder.

c4 Transfer M to right input of adder.

c5 Perform subtraction (correction). Clear Q[0].

c6 Transfer A to output bus.

c7 Transfer Q to output bus.

c8 Transfer word on input bus to Q.

c9 Transfer word on input bus to M.

c10 Clear A, COUNT, and F registers. Figure 5.14

c,, Increment COUNT. Control signals for the twos-

END Completion signal (CU idle). , , !*•_«•

_ & ' complement multiplier.

c10. Methods also exist that attempt to reduce the number of states by merging"compatible" states; see Problem 5.8. To eliminate a flip-flop in this case, wewould need to reduce the number of states of CU from eight to four or fewer, whichis not possible.

In the following example, we apply the two basic hardwired design techniques,classical and one hot, to the multiplier, taking Figures 5.15 and 5.16 as startingpoints.

EXAMPLE 5.2 IMPLEMENTING A MULTIPLIER CONTROL UNIT. The multi-plier control unit CU is small enough that the classical design approach can be appliedto it. This technique uses the minimum number of flip-flops, in this case three, whoseoutputs D2DXDQ denote CU's internal state. We make the natural assignment of thebit-pattern D2DXD0 = i to state S,, yielding 50 = 000, 5, = 001, 52 = 010, and so on.The remaining problem is to design the combinational logic circuit portion C of CU,a straightforward but fairly tedious task, because C has six inputs BEGIN, Q[0],COUNT7,D2,D,,D0and 11 outputs: D2+,DpD0V0,C2,c5,c6,<:7,<:g,c1,,END.

The behavior of CU's combinational logic C is defined by an excitation table,which is obtained by replacing the symbolic states of Figure 5.16 with the correspond-ing 3-bit patterns. It remains to design C. Figure 5.17 shows the results of applying thetwo-level optimization program espresso to this problem. The espresso program imple-ments an efficient algorithm that finds a minimum or near-minimum number of prod-uct terms (prime implicants), in this case only 13, needed to define C. The inputdescription (Figure 5.17a) is a 64-row excitation table for C written in espresso's PLA-style format. For example, the last row of Figure 5.17a

111111 00000001000

specifies the state transition from 57 to 50, with input combination BEGIN Q[0]COUNT7 =111 and active output c1 = 1. The last row of espresso's solution (Figure5.176)

— 110 11100010000

specifies the transition from S6 to 57, with input combination BEGIN Q[0] COUNT7 =— denoting don't cares, and active output signal c6 = 1. This captures the fact that Sb isalways followed by 57, independent of CU's primary input signals. A NAND realiza-tion of the multiplier CU appears in Figure 5.18. which directly realizes the minipiumSOP form obtained via espresso.

Control Design

SECTION 5.1Basic Concepts

We can also implement CU directly from the state table of Figure 5.16, or, equallyeasily, from the flowchart of Figure 5.15 by the one-hot method. Eight flip-flops areneeded to accommodate CU's eight states S0:S7. Equations (5.10) and (5.11) give thenext-state and output equations. The next-state equations are

D0+ = D0- BEGIN + D-

ZV = D,

D3+ = D2Q[0] + D4-0[O]- COUNT!

D4+ = D2 Q[0] + D3 + D4- Q[0] ■ COUNT!

D5+ = D4Q[0]COUNT7

D6+ = D5 + D4- Q[0] ■ COUNT!

D7+ = D6

c

J

1 ,, c9> c10

A:=0COUNT := 0

F:=0M := INBUS

Q := INBUS
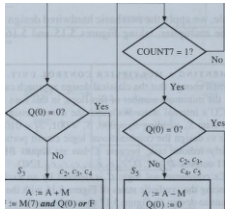
A := A + MF := M(7) and Q(0) or F

co.q.c,,

A(0) := F

A(6:0).Q:= A.Q(7:1)

COUNT := COUNT+1

OUTBUS := Q

OUTBUS

f End J

Cycle 0

Cycles 1 to 7

Figure 5.15

Flowchart for the twos-complement multiplier.

Cycle 8

Cycle 9

Inputs: BEGIN Q[0] Count7

Outputs

State 000 001 010 Oil 100 101 110 111 c0 c, c, c, c, c. c, c7 c« c, c1ft c,, END

SoS2

s*s4

S*

SoS2S3

s*

S3

•Jfl Sn

So

s2

S3

s,s5s6s7sn

Si

s2s4

S4S4

5, 0000000000001

52 0000000001100

53 0000000010000

54 0011100000000

55 11000000000105656 00111100000005757 0000001000000Sn 5n 0000000100000

Figure 5.16

State table for the multiplier control unit.

.model.inputs.output

.i 6

.0 11

.p 64

000000

001000

010000

011000

100000

101000

110000

111000

000001

001001

010001

011001

100001

101001

110001

111001

000010

001010

mult007.pla

BGN Q0 CT7 D2 Dl DOs D2+ D1+ D0+ cO c2 c5 c6c7 c8 c9 END

00000000000000000000000100001000010000100010000100001000010000100001000010000100010001000010000

00000100000100000100000100000100000100000100000100001000010000100001000010000100001000010000100001000010000100000100

101111110111

111111

.end

00000001000000000001000000000001000

(a)

model   milt005.pla

.inputs BGN Q0 CT7 D2 Dl DO

.output s D2+ D1+ D0+ cO c2 c5 c6

          c7 c8 c9 END

.i 6

.0 11

.p 13

-011-0 01000000000

1--000 00100000000

-10100 01110000000

-11100 10110000000

---111 00000001000

-0-010 10000000100

-0-100 10010000000

---011   10001000000

-1-010 01100000100

---000   00000000001

---001   01000000010

---101   11001100000

---110   11100010000

. e

(*)

Figure 5.17

Design of combinational part of the multiplier control unit by espresso: (a) input data(excitation table) and (b) output data (optimized SOP specification).

The output equations are

c0- c, - cu - D4

c7 = D7cg = D2

END = D0

The NAND circuit implied by these equations appears in Figure 5.19.

Despite having more flip-flops, the one-hot design is better in many ways than theclassical design. The one-hot design has fewer and generally smaller gates in its combi-

COUNT7
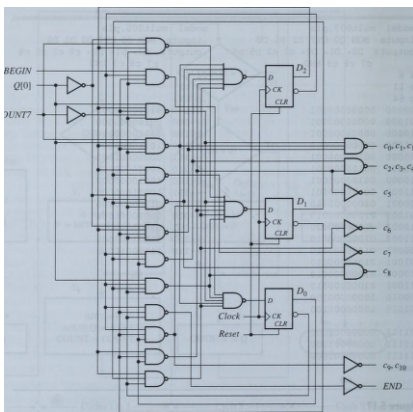
Figure 5.18

All-NAND classical design for the multiplier control unit.

national logic because entry to a particular state depends on a small number of primaryand secondary (state) input variables—as few as one state variable in several cases.This dependence also holds for the output functions, since most of the primary outputsignals can be taken directly from the corresponding hot-state variable. Another pointworth noting is that the one-hot CU's structure closely follows that of its state behav-ior, as exemplified by the flowchart specification (Figure 5.15). Consequently, the one-hot design is easier to understand and easier to modify (should that be necessary) thanthe classical design.

BEGIN

Q[0]

COUNT7
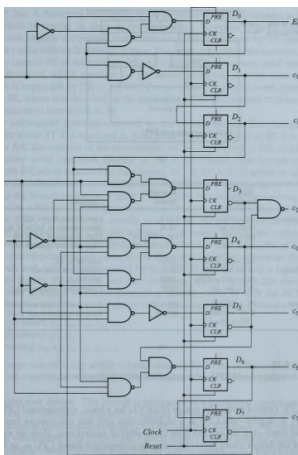
END

c9> cio

c2' c3' c4

co.ci.cu

Figure 5.19

All-NAND one-hot design for the multiplier control unit.

Sequential circuit

Control signals

no r~AR~i i pc 'i

Program control unit PCU

To M and IO devices

A_

System bus

DR

AC

rr ti

Arithmetic-logic unit

Data processing unit DPU

(a)

Type	HDL format	Assembly format		Comment
Data transfer	AC := M(X)	LDX		Load X from M into AC
	M(X) := AC	STX		Store contents of AC in M as X
	DR := AC	MOV DR,	AC	Copy contents of AC to DR
	AC := DR	MOV AC,	DR	Copy contents of DR to AC
Data	AC	= AC + DR	ADD	Add DR to AC
processing	AC	= AC - DR	SUB	Subtract DR from AC
	AC	= AC and DR	AND	And DR to AC
	AC	= not AC	NOT	Complement contents of AC
Program	PC := M(adr)	BRA adr		Jump to instruction with address adi
control	if AC = 0 then PC := M(adr)	BZ adr		Jump to instruction adr if AC = 0

(b)

Figure 5.20

An accumulator-based CPU: (a) organization and (b) instruction set.

CPU control unit. The design of the CU for a basic, nonpipelined CPU differs mainly in degree—the CPU is a multifunction unit that can contain hundreds of control lines—but not in kind from the multiplier control unit. Here we examine a few of the design issues involved, using an accumulator-based CPU as an example. In section 5.3.3 we will discuss the complex control problems associated with pipe-lined CPUs.

The accumulator-based CPU introduced in section 3.1.1 has the overall organi- 327 zation depicted in Figure 5.20a (which repeats Figure 3.3). This CPU consists of a datapath unit DPU designed to execute the set of 10 basic single-address instruc-tions listed in Figure 5.20b. The instructions are assumed to be of fixed length and to act on data words of the same fixed length, say 32 bits. The program control unit PCU is responsible for managing the control signals linking the PCU to the DPU, as well as the control signals between the CPU and the external memory M.

To design the PCU, we must first identify the relevant control actions (micro-operations) needed to process the given instruction set using the hardware from Figure 5.20a. A flowchart description of the behavior of the CPU appears in Figure 5.21, which is similar in form to the multiplier's flowchart (Figure 5.15). All instructions require a common instruction-fetch step, followed by an execution step that varies with each instruction type. The fetch step copies the contents of the program counter PC to the memory address register AR. A memory-read operation is then executed, which transfers the instruction word / to memory data register DR; this is expressed by DR := M(AR). /'s opcode is transferred to the instruction register IR, where it is decoded; at the same time PC is incremented to point to the next consecutive instruction in M.

The subsequent operations depend on the opcode pattern. For example, the store instruction ST X is executed in three steps: the address field of ST X is trans-ferred to AR, the contents of the accumulator AC are transferred to DR, and finally the memory write operation M(AR) := DR is performed. The branch-on-zero instruction BZ adr is executed by first testing AC. If AC * 0, no action is taken; if AC = 0, the address field adr, which is in DR(ADR), is transferred to PC, thus effecting the branch operation. Figure 5.21 implies that instruction fetching takes three cycles, while instruction execution takes from one to three cycles. As we will see later, RISC processors are usually designed so that all instruction execution times are equalized to one CPU clock period T_c in length, making the cycles asso-ciated with the register-transfer operations in Figure 5.21 into subcycles of T_c.

The microoperations appearing in the flowchart implicitly determine the con-trol signals and control points needed by the CPU. Figure 5.22b lists a suitable set of control signals for the CPU and their functions, while Figure 5.22a shows the approximate positions of the corresponding control points in both the PCU and DPU. These control lines can be placed in the three basic groups defined earlier.

• Function select: c_2, c_9, c_{10}, cM, c_{12}.

• Storage control: c,, c_8.

• Data routing: c_0, c_3, c_4, c_5, c_6, c_7.

Here storage control refers to the external memory M. Many of the control signals transfer information between the CPU's internal data and control registers.

Control unit design. The overall organization of a hardwired control unit that implements the flowchart of Figure 5.21 appears in Figure 5.23. It is assumed that the opcode stored in the instruction register IR is decoded into 10 signals, one per instruc-tion type, which along with BEGIN and a status signal (AC = 0) form the inputs to the main sequential circuit FSM that generates the control signals c_0:c_n. Hence FSM has 12 primary inputs and 13 primary outputs. The number of internal states can be esti-

mated from Figure 5.21. If each distinct action box is assigned to a different state.

r-\

w

£

U

.< /

k

~3!3

U   ^n

<

a: >•c   a:Q<

CO

iv;

Qii

H   oo   fc

U

0   <r

£   o

ll

-    U

00   <

a:

Q   Q

R   ■*s

s

<   QU

<

o   II

00   U<

a:

m   Q

5

U
00   <

a: fe

(- », Q "£   ll

«)S *s -a _   U<

BSS I
regiield Q a:n
s Recode
ter

aloraddrdataode: ion (opemory■n coun < +u<

E = oO i

o S a ai

3 § E n " IR = InstruM = MainPC = Progr QU< 90 00 u<

ii n n o <

>
u a: a: a: a o
2

a:Qii
r)
<<QQC

u

< *r <

a:
Q 00
>

U<
O2 ll

a:

C

s~\ >< 00

P a;

—1 H u U

B 00 a: <■

H Q II

n a: a:<

u
X < 05 oo 2
■o

a? — a? £ a?

/ \ c U < + C o O < a:

^/?SYL 0. ? Q J a:a ? U

, ' V> o/ a: n ii n u

•fl- sc U ,J a: ry; <r

^/ 03 >: a oo £ a: oo < 05 0 oo

a,U
in ig
o

3

^ Control

circuit • • ^ signals

I

i>

'5 -

1 IR 1   AR | PC [

C4^i3R(OP)   \   DRiADR)

M andIO devices

ci

Systembus

c8

^

1

i

u

i   t ■ „

c6 ~f>)
c9 —
C|0   Arithmetic-logic unit
cn
c12
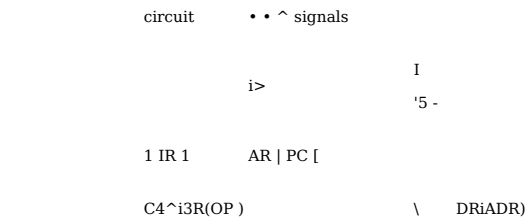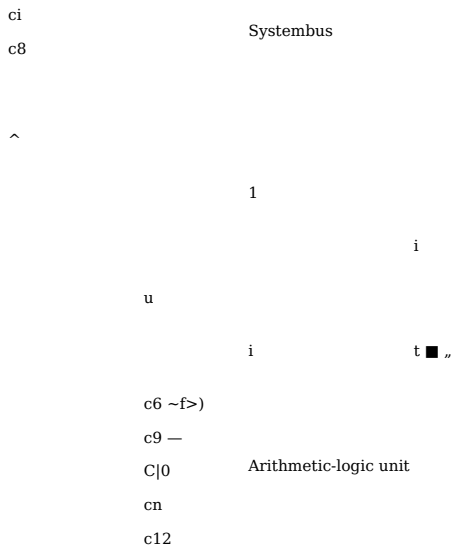
h
329

CHAPTER 5 Control Design

Control signal Operation controlled

Co   AR   = PC

C| DR:=M(AR)

<2 PC:=PC+ 1

c3   PC := DR(ADR)

Q   IR:=DR(OP)

c5   AR   = DR(ADR)

c6   DR   = AC

Cj   AC   = DR

Cg   M(AR) := DR

c9   AC   = AC + DR

CI0 AC   = AC - DR

cn   AC   = AC and DR

c12 AC   = not AC

(a)

Figure 5.22

(a) Control points and (b) control signal definitions for the accumulator-based CPU.

(b)

Statussignals

BEGIN

(AC = 0)

Finite-state

machine

FSM

*

i   i i v i i

LD ST

Decoder   BZ

it

Controlsignals

Instructionregister

IR

Figure 5.23

Organization of a hardwired control unit for the accumulator-based CPU.

330

SECTION 5.1 Basic Concepts

then there are 13 states S0:Sl2, as indicated on the flowchart. Implementing the result-ing 13-state machine via the classical or one-hot methods is straightforward.

EXAMPLE S.3 IMPLEMENTING A PROGRAM CONTROL UNIT. The circuit

FSM of Figure 5.23 issues the control signals governing instruction processing in the 10-instruction accumulator-based CPU. Its behavior is defined by the flowchart in Fig-ure 5.21. We will implement FSM using a minor variant of our earlier one-hot method that reduces the number of flip-flops needed, while maintaining the simplicity of a one-hot design. Most of the states S4:S]2 identified in Figure 5.21 for the execution phase of the instruction can be distinguished by the opcode-type signals LD, ST, and so on, which are primary inputs of FSM. If we do not require FSM to be a Moore machine, we can coalesce the states into a smaller set whose output actions (the control signals they activate) are determined by FSM's primary inputs as well as its states. Specifically, we can replace the states in the execution phase by just three states 54*,55*,56*, all of which are visited in sequence by the load and store instructions, but which reduce to a single state 54* for the remaining instructions. We can also reduce the instruction-fetch phase to a sequence of three states by merging 50 and Sl into one state Sx* so that, whether active or inactive, FSM performs the operation AR := PC.

The resulting machine has the state behavior depicted by a Mealy-type state transi-tion graph in Figure 5.24. This figure follows the condensed style of Figure 5.11 whereonly the signals that directly affect or are affected by each state are shown. For exam-ple, the state transition graph implies that when in state S5*, a transition is made to 56* with output cx = 1 (the only active output) if the current instruction is of type LD. This event is indicated by the label LD/c, on the 55*-to-56* transition arrow. If the current instruction is ST, however, then only c6 becomes 1, as indicated by the label ST/c6. No other instruction types allow FSM to enter state S5*. When the transition from state 5, to Sj is automatic, that is. it is independent of the primary input signals, we used a label of the form 0/0,.

Let us implement FSM using six D flip-flops, with the output D, of the ith flip-flop forming the hot variable for state 5, or S*. We can now write down a set of logic equations directly from Figure 5.24 that define FSM.

BEGINIc0

MOVl/c6;

MOV2/c7;

ADD/c9;

SUB/c,0:

AND/c,,;

NOT/c12;

BRA/c3;

BZ and (AC = 0)/c3

BZ and (AC = O)/0
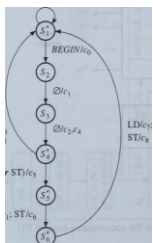
(LD or ST)/c5

LD/c,; ST/c6

Figure 5.24

State transition graph for the accumulator-based CPU.

$D_{,+} = D, BEGIN + D4(M0V1 + M0V2 + ADD + SUB + AND$

$+ NOT + BRA + BZ) + D6D2+ = DV BEGIN$

331

D3+- -.D2

d;- :D3

D5+- = D4-(LD + ST)

D6+- -.D5

UK >ns also follow immedia

c0 = D,

c2 = c4-- :D3

c5 = D4 (LD + ST)

cl = D4 MOV2 + D6LD

c9 = D4 ADD

ci   = D2 + DyLD

c3   = D4 (BRA + BZ •(AC: = 0))

c6 = :D4   MOV1 + »5 ST

c8: -D6   ST

c10: ■-D4 SUB

c12: --D4   NOT

These equations lead to the logic circuit in Figure 5.25.

CHAPTER 5Control Design
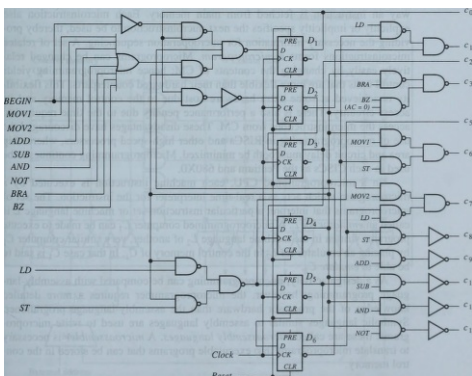
Figure 5.25

One-hot implementation of the CPU state transition graph of Figure 5.24.

332

SECTION 5.2

Microprogrammed

Control

5.2

MICROPROGRAMMED CONTROL

We turn next to the design of control units that use microprograms to select, inter-pret, and execute a processor's instruction set.

5.2.1 Basic Concepts

An instruction is implemented by a sequence of one or more sets of concurrentmicrooperations. Each microoperation is associated with a group of control linesthat must be activated in a prescribed sequence to trigger the microoperations. Asthe number of instructions and control lines can be in the hundreds, a hardwiredcontrol unit is difficult to design and verify, even with good CAD tool support.Furthermore, such a control unit is inherently inflexible in the sense that changes,for example, to correct design errors or update the instruction set, require that thecontrol unit be redesigned.

Microprogramming [Lynch 1993] is a method of control-unit design in whichthe control signal selection and sequencing information is stored in a ROM orRAM called a control memory CM. The control signals to be activated at any timeare specified by a microinstruction, which is fetched from CM in much the sameway an instruction is fetched from main memory. Each microinstruction alsoexplicitly or implicitly specifies the next microinstruction to be used, thereby pro-viding the necessary information for microoperation sequencing. A set of relatedmicroinstructions forms a microprogram. Microprograms can be changed rela-tively easily by changing the contents of CM; hence microprogramming yieldscontrol units that are more flexible than their hardwired counterparts. This flexibil-ity is achieved at some extra hardware cost due to the control memory and itsaccess circuitry. There is also a performance penalty due to the time required toaccess the microinstructions from CM. These disadvantages have discouraged theuse of microprogramming in RISCs and other high-speed processors, where chiparea and circuit delay must both be minimized. Microprogramming continues to beused in such CISCs as the Pentium and 680X0.

In a microprogrammed CPU, each machine instruction is executed by amicroprogram which acts as a real-time interpreter for the instruction. The set ofmicroprograms that interpret a particular instruction set or machine language L iscalled an emulator for L. A microprogrammed computer C, can be made to executeprograms written in the machine language L2 of another, very similar computer C2by placing an emulator for L2 in the control memory of Cj. In that case C, is said tobe able to emulate C2.

As a design activity, microprogramming can be compared with assembly-lan-guage programming; however, the microprogrammer requires a more detailedknowledge of the processor hardware than the assembly-language programmer.Symbolic languages similar to assembly languages are used to write micropro-grams: these are called

microassembly languages. A microassembler is necessary to translate microprograms into executable programs that can be stored in the con-trol memory.

Control unit organization. In its simplest form a microinstruction has two parts: a set of control fields that specify the control signals to be activated and an

address field that contains the address in CM of the next microinstruction to be executed. In the original scheme proposed by Maurice V. Wilkes, the inventor of microprogramming, each bit k_i of a control field corresponds to a distinct control line c_i [Wilkes 1951]. When $k_i = 1$ in the current microinstruction, c_i is activated: otherwise c_i remains inactive. Figure 5.26 shows a microprogrammed control unit designed in this style. The control memory CM is implemented by a ROM of the type discussed in section 2.2.2. The left part (AND plane) of the ROM decodes an address obtained from the control memory address register (CMAR). Each address selects a particular row in the right part (OR plane) of the ROM that contains a microinstruction composed (in this small example) of a 6-bit control field and a 3-bit address field. When the top-most row in Figure 5.26, which represents the microinstruction with address 000, is selected, the control signals c_0, c_2, and c_4 are activated, as indicated by the xs in the control field. At the same time, the contents of the address field $a_2 a_x a_Q = 001$ are sent to the CMAR, where they are stored and used to address the next microinstruction to be executed.

As Figure 5.26 indicates, the CMAR can be loaded from an external source as well as from the address field of a microinstruction. The external source typically provides the starting address of a microprogram in the CM. A specific micropro-gram prestored in CM executes (interprets) each instruction of a microprogrammed

333

CHAPTER 5 Control Design

External condition

Control memory vCM

o-

o

D-D-D-D-

Control field c5 c, c3 c2 c, c0

-x *-

Address field a2 fl| oq

-x x

o

D—T

^MaA WWW K?W

I L_ Control signals

CMAR

/ Mux \

tt:

/ /

External address

Figure 5.26

Basic structure of a microprogrammed control unit.

334

SECTION 5.2

Microprogrammed

Control

CPU. The instruction's opcode, after suitable encoding, provides the starting address for its microprogram.

Every program control unit should be able to respond to external signals or conditions. We can satisfy this requirement by introducing some form of switch S controlled by an external condition that allows the current microinstruction to select one of two possible address fields. Thus in Figure 5.26, the third micro-instruction may be followed by the microinstruction with address 100 or 101, as determined by the external condition. This feature makes conditional branches within a microprogram possible.

Many modifications to the preceding design have been proposed over the years. A major area of concern is the microinstruction's word length, since it greatly influences the size and cost of the CM. Microinstruction length is deter-mined by three factors:

• The maximum number of simultaneous microoperations that must be specified, that is, the degree of parallelism required at the microoperation level.

• The way in which the control information is represented or encoded.

• The way in which the next microinstruction address is specified.

Control memories are usually ROMs, so their contents cannot be altered on-line. Normally there is no need to change the CM except to correct design errors or to make minor enhancements to the system. It was recognized from the beginning, however, that the CM could be a read-write memory or RAM. Wilkes observed that such a device, called a writable control memory (WCM), would have a number of "fascinating possibilities," but doubted that its cost could be justified [Wilkes1951]. Perhaps the most interesting feature of a WCM is that it allows us to change a processor's instruction set by changing the microprograms that interpret the instruction set. Thus we can, in principle, provide the same machine with several different instruction sets that can be tailored to specific applications. A processor with a WCM is said to be dynamically microprogrammable because the control memory contents can be altered under program control.

Parallelism in microinstructions. Microprogrammed processors are frequently characterized by the maximum number of microoperations that a single microin-struction can specify. This number ranges from one to several hundred.

Microinstructions that specify a single microoperation are similar to conven-tional machine instructions. They are relatively short, but due to their lack of paral-lelism, more microinstructions are needed to perform a given operation. The format of the IBM System/370 Model 145, which is shown in Figure 5.27, is repre-sentative of this type of microinstruction. It consists of 4 bytes (32 bits). The left-most byte (shaded) is an opcode that specifies the microoperation to be performed. The next 2 bytes specify operands, which, in most cases, are the addresses of CPU registers. The right-most byte contains information used to construct the address of the next microinstruction.

Microinstruction formats take advantage of the fact that, at the microprogram-ming level, many operations can be performed in parallel. If all useful combina-tions of parallel microoperations were specified by a single opcode, the number of opcodes would, in most cases, be enormous. Furthermore, an opcode decoder of considerable complexity would be needed. To avoid these difficulties, it is usual to

16

24

31

1 1 1 1 1 1 1   i i i i i i i   i i i i i i i   i

v. .A A. A J

V   V

Control field(opcode) Operand 1 Operand 2 CM addressinginformation

Microinstruction format of the IBM System/370 Model 145.

divide the microoperation specification part of a microinstruction into k disjointcontrol fields. Each control field handles a limited set of microoperations, any oneof which can be performed simultaneously with the microoperations specified bythe remaining control fields. A control field often specifies the control-line valuesfor a single device such as an adder, a register, or a bus. In the extreme case repre-sented by Figure 5.26, there is a 1-bit control field for every control line in the sys-tem.

Figure 5.28 shows another microinstruction style, that of the IBM System/360Model 50. It encompasses 90 bits, which are partitioned into separate fields for var-ious purposes. There are 21 fields, shown shaded in Figure 5.28, which constitutethe control fields. The remaining fields are used to generate the next microinstruc-tion address and to detect errors by means of parity bits. For example, the 3-bitcontrol field consisting of bits 65:67 controls the right input to the main adder ofthe CPU in question. This field indicates which of several possible registers shouldbe connected to the adder's right input. Bits 68:71 identify the function to be per-formed by the adder; the possibilities include binary addition and decimal additionwith various ways of handling input and output carry bits.

The scheme of Figure 5.26 with a control field for every control signal iswasteful of control memory space because most of the possible combinations ofcontrol signals are never used. Consider, for instance, the register R of Figure 5.29,

335

CHAPTER 5Control Design

0 1

19

40

45

J I I I I I I I I I I I L

J I I I I I l_

I I

Parity ofbits 0-30

VCM addressing information

Unused

IParity of bits 32-55

56

64

72

S3

89

Parity of bits 57-89

V^

CM addressinginformation Unused

Ninety-bit microinstruction format of the IBM System/360 Model 50 (shaded areas arecontrol fields).

336

SECTION 5.2

Microprogrammed

Control

which can be loaded from any of four independent sources under the control of thefour separate signals c_0, c_1, c_2, c_3, as indicated abstractly in Figure 5.29a. A straight-forward implementation of the associated control points using an encoder and amultiplexer appears in Figure 5.29b. Suppose that the c_i's are derived from amicroinstruction control field in which there is 1 bit for each control signal. Thisresults in the 4-bit control field shown in Figure 5.30a. Only the five control-fieldpatterns shown in Figure 5.30a are valid, since any other pattern will create a con-flict by attempting to load R from two or more independent sources simulta-neously. These five patterns can be also encoded into a field $K = k_f j c x k_2$ of width$|\sim \log 25] = 3$ bits, as shown in Figure 5.306, thus reducing the width of the controlfield from 4 to 3 bits. In general, any n independent control signals or microopera-tions can be encoded in a control field of $[\log_2(n + 1)]$ bits, assuming the need tospecify a no-operation condition when no control signal is active.

The unencoded format of Figure 5.30a has the advantage that all the controlsignals are individually identified in, and can be obtained directly from, the micro-instruction. The encoded control signals k_0, k_1, k_2 of Figure 5.306 must be passedthrough a decoder if we wish to extract the four original control signals c_0, c_1, c_2, c_3

x0 xl x2 x$
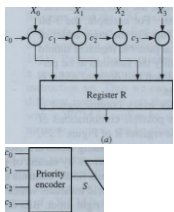
i i i L

4-way multiplexer

LOAD

Register RI

m

A register that can be loaded from four independent sources: (a) abstract representation and(b) possible implementation.

1 0 0 0 R:=X0

0 1 0 0 R:=X,

0 0 1 0 R:=X2

0 0 0 1 R:=X3

  0 0 0 0   No operation

(a)

*0 *1 *2

0 0 0 R:=X0

0 0 1 R:=X,

0 1 0 R:=X2

0 1 1 R:=X3

  1 0 0   No operation

(*)

Figure 5.30

Control field for the circuit of Figure 5.29: (a) unencoded format and (b) encoded format.

Often we can use the encoded control signals directly so that no decoding is 337needed. For example, in the present example, we can connect the two signals kxk2of Figure 5.306 directly to the select inputs S of the multiplexer in Figure 5.29b,thereby eliminating the priority encoder. The complemented control signal ko canthen be connected directly to the LOAD input of the register R to complete thedesign.

Horizontal versus vertical. Microinstructions are commonly divided into twotypes. Horizontal microinstructions have the following general attributes:

• Long formats.

• Ability to express a high degree of parallelism.

• Little encoding of the control information.

Vertical microinstructions, on the other hand, are characterized by

• Short formats.

• Limited ability to express parallel microoperations.

• Considerable encoding of the control information.

The format of the IBM System/360 Model 50 shown in Figure 5.28 is represent-ative of horizontal microinstructions, while that of the System/370 Model 145shown in Figure 5.27 is representative of vertical microinstructions.

Other definitions of horizontal and vertical are found in the literature. One isbased on the degree of encoding: a horizontal microinstruction format allows noencoding of control information, whereas a vertical format does. An alternativedefinition is based on the degree of parallelism. A vertical microinstruction canspecify only one microoperation (no parallelism), while a horizontal microinstruc-tion can specify many microoperations. These definitions are not independent,since a large amount of parallelism implies little encoding, and vice versa. Forexample, the format of Figure 5.31a is horizontal and that of Figure 5.31c is verti-cal under both of the preceding definitions.

Vertical microinstructions are broadly similar to RISC instructions, both in thesmall amount of parallelism they specify and in their single-cycle execution style.Computers have also been designed with long and highly parallel instruction for-mats that resemble horizontal microinstructions; see problem 5.36.

Microinstruction addressing. Each microinstruction in the basic design ofFigure 5.26 contains within itself the address of the next microinstruction to beexecuted. In the case of branch microinstructions, two possible next addresses areincluded. This explicit address specification has the advantage that no time is lostin microinstruction address generation, but it is wasteful of control memory space.The address fields can be eliminated from all but branch instructions by using amicroprogram counter |iPC as the primary source of microinstruction addresses.Its role is analogous to that of the program counter PC at the instruction level.Since only instructions have to be fetched from the control memory, |iPC is alsoused as the control memory address register CMAR.

Conditional branching is a desirable feature in microprograms just as it is inprograms, and it can be implemented in various ways. The condition to be tested isoften a status signal generated by the datapath being controlled. If several such

Control fields

VControl lines

(a)

Control fields

Decoder0

\\ !! i!

Decoder1

Single control field

Decoder

" TT TTTT TTTTTT"

(a)

i i i

i   ii '' ''

Decoder

J

V

Control lines

Control lines(c)

Figure 5.31

Control-field formats: (a) no encoding; (b) some encoding; (c) complete encoding.

conditions exist, a condition-select subfield is included in the microinstruction for-mat to specify which of the possible conditions is to be tested. The branch addresscan be in the microinstruction itself, in which case it is loaded into CMAR when abranch condition is satisfied. Control memory space can be conserved by not stor-ing a complete address field in the microinstruction, but by storing instead somelow-order bits of the address. This technique restricts the range of branch instruc-tions to a small region of the control memory.

An alternative approach to conditional branching is to allow the condition vari-ables to modify the contents of CMAR directly, thus eliminating wholly or in partthe need for branch addresses in microinstructions. For example, let the conditionvariable OVF indicate an overflow condition when OVF = 1, and the normal no-overflow condition when OVF = 0. Suppose we want to execute a SKIP ONOVERFLOW microinstruction. We can connect OVF to the count-enable input ofU.PC at an appropriate point in the microinstruction cycle, thereby allowing theoverflow condition to increment U.PC an extra time, thus performing the desiredskip operation.

Microoperation timing. So far we have assumed that a microinstruction acti-vates a set of control signals for an unspecified time during the microinstruction'sexecution cycle. A single clock signal synchronizes the control signals, and its

period can be the same as the microinstruction cycle period. This mode of controlhas been termed monophase. The number of microinstructions to specify a particu-lar operation can be reduced by dividing the microinstruction cycle into severalsequential subperiods or (clock) phases. A control signal is typically active duringonly one of the phases. This polyphase mode of operation permits a single microin-struction to specify a short sequence of microoperations for some increase in thecomplexity of the microinstruction format.

Consider a microinstruction that controls the register-transfer operation

R :=yCR„ R2)

where R can be R, or R2. This operation can be performed in several phases; thefollowing four-phase interpretation is representative.

• Phase Oj: Fetch the next microinstruction from the control memory CM.

• Phase 02: Transfer the contents of registers R, and R2 to the inputs of the/unit.

• Phase 03: Store the result generated by the / unit in a temporary register orlatch L.

• Phase 04: Transfer the contents of L to the destination register R.

Figure 5.32 shows the timing signals associated with these four phases.

We have also assumed that the influence of a microinstruction control field islimited to the period during which the microinstruction is executed. We can lift thisrestriction by storing the control field in a register that continues to exercise controluntil a subsequent microinstruction modifies it. This technique is called residualcontrol and is particularly useful when microinstructions are used to allocate theresources of a system. For example, a connection between two units can be estab-lished by a microinstruction and maintained for an arbitrarily long period of timevia residual control.

339

<t2.

4V

m

Generate next microinstructionaddress and fetch from CM

Gate input registers to/unit

Store result in temporary register

Gate result to output register

Microinstructioncycle

Figure 5.32

Timing diagram for a four-phase microinstruction.

Time

340

Conditionselect

Externaladdress

Externalconditions

Branchaddress

Controlfields

(a)

s   Branch address

Mux

<^   Load   ■'   ii

uPC

Incremt   Micropro

*- *   "

Control
L   memory
CM

Condi
tionion i   ,
select

A

1 1 1

J

Control

fields

Decoders

» 1   r   i   i
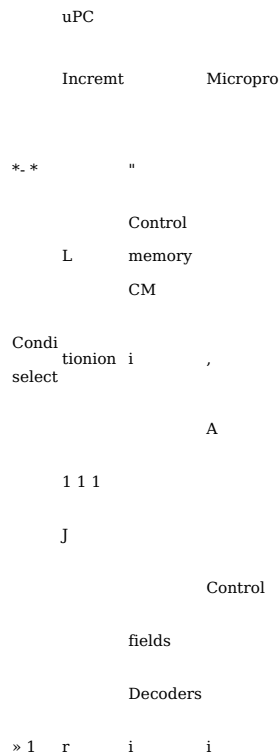
Microinstructionregister p.IR

Control signalsto data processing unit

(b)

Figure 5.33

Typical microprogrammed controller: (a) microinstruction format and (b) controlunit organization.

Control unit organization. We now describe the design of a typical micropro-grammed control unit. We use the microinstruction format shown in Figure 5.33a,which has three parts arranged as follows:

• A condition-select field specifies the external condition to be tested in the case ofconditional branch microinstructions.

• An address field contains the next-address field to be used when a branch condi-tion is satisfied. A microprogram counter (IPC provides the next microinstruc-tion address when no branching is needed.

• The rest of the microinstruction specifies in encoded or unencoded format thecontrol signals that are activated to perform the desired microoperations.

Figure 5.33b depicts a control unit designed around this microinstruction for- 341mat. The counter uPC is the address register for the control memory CM. The con-tents of the addressed word in CM are transferred to the microinstruction registeruTR. The control fields are decoded if necessary uPC and produce control signals for thedata-processing unit; [iPC is then incremented. If a branch is specified by themicroinstruction in |iIR, the contents of the microinstruction's address field areloaded into pPC.

In the scheme of Figure 5.33a, the condition-select field controls a multiplexerthat activates the parallel-load control input of pPC based on the status of someexternal condition variables. Suppose that two condition variables vx,v2 must betested. A condition-select field s0s] of 2 bits suffices, with the following interpreta-tion:

«0 *i Meaning

0   0 No branching

0   1 Branch if v, = 1

1   0 Branch if v2 = 1

1   1 Unconditional branch

The multiplexer has four inputs x0, xu x2, x3, where xt is routed to the multiplexer'soutput when SqS^ = L Hence we require x0 = 0, xx = vv x2 = v2, and a:3 = 1 to controlthe loading of microinstruction branch addresses into uPC in this case.

Finally, a provision is made for loading uPC with an address from an externalsource. This address is used to enter the starting address of the desired micropro-gram in cases where CM contains more than one microprogram.

EXAMPLE 5.4 THE AMD 2909 BIT-SLICED MICROPROGRAM SEQUENCER

[MICK AND BRICK 1980; ADVANCED MICRO DEVICES 1985]. Like the 2901 4-

bit ALU slice (Example 4.5), the 2909 microprogram sequencer is a member of the2900 family of microprocessor components, now found mainly in VLSI cell libraries.It generates microinstruction addresses for a control memory CM and comprises amicroprogram counter uPC and all the logic needed for next-address generation.Devices of this type are termed microprogram sequencers. The 2909 thus replaces uPCand the multiplexer appearing in Figure 5.33; it also adds a stack to implement subrou-tine calls at the microprogram level. Figure 5.34 shows the internal organization of the2909. It handles addresses that are only 4 bits long, thus limiting a single copy of the2909 to controlling a 16-word CM. However, the 2909 is bit sliced, so k copies ofthe 2909 can be cascaded to make a microprogram sequencer for 4^-word addresses.Three copies of the 2909 connected as in Figure 5.35 can process 12-bit addresses andsupport a 4096-word control memory.

The function of a microprogram sequencer is to transfer an address from one ofseveral internal and external address sources to an output bus—the 4-bit bus Y in the2909 case—that is connected to the address bus of CM. The 2909 has four separateaddress sources: its microprogram counter uPC, an external bus D, a register R that isattached to a second external bus, and a four-word internal stack ST. uPC is actuallyimplemented by a 4-bit register of the same name and by a separate incrementer, asshown in Figure 5.34. In every clock cycle this logic circuit performs the operation __

cout.uPC:=Y + cin (5.12)

where cin and cout are carry-in and carry-out signals, respectively. By connecting thecoul output line of each 2909 in an array of k 2909s to the cin input of the 2909 to its left,the operation (5.12) can be extended to addresses of arbitrary length.

If a sequence of microinstructions without branches is being executed, then (5.12)alone suffices for microinstruction sequencing. Many^ microprograms, however,involve some branching to nonconsecutive addresses in CM. A branch address is madeavailable as the address of the next microinstruction by connecting the appropriateaddress field of the current microinstruction in the external microinstruction registerp.IR to the 2909's D or R bus in the manner of Figure 5.33b. The stack ST serves as the

Inputaddress R

Inputaddress D

Registerenable RE

Register R

Addressselect S

OR,

4 x 4-bitstack ST

MultiplixerMUX

OR2OR,ORoCarry- «out c„„,

ZERO

Stack

enable FE

Push/popcontrol PUP

Microprogramcounter uPC

WW

nil

Incrementer
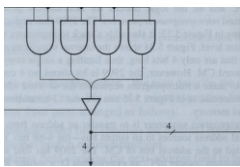
Outputenable OE

Output address Y

Figure 5.34

Structure of the 2909 microprogram sequencer.

Carry

Input addressR

12-'

Input addressD

343

2S -*—*■

12

OR -* 2

ZERO

4-' 4-'

■4

R

D

ORo:OR3

coul

ZERO

2909Y

4/

4-'

R D

SORo:OR3

cout c

ZERO

2909Y

Conditionselect

4-' 4-'

CHAPTER 5Control Design

Miscellaneouscontrol

12-'fOutput address Y

Figure 5.35

Bit-sliced microprogram sequencer for 12-bit addresses.

remaining address source. ST is intended to support subroutine (procedure) calls withinmicroprograms. CALL X is implemented by pushing the contents of u,PC into ST andtaking the next address X from the D or R source. A subsequent return from the micro-subroutine requires popping ST into uPC. Four addresses can be stored in ST, whichallows up to four procedure calls to be nested within a microprogram.

The four possible address sources—(J.PC, D, R, and ST—are connected to a multi-plexer MUX which, as shown in Figure 5.34, is controlled by the two external selectlines S. These lines are typically driven from a 2-bit condition-select field in the currentmicroinstruction; they can also connect to CPU status flags or interrupt request lines.The 2909 has five control lines, denoted OR^ORj and ZERO, that permit externalconditions to modify the address selected by MUX. For example, if ZERO is activated(ZERO = 0), then Y becomes 0000. This line is typically connected to a reset signal,which forces the control unit to begin execution of a microprogram whose startingaddress is all 0s. The OR, lines can force selected bits of Y to 1 to implement condi-tional branches relative to the current address, for instance, to skip the next micron-struction. The stack ST is enabled by the FE (file enable) line, while the push-popselect line PUP causes a push (pop) to be performed when PUP = 1 (0).

Thus microinstruction sequencing by the 2909 is controlled by signals derivedfrom a combination of microinstruction control fields and external conditions. Forexample, suppose the address X is applied to the 2909's input bus D. The following

344

SECTION 5.2

Microprogrammed

Control

microinstruction control fields

S,FE,PUP,OR,ZERO = 11,1,40000,1

(5.13)

implement the operation go to X. The effect of (5.13) is to disable ST and the OR-ANDaddress-modification logic while routing the desired branch address X from D to Y.The microoperation CALL X, where X is stored in the R register, is specified by

S,FE,PUP,OR,ZERO = 01,0,1,0000,1while RETURN is implemented by

S,FE,PUP,OR,ZERO= 10,0,0,0000,1

5.2.2 Multiplier Control Unit

Several hardwired control unit designs for a sequential twos-complement multipli-cation circuit were presented in Example 5.2. Now we examine the design of amicroprogrammed control unit for the same multiplier. The design process canbegin with either the flowchart of Figure 5.15 or the state table of Figure 5.16, bothof which define the flow of control and identify the control signals to be activated.An HDL description like that of Figure 4.13 may be more appropriate, however,since it is essentially the required microprogram written in symbolic form. Figure5.36a repeats this HDL description of the multiplier in a format in which everystatement corresponds to a distinct microinstruction, implying that a microprogramof 10 microinstructions is sufficient.

Microprogram structure. As a first attack, we use the microinstruction formatof Figure 5.33a, which has three parts: a condition-select field, a branch address,and a set of control fields. An address field of 4 bits can address up to 16 micron-structions. Initially, no encoding of control signals will be done, so that there arethirteen 1-bit control fields, one for each of the control lines cQ,cx,.. .,cn,END. Thecontrol unit has the general organization of Figure 5.33b, which has a micropro-gram counter |iPC as the control memory address register. During each micron-struction cycle, |iPC is incremented to produce the address of the nextmicroinstruction. In the case of a branch microinstruction, the address stored in thecurrent microinstruction is the branch address. We eliminate the need for an exter-nal address input by storing the first microinstruction in address 0 of CM and sim-ply resetting |iPC to 0 at the start of multiplication.

Every microinstruction can specify a branch address and so can implement aconditional or unconditional branch. The condition-select field has to indicate oneof four conditions:

• No branching

• Branch if Q[0] = 0

• Branch if COUNT7 = 0

• Unconditional branch

Hence a 2-bit condition-select field is needed. We conclude that a 19-bit micron-struction word is sufficient when a full horizontal version of the format in Figure5.33a is used.

345

Address

Microoperations

Control signalsactivated

BEGIN: A:=0, COUNT := 0, F := 0, M:=INBUS;

INPUT: Q := INBUS;

TEST1: if Q[0] = 0 then go to RSHIFT:

ADD: A[7:0] := A[7:0] + M[7:0], F := (M[7] and Q[0]) or F;

RSHIFT: A[7] := F, A[6:0].Q := A.Q[7:1], COUNT := COUNT + 1,if COUNT7 = 0 then go to TEST1;

TEST2: if Q[0] = 0 then go to OUTPUT1;

SUBTRACT: A[7:0] := A[7:0] - M[7:0], Q[0] := 0;

OUTPUT 1: OUTBUS := A;

OUTPUT2: OUTBUS := Q;

END: Halt;

c2' c3' c4> c5

END

(a)

Address Condition Branchin CM select

Control fields

0000 00

0001 00

0010 01

0011 00

0100 10

0101 01

0110 00

0111 00

1000 00

1001 11

address	Co	c1	c2	c3	c4	c5	c6	Cn	c8	c9	c10	c11	END
0000	0	0	0	0	0	0	0	0	0	1	1	0	0
0000	0	0	0	0	0	0	0	0	1	0	0	0	0
0100	0	0	0	0	0	0	0	0	0	0	0	0	0
0000	0	0	1	1	1	0	0	0	0	0	0	0	0
0010	1	1	0	0	0	0	0	0	0	0	0	1	0
0111	0	0	0	0	0	0	0	0	0	0	0	0	0
0000	0	0	1	1	1	1	0	0	0	0	0	0	0
0000	0	0	0	0	0	0	1	0	0	0	0	0	0
0000	0	0	0	0	0	0	0	1	0	0	0	0	0
1001	0	0	0	0	0	0	0	0	0	0	0	0	1

(b)

Figure 5.36

(a) Symbolic and (b) binary microprogram for twos-complement multiplication.

CHAPTER 5 Control Design

It is now easy to construct a binary microprogram that implements the multi-plication algorithm. The symbolic microprogram of Figure 5.36a is converted lineby line into the bit patterns shown in Figure 5.36b. Consecutive microinstructionsare assigned to consecutive addresses, and the appropriate condition-select bits areinserted (00 denotes no branching; the remaining condition codes are easilydeduced). When multiplication is completed, the microprogram enters a waiting(halt) state by repeatedly executing the no-operation microinstruction in CM loca-tion 1001. It remains in this state until jiPC is reset by the arrival of an externalBEGIN signal. The structure of the resulting control unit appears in Figure 5.37.

346

SECTION 5.2

Microprogrammed

Control

BEGIN0

oioj

COUNT7

1

Timinglogic

MUX

Conditionselect

Branchaddress

Increment

Load

uPC

Reset

Control

memory

(10 X 19 bits)

uIR

{c,}, END *•

Figure 5.37

Microprogrammed control unit for the twos-complement multiplier.

Control-field encoding. Few of the 213 possible control-field patterns allowedby the microinstruction format of Figure 5.37 are actually useful or needed. Figure5.36 shows that several sets of control signals are always activated simultaneously;hence a single 1-bit control field suffices for each such set. The 8 bits reserved forthe 3 sets {c0,cucu}, {c2,c3,c4}, and {c9,cw} can be replaced by 3 bits, yielding theshort, horizontal format given in Figure 5.38.

The number of control bits can be reduced further by encoding the controlfields. Since there are only 10 distinct microinstructions in the multiplicationmicroprogram (Figure 5.36), we can encode the control signals in a single 4-bitcontrol field, yielding a purely vertical format. However, this design severely

1   I 1 1

Co c3 c5 c6 cl CS   <9
                       c10

                E
Conditionselect Branchaddress
                ND

        c sntro   1 fieli is
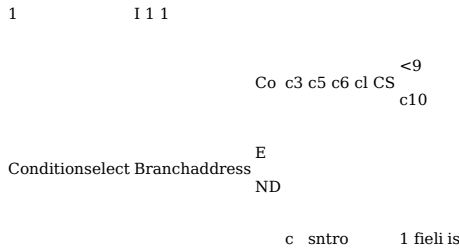
Figure 5.38

Horizontal microinstruction format after removing redundantcontrol fields.

limits our ability to subsequently modify the microinstruction set. Suppose that the 347

microinstruction format we are designing will be used for more applications than

the control of multiplication—note that the multiplier has most of the components control Desi n

of a sequential ALU. It is therefore of interest to encode the microinstructions so

that microinstructions as yet unspecified can easily be added later.

A systematic method of encoding is to divide the control signals into sets thatare compatible in the sense that no two members of set, called a compatibilityclass, are ever active at the same time. The four control signals $Cq^{\wedge\wedge\wedge}$ that loadthe same register R in Figure 5.33 are examples of a compatibility class. We cannow state the following optimization problem, which aims to minimize the totalsize of the control fields needed to implement a particular set of microinstructions:Find a collection of compatibility classes {C,} of control signals such that

• Every control signal is contained in at least one {C,}.

• The function $W = \Sigma \lceil \log_2(|C,| + 1) \rceil$ is a minimum, where lc,l is the number ofsignals in {C,}.

Here W represents the combined width of all the microinstruction's control fields.A solution to this problem minimizes the number of control bits while keeping themaximum degree of parallelism inherent in the original microinstruction set. Onlythe control fields are being minimized; the next-address and condition-select fieldsare unaffected.

Many general solutions, both exact and heuristic, to the foregoing problem arediscussed in the literature on microprogramming [Lynch 1993]. The problem iseasy to solve in the case of the twos-complement multiplier, where we have 10microinstructions and, after eliminating redundant bits from the control field, eightcontrol signals {c0,c2,c5,c6,c7,c8,c9,END}; see Figure 5.38. There are only twoincompatible microoperations, namely c2 and c5, which are activated together inthe SUBTRACT microinstruction. Hence the two largest or maximal compatibilityclasses (MCCs) are C0 = {c0,c2,c6,c7,c8,c9,END} and C, = {cq,c5,c6,c-j,c&,c9, END},so a format containing two encoded control fields suffices. There are several waysto choose subsets of C0 and Cx that cover all control signals and yield a value offive for the cost function W. For example, we can set C'0 = {c0,c2,c6} and C\ ={c5,c7,c8,c9,END}. The resulting microinstruction has the format shown in Figure5.39 and requires a pair of decoders to extract the control signals. The fact thatthere are only two control fields indicates that little inherent parallelism exists inthe multiplication algorithm.

Encoding by function. A drawback of the minimum-width control format ofFigure 5.39 is that functionally unrelated control signals are combined in the samecontrol field, while related signals are derived from different control fields. Forexample, both C'0 and C\ control the transfer of information to OUTBUS. Thislack of functional separation makes the writing of microprograms difficult, sincethe microprogrammer must associate several unrelated opcodes with each controlfield. An encoded format in which each control field specifies the control signalsfor one component or for a related set of operations is preferred, even though morecontrol bits may be needed.

On examining the multiplier circuit, we see that there are five components tobe controlled: the adder, the A.Q register-pair, the external iteration counterCOUNT, and the two data buses INBUS and OUTBUS. Each component has its

348

SECTION 5.2

Microprogrammed

Control

Conditionselect

Branchaddress

Control fields

Decoder0

T tc0 c2 c6

C,, C4

C,

l , l

Decoder

tcio

Figure 5.39

Vertical microinstruction formatwith maximum parallelism andminimum control-field width.

own set of functions, suggesting the encoding-by-function format of Figure 5.40a.Possible control-field assignments and their interpretation appear in Figure 5.40&.Note that this ad hoc encoding has combined the "incompatible" control signals c2and c5. This is unlikely to be of concern, however, if the microinstruction set islater enlarged—there is no obvious functional advantage in keeping c2 and c5 inseparate fields. The assignment of a separate control field to INBUS is of question-able wisdom. It prevents INBUS from transferring data to two or more destina-tions, such as Q and M, simultaneously. Such a transfer could be useful, forexample, to clear both registers at once. It might be better to associate a controlfield with each register that is a potential destination of INBUS rather than withINBUS itself.

Multiple microinstruction formats. In the original multiplication micropro-gram, several microinstructions are used only for next-address generation and donot activate any control lines. This suggests that we can reduce microinstructionsize by using a single field to contain either control information or addressinformation. We then obtain

two distinct microinstruction types—branch micro-instructions, which specify no control information, and action or operate microin-structions, which activate control lines but have no branching capability. Notethat this approach is almost always used at the instruction level. The division ofmicroinstructions into the branch and operate types is rather natural, since thebranch microinstructions control the internal operations of the control unit, whilethe operate microinstructions control the external datapath.

Suppose that we want to use unencoded control fields for the twos-comple-ment multiplier, which requires 8 control bits, as seen from Figure 5.38. Now wedefine a microinstruction format having two parts, a 2-bit condition-select fieldwith the same meaning as before, and an 8-bit field that can contain either a branchaddress or control information. The condition-select code 00, which denotes nobranching, serves to identify the operate microinstructions. The remaining threeselect field codes identify conditional and unconditional branches. We thus obtainthe two 10-bit microinstruction formats of Figure 5.41a. Note that the additionaladdress bits enable us to write microprograms containing up to 28 = 256 instruc-

Control fields

14 13 8   (

i   i i i i i i

CHAPTER 5Control Design

Branchaddress

ADDER SHIFT COUNT INBUS OUTBUS

(a)

Controlfield

Bitsused

Code

Microoperationsspecified

Control signalsactivated

Control field	Bits used	Code	Microoperations specified	Control signals activated
ADDER	7,8	00	No operation	
		01	A := A + M, set F	c2,c3,c4
		10	A := A - M, Q[0] := 0	c2,c3,c4,c5
		11	Unused	
SHIFT	6	0	No operation	
		1	Right-shift A.Q, set A[7]	c0
COUNT	5,4	00	No operation	
		01	Clear COUNT, A, F	c10
		10	COUNT := COUNT + 1	c11
		11	Unused	
INBUS	3,2	00	No operation	
		01	Q := INBUS	c8
		10	M := INBUS	c9
		11	Unused	
OUTBUS	1,0	00	No operation	
		01	OUTBUS := A	c6
		10	OUTBUS := Q	c7
		11	Unused	

(b)

Figure 5.40

(a) Microinstruction format with control fields encoded by function and (b) their interpre-tation.

tions. Because we have destroyed the capability of every microinstruction toimplement a two-way branch, some operations need more microinstructions.

Figure 5.41b shows a microprogram for twos-complement multiplicationusing the formats of Figure 5.41a. This microprogram is somewhat easier to derivefrom the flowchart (Figure 5.15) than was the earlier microprogram (Figure 5.36)because we can now transform decision blocks directly into branch microinstruc-tions, while activity boxes are transformed into operate microinstructions. The con-trol unit of Figure 5.37 is easily modified to handle these new microinstructionformats: The condition-select field is used to control a demultiplexer that routesbits 0:7 either to external control lines (operate microinstructions) or to the branchaddress logic (branch microinstructions).

Multiplication and division cannot be bit sliced in the same way as addition,subtraction, or shifting. However, these operations can be implemented in a bit-

350	Control fields	Branch address

9 8   A.   ^v

r7 6 5 4 3 2 1 0   9 8 r7 6 5 4 3 2 0

Microprogrammed

Control

co   c2 c$ c6 cl c8 cg END

Condition
select

0 1 Branch if Q[0] = 0

1 0 Branch if COUNT7 = 0

1 1 Unconditional branch

(a)
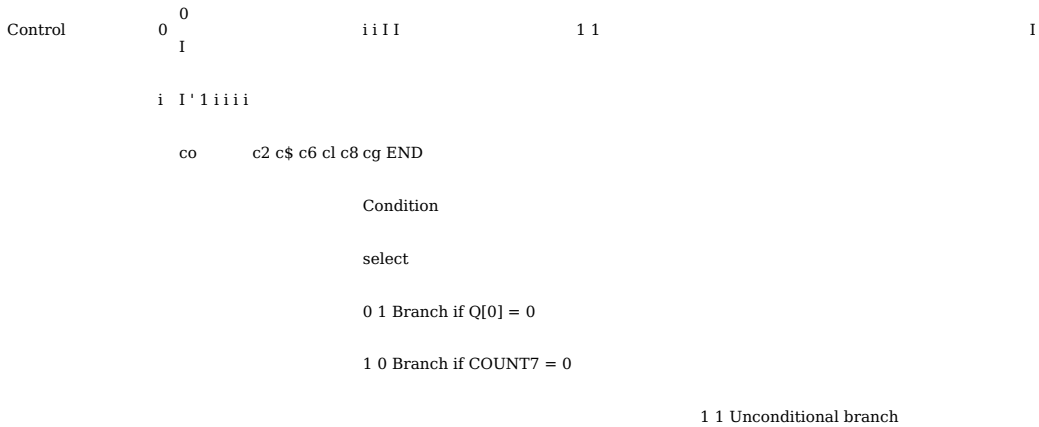
Address	Condition select 9 8	7	6 5	Branch address or control bits		
in CM				4 3 2 1 0	Comment	
0000	0	0	0 0	0 0 0 0 0 1	0 BEGIN	
0001	0	0	0 0	0 0 0 0 1 0	0 INPUT	
0010	0	0	1 0	0 0 0 0 1 0	0 TEST1	
0011	0	0	0 0	1 0 0 0 0 0	0 ADD	
0100	0	0	0 1	0 0 0 0 0 0	0 RSHIFT	
0101	1	0	0 0	0 0 0 0 0 1	0 RSHIFT BRANCH	
0110	0	0	1 0	0 0 0 1 0 0	0 TEST2	
0111	0	0	0 0	1 1 0 0 0 0	0 SUBTRACT	
1000	0	0	0 0	0 0 1 0 0 0	0 OUTPUT1	
1001	0	0	0 0	0 0 0 1 0 0	0 OUTPUT2	
1010	0	0	0 0	0 0 0 0 0 0	1 END	
1011	1	1	0	0 0 0 1 0 1	1 HALT	

00

Figure 5.41

(a) Multiple microinstruction formats and (b) multiplication microprogram that uses theseformats.

sliced ALU under the control of a microprogram that implements one of the shift-and-add/subtract algorithms described in section 4.1, as the next example demon-strates.

EXAMPLE 5.5 TWOS-COMPLEMENT MULTIPLICATION IN A BIT-SLICED

alu [mick and brick 1980]. The AMD 2901 is a 4-bit ALU slice, which isdescribed in Example 4.5. A set of k copies of the 2901 can easily be connected to per-form the basic ALU operations (twos-complement addition and subtraction, as well asthe standard logical operations) on 4A:-bit data words. We now explain how such anarray can implement twos-complement multiplication under microprogram control.

Figure 5.42 shows a four-slice 2901 circuit that is configured to multiply 16-bittwos-complement numbers via the Robertson algorithm. The roles of the accumulatorA, the multiplicand register M, and the multiplier register Q are assigned to the 290l'sR(B), R(A), and Q registers, respectively. R(A) and R(B) are in the 2901's register file(referred to as the RAM in 2900 literature), while Q is a "'quotient" register intended tosupport sequential multiplication and division algorithms. The register addresses Aand B are determined by external signals placed on the corresponding 4-bit RAMaddress buses. The shift lines Q0 and Q3 serve to link the Q registers in the four 2901s

to form a 16-bit Q register that can be right shifted via the 2901's Q shifter (Figure4.36). In the same fashion the RAM0 and RAM3 shift lines effectively link the slices ofR(B), allowing it to serve as the 16-bit accumulator. A connection from RAM0 on theright-most (least significant) slice to Q3 on the left-most (most significant) slice linksthe 16-bit R(B) and Q registers to form the 32-bit shift register—A.Q in the originaldesign—where the product will eventually be stored. Finally, as the contents of theA.Q register-pair are right shifted by the 2901's RAM shifter, the sign bit of the partialproduct should be entered into the most significant bit position of A.Q, that is, A[15].

351

DRAM3 RAMq   DRAM3 RAMo   D   D

KAM3 KAM0   RAM3 RAMq

(a)

Accumulator A r

Q[

Multiplicand register M

R(B)   R(A)

16-bit
^oui Parallel
adder

Q

Multiplier register Q

(*)
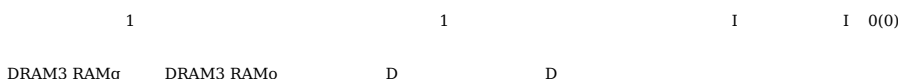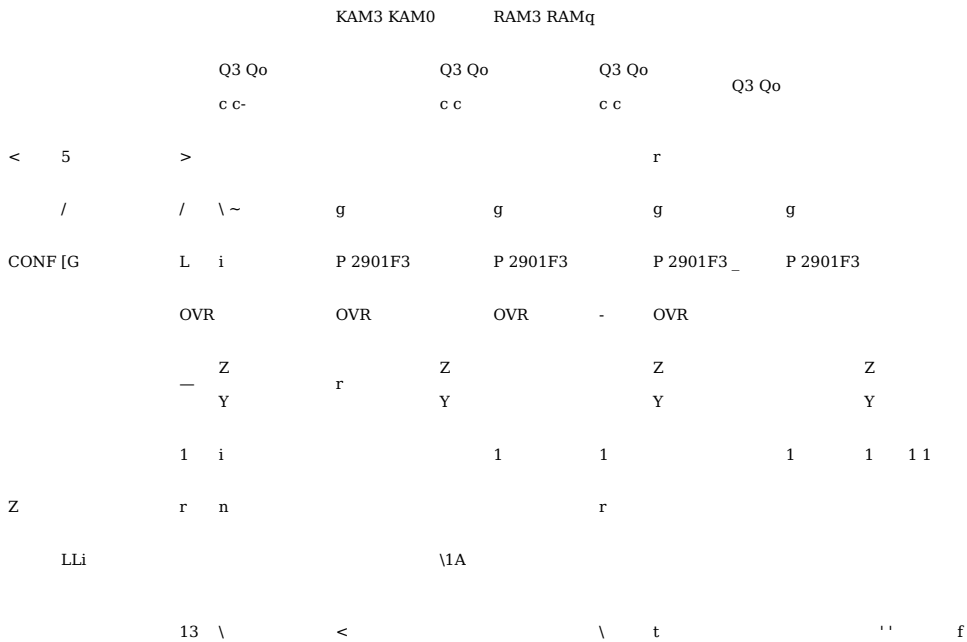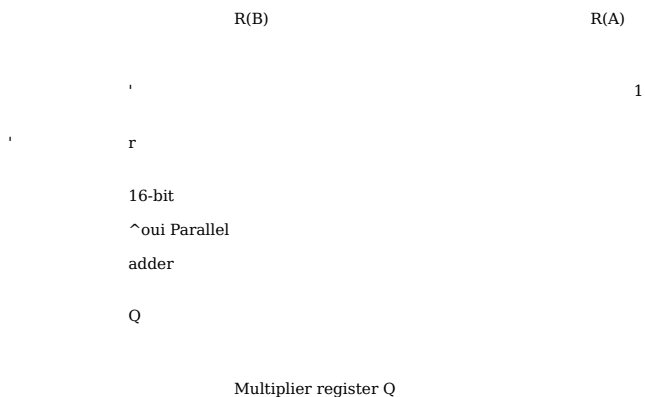
Figure 5.42

A four-slice 2901 array configured for 16-bit multiplication: (a) slice interconnections and (b) register assignments.

SECTION 5.2

Microprogrammed

Control

In our previous implementations of Robertson's algorithm, for example, in the micro-programs of Figures 5.36 and 5.41, this bit was computed via the formula

(5.14)

$¥ := (M[n-1] \text{ and } Q[0]) \text{ or } F$

It can be shown that (5.14) is equivalent to «

$F := F[«-1]jcorOVF$ (5.15)

where ¥[n − 1] is the sign bit F_3 of the result generated by the most significant 2901 slice and OVF is the overflow signal (based on twos-complement addition) produced by that same slice. Equation (5.15) is implemented by the XOR gate appearing in Fig-ure 5.42a.

The microoperations performed by the 2901 are specified by the set of control sig-nals I = IS,IF,ID listed in Figure 4.38; these are typically used as microinstruction con-trol fields of the encoding-by-function type illustrated by Figure 5.40. The design of the 2901 control fields makes it possible to implement conditional microinstructions in clever ways. In particular, if the bits of the IF function-select field are treated as condi-tion variables, then the selected operation varies with the condition values. For instance, IF = 000 specifies add with carry; if the middle bit of IF is a condition variable and is changed to 1, we get IF = 010, which specifies subtract with borrow.

The central operation in binary multiplication, which is a conditional add followed by a right shift, where the condition variable is the current multiplier bit x; stored in Q[0], can be implemented by a single, carefully constructed 2901 microinstruction. This operation is expressed as follows in HDL format:

if Q[0] = 1 then R(B) := R(B) + R(A); R(B).Q := R(B).Q/2;

(5.16)

Let Q[0] be applied as a condition variable to the middle line of the 290l's input source control field Is. Then (5.16) is realized by the following microinstruction:

IS,IF,ID = 0 Q[0] 1,000,100

(5.17)

IF = 000 specifies the add-with-carry operation whose source and destination operandsare determined by Is and ID, respectively. Changing Is from 001 to 011 changes theoperation defined by (5.17) from Y := R(A) + R(B) to Y := 0 + R(B), effectively skip-ping the add step. ID = 100 causes the result F to be right shifted before loading intoR(B); it also right shifts Q as required by (5.16).

Since we also need to make Is a constant at other times, we would implement(5.17) by connecting the middle bit of the Is control field and Q[0] to the data inputs ofa two-way 1-bit multiplexer MUXQ. The output of MUXQ would be the final controlsignal c. Another 1-bit control field CQ would be added to the microinstruction formatto drive the select input of MUXQ and thus determine whether c = Q[0] or the currentvalue in the Is field.

Sixteen-bit multiplication requires (5.16) to be executed 15 times and be followedby a subtraction step that is again conditional on the value of Q[0]. A complete micro-program along these lines appears in Figure 5.43. It implements the same basic algo-rithm we have used earlier, with various modifications geared to the particular featuresof the 2900 series, and is designed to produce a very short multiplication micropro-gram. It must be possible to configure the 2901 ALU as in Figure 5.42 under control ofa microprogram sequencer such as the AMD 2909 (Example 5.4) or 2910. The latterincludes a counter that can be automatically decremented in every clock cycle and socan serve as the multiplier's iteration counter COUNT. On setting the 2910 to a specialrepeat mode, the microprogram sequencer will continue to output the address of thecurrent microinstruction, and so repeatedly execute that microinstruction, untilCOUNT becomes zero.

Microoperations

Comment

Q:=0orR(A)

R(B):=0a/u/R(B)

COUNT := 15; whUe COUNT * 0 do

begin A := A x Q[0] + M; right-shift A.Q;COUNT := COUNT - 1 end

A := A x Q[0] - M

R(B) :=0 or Q

Move multiplier X to Q from R(A)

Clear accumulator A = R(B)

Conditional add and shift repeated15 times

Conditional subtract. Product = A.QMove low half-product from Q to R(B)

(a)

COUNT A

If It

CONFIG REPEAT Comment

...ddddd 0000 dddd 100 011 000 d

...ddddd dddd 0011 011 100 011 d

...01111 0001 0011 040 000 100 0

...ddddd 0001 0011 0^0 001 100 1

...ddddd dddd 0010 010 011 010 d

0 0 Move multiplier X to Qfrom R[0]

0 0 Clear accumulator R[3]

1 1 Conditional add and shiftrepeated 15 times

1 0 Conditional subtract.Product = R[3].Q

0 0 Move low half-product fromQ to R[2]

(b)

Figure 5.43

(a) Symbolic and (b) binary microprogram for twos-complementmultiplication in the 2901-based processor.

353

CHAPTER 5

Control

Design

Figure 5.43 gives in both symbolic (HDL) and binary form a multiplication micro-program containing only five microinstructions. The first two microinstructions initial-ize the multiplication, assuming that the multiplicand M is already stored in registerR[l]. The third microinstruction implements (5.17) and is executed 15 times. Thefourth microinstruction implements the conditional subtraction (correction step)needed to accommodate a negative X. while the last instruction transfers the contents ofthe Q register to the register file; at the end the data is stored as follows: R[0] = multi-plier, R[l] = multiplicand, and R[3].R[2] = product.

Most of the control fields appearing in Figure 5.43fc specify control signalsapplied to the 2901 ALU. The CONFIG field is intended to produce the special multi-plication configuration of Figure 5.42. which requires various control points (not allshown) to establish links such as that from the output of the XOR gate to the RAM,input of the left-most slice, from Q[0] to the middle signal derived from Is. and so on.An address field called REPEAT is also defined, with REPEAT = 0 meaning that thenext microinstruction, whose address is generated by the microprogram counter (iPC.immediately follows the current microinstruction. REPEAT = 1 means that the currentmicroinstruction is to be repeated until the automatically decremented COUNT registerreaches zero.

354

5.2.3 CPU Control Unit

SECTION 5.2M icroprogrammedControl

This section considers the design of the microprogrammed control units and micro-programs for use in the CPU of nonpipelined, general-purpose computers.

Basic emulator. First we reexamine the accumulator-based CPU for whichwe developed a hardwired program control unit in section 5.1.3. The organizationof this CPU and its 10-member instruction set appear in Figure 5.20. The 13 con-trol signals listed in Figure 5.22 define the basic microoperations that are availableto the microprogrammer. (We will later extend this to a more realistic set.) To sim-plify the presentation, we will give the microinstructions only in symbolic formusing our HDL.

Suppose that you want to write an emulator for the target instruction set whosemembers, which are defined in Figures 5.20 and 5.21, are LD, ST, MOV1, MOV2,ADD, SUB, AND, NOT, BRA, and BZ. The microoperations that implement thevarious instructions appear in Figure 5.22, from which the required microprogramsare easily deduced. The microprogram selected to emulate each instruction is iden-tified by the instruction's opcode; hence the contents of the instruction register IRdetermine the microprogram's starting address. We will use the unmodified con-tents of IR as the microprogram address for the current instruction. We will furtherassume that each microinstruction can specify a branch condition, a branch addressthat is used only if the branch condition is satisfied, and a set of control fieldsdefining the

microoperations to be performed. These microinstruction fields caneasily be adapted to a variety of formats (horizontal, vertical, encode-by-function,and so on) as discussed earlier.

Figure 5.44 lists a complete emulator for the given instruction set in symbolicform; the conversion of each microinstruction to binary code is straightforward.

FETCH:

LD:

ST:

MOV1:

MOV2:

ADD:

SUB:

AND:

NOT:

BRA:

BZ:

AR := PC;DR := M(AR);PC:=PC+ l.IR:=DR(OP):go to IR:

ARDRAC

ARDR

DR(ADR);= M(AR);= DR. go to FETCH;

= DR(ADR);= AC;

M(AR) := DR, go to FETCH;

DR := AC. go to FETCH:

AC:=DR, go to FETCH:

AC := AC + DR, go to FETCH;

AC := AC - DR. go to FETCH;

AC := AC and DR. go to FETCH;

AC := not AC, go to FETCH;

PC := DR(ADR), go to FETCH;

if AC = 0 then PC := DR(ADR). go to FETCH;

Figure 5.44

A microprogrammed emulatorfor a small instruction set.

CHAPTER 5Control

This emulator contains a distinct microprogram for each of the ten possible 355instruction execution cycles and another microprogram called FETCH—note howthe name of the microprogram corresponds to its address in the emulator code—which controls the instruction fetch cycle. The go to IR microoperation is imple- Designmented by (i.PC := IR, which transfers control to the first microinstruction in themicroprogram that interprets the current instruction. Depending on the micro-instruction format chosen, either such branch operations can be included in a gen-eral operate-with-branching format or separate branch microinstructions can bedefined. Figure 5.44 assumes that \iPC is the default address source for microin-structions and is incremented automatically in every clock cycle.

Suppose that because of a design error, or because of a late modification to thespecifications of the instruction set, we need to introduce a new instruction calledCLEAR whose function is to reset all bits of the accumulator AC to 0. Although nocontrol line to clear AC was included in the CPU, we can still write a micropro-gram to implement the CLEAR instruction using only the preexisting microopera-tions.

CLEAR: DR := AC;

AC := not AC;

AC := AC and DR, go to FETCH;

By storing this new microprogram in the control memory, CLEAR can be added tothe instruction set with either no changes, or very minor ones, to the CPU hard-ware. Such flexibility is a key advantage of microprogramming over hardwiredcontrol.

Extensions. We will now add to the CPU structure of Figure 5.22 the circuitsto implement fixed-point multiplication and division using sequential algorithms ofthe type discussed in Chapter 4. Two major new registers are required—a multiplier-quotient register MQ and a counter called COUNT, which counts the number of iter-ations (add/subtract and shift steps) used during multiplication or division. Thememory data register DR will be assigned the role of multiplicand or divisor registerMD when appropriate.

Figure 5.45a shows the modified CPU; the number of control signals has morethan doubled to 29. These signals are denoted c_0:c_2& and defined in Figure 5.45b:c_0:c_{12} correspond to the control signals of the original CPU in Figure 5.22. Severalof the control signals listed in Figure 5.45b implicitly cause flag (status) bits to beset or reset. For example, if overflow occurs during addition or subtraction, whichare controlled by c_9 and $c_{,0}$, respectively, then OVR is set to 1; otherwise OVR isreset to 0. The three flag bits FLAGS, the least significant bit MQ[0J of the multi-plier-quotient register, and the signal COUNT = n - 1 all serve as branch condi-tions that the microprogrammed control unit can test.

Figure 5.46 lists a symbolic microprogram for this CPU that implements theRobertson multiplication algorithm for twos-complement numbers first introducedin Example 4.2. A special-purpose microprogrammed controller for this type ofmultiplication was developed in section 5.2.2. The microprogram 2Cmult givenhere is essentially the same as the one defined previously for the stand-alone multi-plier (refer to Figure 5.42a). In this symbolic form, the microprogram is also simi-lar to the original HDL description of the Robertson algorithm (Figure 4.13). We

356

SECTION 5.2M icroprogrammedControl

assume that before 2Cmult is executed, the multiplier operand X is placed in MQand the multiplicand Y is in DR. Each statement in Figure 5.46 represents a singlemicroinstruction.

The general three-part microinstruction format comprising a condition-selectfield, a branch-address field, and a set of control fielcfs will be used for 2Cmult.Five conditions to be tested are identified in Figure 5.45a: AC = 0, AC < 0, MQ[0],COUNT =•** — 1, and OVR. the overflow indicator. Adding the possibilities of anunconditional branch and no branching, we obtain seven branch-condition codesthat can be represented by a 3-bit condition-select field.

Various control signals can be grouped together in common encoded fields toreduce the microinstruction size. We can identify many of these fields from thelist of control signals without reference to the actual microinstructions that are tobe implemented. For example, three control signals cv c6, and c-,0 transfer data toDR. Since they are mutually exclusive (compatible), we can encode them in a

co ci

C2S

it - T

<23c25

Microprogrammed

control

unit

COUNT = n - 1

OVR

COUNT

AC = 0

AC<0

MQ[0]

GO I~ar—i \~K3-c2

C4-Q

M and10 devices

DR(OP)

VT

DR(ADR)

SystemBus

DR

c6:c7-c,:c,2-

c17:c20-c26:c27-

] | AC HJ)-r~MQ

rTTT Tl

c21c22

y

Arithmetic-logicunit

FLAGS

c28

Controlsignal

c2c3

c5c6<7c8c9c10

c\\

Cft

c13C{4

Cl5^16

cn

cn

Cl9c20

H\

C22C23C24^25<^26C27c28

Operationcontrolled

AR := PC

DR := M(AR)

PC := PC + 1

PC := DR(ADR)

IR := DR(OP)

AR := DR(ADR)

DR := AC

AC := DR

M(AR) := DR

AC := AC + DR

AC := AC - DR

AC := AC and DR

AC := not AC

RSHIFT AC

LSHIFT AC

RSHIFT AC.MQ

LSHIFT AC.MQ

AC:=0

AC[n-l]:=F

MQ := DR

MQ[0]

DR := MQ

MQ[0] := 1

MQ[0] := 0

COUNT := COUNT + 1

uPC := IR

COUNT := 0

F:=0

F:=l

FLAGS := 0

(a)

Figure 5.45

(a) Control points and (b) control signal definitions for die extended CPU.

(*)

_————————^——— 357

Address Microinstruction

— CHAPTER 5

BEGIN: A := 0, COUNT := 0, F := 0; Control

TEST1: if MQ[0] = 0 then go to RSHIFT: Design

ADD: AC := AC + DR, F := (DR[«-1] and MQ[0]) or F;

RSHIFT: AC [n-\ ] := F, AC.MQ := RSHIFT(AC.MQ), COUNT := COUNT + 1,

if COUNT * n-\ then go to TEST1;

TEST2: if MQ[0] = 0 then go to FETCH;

SUBTRACT: AC := AC - DR, MQ[0] := 0, go to FETCH;

Figure 5.46

Twos-complement multiplication microprogram 2Cmult for the extended CPU.

2-bit field. Note that one more bit pattern must be reserved for the no-operationcase. Similarly, we can combine the many control signals that alter the contentsof AC.

Suppose that we have decided not to encode the control signals. This decisionimplies that the condition-select and control fields occupy 32 bits of each microin-struction. Suppose further that an 8-bit branch address denoting a complete CMaddress is included in each microinstruction; a CM storing up to 256 forty-bitwords is therefore supported. Figure 5.47 shows a possible organization for theCPU control unit with the foregoing design assumptions. As in our previousdesigns, external conditions control the loading of branch addresses into the (IPC.In addition, the (IPC can be loaded from the instruction register IR via a logic cir-cuit K (typically a ROM or PLA), which maps instruction opcodes onto microin-struction addresses.

Microprogram sequencers. It is possible to place all the circuitry required togenerate microinstruction addresses in a single IC or cell called a microprogramsequencer, a simple example of which, the AMD 2909, was discussed earlier(Example 5.4). A microprogram sequencer is a general-purpose building block formicroprogrammed control units. It contains a microprogram counter [IPC, as wellas the logic needed for conditional branching and transferring control betweenmicroprograms. A control unit can be constructed from three components: a RAMor ROM used as the control memory, a microinstruction register, and a micropro-gram sequencer. Figure 5.48 shows a microprogrammed CU designed in this way.The microinstruction register can be implemented as a two-stage pipeline to allowmicroinstruction fetching and execution to be overlapped.

Microprogram sequencers are mainly found in standard-cell families like the2900 series intended for the design of both general-purpose and application-specificprocessors. They are also used in CISC CPUs such as the Motorola 680X0. Becauseof IC component density and pin restrictions, early microprogram sequencers likethe 2909 were relatively simple and had to be bit sliced to allow control units ofpractical size to be constructed from them. Subsequent advances in VLSI technol-ogy have enabled more powerful and self-contained control units of this kind to bebuilt.

358

SECTION 5.2

Microprogrammed

Control

AC = 0

AC<0

MQ[0]

COUNT = n- 1

OVR

MUX

Conditionselect

3 >'

Instructionregister

1R

Opcode

K,

Branch address

Load branchaddress

u

uPC

IncrementReset

Control

memory

(256x40 bits)

40

U.IR

>29 Control signals

Figure 5.47

Microprogrammed control unit for the extended CPU.

Control

memory

CM

CM address

Microprogram sequencer

Micro-

instruction

ulR

Branch address

IR ••

Datapath

unit

Opcode

To main memory and IO devices

Figure 5.48

Microprogrammed CPU employing a microprogram sequencer.

EXAMPLE 5.6 THE TEXAS INSTRUMENTS 890 MICROPROGRAM SEQUENCER [TEXAS

instruments 1985. lackzo et al. 1986]. This circuit, whose full designation is the SN74AS890, is a member of the Texas Instruments 88X microprocessor component family, which was introduced in the mid-1980s aimed at the design of general-purpose CPUs. It can be considered a natural evolution of the 2909-class microprogram sequencers to accommodate larger address sizes (and hence larger control memories), more address sources, and more-flexible operating modes. Packaged in a single 70-pin IC, the 890 has a number of features intended to simplify the development of micropro-grams. The address size is 14 bits, enabling a single 890 to manage a control unit con-taining a 16K-word CM for the storage of microprograms; consequently, it is not bitsliced. The corresponding datapath member of the 88X series is the 888 (SN74AS888), an 8-bit ALU slice. The architecture of the 888 is almost identical to that of the 29014-bit ALU considered earlier (Examples 4.5 and 5.5) except for its larger word size.

Figure 5.49 depicts the internal organization of the 890. Like the 2909 (Figure 5.34), the CM address sources are a small set of external buses and internal registers, including a microprogram counter uPC and a LIFO stack. The (J.PC is implemented as

IO bus DRA

14

IO bus DRB

RAOEOSEL

Mux \

Register/counter A

t L_

Register 3

. —^—*control

Address ^

modify B3:B0

MUX2:0 *-

CC

14

14

B>

Register/counter B~I 1

9 x 14-bit stack STK

RBOE

ZERO

Stack

■* y- operations

S2:S0

-*- Stack status

/ Mux \

INT

'14,

I

Microprogramcounter U.PC

Interruptreturn register

Y outputmultiplexer

Incrementer

YOE

V

INC

IO bus Y

Figure 5.49

Structure of the Texas Instruments 890 microprogram sequencer.

SECTION 5.2

MicroprogrammedControl

a 14-bit register with a separate incrementer, as in the 2909, and is the usual source ofmicroinstruction addresses. There are two main external sources of branch addresses,the buses DRA and DRB, while the Y bus serves as the main output address bus. Allthree buses are 14 bits wide and, for added flexibility, all are bidirectional. DRA, DRB,and Y may be compared with the 2909's D, R, and Y buses, respectively. The 4-bit B(branch) bus replaces the four least significant bits of addresses on the DRA and DRBbuses to implement conditional 32-way branches: the B3:B0 lines therefore correspondroughly to the ORjiORg and ZERO lines used for address modification in the 2909.The DRA and DRB buses also have registers/counters A and B, respectively, associ-ated with them. A and B can serve either as independent address sources or else as iter-ation counters when executing a loop in a microprogram.

The 890 has a nine-word stack STK to implement subroutine calls and interrupts.Three control lines S2:S0 allow various stack operations including push, pop, reset,and hold to be specified by microinstruction control fields. A push operation (withINT = 0) places the contents of |iPC at the top of the stack; a pop operation transfersthe address at the top of the stack to the Y bus via the Y multiplexer. In addition to theusual stack pointer SP for automatically tracking the top of the stack, a second pointerregister, the read pointer RP. reads out the contents of the stack word by word to the890's DRA port. This readout process, which does not alter the contents of the stack orSP, can be used to backtrack through a sequence of subroutine calls or interrupts toidentify the cause (for instance, overflow) and the location in CM of a problem occur-ring during microprogram execution.

In summary, the 890 can output to the Y bus a 14-bit microinstruction addressderived from four sources: the microprogram counter |iPC, the stack STK, the DRAbus, or the DRB bus. The addresses on the DRA and DRB buses can be obtained eitherexternally or from the 890's internal A and B registers. The DRA/DRB addresses canalso be modified by the B3:B0 lines, which, in conjunction with the control inputsMUX2:MUX0 and CC of the Y multiplexer, support the implementation of manykinds of conditional and unconditional branching. The "condition code" bit CC isdesigned to add a simple two-way branch option to most microinstructions.

Consider the execution in cycle i of a microinstruction /(ADR) stored at controlmemory address ADR. If no branching is specified, then during clock cycle /, uPCwrites the address ADR to the Y bus; at the same time it reads in the next addressADR + 1 from the incrementer. In this way the 890 is ready to execute the instruction/(ADR + 1) in cycle / + 1. Sometimes it is desirable to allow a status signal /-"from thedatapath unit to control the operation of the incrementer so that under certain condi-tions (illustrated later) no incrementing occurs: in such a case the execution of/(ADR)is repeated in cycle i + \. The action (the execution of some microinstruction) that setsup the relevant value of F to block the increment in cycle ; must therefore occur incycle i - 1 or earlier.

Figure 5.50 presents a few examples of the huge number of possible branch micro-operations that the 890 can implement (along with a full range of datapath operationsthat we do not consider here). We show only the principal control fields associatedwith program control in microinstructions. The first "continue" or "no operation"(NOP) microinstruction is intended merely to replace the current contents of U.PC byU.PC + 1; that is.

p.1,: uPC:=uPC + l;

{Continue}

The condition code CC and increment bit INC must be set to 1 at least one cycle ear-lier. The control signal combination MUX2:MUX0,CC =1001 selects U.PC as the datainput of the Y multiplexer and applies it to the incrementer, which then outputs |iPC +INC to |iPC. The control field values S2:S0 = 111 and OSEL = 0 are needed to inacti-vate stack operations. The remaining control bits denoted by d are don't cares. The sec-

Instruction	CM address	MUX2: MUX0	S2:S0	R2:R0	OSEL	CC	INC	1	DRA	DRB	
(Setup)		ddd	ddd	ddd	d		1	!	..ddd	..ddd	Control
Continue (NOP)	0001	100	111	ddd	0	d	d		..ddd	..ddd	Design
(Setup)		ddd	ddd	ddd	d		1	d	..ddd	..ddd	
Branch to 5	0001	000	111	ddd	0	d	d		..0101	..ddd	
(Setup)		ddd	ddd	ddd	d		1	1	..ddd	..ddd	
Branch to 5 if CC= 1	0001	110	111	000	0	d	d		..0101	..ddd	
(Setup)		ddd	ddd	ddd	d		1	1	..ddd	..ddd	
Loop until A = 0	0001	110	100	ddd	0	1	1		..ddd	..ddd	
	0010	110	111	010	0	0	1		..ddd	..ddd	
	0011	000	010	000	1	1	1		..ddd	..ddd	
(Setup)		ddd	ddd	ddd	d		1	1	..ddd	..ddd	
Call subroutine (at 5)	0001	000	110	ddd	d	d	d		..0101	..ddd	
(Setup)		ddd	ddd	ddd	d		0	d	..ddd	..ddd	
Return from subroutine	0001	010	011	000	d		0	d	..ddd	..ddd	

Figure 5.50

Sample branch microinstructions for the 890 microprogram sequencer.

ond example in Figure 5.50 is a simple unconditional branch to the address on the DRA bus; the combination MUX2:MUX0,CC = 0000 makes Y = DRA. Changing MUX2:MUX0 to 110 as is done for the third microinstruction "branch to 5 if CC = 1,'"ensures that the branch occurs only if CC = 0; that is,

\ily if CC = 0 then Y = DRA else uPC := uPC + 1;

The fourth example employs a sequence (microprogram) of three microinstructions|!l4 ,:ij.I4 3 to implement "loop until A = 0" as follows:

Hl4J: uPC:=uPC + 1, STK(SP) := uPC, SP := SP + 1: {Continue, push uPC}

1^4.2

uPC := uPC + 1, A := DRA; {Continue, load register A}

(5.18)

ILII43: A:=A- 1, if A*0then Y = STK(SP)

else (IPC := uPC + 1. SP := SP - 1: {Decrement A. branch to stack if

A = 0. pop}

The call and return microinstructions have similar interpretations.

Normally |iPC contains the address whose value is one plus the address ADR of the currently executing microinstruction. The interrupt return register IRR is designed to operate in parallel with uPC but contains the current address ADR rather than ADR + 1. This feature permits an interrupt to be implemented in the following way that has zero latency. The interrupting device disables the 890's Y bus by setting YOE to one. It then places a new address ADR1 (the interrupt vector) on Y, forcing a trans-fer to CM address ADR1, which is the start address of the interrupt-servicing routine P. The microinstruction at address ADR1 must be designed to push IRR into the stack(which requires INT = 1). thus saving the return address of the interrupted program.

Nanoprogramming. In most microprogrammed processors, an instruction fetched from memory is interpreted by a microprogram stored in a single control

362

memory CM. In a few machines, however, the microinstructions do not directly issue the signals that control the hardware. Instead, they are used to access a second control memory called a nanocontrol memory nCM that directly controls the hard-ware. In such cases there are two levels of control memories, a higher-level one termed a microcontrol memory pCM whose contents are microinstructions and the lower-level nCM that stores nanoinstructions—see Figure 5.51. The nanoprogram-ming concept was first used in the QM-1 computer designed around 1970 by Nano-data Corp. It is also employed in the Motorola 680X0 microprocessors series[Stritter and Tredennick 1978].

Consider a nanoprogrammed computer in which U.CM and nCM have dimen-sions Hm x Wm and Hn x Wn, respectively. The advantage of this two-level control design technique is that it can reduce the total size S2= Hmx Wm + HnxWn of the control memories, which translates to smaller chip area in the case of a one-chip CPU like the 680X0. Typically, the microprograms are encoded in a narrow verti-cal format so that although Hm is large, Wm is small. Nanoinstructions, on the other hand, usually have a highly parallel horizontal format making Wn large. If one nanoprogram can interpret many microinstructions, then H,, can be kept relatively small so that S2 < 5, = Hm x Wn, which is roughly the size of a comparable single-level control unit. The potential for reducing the total size of the control memories is the main reason for the use of nanoprogramming in the 680X0 series. Another advantage is the greater design flexibility that results from loosening the bonds between instructions and hardware with two intermediate levels of control rather than one. These advantages motivated the QM-1, which had the goal of efficiently emulating the instruction sets of a wide variety of different computers. The main disadvantages of the two-level approach are a loss of speed due to the extra mem-ory access for nCM and a more complex control-unit organization.

From instruction register IR

UPC

Microcontrol

memory

uCM

Microinstruction reg. (ilR

1 I

nPC

Nanocontrol

memory

nCM

w

Nanoinstruction reg. nlR

Control signals

Figure 5.51

Two-level control store organization for nanoprogramming.

To see the savings in control-memory size that can result from the use of nano-programming, consider the analysis carried out by the designers of the 68000 microprocessor [Stritter and Tredennick 1978]. Suppose that one- and two-level control stores are characterized by the parameters shown in Figure 5.52. A one-level conventional CM is assumed to store Hm horizontal microinstructions each with a format consisting of N control bits and |~log2//m~| next-address bits. The size of this memory is therefore

S, = //m(/V+riog2//ml) (5.19)

In the two-level organization (Figure 5.52b), the microcontrol memory uCM again stores Hm microinstructions, but the TV-bit control fields are transferred to nCM. In place of the latter, each microinstruction in pCM contains a [log2 Hn~\-b\l address to specify any nanoinstruction location in nCM. It is assumed that little or no branching takes place among nanoinstructions, so no explicit address bits are included in the model of nCM. Thus the size of the two-level control store is

S2 = Hm([\og2 Hm~\ + [log2 Hn-} + NH„

(5.20)

Suppose that all the control-bit patterns in nCM are different so that each repre-sents a unique control state associated with the given instruction set. We can write Hn = rHm, where r is the ratio of the number of unique control states to the total number Hn of control states needed to implement all instructions. Substituting into (5.20) yields

S1 = Hm(\\og2Hm\ + \\og2rHn-\+rN)= Hm{2\\og2Hm-\ + \\og2r^+ rN)

363

(5.21)

log2 Hn

Address

Control

memory

CM

Microcontrolmemory (iCM

log-, H,„ i log: H„i ►+« »

Control

T

Hn

1

v

Addresses

Nanocontrolmemory nCM

vControl

(a)

Figure 5.52

Control memory models: (a) one level and (b) two level.

(*)

The following parameters are cited for the 68000 design: TV = 70, Hm = 650,

section 5 3 and r = 0.4 so that Hn = 260. Substituting into (5.20) and (5.21), we obtain 5, =

Pipeline Control 52,450 and S2 = 30,550. Consequently, the use of nanoprogramming saves a total

of 52,450 - 30,550 = 21,850 bits of control storage (42 percent of S{). In general,

two levels of control memory require less memory sp^ce if 52 < S,. Hence from

(5.19) and (5.21), we conclude that the inequality

^>riog2//m]+ri°g2''i+^

must be satisfied.

5.3

PIPELINE CONTROL

Pipelining provides a basic way to speed up arithmetic operations, as we saw inChapter 4. It is also used to implement the entire instruction-processing behavior ofhigh-performance CPUs, a topic we examine in this section.

5.3.1 Instruction Pipelines

During program execution, instructions pass through a sequence of processingsteps that lend themselves naturally to pipelining. Consequently, a CPU can beorganized as one or more pipelines, whose various stages fetch opcodes and oper-ands, execute instructions, and store results in local registers or external memory.In general, an instruction pipeline is a multifunction, reconfigurable pipelinedesigned to speed up a computer's performance by efficiently overlapping the pro-cessing of instructions. Such pipelines are contrasted with arithmetic pipelines ofthe type covered in section 4.3.2, which can, however, be built into instructionpipelines to implement the execution stages. An instruction pipeline is normallyinvisible to programmers and managed automatically by program compilers and bythe CPU's internal program-control unit. Instruction pipelines were first used in theIBM 7030 (also known as Stretch) and a few other computers of the 1960s. Theyreemerged in the 1980s as key contributors to the high performance achieved byRISCs. Instruction pipelining has also been successfully incorporated into CISCssuch as the 80X86/Pentium series, beginning with the 80486 microprocessor in1989.

Pipeline structure. The general structure of a pipeline of m stages Sl,S2,...,Smappears in Figure 5.53 (which repeats Figure 4.47). When S, has computed itsresults, it passes them, along with any unprocessed input operands, to 5,+ 1 for fur-ther processing, and S, receives a new set of operands from $,-_,. Thus the pipelinecan contain up to m independent data sets, all in different stages of computation.Buffer registers and other synchronization logic are placed between stages so thatthe stages do not interfere with one another. The performance speedup of aninstruction pipeline derives from the fact that up to m independent instructions canbe in progress simultaneously in the m stages.

The simplest instruction pipeline breaks instruction processing into two parts:a fetch stage 5, and an execute stage 52. Thus a two-stage pipeline increases

Data

Control unit

" "   V \i \1 If

Ri —*- C-   R2 ►- C- H. ... -> R,., ►   c

   i   m

Dataout

Stage Sx

Figure 5.53

Structure of an m-stage pipeline.

Stage S2

Stage Sn

CHAPTER 5

Control

Design

throughput by overlapping instruction fetching and instruction execution. Whileinstruction /, with address A, is being executed by stage S2, the instruction Ij+l withthe

next consecutive address Ai+l is fetched from memory by stage 5,. If on execut-ing /, in S2 it is determined that a branch must be made to a nonconsecutive addressAj*Ai+l, then the prefetched instruction /,+1 in 5, has to be discarded. As we willsee later, techniques exist to minimize the negative effect of branch instructions onpipeline performance.

Figure 5.54 shows an implementation of a two-stage instruction pipeline that iscommon in microprogrammed CPUs. It is the generic microprogrammed controlunit of Figure 5.47 repackaged into two sequential stages. The fetch stage S, con-sists of the microprogram counter (IPC, which is the source for microinstructionaddresses, and the control memory CM, which stores the microinstructions. (CM issometimes considered to lie outside the pipeline proper, with the task of feedingmicroinstructions "into" the pipeline.) Observe how p:PC is appropriately posi-tioned to be the buffer register for Sv It is only necessary to increment (IPC toobtain the next consecutive microinstruction address, which is then fetched whilethe current microinstruction is being executed in stage 52. The execution stage S2contains the microinstruction register |iIR, the decoders that extract control signalsfrom the microinstructions in filR, and the logic for choosing branch addresses.Another preexisting register, this time |iIR, acts as the buffer register for stage 52.Microinstruction execution is much simpler than the corresponding task at theinstruction level. It involves decoding the control and condition-select fields of thecurrent microinstruction |il stored in (J.IR, as well as distributing the resulting con-trol signals. If fil specifies branching, the branch address is obtained directly fromfil itself and fed back to Sv There the branch address is loaded into |iPC, replacing(IPC's previous contents and causing any ongoing fetch operation in 5, to beaborted.

Multistage pipelines. An m-stage instruction pipeline can overlap the pro-cessing of up to m instructions, so it is desirable to use more than two stages tomaximize instruction throughput. The value of m depends on the maximum num-ber of stages into which instruction processing can be efficiently broken. Thisnumber in turn depends on the complexity of the instruction set. the organization

366

SECTION 5.3Pipeline Control

External address

i   u   ii   iStage

Microprogram counteruPC

' i

Control

Branch .address   memory

CM

ii   StageSi

Microinstructionregister uIR

'   \

External   Next-addresslogic   Decoders

:>nditions   ■*—

i
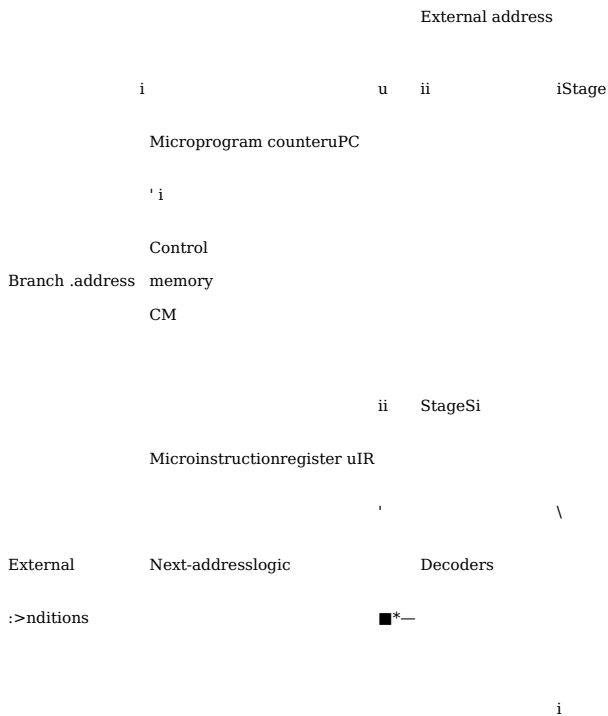
Control signals

Figure 5.54

Two-stage pipelined microprogram control

unit.

of the external memory M, and the way in which the CPU's datapath is imple-mented. In practice, the number of pipeline stages ranges from three (in the case ofthe ARM6) to a dozen or more. Pipeline structure is complicated by the provisionof alternative (parallel) stages, feedback paths, and feedforward (bypass) features.Figure 5.55 shows a CPU organization that implements a four-stage instruc-tion pipeline. We assume that the CPU is directly connected to a cache memory,which is split into instruction and data parts, called the I-cache and D-cache,respectively. This splitting of the cache permits both an instruction word and amemory data word to be accessed in the same clock cycle. Each stage makes use ofcertain common resources such as the cache and the register file RF, which can beregarded as external to the pipeline proper. The four stages 5,:54 of Figure 5.55perform the following functions:

1. IF: instruction fetching and decoding using the I-cache.

2. OL: operand loading from the D-cache to RF.

Mainmemory

Instructions

I-cache

Fetch anddecode

- PC

- IR

5[! instructionfetch (IF)

Mainmemory

L Data

ii

D-cache

ii

u

Data readlogic   ALU

                                                              Data writelogic

                if   1   y   n

                              RegisterfileRF

52: operandload (OL)   53: ALU op-eration (EX)   S4: operandstore (OS)

                    i   A   ' '
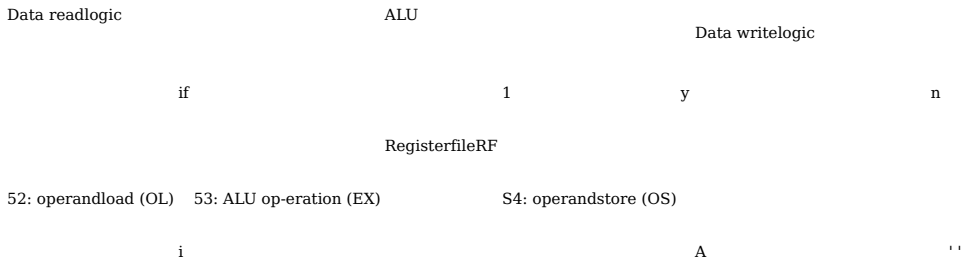
367

CHAPTER 5

Control

Design

Figure 5.55

Organization of a CPU incorporating a four-stage instruction pipeline.

3. EX: data processing using the ALU and RF.

4. OS: operand storing to the D-cache from RF.

Stages S2 and 54 implement memory load and store operations, respectively, andare tailored to a load-store architecture. Stages S2, S3, and S4 share the CPU's localregisters in RF; these registers act as interstage buffer registers. The CPU's ALU isin stage 53 and implements data-transfer and data-processing operations of the reg-ister-to-register type. If each stage completes its operation in a single CPU clockcycle of period Tc, the pipeline and the CPU as a whole can be clocked at a fre-quency of/= \ITC. At its maximum execution rate, which implies that no delaysoccur due to instruction branching, cache misses, or other causes, an ideal perfor-mance level of 1 clock cycle per instruction, or a CPI of 1, can be achieved.

We can vary the organization shown in Figure 5.55 in many ways to tradehardware cost for performance. For example, a less expensive D-cache cannot per-form loads and stores simultaneously, in which case we can implement D-cacheaccesses in a single stage, thus merging S2 and 54 into a single load-store stage.Memory or register-file accesses are complicated by addressing modes such asindexing, which require an ALU to calculate a memory address before the accessoperation proper can be initiated. In such cases it may be desirable to add a stage,that is, a separate clock cycle, for operand address calculation. Instructions such asthe more complicated arithmetic operations require multiple clock cycles for theirexecution; hence they require multiple cycles through the execution stage of apipelined CPU. Such considerations, and the hardware/performance trade-offs they

368

SECTION 5.3Pipeline Control

entail, give rise to the many different instruction pipeline organizations in contem-porary computers.

EXAMPLE 5.7 PIPELINE ORGANIZATION OF THE MIPS R2/3000 [KANE

and heinrich 1992]. The R2000 and R3000 are.early members of the MIPSRX0O0 series of RISC microprocessors, which we discussed in Examples 3.5 and 3.7.They implement the same MIPS-I instruction-set architecture and have nearly identicalCPU organizations, so we will treat them as a single machine denoted by R2/3000.Later members of the same series have numerous architectural extensions and far morecomplex instruction pipelines.

The R2/30O0 employs a five-stage instruction pipeline whose stages have the fol-lowing functions designed to meet the goal of completing one instruction per clockcycle:

1. IF: instruction fetching using the I-cache.

2. RD: operand loading (reading) from the register fde RF while decoding the fetchedinstruction.

3. EX: data processing using the ALU and RF as needed.

4. MA: operand accessing (load or store) using the D-cache.

5. WB: operand storing (writing back) to RF.

Comparing this pipeline organization with that of Figure 5.55, we see that the first andthird stages are roughly the same. The R2/3000's instruction-fetch (IF) stage is compli-cated by the use of virtual memory, which requires that the (virtual) addresses appear-ing in the input instruction stream be translated on the fly into physical addressescorresponding to the available main memory. Consequently, instruction decoding isdeferred to the second stage of the pipeline. This operand-read (RD) stage also trans-fers any needed input operands from the CPU's 32-word register file RF in preparationfor execution in stage 3 (EX). All memory data accesses (D-cache loads and stores) usestage 4 (MA), which transfers a data word between the CPU and the D-cache. The fifthor "write back" (WB) stage is used by load instructions to write a word fetched fromthe data cache into RF. The result of an ALU operation is also stored in RF during theWB stage.

Like other RISCs, the R2/3000 aims at single-cycle execution of its instructions.Figure 5.56 shows the ideal situation when, after a start-up phase during which it fillsup, the instruction pipeline is fully utilized and outputs a new result every clock cycle.

        IF

        RD

        EX /. /, h u h h h h h

        MA

        WB

            'i h h h h h h h

Stage   h h h h h k h

              h h h h h

              h h h h h
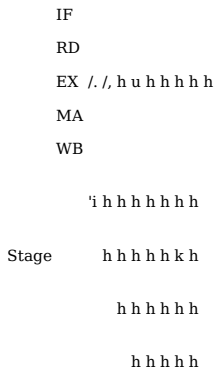
Instruction fetch (I cache)Read from register fileExecute ALU operationMemory access (D cache)Write back to register file

1 23456789

Time (clock cycle)

Figure 5.56

Maximum-rate instruction execution in the R2/3000 instruction pipeline.

CHAPTER 5Control

If, in this "streaming" mode of operation, an instruction computes a new result in 369clock cycle i during the instruction's EX phase, that result can be used by

anotherinstruction in cycle i + 1. In some cases, notably load and branch instructions, this isnot true, and delays occur due to the effects of an instruction on a subsequent one. Forexample, suppose that an instruction that loads a data word X into a register is immedi- esign

ately followed by an instruction that uses X in its EX stage. Then, as illustrated in Fig-ure 5.57a, a one-cycle gap or delay slot occurs in the instruction stream because an LD(load) instruction's data is not available until after its MA cycle.

Several actions can be taken to deal with this situation:

• The pipeline can be temporarily halted or stalled whenever a load or branch instruc-tion is executed. This action, however, complicates control of the pipeline and thesynchronization of CPU operation and causes a loss in performance.

• A NOP (no operation) instruction can be inserted as shown in Figure 5.57i>, whichhas the effect of synchronizing the issuing and execution of all instructions, none ofwhich now needs to be delayed. This action does not improve the pipeline's perfor-mance, however.

• A nearby instruction that does not depend on X can be taken and repositioned in theinstruction stream—which requires a smart compiler—immediately after the LDinstruction. This approach is illustrated in Figure 5.57c, where the SUB instructionhas been moved to fill LD's delay slot. Restructuring of this type is valid only if itdoes not alter the program's final results; for example, it requires that SUB not usethe data fetched by LD as an input operand. The net effect is to make the pipelineoperate at its maximum rate and to complete the four indicated instructions using onecycle fewer than before.

A similar delay problem arises in the case of branch instructions. The branchaddress computed by an R2/3000 branch instruction / does not become available foruse until fs third (EX) stage, which creates a delay slot in Fs second (RD) stage.Another instruction falling into this delay slot is executed, regardless of whether thebranch is taken or not. Consequently, a compiler inserts a NOP into this slot unless, asin Figure 5.57c, the delay slot can be filled in some useful way that does not change theprogram's overall behavior.

Figure 5.58 summarizes the structure of another multistage instruction pipe-line, that of the Amdahl 470V/7, a 1978-vintage machine designed to be compati-ble with the IBM System/370 series of mainframe computers [Amdahl 1978]. The470V/7's memory system comprises a main memory and a single or unified cache(termed the high-speed buffer in Amdahl literature) intended for both instructionand data storage. The CPU is partitioned into a 12-stage pipeline, whose stageshave the roles listed in Figure 5.58. These perform the same four functions as thegeneric pipeline of Figure 5.55, namely, instruction fetching and decoding (IF),operand loading (OL), instruction execution (EX), and, finally, operand storage(OS). Because of the many addressing modes and instruction types needed to sup-port the 470V/7's CISC architecture, each of the preceding functions is subdividedinto several pipeline stages. The first two stages S{ and S2 communicate with amemory control unit that is responsible for all accesses to main memory and thecache. These stages transfer instructions or data operands between the pipeline andthe cache. All results are checked for errors in stage Su using parity-check codes inmost cases. If an error is detected, the instruction in question is automatically re-executed, an error-recovery technique called instruction retry.

370

SECTION 5.3Pipeline Control

IF   LD ADD ST   SUB

RD   LD   ADD ST   SUB

Stage Ex   LD   *   ADD ST   SUB f

MA   LD   ADD ST   SUB

WB   LD   ADD ST   SUB
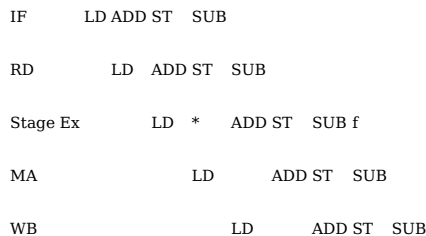
*Input operand ofADD unavailable

3 4 5 6 7

Time (clock cycle)

(a)

IF   LD NOP ADD ST   SUB

RD   LD   NOP ADD ST   SUB

Stage EX   LD   NOP ADD ST   SUB

MA   LD   NOP ADD ST   SUB

WB   LD   NOP ADD ST   SUB

4 5 6 7

Time (clock cycle)

IF   LD SUB ADD ST

RD   LD   SUB ADD ST

Stage EX   LD   SUB ADD ST

MA   LD   SUB ADD ST

WB   LD   SUB ADD ST
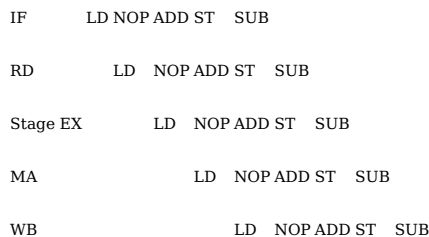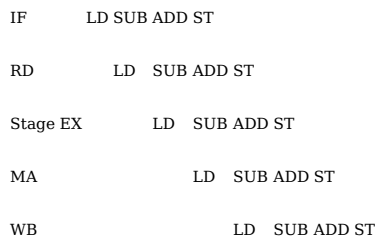
12 3 4 5 6 7 8

Time (clock cycle)

(c)

Figure 5.57

(a) R2/3000 pipeline delay slot caused by load instruction LD;

(b) use of NOP instruction to fill the delay slot; (c) use of SUBinstruction to eliminate the delay slot.

Function

Stage Name

Action performed

Instruction	S,	Instruction address
fetch IF	5,	Start buffer
	si	Read buffer
	s4	Decode instruction
Operandload OL	s5	Read registerCompute addressStart buffer
	s&	Read buffer
Execute	s„	Execute 1
instruction EX	•^10	Execute 2
Operandstore OS	s„	Check resultWrite result

Request next instruction from memory control unitInitiate cache to read instructionRead instruction from cache into I-unitDecode opcode of instruction

Read address (base and index) registersCompute address of current memory' operandInitiate cache to read memory operandRead operands from cache and register file

Pass data to E-unit and begin instruction executionComplete instruction execution

Perform code-based error check on resultStore result

Figure 5.58

The stages of the Amdahl 470V/7 instruction pipeline.

In recent years the large number of pipeline stages illustrated by the 470V/7have become common, because such fine-grained stages enable a pipeline tooperate at higher clock frequencies. Multiple-instruction pipelines are also com-mon, especially in superscalar processors, which can issue (dispatch) two ormore instructions simultaneously. For example, each of the three functional units(E-units) of the PowerPC 601 microprocessor (Example 1.7) is implemented as adistinct pipeline; the structure and relationship of these pipelines are outlined inFigure 5.59 [Becker et al. 1993]. The 601 has an instruction buffer or queue thatstores up to eight instructions which are prefetched from the single (unified)cache memory. In each clock cycle this buffer can send a separate instruction toeach of the pipelined E-units. The two-stage branch-processing unit fetches andprocesses branch instructions. The five-stage fixed-point unit processes fixed-point ALU operations and also handles cache data accesses both for itself and forthe floating-point unit. Some operations, such as multiply and divide, circulaterepeatedly through the execute stage. The floating-point unit supports a fullrange of floating-point instructions, including a compound multiply-and-addinstruction.

371

5.3.2 Pipeline Performance

The goal in controlling a pipelined CPU is to maximize its performance withrespect to target workloads. After reviewing the performance measures applicableto instruction pipelines, we consider the factors that reduce performance and howthey can be overcome.

Performance measures. A pipeline's performance can be measured by itsthroughput in terms of millions of instructions executed per second or MIPS.Another popular measure of performance is the number of clock cycles per

372

instruction or CPI. These quantities are related by the equation

$$CPI = f/MIPS \quad (5.22)$$

where /is the pipeline's clock frequency in MHz. and the values of CPI and MIPSare average figures that can be determined experimentally by processing suites ofrepresentative programs (benchmarks). The maximum value of CPI for a singlepipeline is one, making the pipeline's maximum possible throughput equal to/.This throughput is attained only when the pipeline is supplied with a continuousstream of instructions that keep all its stages busy. Superscalar machines reduceCPI below one by executing several instruction streams simultaneously using mul-tiple pipelines.

Figures 5.56 and 5.57 illustrate a useful way to visualize pipeline behaviorcalled a space-time diagram, which shows the utilization of each pipeline stage asa function of time. In general, a space-time diagram for an /n-stage pipeline hasthe form of an m x n grid, where n is the number of clock cycles to complete theprocessing of some sequence of N instructions of interest. Figure 5.60 shows aspace-time diagram for the four-stage arithmetic pipeline of Example 4.8. which isexecuting a complex vector summation instruction denoted /. An unshaded box inthese figures marks a busy stage 5,, and the box's entry denotes the particularinstruction being processed by 5,. As the shading shows, some stages are not uti-lized at the beginning and end of the instruction sequence, when the pipeline mustbe filled and emptied (flushed), respectively. The stages are also underutilized ifoperands are not available when needed. The ratio of the unshaded (busy) area to

Instructionfetch

(branchinstruc-tions)

Decode

andexecute
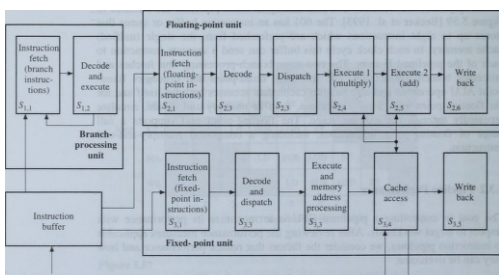
Cache (unified)

Cache (unified)

Figure 5.59

Instruction pipelining in the PowerPC 601.

the total (shaded and unshaded) area of a space-time diagram for an m-stage pipe-line is defined as the efficiency or utilization E{m} of the pipeline. In other words,E(m) is the fraction of time the pipeline is busy. In the case of Figure 5.60, theefficiency is E(4) = 44/76 - 0.58. Note how the instruction reordering shown inFigure 5.57c improves the pipeline's efficiency by eliminating the delay slot.

Another general measure of pipeline performance is the speedup S(m) definedby

373

S(m) =

7X0T{m}

(5.23)

where T(m) is the execution time for some target workload on an wi-stage pipelineand 7"(1) is the execution time for the same workload on a similar, nonpipelinedprocessor. It is reasonable to assume that T{\} <mT(m), in which case S(m) < m. Apipeline's efficiency and speedup are related as follows:

S(m) = m x E(m)

(5.24)

Hence for the example in Figure 5.60 where m = 4 and £(4) = 0.58, the speedup5(4) = 4 x 0.58 = 2.32 and cannot exceed 4. In general, speedup and efficiency pro-vide rough performance estimates which should be used with caution, since theydepend on the programs being run. Their values can change drastically from pro-gram to program, or from one part of a program to another.

Optimizing m. Equation (5.24) suggests that an easy way to improve a pipe-line's performance is to increase the number of stages m. This assumes that thepipeline's processing tasks can be subdivided in a useful way and that the cost ofdoing so is acceptable. Each new stage S, introduces some new hardware cost anddelay due to its buffer register Ri and associated control logic. We now analyze thetrade-offs involved in doing this [Kogge 1981; Hwang 1993]. In particular, we willdetermine the pipeline's performance/cost ratio PCR defined as

PCR-L

where/is the pipeline's clock frequency and K is its hardware cost.

(5.25)

CHAPTER 5

Control

Design

s} //////// / /   /

s2 //////// / /   /

*3   //////// / /   /

54   //////// / /   /
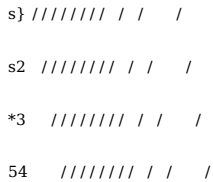
Stage

12 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Time (clock cycle)

Figure 5.60

Space-time diagram for a four-stage pipeline.

374 Suppose the pipeline P has m stages and implements a particular set of opera-

section 5 3 tions (instructions) SI. Let a be the delay (latency) of an efficient, nonpipelined

Pipeline Control processor that also implements SI. It is reasonable to assume that each stage 5, of P

has delay a/m—that is, m times less than the corresponding nonpipelined proces-sor—plus some extra delay b due to 5,'s buffer register /?,-. Hence if Tc = l//is P'sclock period, we can write

Tc = a/m + b (5.26)

The pipeline's hardware cost can be estimated by

K=cm + d (5.27)

where c is the buffer-register cost per stage and d is the cost of the pipeline's (com-binational) data-processing logic. Hence from (5.25), (5.26), and (5.27) we havePCITl= TCK = {aim + b){cm + d), so

PCR = m/[bcm2 + (ac + bd)m + ad] (5.28)

To maximize PCR with respect to the number of stages m, we differentiate (5.28)with respect to m and equate the result to zero. Using the standard differentiation-by-parts formula

we obtain

d ru\ \du u dvd~x\y) vdx v^dx

-j^PCR) = 1/v - m(2bcm + ac + bd)/^ (5.29)

where u-m and v = bcm2 + (ac + bd)m + ad. On equating (5.29) to zero, we getv = m(2acm + ad + be). Substituting for v and solving for m yields the value m t ofm that maximizes PCR, namely,

ladmoPt = Jfc (5-30)

The optimum number of stages is the integer closest to m0 t. Figure 5.61 plots PCRagainst m according to (5.28) for a = d = 5 and b = c = 1. The optimum value of mis five, as predicted by (5.30). Hence in this instance, the maximum throughput perunit of hardware cost or, equivalently, the minimum cost per instruction processed,occurs when the pipeline has five stages.

Collisions. As discussed in section 4.3.2 pipelines can have feedback pathsthat enable a stage to be used repeatedly while processing a single instruction. InFigure 5.60, for example, each stage is used many times while processing theinstruction / (vector summation). A new instruction of the same kind cannot bestarted until clock cycle 17, after / uses stage Sx for the last time. If a secondinstruction is initiated at the wrong time—at t = 9, for instance—then both instruc-tions will attempt to use stage 5, at r = 10, a situation termed a collision. However,a simple add instruction of the kind in Figure 5.57 could be initiated at t = 9, 11, 13,14, or 15 without colliding with the sum instruction /. Thus up to five add instruc-tions, if available for execution at the right times, could be interleaved with theeight-element sum operation, thereby increasing the pipeline's overall efficiency.

Performance/costratio PCR

0.06

0.05

0.04

0.03

0.02

0.01
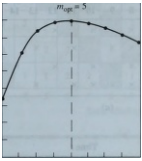
13 5 7 9

Number of pipeline stages m

Figure 5.61

Performance/cost ratio PCR for an/n-stage pipeline.

CHAPTER 5

Control

Design

In general, pipeline collisions of the foregoing type are avoided by carefullyscheduling the times at which new pipeline operations are initiated. We present apipeline control strategy to avoid collisions and maximize performance in a pipe-line with feedback or feedforward connections [Kogge 1981; Stone 1993]. Let P besuch a pipeline that consists of m stages S],S2,...,Sm and executes an instruction oftype /. (Later we will consider the problem of scheduling different types of instruc-tions in the same pipeline.) We can represent /'s usage of the pipeline with a space-time diagram (refer to Figure 5.60), which indicates stage usage in every clockcycle while / is being executed. We will also represent the same information in aslightly different form R called a reservation table. The m rows of R represent thestages of P, while the columns represent the sequence of clock cycles required forone complete execution of / by P. An x is placed at the intersection of row S, andcolumn Cj if stage S, is used by / in clock cycle / =7'. Figure 5.62a shows the reser-vation table corresponding to Figures 4.52 and 5.60. If the method of Figure 4.52 isused to sum the pair of numbers bx,b2, then the small reservation table of Figure5.62b results.

Two operations of type / that are initiated k clock cycles apart collide at stage5, of P, if row i of the corresponding reservation table R contains two xs that areseparated by a horizontal distance of k. In the case of Figure 5.62b, a collisionoccurs at every stage if k = 1, 4, or 5, as is easily verified. For example, if the firstinstruction is initiated at / = 1 and the second at t = 5, in which case k = 4, then bothinstructions will attempt to use all four stages and collide at t = 6. Let F be the setof numbers, called the forbidden list of R, whose entries are the distances, that is,the numbers of clock cycles between all distinct pairs of xs in every row of R. Thecollision conditions for R are characterized by the following easily proven result:Two pipeline instructions initiated k clock cycles apart collide if and only if A: is inthe forbidden list F of R. Thus we can easily meet the fundamental requirement ofavoiding collisions by delaying new instructions by time periods not appearing inthe forbidden list. Much less obvious is how to schedule initiation times that maxi-mize the pipeline's performance.

The maximum number of collision-free operations that can be initiated per unittime under steady-state conditions corresponds to the pipeline's throughput defined

376

SECTION 5.3Pipeline Control

TimerStage 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

5[xxxxxxxx x x x

52 xxxxxxxx x «x x

53 XXXXXXXX X X X

54 XXXXXXXX X X X

(a)

Timef

Stage

5, xx

52 xx

53 x54

Figure 5.62

Pipeline reservation tables for yV-element vector summation: (a) N = 8 corresponding toFigure 5.60 and (b) N = 2.

by Equation (5.22). The delay occurring between the start of two successive, colli-sion-free pipeline instructions is called the initiation latency, or simply the latencyL in this context. Under steady-state operating conditions, L corresponds to thepipeline's CPI and is measured in clock cycles. We now turn to the problem ofdevising control strategies that maximize the performance of a basic, single-func-tion pipeline by determining the best values of L to use for collision-free operation.We denote the minimum value of the initiation latency L, that is, the minimumaverage latency by L^. A simpler goal is to achieve the minimum constantlatency, defined as the smallest fixed value Lcmin of L such that any number ofinstructions can be initiated L clock cycles apart without causing collisions.Clearly, Lmin < Lcrmn. The number Lcmin can be calculated from the forbidden list Fusing the fact that Lcmin is the smallest integer L such that hL is not in F for anyinteger h > 1. The forbidden lists for the reservation tables of Figures 5.62a and5.626 are {1,2, 3,4, 5, 6,7, 8, 9, 10, 11, 12, 13,14, 15} and {1, 4, 5}, respectively.Thus, as observed earlier, successive sum instructions with the reservation table ofFigure 5.62a must be initiated at least 16 clock cycles apart, since the latencies Lminand Lcmin are both 16. In the case of Figure 5.62b, new instructions can be initiatedas few as two cycles apart. However, the minimum constant latency Lcmia * 2,because 2 x 2 = 4 is in F; in this case Lcmin = 3. If instructions are initiated at t = 1and 3, a third instruction cannot be initiated until t = 9, as demonstrated by thespace-time diagram of Figure 5.63a. The average initiation latency for the pipelinescheduling scheme defined by Figure 5.636 is four, because two new instructions

Stage

5, 1 1 2 2   1   2 3 3 4 4   3   4 5

Si   1 1 9 2   1   2 3 3 4 4   3   4

*3   1 1 2 2   1   2 3 3 4 4   3

54   1 1 2 2   1   2 3 3 4 4   3

3 4 5 6 7 8 9 10 11 12 13 14 15 16 17Time (clock cycle) t

(a)

5, 11 22133244355466

52 11 2213324435546

53 1; 221332443554

54 11 22133244355

Stage

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 Time (clock cycle) t

(b)

Figure 5.63

Pipeline scheduling strategies for the reservation table of Figure 5.62ft: (a) nonoptimal and(ft) optimal.

are initiated every eight clock cycles. The minimum average latency is achievedfor this example when new operations are initiated $L^- = 3$ clock cycles apart, as inFigure 5.63ft. Observe that in the latter case the steady-state efficiency of the pipe-line is 100 percent.

Control scheme. An elegant way to control a pipeline for collision-free oper-ation is by computing collision vectors. A collision vector CV for a reservationtable R at time t is a binary vector $cxc_2 cM\_]cM$, where the ith bit c, is 1 if initiat-ing a pipeline instruction at $t + i$ results in a collision; c, is 0 otherwise. An initialcollision vector CV0 is obtained from the forbidden list F of R as follows. ElementCj of CV0 is set to 1 if i is in F, and c, is set to 0 otherwise, for i = 1,2,...,M, whereM is the maximum element in F. A convenient way to store CV is in a shift regis-ter CR = CR,:CRM called a collision register. By inspecting CR, at time /, we candetermine whether issuing a new instruction in the next clock cycle $t + 1$ willresult in a collision. A simple left shift of CR, with the right-most bit CRM set to 0prepares CR, for inspection in the next clock cycle. If we decide to initiate a newinstruction at $t + 1$, then CR is left shifted and its contents are replaced by CR orCV0, where CV0 is the initial collision vector obtained from F as specified "above

and or denotes the bitwise OR operation. These actions ensure that CR defines allthe collision possibilities due either to ongoing pipeline operations or to the newlyinitiated one.

To illustrate the foregoing concepts, consider again the reservation table inFigure 5.62b. Since F = {1,4, 5},M = 5 and the corresponding initial collision vec-tor CV0 is 10011. The collision register CR is initialized to 00000. When it isdecided, say, at t = 0, to start the first pipeline instruction at t = 1, CR is left shiftedand ORed with CV0, resulting in CR = 00000 or 10011 = 10011 = CV0. At t = 1 thenew pipeline instruction is initiated, and CR is again inspected. Since CR, = 1, weconclude that a new instruction must be delayed; CR is merely shifted during thiscycle, changing its contents to 00110. At t = 2, CR contains 00110 with CR, =0,allowing a new instruction to start at t = 3. If a second pipeline operation is initiatedin the next cycle, CR is shifted and ORed with CV0, therefore becoming 01100 or10011 = 11111 at t = 3. Five subsequent shifts are needed before CR, againbecomes zero at t = 8. A third instruction cannot therefore begin until / = 9, asshown by Figure 5.63a. If no new task is started at t = 3, CR is 01100, indicatingthat a new instruction can begin at t = 4. If a new instruction is then initiated at / =4, CR becomes successively 11000 and 11000 or 10011 = 11011, implying that athird instruction can begin at / = 7; Figure 5.63ft depicts this situation. Observe thatat t = 6. CR becomes 01100, repeating the pattern encountered at t = 3.

Task-initiation diagram. We can derive an optimal collision-free schedulefor initiating pipeline operations from the state behavior of the collision registerCR. For this purpose we construct a condensed state-transition graph for CR calleda task-initiation diagram (TID). The states of the TID are all the collision vectors{CV,} formed by the operation CR or CV0, when new pipeline operations can beinitiated. (The other states of CR are formed by shifting these vectors and areexcluded from the TID.) An arrow from CV, to CV indicates that there is asequence of state transitions that changes CR"s state from CV, to CV;; the arrow islabeled with the minimum number of state transitions nVj required. Thus n^- denotesthe minimum latency between the initiations represented by the TID states CV, and

CV0= 10011

CV, = 11011

CV, = 11111

Figure 5.64

Task-initiation diagram (TID)for Figure 5.63ft.

CV-. A closed path or loop in the TID corresponds to the task initiation schedule 379for the pipeline that can be sustained indefinitely without collisions. Let s be thesum of the «,-,■ labels along the arrows forming the loop divided by the number of „ .arrows in the loop. Clearly, s is the average latency of the corresponding schedule Designof pipeline task initiations. Therefore, the average latency of the pipeline is mini-mized by choosing a task sequence corresponding to a loop of the TID with a min-imum value of s, which is then the minimum average initiation latency Lmin [Kogge1981].

Figure 5.64 shows the TID derived from the reservation table of Figure 5.62b,where the initial collision vector CV0 = 10011. This TID is obtained in straight-forward fashion by examining all the possible states and state transitions of thecorresponding CR as described earlier. The states included in the TID are allthose loaded into SR when new operations can be initiated, namely, the threestates CV0 = 10011, CV, = 11011, and CV2 =11111 identified above. For exam-ple, the self-loop labeled 3 on state CV! is a consequence of the followingsequence of state transitions involving CR:

Clock

cycle State of CR Actions taken

Initiate new instruction. Left shift CR.

Left shift CR.

Select new instruction to initiate. Left shift CR. CR := CR orCV0= 11000 or 10011.

Initiate new instruction. Left shift CR.

The TID of Figure 5.64 contains several loops corresponding to pipeline controlstrategies with different average initiation latencies. For example, the loopformed by the two arrows linking CV0 and CV, has an average initiation latencyof (2 + 6)/2 = 4 cycles and corresponds to the space-time diagram of Figure5.63a. The self-loop of state CV, has the minimum average latency Lmin = 3 andtherefore maximizes pipeline performance; using this loop for pipeline controlyields the space-time diagram of Figure 5.63b. The analysis confirms our previousobservation that the optimum scheduling strategy for this example is to initiate anew instruction three cycles after the previous instruction. Hence a simple logiccircuit based on a modulo-3 counter suffices to control this particular pipeline.

The progress of an instruction stream through a pipeline can be delayed by var-ious unfavorable dependency relationships among instructions and their data oper-ands, which are collectively referred to as hazards. We now define the main typesof pipeline hazards and discuss some general ways to detect them and reduce theirimpact on performance.

Control dependencies. Conditional and unconditional jumps, subroutinecalls, and other program-control instructions that involve branching can adverselyaffect the performance of an instruction pipeline. In these cases the address of thenext instruction is not known with certainty until after the program-control instruc-tion / has been executed. Hence the question arises: Which instructions should be

t   11011 =cv.

t+l   10110

t + 2 01100

t'+3 11011 =cv

380 entered into the pipeline immediately after /? If these happen to be the wrong

instructions, that is, / causes a jump to a distant part of the program, then provision

Pi line Control must ^e mac^e t0 cancel tne effects of the partially executed instructions. This pro-

cess is sometimes termed flushing the pipeline and clearly reduces its throughput.In the case of a two-way branch, it is sometimes worthwhile for the compiler or thepipeline's control logic to "guess" the direction of the branch, that is, to anticipatethe outcome of the branch condition test, and enter the instruction at /'s more likelytarget address into the pipeline immediately after /. This process is known as spec-ulative execution. Pipeline flushing is then needed only when the wrong guess hasbeen made.

We can estimate the influence of branch instructions on the performance of aninstruction pipeline as follows. Suppose that the pipeline has m stages and that eachinstruction requires m clock cycles, corresponding to one complete pass throughthe pipeline. If there are no branch instructions in the instruction streams being pro-cessed, then an ideal throughput of one instruction per clock cycle is achieved; thatis, CPI = 1. Let p be the probability of encountering a branch instruction, and let qbe the probability that execution of a branch instruction / causes a jump to a non-consecutive address. Assume that each such jump requires the pipeline to beflushed, destroying all ongoing instruction processing, when / emerges from thelast stage (a pessimistic assumption).

Now consider an instruction sequence of length r that is streaming through thepipeline. The number of instructions causing branches to take place is pqr, andthese instructions are executed at a rate of Mm instructions per cycle. The remain-ing (1 - pq)r nonbranching instructions are processed at the maximum rate of oneinstruction per cycle. Hence the total number of cycles nc needed to process all rinstructions is

nc = pqrm + (1 - pq)r

This implies that the average CPI of the pipeline, which by definition is njr, isgiven by

CPI = 1 + pq{m - 1) (5.31)

with the optimum value CPI = 1 occurring when q = 0, that is, when no branchingoccurs during program execution. Note that a comparable nonpipelined instructionprocessor has CPI = m. If p = 0.2, q = 0.4, and m = 5, which are typical values forinstruction pipelines, then (5.31) implies that CPI = 1.32. Hence, in this case, pipe-lining reduces the number of cycles per instruction from 5 to 1.32, an improvementby a factor of about four. The improvement is less for longer pipelines, since eachbranch to a nonconsecutive instruction address causes more partially processedinstructions to be discarded. A compiler or programmer can increase throughput byemploying fewer branch instructions (to reduce p) and by constructing conditionalbranch instructions so that the more probable results of the condition tests cause nobranching (to reduce q).

Pipelined computers employ various hardware techniques to minimize the per-formance degradation due to branching. The Amdahl 470V/7, for example, hasspecial branch-resolution logic to send the result of a branch condition test from theE-unit to the I-unit before the conditional branch instruction has been completelyprocessed. This logic allows the I-unit to initiate processing of the correct next

CHAPTER 5Control

instruction with a loss of data in only 3 of its 12 pipeline stages. A different 381approach is taken by the IBM 3033. Its cache is divided into three separate instruc-tion buffer areas: One holds a normal sequence of consecutive instructionsprefetched under the assumption that no branches will occur; the other two buffers Designhold prefetched instruction sequences starting at up to two branch addresses speci-fied by previously decoded branch instructions. Thus when the 3033's CPUdecodes an unconditional branch instruction of the form go to A-, and has aninstruction buffer with available space, it proceeds to prefetch and process instruc-tions starting at location Aj. In the case of a two-way conditional branch instructionwith two target branch addresses Aj and Ak, the CPU selects one branch address forprefetching. If, when the conditional branch instruction is subsequently executed, itturns out that the wrong selection was made by the CPU, then time is lost while thecorrect instruction is fetched. If the CPU has anticipated the outcome of the condi-tion test correctly, then the required next instruction is either already in the instruc-tion pipeline or is stored in an instruction buffer.

RISC machines rely on instruction pipelines that overlap instruction fetch andexecute to achieve single-cycle execution for most instructions. As we saw in thecase of the MIPS R2/3000 (Example 5.7), special measures are taken to nullify thedelay slots associated with load and branch instructions. A closely related techniquecalled delayed branching is used in some RISCs to reduce the penalty due to pipelineflushes on program branching. A delayed branch instruction /, causes the instructionI2 immediately following Ix to be executed while the instruction /' at the targetaddress specified by Ix is still being fetched. The execution of/' then follows that ofI2 rather than following that of Ix, as would normally be the case. For example, theIBM 801, the prototype RISC processor, has an alternative branch-with-executeform of every normal branch instruction [Radin 1983]. Thus the instructionsequence

LOAD Rl, A (532)

BNZ L

for the 801 containing the normal conditional branch instruction BNZ (branch ifnonzero) idles the CPU while the instruction at the branch address L is beingfetched. Suppose that BNZ is replaced by the corresponding branch-and-executeinstruction BNZX and the instruction order is reversed as follows:

BNZX L (5.33)

LOAD R1,A

The modified code (5.33) has the same meaning as (5.32), but now the LOADinstruction is executed while the instruction specified by BNZX is being fetched.The compiler of the 801 is able to translate about 60 percent of program branchesinto the more efficient branch-with-execute form.

Data dependencies. An m-stage pipeline operates at its maximum perfor-mance level when it contains m different instructions, each in a different stage ofcomputation. As we have seen, problems can occur if the decision to execute a par-ticular instruction depends on the outcome of an earlier branch instruction. Thisproblem is due to the program's flow of control and so is called a control depen-dency. Other, more subtle data dependencies can exist among the operands being

382 processed by different instructions and can also reduce the pipeline's throughput.

sfction s 3 ^or examPle' suppose that instruction /, changes the contents of register R and that

p r c ntroi ^ ls reac^ ^y a SUDseciuent instruction /, in the generic instruction pipeline of Figure

5.55. If I2 is in stage S2 (operand read) while /, is in stage 53 (execute), then I2 willread an old. and possibly erroneous value of R. since (, does not write its result toR until it reaches stage 54. Thus although the instructions have been dispatched inthe proper order required by the program, their read and write steps can be pro-cessed in a logically incorrect order within the pipeline. This data dependencyproblem is known as a read-after-write (RAW) hazard. It is solved by requiring /,to complete its execution before I2 enters the operand read stage, which may meanreducing the throughput of the pipeline.

To identify hazards of the foregoing type, we consider the sets of input andoutput operands (registers or memory locations) associated with each instruction /entering the pipeline. The set of input operands of /; is defined as the domain of Land is denoted by /)(/•); the set of output operands of / is its range and is denotedby R(Ij). For example, the instruction / for the MIPS R2/3000

ADD R1,R2,R3

which denotes the 32-bit addition Rl := R2 + R3, has the domain D{I) = {R2,R3}and the range R(f) = {Rl}. If I2 follows lx in program order, then a RAW hazardindicating a potential error situation exists if /?(/,) and D(I2) contain a commonoperand. This condition is expressed formally as

R{IX) n D(/2) * 0 (RAW hazard) (5.34)

where n denotes set intersection and 0 denotes the empty set.

A similar problem called a write-after-read (WAR) hazard is present if thecondition

D(IX) n R(I2) * 0 (WAR hazard) (5.35)

holds. In this case an error occurs if the second instruction I2 modifies an operandbefore it can be read by the first instruction /,. Unlike the RAW hazard, a WARhazard cannot occur in a pipeline such as that of the R2/3000 (Figure 5.56) becauseof the relative positions of the read and the write stages. The only stage that readsfrom registers is S2 (RD), which precedes the only stage S5 (WB) that writes to reg-isters, so by the time I2 reaches the write stage, /, has left the pipeline. Only onestage S4 (MA) controls memory reads and writes, and Ix always reaches this stagebefore I2.

A third type of data-dependency hazard is defined by

W(/i) n W(I2) * 0 (WAW hazard) (5.36)

and is known as a write-after-write (WAW) hazard. It is present if the pipelineallows /2 to modify an operand before the same operand is modified by /,. TheRAW, WAR, and

WAW hazards are also known as true, anti, and output datadependencies, respectively. Clearly, data-dependent hazards depend on both thestructure of the instruction pipeline and the order of the instructions that accesscommon registers or memory locations.

A pipeline hazard due to a data dependency can be detected by checking forthe necessary conditions given by (5.34), (5.35), and (5.36), either during compila-

tion (static hazard detection) or at run time (dynamic hazard detection). Such ahazard can be avoided by preventing the second member I2 of a hazardous instruc-tion pair (I\J2) from entering a read or write stage until the first instruction Ix hasexited from the subsequent read or write stage associated with the hazard. As in thecase of the control (branch instruction) hazards discussed earlier, we can avoid thehazard by delaying I2 either by stalling it, preceding it by one or more NOPs, or—most efficiently—reordering the instruction stream so that useful instructions,which neither slow down the instruction stream nor alter program behavior, areplaced between Ix and I2.

Another way to reduce the delays due to hazards is to build into the pipelineextra operand-transfer paths that permit faster exchange of shared informationamong interacting instructions. Consider, for example, a five-stage pipeline likethat of the MIPS R2/3000. A result R computed by the ALU in stage 53 (EX) is notwritten into the register file until stage S5 (WB) two cycles later. By adding anoperand-transfer "forwarding" path Pa from the output S4 to the input of S2 (RD),the result X computed by /, can be made available to I2 one cycle earlier thanbefore. As shown in Figure 5.65, we can even forward X from the output of 53 tothe input of the same stage via another path Pb so that X is supplied with no delaypenalty to an ALU instruction I2 that immediately follows /,. While forwardingpaths of this type reduce the delay penalties associated with hazard avoidance, theyalso add considerable complexity to the pipeline's control logic.

383

CHAPTER 5

Control

Design

1   r

5j: Fetch and decodeinstruction (IF)

■   '

S2: Read fromregister file (RD)

r 1

I ■ r

S3: Perform ALUoperation (EX)

p>   1

r

54: Memoryaccess (MA)

fa

Sy Write toregister file (WB)
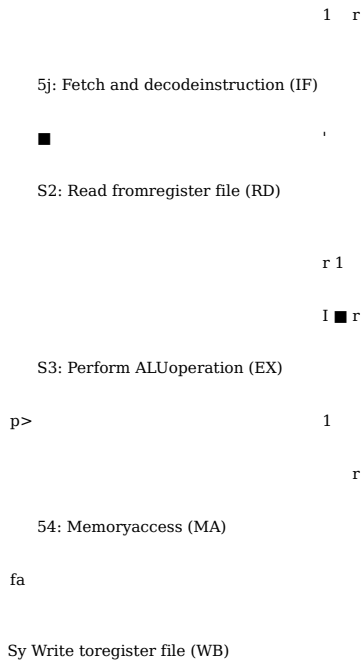
Figure 5.65

Pipeline with forwarding paths to reduce hazard-causeddelays.

384 5.3.3 Superscalar Processing

SECTION 5.3

Pipeline Control Microprocessors such as the PowerPC (Figure 5.59) reach performance levels

greater than one instruction per cycle—that is, a CPI figure less than one—byfetching, decoding, and executing several instructions .concurrently. This mode ofoperation is called superscalar. A superscalar computer has a single CPU thatattempts to exploit the parallelism that is implicit in ordinary (sequential) computerprograms. It is contrasted with a parallel computer, which can have more than oneCPU and is designed to execute programs whose parallelism is explicit at a high,application level; we discuss parallel computers in section 7.3.

Characteristics. Superscalar operation requires a processor to detect andexploit instruction-level parallelism hidden in the programs it executes. A super-scalar CPU has multiple execution units (E-units), each of which is usually pipe-lined, so that they constitute a set of independent instruction pipelines. The CPU'sprogram control unit PCU is designed to fetch and decode several instructionsconcurrently. It can issue or dispatch up to k instructions simultaneously to thevarious E-units where k, the instruction-issue degree, can be six or more usingcurrent technology. The need to process so many instructions simultaneouslywithout performance-degrading conflicts greatly complicates the design of thePCU. Figure 5.66 shows in idealized form the differences in instruction-process-ing abilities between three CPU organizations: a sequential (nonpipelined) proces-sor, a basic pipelined processor, and a superscalar processor, all of which areexecuting the same instruction stream /,,/2,/3,... Assuming that each instructionrequires a total of five cycles, we see that a single five-stage instruction pipeline(k = 1) offers a speedup of 5, while the two-issue (k = 2) superscalar design has apotential speedup of 10. Observe that at the start of cycle 15, the sequential CPUhas completed only two instructions, whereas the pipelined and superscalarmachines have completed 10 and 20 instructions, respectively; moreover, thesuperscalar CPU has already started processing instructions 721 through 730.

As Figure 5.66 illustrates, the presence of k independent m-stage pipelined E-units enables a superscalar CPU to achieve speedup factors approaching k x m,compared to a CPU that has no instruction-level parallelism. Keeping k pipelinesbusy requires the CPU to fetch at least k instructions per clock cycle; hence super-scalar designs place heavy demands on the instruction-fetch logic. The resultinghigh volume of instruction traffic from the program memory to the CPU requiresthe system to have a large, fast cache, often in the form of an instruction-only cache(I-cache) for program storage, complemented by a data-only D-cache for operandstorage. Instruction fetching is supported by an instruction buffer or queue, a stor-age unit within the CPU that serves as a staging area for prefetched and (partially)decoded instructions. The PCU dispatches the instructions from its instructionbuffer to the various E-units for execution.

The PCU of a superscalar machine is responsible for determining when eachinstruction can be executed and for providing it with access to the resources itneeds, such as memory operands, E-units, and CPU registers, in a prompt and effi-cient manner. To do so, it must take the following factors into account:

• Instruction type: For example, a floating-point add instruction has to be issued toa floating-point E-unit and not to an integer E-unit.

Time / (clock cycles):

1 2   3   4   5 6   7 8 9 10   n   12 13 14 15

1*. | ID. |0L, |ex, |OS, || IF, id-,|ol,|ex,|os,| |lF3 ID, |ol3|ex3|os3|

Instruction I,

Instruction /2(a)

Instruction /3

CHAPTER 5

Control

Design

Instruction fetch IF:Instruction decode IDOperand load OL:Execution EX:Operand store OS:

I7' 1 '- b   h   Ml 6 b '.]   b   ho || /|i hi hi Iu* *l

:   1 7> b   b   Ml h b ''   b   b \\ha •u hi /,3 | ' 4 |

     h   A   Ml '-; b '«   b   b\\ b   ho hi hi | ';-;

     /.   Ml 3   b M b   b || h b   ho /„h M

     Mh :   b   b   b k || b h   b   fa | ' 1
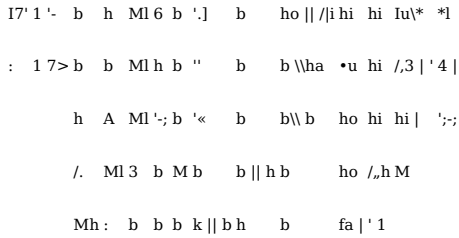
Instruction fetch IF,: l/j T/j

Instruction fetch IF2: A A,Instruction decode ID,: |/i

Instruction decode ID2:Operand load OL,:Operand load OL2:Execution EX,:Execution EX2:Operand store OS,:Operand store OS2:

h

(b)

U '-,

U h

b h

m

/, A

h h

'13 '15 '17 '19

/•.i In h*

I,* I,

In \U

/.1 A

'15 '17 U19 U21 '23

hi 'l4 Ul6

'20 '22 '24

'q '11 ' n ' 1

/n /1

/in /17

An A-.

'7 U> A. /.3 As /.7 A

|/y |/io |/i: I 7711/i6 |/ig \h^\

/5 K7 U9 Ml. /l3 /,5 h

b h \ho\ hi /,4 /.

(C)
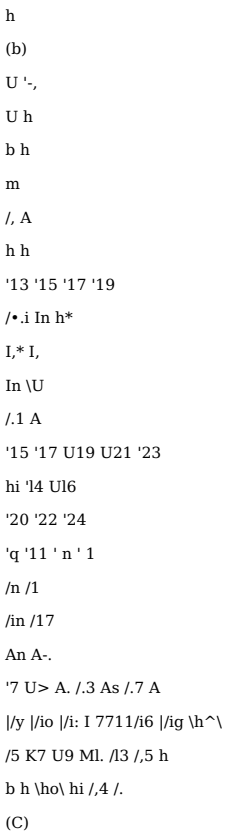
Figure 5.66

Maximum parallelism in (a) a sequential CPU; (b) a CPU with a five-stage instruction pipe-line; (c) a superscalar CPU with two five-stage instruction pipelines.

• E-unit availability: An instruction can be issued to a pipelined E-unit only if nocollisions will result, as determined by the pipeline's reservation table.

• Data dependencies: To avoid conflicting use of registers, data-dependency con-straints among the operands of the active instructions must be satisfied.

• Control dependencies: To maintain high performance levels, techniques areneeded to reduce the impact of branch instructions on pipeline efficiency.

• Program order: Instructions must eventually produce results in the order speci-fied by the program being executed. The results may, however, be computed out-of-order internally to improve the CPU's performance.

Delaying a problematic instruction before it enters an instruction pipeline can pre-vent conflicts. Such static scheduling of instructions can occur during programcompilation, for example, by implementing the collision-avoidance technique

386 discussed in section 5.3.1. We can improve throughput, however, by issuing all

sfction instructions as rapidly as possible and resolving any subsequent conflicts on the

p .. c ' ! fly. We next discuss two control techniques that address these issues: dynamic

instruction scheduling and the branch prediction.

Dynamic instruction scheduling. Sophisticated resource scheduling tech-niques were implemented in some high-performance computers of the 1960s, nota-bly Control Data Corp.'s CDC 6600 and IBM's System/360 Model 91 [Smith1989]. We outline a method known as Tomasulo's algorithm, after its inventor R.M. Tomasulo, who developed it to schedule floating-point instructions in theModel 91 [Tomasulo 1967]. This method is used in several variations for dynamicinstruction scheduling in recent superscalar microprocessors.

Tomasulo's approach provides each shared E-unit F, with a set of reservationstations whose purpose is to receive instructions that use /"",, keep track of theseinstructions* operands, and when all the operand values needed by a waitinginstruction L become available, initiate execution of Ij by Fr A reservation stationcan thus be seen as implementing a virtual E-unit of type Ft to which an instructioncan be sent immediately on decoding it; however, the instruction may not actuallybe executed until some later time. While one instruction / is delayed at a reserva-tion station waiting for operands, another instruction Ik waiting at the same E-unitwhose operands become available sooner can be executed first, even if Ik follows Lin the program order.

To handle data dependencies, operand values can be reassigned to temporary(virtual) registers at the reservation station, a technique referred to as registerrenaming. A

large set of temporary registers is typically needed to support the scheduling of many instructions. Several such registers at different reservation sta-tions can be assigned to the same program variable such as a register operand R[i], which allows several values of R[i] to be maintained concurrently without con-flicts. A temporary register is marked by a "tag" to indicate whether the operand value it contains is valid (to prevent an instruction from reading an obsolete value) and whether there are uncompleted instructions that need that particular value (to prevent premature overwriting of a valid value). A reservation station keeps count of the number of instructions waiting for a data value to appear in its result register R[/]; it does not mark R[/] as free to be updated until all the instructions waiting for R[/]'s new value have received it. The Model 91 employed a special bus, called the common data bus, to automatically route operand values as they became available to the reservation stations of the waiting instructions.

Consider, for example, the following three-instruction sequence:

R[l]R[2]R[3]

= ALPHA Instruction /, (load)

= R[1] + R[2] Instruction U (add)= R[4] + R[5] Instruction 73 (add)

A superscalar CPU can fetch and decode all three instructions simultaneously, or nearly simultaneously. If the current value of the operand ALPHA in the first instruction /, is in main memory, but not in the D-cache (a cache miss), the execu-tion of /j is delayed by several cycles. In that case /, is sent to a reservation station in the memory control logic—which is treated as an E-unit for scheduling pur-poses—and /j's R[l] operand is assigned to a temporary register there, say, TR[3].

CHAPTER 5Control

Execution of /, then stalls until ALPHA arrives and TR[3] is tagged as unavailable. 387(The current value of R[l] is in some temporary register, which can contain a value that is valid for some earlier instructions still in process elsewhere in the CPU.)The second instruction 72 is sent to an add unit where it is delayed by the fact that Designits R[l] operand, which the PCU points to as being assigned to TR[3], is unavail-able; thus 72 is placed in a reservation station at the adder. In the meantime, if all the operand values needed by the third instruction 73 are available, I3 can be exe-cuted—out of order—by the add unit. When ALPHA eventually arrives in the CPU, /, is executed by loading ALPHA into TR[3], whose tag is then changed to indicate that a valid result is now available. At that point 72 can also be executed in the next available cycle of the add unit.

Branch prediction. A two-way conditional branch instruction of the form

if C then Ix else 72 (5.37)

can cause control-dependency delays in an instruction pipeline because the branch's target address, which is the address of either 7, or 72, is not known until the condition C has been computed and checked. The delayed-branching method described in section 5.3.2 is one way to mask such delays and has been imple-mented in many RISC microprocessors. Another increasingly popular and more powerful technique is to predict the value of C, which implies branching to /.-, and then proceed to execute the instructions 7,7+1,7+2,... along the expected path before C's value is known. If the prediction is correct, then a performance gain has been made; if the prediction is wrong, then any instructions executed along the mispredicted path are cancelled. Because of its tentative nature, the execution of instructions before the correct path has been identified is termed speculative.Branch prediction and speculative execution require extensive instruction-level parallelism in the form of multiple E-units, temporary data registers, and so forth,which, as we have just seen, are also needed for dynamic instruction scheduling.Like dynamic scheduling, branch prediction techniques were not widely used until the 1990s.

Computer programs have certain characteristics that make it possible to predict instruction addresses. The normal fetching of instructions from consecutive mem-ory addresses depends on an implicit prediction that consecutively executed instructions have consecutive addresses. This simple prediction fails in the case of branch instructions. However, branch instructions often contain two-way (true-false) conditions of the following form: If the "usual" condition is present, execute /5 execute 72 only when an exceptional condition, for example, the end of a pro-gram loop or an erroneous data value, is encountered. In such cases we can reason-ably predict that the program will branch to the 7, path most of the time.

A superscalar machine can benefit from the simple fixed prediction that the first (second) branch address is the usual one; therefore, it always follows the path corresponding to the condition being true (false). This technique has an accuracy of about 50 percent and costs very little to implement; it is used in such processors as Sun Microsystem's SuperSparc. Clearly, a greater improvement in CPU perfor-mance is possible if branch addresses are predicted correctly most of the time.Accurate predictions can be made by having the CPU dynamically monitor condi-tional branches as they are processed and maintain a record of the paths usually

388

SECTION 5.3Pipeline Control

followed, for example, paths around a program loop. Such schemes involve trade-offs between prediction accuracy and control-hardware cost [Uht. Sindagi, andSomanathan 1997]. We will now describe the simplest such method, l-bit dynamic-branch prediction, which is implemented in the Digital Alpha 21064 superscalar microprocessor, where it is reported to produce a branch-prediction accuracy close to 80 percent.

The idea behind 1-bit branch prediction is to assign a control bit p to a branch instruction / like (5.37) when it is first executed; the CPU then uses the value of /? to predict /'s branching behavior in the future. The prediction rule of this method is that / will branch to the same instruction as it did the last time it was executed.Thus when iterating through a loop controlled by /, once the loop execution path is entered, p predicts that the same loop path will be followed each time / is encoun-tered. Of course, a misprediction eventually results when the loop is exited, but p can be expected to be right most of the time. The two states of /? have the following interpretation for (5.37): p = 1 predicts that next instruction will be /(—that is, Cwill be 1; p = 0 predicts that next instruction will be I2—that is, C will be 0. Figure5.67 illustrates the state behavior of p. The eventual outcome of each condition test determines p's next state: p remains unchanged if C's value agrees with the latest prediction made by p\ otherwise, p is changed.

Methods that record more detailed information about a branch instruction's history can replace the 1-bit prediction scheme; see problem 5.37. It is convenient to store the branching statistics (p in the above case) in a table—the branch history table—along with the address of / and of the instruction to which / currently branches. For rapid access, we can place the branch history table in a cachelike memory in the CPU called a branch target buffer (BTB); see Figure 5.68. The BTB is used as follows: Instruction requests are sent simultaneously to the I-cache and the BTB. If a match is found in the BTB, the accompanying, predicted branch tar-get address is read out. Execution proceeds along the instruction path defined by the branch target address, with all results considered speculative until the outcome of the branch condition test becomes available. When execution of the branch instruction is completed, its target address is updated in the BTB, which permits mispredicted targets to be replaced; the branch instruction's prediction statistics are also updated.

We conclude with an example of a microprocessor that implements all the methods discussed so far for exploiting instruction-level parallelism.
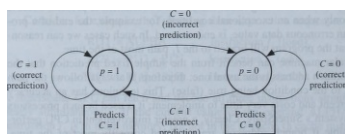
Figure 5.67

State behavior of 1-bit dynamic branch prediction method.

Branch instructionaddresses

Branchhistorytable

Match

Branch targetaddresses

Predictionstatistics

" 1

Comparison Predicted Branchtarget history

i   i   address update

Instruction address

Figure 5.68

Organization of a branch target buffer (BTB).

EXAMPLE 5.8 THE MIPS R10000 SUPERSCALAR MICROPROCESSOR [YEA-ger 1996]. This member of the MIPS RX000 microprocessor family was deliveredin 1996. It employs the 64-bit MIPS-IV architecture, which is backward compatiblewith that of the 32-bit R2/3000 (Example 5.7). The R10000 is a single-chip, supersca-lar microprocessor that can issue four instructions per clock cycle. At a clock fre-quency of 200 MHz, it can therefore operate at a CPI of 0.25 clock cycles perinstruction, which is equivalent to a peak MIPS throughput of 800 million instructionsper second. The initial version of the R10000 contains some 6.8 million transistors.

The RIOOOO's overall organization appears in Figure 5.69. This microprocessor'shigh performance is due mainly to its fast clock and to the presence of five independentand pipelined E-units: two for executing fixed-point instructions (the integer E-units).two for floating-point instructions (the floating-point E-units), and one for load andstore instructions (the load/store unit, which handles address calculations). The lengthof these execution pipelines varies from three to five stages, and each is preceded by acommon two-stage pipeline for fetching and decoding instructions. Consequently, aninstruction can pass through as many as seven consecutive pipeline stages; see Figure5.70. The fixed-point pipelines employ two 64-bit integer ALUs and a 64-word registerfile. The fixed-point pipelines are designed for 64-bit floating-point numbers using theIEEE 754 format; they are supported by another 64-word register file. To keep thepipelines as full as possible requires an interface to external memory that has very highbandwidth. The R10000 contains a primary (level 1) cache composed of a 32KB I-cache and a 32KB D-cache. The primary cache can be backed up by a much larger sec-ondary (level 2) cache that is off-chip and is linked to the CPU by a dedicated bus.

In searching for parallelism that it can exploit, the CPU prefetches and examinesup to 32 consecutive instructions, representing a large block (window) of instructionsfrom the program being executed. Four consecutive instructions are fetched simulta-neously from the I-cache. They are usually decoded in the next clock cycle andplaced in three queues for execution by the various pipelines. The queues, whichcombine the functions of instruction buffers and reservation stations, dispatch instruc-tions to E-units where they can be executed out of order. Each queue's control logicperforms dynamic scheduling to determine when the operands and executionresources needed by its instructions become available. Various methods, including aregister renaming scheme that exploits the RIOOOO's large register files, resolve data

Main memory and IO

Systembus

11

Secondarycache

I-cache

(primary instruction

cache)
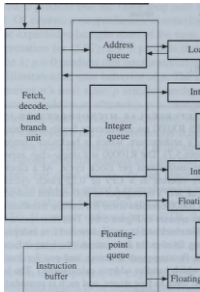
ii

Secondarycache bus

D-cache

(primary datacache)

Load/store unit (pipeline 11 u

Integer ALU1 (pipeline 2)

I * t

Integerregister file

TT

Integer ALU2 (pipeline 3)

Floating-point adder (pipeline 4)

Floating-pointregister file

ZZ

*■ Floating-point multiplier (pipeline 5)

Figure 5.69

Organization of the MIPS R10000 microprocessor.

and control dependencies. Branch prediction is implemented by a 512-entry branch-history table, which permits up to four branch paths to be executed speculatively atthe same time.

5.4SUMMARY

A digital system such as a CPU is usually partitioned into control and data-processing units. The function of the control unit is to issue to the data-processingunit control signals that select and sequence the data-processing operations. There

Stage 1

Stage 2

Stage 3 Stage 4 Stage 5 Stage 6 Stage 7

391

I-cache

Fetchinstruction

Pipeline 1: Load/store

Pipeline 2: Integer E-unit

Pipeline 3: Integer E-unit

Pipeline 4: Floating-pointE-unit (adder)

Pipeline 5: Floating-pointE-unit (multiplier)

Decodeinstruction

Instructionbuffer

Readregister

Computeaddress

Readregister

ExecuteALU1

Readregister

ExecuteALU2

Readregister

Align

Readregister

Multiply

Figure 5.70

Instruction pipelining in the R10000.

Load

Writeregister

Writeregister

Writeregister

Add

Pack

Writeregister

Sumproduct

Pack

Writeregister

CHAPTER 5

Control

Design

are two general types of complex controllers: hardwired and microprogrammed. Ahardwired control unit employs fixed logic circuits to generate the control signals.A microprogrammed control unit stores the control signals in sequences of micro-instructions (microprograms) in a control memory. Microprogramming provides asystematic and flexible method for control-unit design, since the control functionscan easily be changed by changing the stored microprograms. On the other hand,microprogrammed control units are generally larger and slower than the corre-sponding hardwired units.

We have considered two general approaches to hardwired control design thatare suitable for fairly small control units. The main design steps are state tablespecification, state code assignment, and design of the combinational logic thatimplements the next-state and output functions. The so-called classical methodminimizes the number of flip-flops used to encode and store the state information,requiring only $\lceil \log_2 n \rceil$ flip-flops for an n-state controller. The one-hot method pro-duces a circuit that contains n flip-flops but is easier to design and debug. Eachstate is assigned an n-bit binary code containing a single 1. This state-encodingscheme permits the next-state and output functions to be directly specified in a reg-ular and easily implemented form.

A microprogrammed control unit's state information is centered in the con-trol memory CM. The control unit also contains logic to generate microinstruc-tion addresses and to fetch and decode the microinstructions from CM. Themethods used for program control at the instruction level, for example, subrou-tine calls, can also be implemented at the microinstruction level. Microinstruc-tion formats fall into two groups: horizontal and vertical. Horizontal micro-instructions are characterized by long formats, little encoding of the controlfields, and the ability to control many microoperations in parallel. Vertical micro-instructions have short formats, considerable control-field encoding, and limited

392

SECTION 5.5Problems

parallelism. A few processors use two levels of microprogramming for addedflexibility: microinstructions are interpreted by nanoinstructions that directly con-trol the hardware.

We can improve the performance of a CPU by structuring its program-controland execution logic in the form of one or more pipelines. An m-stage instructionpipeline overlaps the execution of up to m separate instructions, allowing the per-formance level of one cycle per instruction (CPI) to be achieved. The simplesttwo-stage pipeline overlaps (micro) instruction fetching and execution; a typicalinstruction pipeline has five or more stages. Proper operation of a pipeline requiresthe avoidance of collisions, which occur when two instructions attempt to use thesame stage simultaneously, and hazards due to various data and control dependen-cies among instructions. Superscalar processors achieve CPI levels less than oneby executing several instructions in parallel using multiple instruction pipelines.Complex control methods such as dynamic instruction scheduling and branch pre-diction are required for efficient superscalar computation.

5.5PROBLEMS

5.1. Construct a state table corresponding to the state transition graph of Figure 5.71. Is thisa Mealy or a Moore machine?

5.2. (a) Using the notation of Figure 5.4, devise a general procedure to convert a Mealystate table into an equivalent Moore state table. [Hint: Since the Mealy table can haveseveral outputs associated with each current state S_i, consider "splitting" S_i, into a set ofstates, each of which represents S_i for some fixed output combination.] (£>) Constructthe usual two-state Mealy table for a serial adder, and apply your conversion procedureto obtain an equivalent Moore state table.

5.3. Construct a Moore state table that is equivalent to the Mealy state table for the 4-bit-stream serial adder appearing in Figure 2.12.

5.4. Figure 5.72 shows the logic circuit of a DRAM interface controller intended for usewith a certain microprocessor. (Some output circuitry has been omitted for

simplicity.)This 10-state finite-state machine is implemented with 10 flip-flops and is referred toas "one-hot encoded." However, while 9 of the 10 states have the normal one-hot state

0/00

RESET
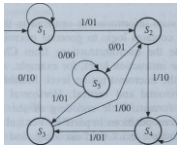
Figure 5.710/00 State transition graph of a five-statesequential circuit.

5.5. Modify the procedure given in the text for designing one-hot Moore machines to applyto Mealy machines.

encoding (exactly one flip-flop output is 1), the reset state 50 is encoded as 3930000000000, rather than as 1000000000. (a) Suggest a reason for using the all-0 codefor S0. (b) Construct a complete state diagram (state transition graph) for this controller.

CHAPTER 5

Control

Design

5.6. Use the classical method to design the DMA controller of Example 5.1 with a mini-mum number of D flip-flops. Use NAND gates to implement the combinational logic.

5.7. A tennis-scoring device TS is to be constructed that determines the winner in a two-person game of tennis. TS has inputs x]jc2 and outputs c,,c->. Input a, is set to 1 when-ever player i scores a point and is set to 0 otherwise. Input ~, is set to 1 whenever playeri wins a game; it is 0 otherwise. The rules of tennis can be stated succinctly as follows:To win a game, a player must win at least four points and must be at least two pointsahead of the other player, (a) Construct a state table that defines the behavior of TS.(b) Estimate as accurately as you can the minimum number of flip-flops needed to im-plement TS using the classical and one-hot design methods.

5.8. Close scrutiny of the multiplier behavior defined by Figure 5.15 shows that certainstates can be merged. Consider states S2 (load register Q) and 56 (output register Q).To enter state S2 requires COUNT = 0 and therefore the control signal COUNT7 = 0,whereas S6 is entered only after COUNT7 becomes 1. Hence we can merge 52 and 56to form a single state S26 if when entering S26 with COUNT7 = 0, we match 5-,'s next-state and output behavior, but when entering S?6 with COUNT7 = 1. we match 56'sbehavior. In the first case the next state is 53 or 54 depending on Q[0], and the activeoutput is cg; in the second case the next state is S7, and the active output is c7. Notethat since the outputs associated with S26 will depend on the primary inputs, we willhave Mealy-type behavior. (<•) Identify a second pair of states from Figure 5.15 thatcan be merged in this manner and explain their relationship, (b) Construct a state tablein the style of Figure 5.16 for a reduced, six-state control unit.

5.9. Some early computers had small instruction sets, but did not restrict memory access toload and store instructions (load/store architecture), and so needed fewer CPU regis-ters than a modern RISC. Figure 5.73 shows the instruction set for a CPU of this type

REF_REQUEST —JDRAM_SEL

CLOCK

RESET

D Q

>CKcut

IDLE

RW1

RW2

RW3

:D-

D Q>CK

CLR

-i- D Q -i- D O -L D

Q

->CK

CLR

->CK

CLR

RW4

>CK

CLR

REF1

REF2

REF<

REF4

D Q

>CK I—PCK

CLR

IT

D Q

D Q>CK

CLR

HZ

D Q

>CK

CLR

x=

D Q>CK

CLR

RAS_PRECHARGE

Figure 5.72

A 10-state control unit with modified one-hot state encoding.

SECTION 5.5Problems

Type	HDL format	Assemblyformat	Comment
Data transfer AC	= M(X)	LDX	Load X from M into AC
	M(X) := AC STX		Store contents of AC in M as X
Data	AC := AC + M(X) ADDX		Add X to AC
processing	AC	= AC - M(X) SUBX	Subtract X from AC
	AC	= AC and M(X) ANDX	And X to AC
	AC	= not AC NOT	Complement contents of AC
Program	PC := M(adr)	BRAadr	Jump to instruction with address adr
control	if AC = 0 then PC := U(adr)	BZadr	Jump to instruction adr if AC = 0

Figure 5.73

Modified instruction set for an accumulator-based CPU.

derived from that of Figure 5.20. The data-processing instructions now reference mainmemory M, and the data register DR is no longer visible to a program. (It is still usedinternally for memory access operations, however.) Construct a flowchart in the styleof Figure 5.21 for the modified CPU.

5.10. Suppose that the accumulator-based CPU of Figures 5.20 through 5.24 is enlarged toinclude the following instructions: (1) A left-shift instruction LSH that implementsAC := AC[/? - 2:0].0; (2) an add-with-carry instruction ADC that computes AC + DR +CY, where CY is a new carry flag that is set (reset) whenever an arithmetic instructioncauses (does not cause) AC to overflow; (3) a skip-on-carry instruction SKC that caus-es the CPU to skip the next consecutive instruction if and only if CY = I. (a) Show thechanges that need to be made to the flowchart of Figure 5.21 to incorporate the newinstructions, (b) Specify a minimal set of new control signals that should be added tothe list of Figure 5.22b to support the three new instructions.

5.11. Consider the design of the control circuit FSM for the accumulator-based CPU definedby Figures 5.20 through 5.24. Assume that it must have the 13 internal states 50:5:2de-fined by Figure 5.21 and is to be implemented as a Moore machine using the one-hotmethod with D flip-flops and NAND gates. Assign the hot variable Di to state 5, andobtain a complete set of next-state and output equations for FSM in sum-of-productsform. Estimate the number of NAND gates (including inverters) needed to constructFSM in this way, assuming a D flip-flop is equivalent to five NANDs.

5.12. Answer the following questions concerning the microprogrammed control unitshown in Figure 5.26. (a) What control signals are activated by the microinstruction/5 with address a2a^a0 = 101? (b) What microinstruction is loaded into CMAR after/5? (c) Suppose that all the control functions performed by the top two microinstruc-tions /0 and /, can be carried out simultaneously. Devise a single microinstructionthat can replace both /0 and /,.

5.13. A certain processor has a microinstruction format containing 10 separate control fieldsC0:C9. Each C, can activate any one of n, distinct control lines, where «, is specified asfollows:

i = 0

n.- = 4

5 616 7

8 98 22

What is the minimum number of control bits needed to represent the 10 control fields? 395What is the maximum number of control bits needed if a purely horizontal format isused for all the control information? CHAPTER 5

Control

5.14. Draw a logic diagram showing how to construct a microprogram sequencer for (a) a Design64 x 12-bit control memory and (b) a 12 x 64-bit control memory, using one or more

copies of the AMD 2909.

5.15. Using the format of Equation (5.14), specify the control signals needed to performthe following microoperations in a 2909-based microprogram sequencer: (a) CALLX, where X is the address on the D bus; (b) go to 0 if external condition C, =1; and(c) repeat the last microinstruction.

5.16. Describe the changes that must be made to the hardware and the microprogram for the16-bit twos-complement multiplier described in Example 5.5 in order to do the follow-ing: (a) 12-bit twos-complement multiplication; (b) 16-bit twos-complement multipli-cation with the following register assignment: R[3] = multiplier, R[2] = multiplicand,and R[1].R[0] = product.

5.17. Design the control logic that is driven by the CONFIG control field appearing in Fig-ure 5.42. In other words, show in detail how the 2901-based processor is dynamicallyreconfigured while executing the multiplication microprogram of Figure 5.43.

5.18. Use the information in Figure 5.50 and the text to determine the microoperations thatimplement the call and return microinstructions for the 890 microprogram sequencer.Express each microinstruction in generic HDL format, as in (5.18).

5.19. You are to design a microprogrammed controller for a fixed-point divider that uses the circuit of Figure 4.23 and the nonrestoring division algorithm of Figure 4.24. The di-vider should handle both positive and negative integers having a 16-bit sign-magnitude format, (a) List all the required control signals and the microoperations they control.(b) Design a microinstruction format of the type shown in Figure 5.40 in which the con-trol fields are encoded by function in an efficient manner.

5.20. A microprogrammed control unit is to be designed for a floating-point adder with the general structure shown in Figure 4.44. A number of the form M x B is represented by a 32-bit word comprising a 24-bit mantissa, which is a twos-complement fraction, and an 8-bit exponent, which is a biased integer. The base B is two. (a) Using our HDL, give a complete listing of a symbolic microprogram to control this adder, (b) Derive a suitable microinstruction format that uses unencoded control fields.

5.21. A conventional microprogrammed CPU is being redesigned for implementation as a one-chip microprocessor. At present it has a single 256 x 80-bit control memory and employs a highly parallel horizontal microinstruction format in which every instruction contains one 8-bit branch address. It is estimated that in a two-level organization of the control unit, only about sixty-four 300-bit nanoinstructions would be needed to imple-ment the current instruction set. If the total size of the control memories is the major cost consideration, should the new microprocessor have one- or two-level control1 Show your calculations and state all your assumptions.

5.22. A pipeline P is found to provide a speedup of 6.16 when operating at 100 MHz and an efficiency of 88 percent, (a) How many stages does P have? (b) What are P s MIPS and CPI performance levels?

396

SECTION 5.5 Problems

Stage

Si

s3

Time /

Figure 5.74

Reservation table for a three-stage pipeline.

5.23. The hardware cost of a new m-stage, single-function pipeline is approximated by 22m + 30. The latency of the function to be executed is 90 ns if pipelining is not used. The pipelined implementation's interstage buffers are expected to add an addi-tional 10m ns to this latency. Estimate the number of stages needed to optimize the pipeline's performance/cost ratio.

5.24. (a) For the pipeline reservation table appearing in Figure 5.74, calculate the forbid-den set F, the minimum constant latency Lcmin, and the minimum average latency Lmin.(b) Suppose that the pipeline is to be operated with a constant latency L such that the resulting pipeline efficiency is as close to 0.5 as possible. What is L in this case?

5.25. Construct a task initiation diagram (TID) for the pipeline reservation table appearing in Figure 5.74 and calculate the pipeline's minimum average latency Z.mjn.

5.26. For the pipeline reservation table appearing in Figure 5.75, calculate the forbidden set F, the minimum constant latency Lcmn. and the minimum average latency Lmin. Also construct a task initiation diagram for this pipeline.

5.27. Prove informally the following general property of a single-function pipeline. If K is the maximum number of x's in any row of the pipeline's reservation table, then K <Lmjn, the minimum average latency. This result provides a useful lower bound on Lmin.

5.28. Consider the following seven-instruction fragment of assembly language code for the MIPS RX000. Recall that the RX000 has no explicit flag bits and that the general reg-ister R0 always stores the constant zero.

{Set on less than: if Rl < R2, then set R7 to 1, else set

R7to0}{Branch on equal: if R7 = 0. then PC := OUT1}{No operation }

{Add unsigned: R3 := R2 + R0}{Branch unconditionally: PC := OUT2}{No operation}{Add unsigned: R3 := Rl + R0}

(a) What is the program's purpose? (b) What is the role of its two NOP instructions?(c) Redesign this program to reduce the number of instructions from seven to four (oras much as you can).

5.29. The following code fragment is to be executed in the six-stage instruction pipeline of Figure 5.76. Assume that every instruction must pass through all stages, including the three execution stages.

    SLT   R7.R1.R2

    BEQ   R7,R0,OUTl

    NOP

    ADDU R3,R2,R0

    B   OUT2

    NOP

OUT1 ADDU R3,R1,R0

OUT2

Stage

s2

Si

s*

Ss

Timef

397

CHAPTER 5

Control

Design

Figure 5.75

Reservation table for a five-stage

pipeline.

Load constant A into general register r4

Load constant B into general register r5

Add r4 and r5 and put the sum in r8

Store r8 in the memory location addressed by rl

Load constant C into general register r6

Add r5 and r6 and put the sum in r9

Store r9 in the memory location addressed by r2

(a) Construct a space-time diagram in the style of Figure 5.57 for this program, and determine how many cycles are needed to completely execute it. (b) Determine a valid reordering of the program that will reduce its execution time. Construct the space-time diagram for the reordered program.

Id   r4,#A

Id   r5,#B

add r8, r4, r5

St   m(rl), r8

Id   r6,#C

add r9, r5, r6

St   m(r2), r9

'   r

S,: Fetch (IF)

'   r

S2: Decode(DE)

'   t

S3: Execute (El)

i   t

S4: Execute (E2)

i   <

S5: Execute (E3)

'   i

S6: Write back (WB)

Figure 5.76

Six-stage instruction pipeline.

SECTION 5.5 Problems

Id   r4,#A

Id   r5, #B

Id   r6,#C

Id   r9, #0

beq r4, r5, adrl

add r9, r4, r5

mul r9, r9, r9

mul r9, r9. #1

St   m(rl),r9

398 5.30. (a) The three conditions (5.34), (5.35), and (5.36) for RAW, WAR, and WAW data-

dependent hazards are considered to be necessary, but not sufficient, for an instruction pipeline to produce invalid results. Show by means of an example why these conditions are not sufficient, (b) Conspicuous by its absence from the above set is the read-after-read (RAR) condition D(/,) n D(I2) * 0. Explain why Jhis condition is not a hazard.

5.31. Consider the following assembly-language program for a hypothetical RISC:

Load constant A into general register r4 Load constant B into general register r5 Load constant C into general register r6 Clear general register r9 If r4 = r5 then go to adrl Add the sum of r4 and r5 to r9 Square the contents of r9 Increment r9 by one adrl: st m(rl), r9 Store r9 in the memory location addressed by rl

Identify all possible RAW,WAR, and WAW hazards that are present if nothing is known about the structure of the RISC*s instruction pipeline.

5.32. Suppose the code fragment in problem 5.31 is processed by the four-stage instruction pipeline of Figure 5.55. Assume that data reads (from registers and/or memory) can oc-cur only in stage 52. while data reads and writes can occur only in stage SA. Identify all RAW, WAR, and WAW hazards that are present in this case.

5.33. Consider the five-stage instruction pipeline of Figure 5.65. Assume that the program counter can be changed only by program-control instructions in the same manner as a general register. What delay penalty is associated with a branch instruction? By how much can the use of forwarding paths reduce this penalty?

5.34. (a) Explain why one is a lower bound on the CPI of conventional, nonsuperscalar mi-croprocessors, (b) Name and briefly describe two techniques superscalar microproces-sors use to make CPI less than one.

5.35. Early RISCs such as the IBM 801, which are not superscalar, use a branch-and-execute instruction to eliminate the pipeline delay slots caused by branch-instruction latency, as illustrated by (5.33). (a) This feature was deliberately excluded from the later POWER and PowerPC architectures because "it poses a severe handicap for supersca-lar processors." Explain this statement, (b) Suggest a reason why an even later super-scalar microprocessor, the MIPS R10000, has the branch-and-execute feature.

5.36. Instead of using conventional instructions and pipelining, we can achieve superscalar performance by employing a very long instruction word (VLIW) to control multiple E-units and other CPU resources in much the same way a microinstruction controls mul-tiple resources that execute microoperations. The programmer or compiler determines the control fields of the VLIW instructions and specifies the resources of a VLIW pro-cessor to be used in each clock cycle. Like horizontal microinstructions, VLIW instruc-tions aim to maximize the number of operations done in parallel and require simple decoding logic. Superscalar VLIW computers have not been commercially successful. Suggest three reasons for this lack of success.

5.37. One-bit branch prediction can be extended by using 2 bits to record the outcomes of the last two executions of each conditional branch instruction. Devise such a 2-bit pre-

diction method and explain it using a state diagram like that of Figure 5.67. State 399 clearly the rationale for your method's prediction rules.

5.6 REFERENCES

1. Actel Corp. FPGA Data Book and Design Guide. Sunnyvale, CA, 1994.

2. Advanced Micro Devices Inc. Bipolar Microprocessor and Logic Interface (Am2900 Family) Data Book. Sunnyvale, CA, 1985.

3. Amdahl Corp. Amdahl 470V/7 Machine Reference Manual, Publ. G1003 0-01/A. Sunnyvale, CA, 1978.

4. Baranov. S. Logic Synthesis for Control Automata. Dordrecht. The Netherlands: Kluwer. 1994.

5. Becker, M. C. et al. "The PowerPC 601 Microprocessor." IEEE Micro, vol. 13 (October 1993) pp. 54-68.

6. Cormen, T. H., C. E. Leiserson, and R. L. Rivest. Introduction to Algorithms. New York: McGraw-Hill, 1990.

7. Hayes, J. P. Introduction to Digital Logic Design. Reading, MA: Addison-Wesley, 1993.

8. Hwang, K. Advanced Computer Architecture. New York: McGraw-Hill, 1993.

9. Kane, G. and J. Heinrich. MIPS RISC Architecture. Englewood Cliffs, NJ: Prentice-Hall, 1992.

10. Kogge, P. M. The Architecture of Pipelined Computers. New York: McGraw-Hill, 1981.

11. Lackzo, F. et al. "32-Bit CPU Design with the 'AS888/'AS890." ACM Sigmicro News-letter, vol. 17 (July 1986) pp. 8-13.

12. Lynch, M. A. Microprogrammed State Machine Design. Boca Raton, FL: CRC Press, 1993.

13. Mealy, G. H. "A Method for Synthesizing Sequential Circuits." Bell System Technical Journal vol. 34 (1955) pp. 1045-79.

14. Mick, J. and J. Brick. Bit-Slice Microprocessor Design. New York: McGraw-Hill. 1980.

15. Moore, E. F. "Gedanken Experiments on Sequential Machines." Annals of Mathematics Studies, no. 34 (Automata Studies). Princeton, NJ: Princeton University Press, 1956.

16. Radin, G. "The 801 Minicomputer." IBM Journal of Research and Development, vol. 27 (May 1983) pp. 237-46.

17. Smith, J. E. "Dynamic Instruction Scheduling and the Astronautics ZS-1." IEEE Com-puter, vol. 22 (July 1989) pp. 21-35.

18. Stone, H. S. High-Performance Computer Architecture. 3rd ed. Reading, MA: Addison-Wesley, 1993.

19. Stritter. S. and N. Tredennick. "Microprogrammed Implementation of a Single-Chip Com-puter." Proceedings of the 11th Microprogramming Workshop (December 1978) pp. 8-16.

20. Texas Instruments Inc. SN74AS888/SN74AS890 Bit-Slice Processor User's Guide. Dal-las: 1985.

21. Tomasulo, R. M. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units." IBM Journal of Research and Development, vol. 11 (January 1967) pp. 25-33.

22. Uht, A. K., V. Sindagi. and S. Somanathan. "Branch Effect Reduction Techniques." IEEE Computer, vol. 30 (May 1997) pp. 71-81.

23. Wilkes, M. V. "The Best Way to Design an Automatic Calculating Machine." Report of the Manchester University Computer Inaugural Conference, 1951, pp. 16-18. [Reprint-ed in E. E. Swartzlander (ed.) Computer Design Development: Principal Papers. Roch-elle Park, NJ: Hayden. 1976, pp. 266-270.]

24. Yeager, K. C. "The MIPS R10000 Superscalar Microprocessor." IEEE Micro, vol. 16 (April 1996) pp. 28-U).

CHAPTER 5

Control

Design

CHAPTER 6

Memory Organization

This chapter is concerned with the design of a computer's memory system and its impact on performance. The characteristics of the most important storage-device technologies are surveyed. The behavior and management of multilevel hierarchi-cal memory systems are discussed, and cache memories are examined in detail.

6.1

MEMORY TECHNOLOGY

Every computer contains several types of devices to store the instructions and data required for its operation. These storage devices plus the algorithms—imple-mented by hardware and/or software—needed to manage the stored information form the memory system of the computer.

6.1.1 Memory Device Characteristics

A CPU should have rapid, uninterrupted access to the external memories where its programs and the data they process are stored so that the CPU can operate at or near its maximum speed. Unfortunately, memories that operate at speeds compara-ble to processor speeds are expensive, and generally only very small systems can afford to employ a single memory using just one type of technology. Instead, the stored information is distributed, often in complex fashion, over various memory units that have very different performance and cost.

Memory types. The information-storage components of a computer can be placed in four groups, as illustrated in Figure 6.1.

400

• CPU registers. These high-speed registers in the CPU serve as the working memory for temporary storage of instructions and data. They usually form a gen-eral-purpose register file for storing data as it is processed. A capacity of 32 datawords is typical of a register file, and each register can be accessed, that is, read from or written into, within a single clock cycle (a few nanoseconds).

• Main (primary) memory. This large, fairly fast external memory stores programs and data that are in active use. Storage locations in main memory are addressed directly by the CPU's load and store instructions. While an IC technology similar to that of a CPU register file is used, access is slower because of main memory's large capacity and

the fact that it is physically separated from the CPU. Mainmemory capacity is typically between 1 and 210 megabytes, where a megabyte,also denoted 1 MB, is 220 bytes, and 210 MB = 230 bytes is referred to as agigabyte (1 GB). Access times of five or more clock cycles are usual.

• Secondary memory. This memory type is much larger in capacity but also muchslower than main memory. Secondary memory stores system programs, largedata files, and the like that are not continually required by the CPU. It also acts asan overflow memory when the capacity of the main memory is exceeded. Infor-mation in secondary storage is considered to be on-line but is accessed indirectlyvia input/output programs that transfer information between main and secondarymemory. Representative technologies for secondary memory are magnetic harddisks and CD-ROMs (compact disk read-only memories), both of which haverelatively slow electromechanical access mechanisms. Storage capacities ofmany gigabytes are common, while access times are measured in milliseconds.

• Cache. Most computers now have another level of IC memory—sometimes sev-eral such levels—called cache memory, which is positioned logically betweenthe CPU registers and main memory. A cache's storage capacity is less than thatof main memory, but with an access time of one to three cycles, the cache ismuch faster than main memory because some or all of it can reside on the sameIC as the CPU. Caches are essential components of high-performance computers

401

CHAPTER 6

MemoryOrganization

Cache(level 2)

ICs 2:m

CPU

Cache(level 1)

Register

file

IC 1 (microproces ior)

Mainmemory

ICs m:n

Secondarymemory

Hard disks, etc.

Figure 6.1

Conceptual organization of a multilevel memory system in a computer.

402 that aim to make CPI < 1. Unlike the three other memory types, caches are nor-

mally transparent to the programmer. Together, a computer's caches and main

M T . . memory implement the external memory M addressed directly by CPU instruc-

tions.

The goal of every memory system is to provide adequate storage capacity withan acceptable level of performance and cost. We can achieve these goals byemploying several memory types—with different cost/performance ratios—thatare organized to provide a high average performance at a low average cost per bit.The individual memory units form a multilevel hierarchy of storage devices, assuggested by Figure 6.1. Successful operation of the hierarchy requires automaticstorage-control methods that make efficient use of the available memory capacity.These methods should free the user from explicit management of memory space.They should also free programs from the particular memory environment in whichthey are executed.

Performance and cost. The computer architect can choose from a bewilder-ing variety of memory devices that employ various electronic, magnetic, and opti-cal technologies and offer many cost/performance trade-offs [Cook and White1994; Prince 1996]. However, all memories are based on just a few physical phe-nomena and organizational principles. We now examine the features common tothe devices used to build cache, main, and secondary memories.

The most meaningful measure of the cost of a memory device is the purchaseprice to the user of a complete unit. The price should include not only the cost ofthe information storage medium itself but also the cost of the peripheral equipment(access circuitry) needed to operate the memory. Let C be the price in dollars of acomplete memory system with S bits of storage capacity. We define the cost c ofthe memory as follows:

C

c = — dollars/bit

The performance of an individual memory device is primarily determined bythe rate at which information can be read from or written into the memory. A basicperformance measure is the average time to read a fixed amount of information, forinstance, one word, from the memory. This parameter is called the read accesstime, or simply the access time, of the memory and is denoted by t_A. The writeaccess time is defined similarly; it is often, but not always, equal to the read accesstime. The access time depends on the physical nature of the storage medium and onthe access mechanisms used. It is calculated from the time the memory receives aread request to the time at which the requested information becomes available atthe memory's output terminals.

Clearly, low cost and short access time are desirable memory characteristics;unfortunately, they also tend to be incompatible. Memory units with fast access areexpensive, while low-cost memories are slow. Figure 6.2 shows the relationshipbetween cost c and access time t_A for some recent memory technologies. Thestraight line AB approximates this relationship. If we write $t_A = 10$-v and $c = 10$\then $y \sim mx + k$, where m denotes the slope of AB and k is a constant. Hence $t_A \sim 10$""+* = kcm + k". From the data in Figure 6.2, we can conclude that m « -0.5.Hence to decrease t_A by a factor of 10, the cost c must increase by about 100.

100

10

A \

1

io-1

| io-1 io-3

c 10-1

£

% IO"5

V

u

< IO"6

io-7

10"8

io-9

Magnetic tapes

Optical disks (CD-ROMs, etc.)

Magnetic disks(hard disks)

10-

10-
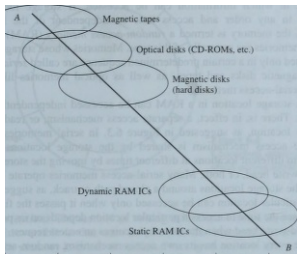
10"8 10"

Cost c (dollars/bit)

io-

io-

Figure 6.2

Access time versus cost for representative memory technologies.

Manufacturing improvements have steadily reduced the storage cost per bit c for the principal memory technologies. This trend is especially striking in the case of the IC RAMs used to construct main and cache memories, where the storage density per IC has increased steadily while the cost per IC has remained fairly con-stant. The state of the art in RAM manufacture circa 1975, 1985. and 1995 is repre-sented by single-chip RAMs of capacity 4 Kb, 256 Kb, and 16 Mb, respectively. Here 1 Kb denotes a kilobit and equals 210, or 4096 bits, while 1 Mb denotes a megabit and equals 220, or 1,048,576 bits. At a typical introductory price of S40 for each chip type, the cost per bit c fell from around 0.01 dollars per bit in 1975 to 0.00015 dollars per bit in 1985 and to 0.0000024 dollars per bit a decade later. Similar developments have taken place in other technologies, notably magnetic(hard) disk memories, as storage density has increased steadily with little change in the cost per memory unit.

Although storage density has grown rapidly for the principal memory technologies, access times have decreased at a much slower rate. This disparity has tended to aggravate the speed mismatch—the von Neumann bottleneck—between the CPU and M. Memory speed has increased slowly, but the computing speed of microprocessors has spurted, along with their ability to produce and consume ever-increasing amounts of information. As we will see in this chapter, various design techniques can increase the effective rate at which the CPU can access the informa-tion stored in its memory system.

404

SECTION 6.1 Memory Technology

Access modes. A fundamental characteristic of a memory is the order or sequence in which information can be accessed. If storage locations can be accessed in any order and access time is independent of the location being accessed, the memory is termed a random-access memory (RAM). IC (semicon-ductor) memories are generally of this type. Memories whose storage locations can be accessed only in a certain predetermined sequence are called serial-access mem-ories. Magnetic disks and tapes, as well as optical memories like CD-ROMs, employ serial-access methods.

Each storage location in a RAM can be accessed independently of the other locations. There is, in effect, a separate access mechanism, or read-write "head," for every location, as suggested in Figure 6.3. In serial memories, on the other hand, the access mechanism is shared by the storage locations and must be assigned to different locations at different times by moving the stored information, the read-write head, or both. Many serial-access memories operate by continually moving the storage locations around a closed path or track, as suggested by Figure 6.4. A particular location can be accessed only when it passes the fixed read-write head. Hence the time to access a particular location depends on its position relative to the read-write head when the memory receives an access request.

Since every location has its own access mechanism, random-access memories tend to be more costly than the serial type. In serial-access memories, however, the time required to bring the desired location into correspondence with a read-write head increases the effective access time, so serial access tends to be slower than random access. Thus the type of access mode contributes significantly to the inverse relationship between cost and access time. In Figure 6.2, for example, the random-access technologies (dynamic and static RAMs based on ICs) and serial-access technologies (magnetic disks, magnetic tapes, and optical disks) are clearly separated into two groups.

Memory devices such as magnetic hard disks and CD-ROMs contain many rotating storage tracks. If each track has its own read-write head, the tracks can be accessed randomly, but access within each track is serial. In such cases the access mode is semirandom. Note that the access mode is a function of both memory organization and the inherent characteristics of the storage technology. The IC technologies used for RAMs can also be used to construct serial-access memories; the converse is not true, however.

Read-write head selector

4 A A A A A A a]! ji | v v | v J J

Read-writeheads

Storagelocations

Figure 6.3

Conceptual model of a random-access memory.

Read-writehead

Storagelocations
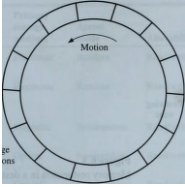
Figure 6.4

Conceptual model of aserial-access memory.

CHAPTER 6

Memory

Organization

Memory retention. The method of writing information into a memory can bepermanent or irreversible in that once information has been written, it cannot bealtered while the memory is in use or on-line. Printing on paper is an example of apermanent storage technique. Memories whose contents cannot be altered on-line—if they can be altered at all—are read-only memories (ROMs). A ROM istherefore a nonerasable storage device. ROMs are widely used to store control pro-grams such as microprograms. Compact disk (CD) ROMs are a class of noneras-able secondary memory devices developed in the 1980s that employ an optical(laser) read-write mechanism. A standard (12 cm diameter) CD-ROM has a capac-ity of about 600 MB and is used to store large program and data files. Semiconduc-tor ROMs whose contents can be changed off-line—and with some difficulty—arecalled programmable read-only memories (PROMs). Programmable CDs arereferred to as CD-recordable (CD-R) disks.

Memories in which reading or writing can be done with impunity on-line arecalled read-write memories to differentiate them from ROMs. All memories usedfor temporary storage purposes are read-write memories. Unless otherwise speci-fied, we will use the terms memory and RAM to mean read-write memories.

In some technologies the stored information is lost over a period of time unlesscorrective action is taken. Three characteristics of memories that destroy informa-tion in this way are destructive readout, dynamic storage, and volatility. In somememories the method of reading the memory destroys the stored information; thisphenomenon is called destructive readout (DRO). Memories in which reading doesnot affect the stored data have nondestructive readout (NDRO). In DRO memorieseach read operation must be followed by a write operation that restores the mem-ory's original state. This restoration is carried out automatically using a buffer reg-ister, as shown in Figure 6.5. The read transfers the word at the addressed (shaded)location to the buffer register where it is available to external devices. The contentsof the buffer are automatically written back into the original location.

Certain memory devices have the property that a stored 1 tends to become a 0,or vice versa, due to some physical decay process. For example, in some IC memo-ries, an electric charge in a capacitor represents a stored 1; the absence of a storedcharge represents a 0. Over time, a stored charge tends to leak away, causing a loss

SECTION 6.1Memory Technology

Destructiveread
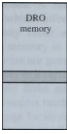
Bufferregister

Restoringwrite

Figure 6.5

Memory restoration in a destructivereadout (DRO) memory.

of information unless the charge is restored by a process called refreshing. Memo-ries that require periodic refreshing are called dynamic memories, as opposed tostatic memories, which require no refreshing. (Note that in this context dynamicand static do not refer to the presence or absence of mechanical motion in the stor-age device.) Most memories that employ magnetic or optical storage techniquesare static. Main memories are usually built from dynamic ICs referred to asdynamic RAMS (DRAMs). ICs can also implement static memories referred to asstatic RAMS (SRAMs). As Figure 6.2 indicates, SRAMs tend to be faster, that is,have lower access time, than DRAMs, but the cost per bit of SRAMs is higher.SRAMs are often used to build caches. A dynamic memory is refreshed in muchthe same way that data is restored in a DRO memory. The contents of every loca-tion are sent periodically to buffer registers and then returned in amplified form totheir original locations.

Another physical process that can destroy the contents of a memory is theremoval or failure of its power supply. A memory is volatile if the loss of powerdestroys the stored information. Information can be stored indefinitely in a volatilememory by providing batten,' backup or other means to maintain a continuous sup-ply of power. Most IC memories are volatile, while most magnetic and opticalmemories are nonvolatile.

Figure 6.6 summarizes these characteristics for some important contemporarymemory technologies.

Other characteristics. We defined the access time tA as the time between thereceipt of a read request signal by a memory and the delivery of the requestedinformation to its output terminals. Some DRO and dynamic memories cannot ini-tiate a new access until a restore or refresh operation has been carried out. There-fore, the minimum time that must elapse between the start of two consecutiveaccess operations can be greater than fA. This elapsed time is called the cycle timetM of the memory and represents the time needed to complete a read or write oper-ation.

The maximum amount of information that can be transferred to or from thememory per unit time is the data-transfer rate or bandwidth frM and is measured in

Technology	Primary storage medium	Accessmode	Alterability	Permanence	Typicalaccesstime t_A 40/	
						CHAPTER 6Memory
Bipolarsemiconductor	Electronic	Random	Read/write	NDRO, volatile	10 ns	Organization
Metal oxide semiconductor (MOS)	Electronic	Random	Read/write	DRO or NDRO.volatile	50 ns	
Magnetic (hard)disk	Magnetic	Semirandom	Read/write	NDRO.nonvolatile	10 ms	
Magneto-opticaldisk	Optical	Semirandom	Read/write	NDRO.nonvolatile	50 ms	

NDRO.

Compact diskROM	OpticalMagnetic	Semirandom	Read only	nonvolatile NDRO,nonvolatile	100 ms1 s
Magnetictape cartridge	Serial		Read/write		

Figure 6.6

Characteristics of some common memory technologies.

bits or words per second. If vv is the number of bits that can be transferred simulta-neously to or from the memory, then $b_M = w/t_M$ bits/s. If $t_M = t_A$, then $b_M = w/t_A$. Some memory types, particularly serial memories, require a long access time t_A toinitiate a new access operation; once the operation is initiated, however, data trans-fer can proceed at a rate b_M much greater than w/t_A. In such cases the manufacturerprovides independent specifications for t_A, f_M, b_M, and related performance param-eters.

Finally, we mention reliability, which is measured by the mean time beforefailure (MTBF). In general, memories with no moving parts have much higher reli-ability than memories such as magnetic disks, which involve considerable mechan-ical motion. Even in memories without moving parts, reliability problems arise,particularly when very high storage densities or data-transfer rates are used. Error-detecting and error-correcting codes can increase the reliability of any memory.

6.1.2 Random-Access Memories

RAMs are distinguished by the fact that each storage location can be accessedindependently with fixed access and cycle times that are independent of the posi-tion of the accessed location.

Organization. Figure 6.7 shows the main components of a RAM device suchas a DRAM IC. At its heart is a storage unit composed of a large number (2^m) ofaddressable locations, each of which stores a w-bit word. Individual bits are notdirectly addressable unless $w = 1$. A RAM of this sort is referred to as a $2^m \times n$ -bitor 2^m-word memory. The RAM operates as follows: First the address of the targetlocation to be accessed is transferred via the address bus to the RAM's address

408

wbits
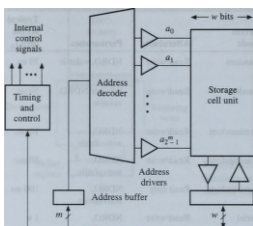
Control Addresslines bus

Read/write drivers(sense amplifiers)

Data buffer

Databus

Figure 6.7

One-dimensional (1-D) random-access memory unit.

buffer. The address is then processed by the address decoder, which selects therequired location in the storage cell unit. A control line indicates the type of accessto be performed. If a read operation (load) is requested, the contents of theaddressed location are transferred from the storage cell unit to the data buffer andfrom there to the data bus. If a write (store) is requested, the word to be stored istransferred from the data bus to the selected location in the storage unit. Since it isnot usually necessary or desirable to permit simultaneous reading and writing, theinput and output data buses are often combined into a single, bidirectional data bus.

The storage unit is made up of many identical 1-bit memory cells and theirinterconnections. The actual number of lines connected to the cell and their func-tions depend on the memory technology and the addressing scheme in use. Eachcell is connected to a set of data, address, and control signals. One physical lineoften has several logical functions; for example, it can serve as both an address anddata line. In each line connected to the storage cell unit, we can expect to find adriver that acts as either an amplifier or a transducer of physical signals. Thus wesee in Figure 6.7 several sets of drivers for the address and data lines. The drivers,decoders, and control circuits form the access circuitry of the RAM and can have asignificant impact on the total size and cost of the memory.

A RAM's storage cells are physically arranged into regular arrays to reducethe cost of the connections between the cells and the access circuitry. The memoryaddress is partitioned into d components so that the address A, of cell C, becomes ad-dimensional vector $(A_iA, A_{i2},...,A_{Ld}) = A_t$. Each of the d parts of the address wordgoes to a separate address decoder and a separate set of address drivers. A cell isselected by simultaneously activating all d of its address lines. A memory unit withthis kind of addressing is said to be d-dimensional. Thus the basic RAM of Figure6.7 is one-dimensional (1-D).
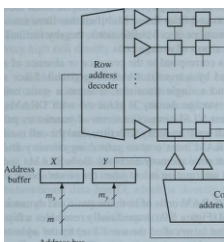
Storage cell array

Columnaddress decoder

Figure 6.8

Two-dimensional (2-D) RAM addressing scheme.

409

The most common RAM organization is the two-dimensional (2-D) or row-column scheme shown in Figure 6.8, where, for simplicity, the data and control cir-cuits are omitted. Here the m-bit address word is divided into two parts, X and Y, consisting of mx and my bits, respectively. The cells are arranged in a rectangular array of Nx < 2m* rows and Ny < 2mv columns, so the total number of cells is N =NxNy. A cell is selected by the coincidence of signals applied to its X and Y address lines. The 2-D organization requires much less access circuitry than a 1-D organi-zation for the same storage capacity. For example, if 7Vr = 7VV = JN, the number of address drivers needed is ijN, whereas the 1-D RAM of Figure 6.7 has N = NxNy address drivers. Instead of a single one-out-of-TV address decoder, two one-out-of-Jn address decoders suffice. In addition, the 2-D organization is a good match for the inherently two-dimensional layout structures allowed by VLSI technology.

Semiconductor RAMs. Semiconductor memories in which the storage cells are small transistor circuits have been used for high-speed CPU registers since the 1950s. It was not until the development of VLSI in the 1970s that producing large RAM ICs suitable for main-memory and cache applications became economical. Single-chip RAMs can be manufactured in sizes ranging from a few hundred bits to 1 Gb or more. Both bipolar and MOS transistor circuits are used in RAMs. but MOS is the dominant circuit technology for large RAMs. Current IC manufactur-ing limitations make it impossible to manufacture, say, a terabit (240-bit) RAM on a single IC chip. Consequently, very large semiconductor RAMs must be con-structed from a set of smaller RAM ICs.

As observed earlier, semiconductor memories fall into two categories—SRAMs and DRAMs—whose data-retention methods are static and dynamic.

410

respectively. SRAMs consist of memory cells that resemble the flip-flops used in processor design. SRAM cells differ from flip-flops primarily in the methods used to address the cells and transfer data to and from them. Multifunction lines mini-mize storage-cell complexity and the number of cell connections, thereby facilitat-ing the manufacture of very large 2-D arrays of storage cells.

In a DRAM cell the 1 and 0 states correspond to'the presence or absence of a stored charge in a capacitor controlled by a transistor switching circuit. Since a DRAM cell can be constructed around a single transistor, whereas a static cell requires up to six transistors, higher storage density is achieved with DRAMs. Indeed, DRAMs are among the densest VLSI circuits in terms of transistors per chip. The charge stored in a DRAM cell tends to decay with time, and the cell must be periodically refreshed. Hence a DRAM must contain refreshing circuitry and interleave refreshing operations with normal memory accesses. Both SRAMs and DRAMs are volatile, that is, the stored information is lost when the power source is removed.

Figure 6.9 shows examples of MOS RAM cells of both the static and dynamic varieties. The six-transistor SRAM cell (Figure 6.9a) superficially resembles a flip-flop. A signal applied to the address line (also called the word line) by the address decoder selects the cell for either the read or write operation. The two data lines (also called bit lines) are used in a complex way [Weste and Eshraghian 1992] to transfer the stored data and its complement between the cell and the data drivers.

Figure 6.9b shows a particularly simple and useful memory cell based on dynamic charge storage. This one-transistor DRAM cell comprises an MOS tran-sistor T, which acts as a switch, and a capacitor C, which stores a data bit. Apart from power and ground, the cell has only two external connections: a data (bit) line and an address (word) line. To write information into the cell, a voltage signal (either high or low, representing 1 and 0, respectively) is placed on the data line. A signal is then applied to the address line to switch on T. This action transfers a charge to C if the data line is 1; no charge is transferred otherwise. To read the cell, the address line is again activated, transferring any charge stored in C to the data

Address line

Data line D

Power

-L-

Ground(a)

*r *-<'

Address line

T

Data line D

Ground

Data line D

(b)

Figure 6.9

(a) Static and (b) dynamic RAM cells in MOS technology.

line where it is detected. Since the readout process is destructive, the data being read out is amplified and subsequently written back to the cell; this process may be combined with the periodic refreshing operation required by dynamic memories. The advantages of this DRAM cell are its small size, which means that ICs with very high cell density can be manufactured, and its low power consumption.

411

CHAPTER 6

MemoryOrganization

RAM design. A RAM IC typically contains all required access circuitry, including address decoders, drivers, and control circuits. Figure 6.10 shows a generic 2m x w-bit RAM IC and identifies its control lines. WE is the write-enable line; a memory write (read) operation takes place if WE = 1 (0). A second control line, the chip-select line CS, triggers a memory operation. A word is accessed for either reading or writing only when CS is activated. This line signals that the data bus has a word ready to be written into the RAM or, in the case of a read operation, that the data bus is ready to receive a data word. The RAM of Figure 6.10 has a bidirectional data bus D, which is directly wired to all addressable storage loca-tions, and so it requires a third control line, output enable OE. In write (input) oper-ations this line is deactivated (OE = 0), allowing D to act as an input bus to all storage locations. Of course, only the addressed location actually stores the word received on D. In read (output) operations, OE must be activated (OE = 1) so that only the addressed memory location transfers its data to D.

A memory-design problem that the computer architect may encounter is the following: given that N x w-bit RAM ICs denoted MNw are available, design an N' x w'-bit RAM, where N' > N and/or w' > w. A general approach is to construct a p x q array of the MNw ICs, where p = \N'/N\ q = fw'/w~|, and [x~] denotes the smallest integer greater than or equal to x. In this IC array each row stores N words (except possibly the last row), while each column stores a fixed set of w bits from every word (except possibly the last column). For example, to construct a 1GB RAM using 64M x 1-bit RAM ICs requires p = 16, q = 8, and a total of pq = 128 copies of the 64Mb RAM. When N' > N, additional external-address-decoding cir-cuitry is usually required.

Consider the task of designing an N x 4w-bit RAM using N x w-bit ICs of the type appearing in Figure 6.10. Clearly, four ICs are needed to quadruple the word size in this way, since p - 1 and q = 4. The four are arranged in the 1 x 4 array con-figuration of Figure 6.11. Each RAM IC contains a w-bit slice of every stored word. Note how all the address and control lines are connected in exactly the same

Address A
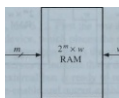
*—»• DataD

Output enable OEWrite enable WEChip select CS

Figure 6.10

A RAM IC showing its major external connections.

412

way to each IC. Their w-bit data buses are concatenated to form a single 4u -bit bus, as indicated.

Now suppose we want to increase the number of stored words by a factor of four. This time p = 4 and q = 1, and again we need four RAM ICs. The number of addresses has quadrupled, hence two lines are added «to the address bus. Further-more, a one-out-of-four address decoder must be introduced, as shown in Figure 6.12, to decode the extra address bits. The original m address lines are connected to the m-bit address bus of every RAM IC; the new lines are the main inputs to the decoder. Each decoder output line is connected to the CS inputs of the RAMs in the same row, ensuring that the row has a unique address. The output buses of all RAM ICs in the same column are designed so they can be wired together without addi-tional logic. (This tristate busing technique is explained in section 7.1.1.) The remaining control lines WE and OE are attached to every RAM IC as before. The external CS line is connected to an enable input of the decoder. Making this line 0 forces all CS lines to the individual RAM ICs to 0 so that they are all deactivated and no memory operation takes place.

EXAMPLE 6.1 A COMMERCIAL 64Mb DRAM CHIP IMICRON TECHNOLOGY1997]. The Micron Technology MT4LC8M8E1, which we will call the 8E1 for short, is a commercial DRAM chip introduced in 1996. It stores 64 Mb, that is, 226 bits of data, in single-transistor storage cells of the kind shown in Figure 6.9b. The stored information is organized as 223 8-bit bytes, so the 8E1 is also referred to as an 8M x8-bit DRAM. The memory address size m = 23, and the data word size w = 8.

The internal structure of the 8E1 appears in Figure 6.13. Two-dimensional addressing is employed, with the 23-bit address broken into two parts: a 13-bit row address and a 10-bit column address. Only 13 external address lines are used, allowing the 8E1 to be housed in a small, 32-pin package, which implies that row and column addresses must be multiplexed over the address bus, a common tactic in large RAM chips. This multiplexing is controlled by two lines: RAS (row address select) and CAS (column address select), which replace the generic CS control line of Figures 6.10,

DataD

4tv

w   W   H   W

2mxwRAM   2mxwRAM 2mxwRAM   2mXH-RAM

m   CS WE OE   CS WE OE   CS WE OE CS WE OE

Address A ■ < >   4   i   4 »

i ►   <   >   1 >—
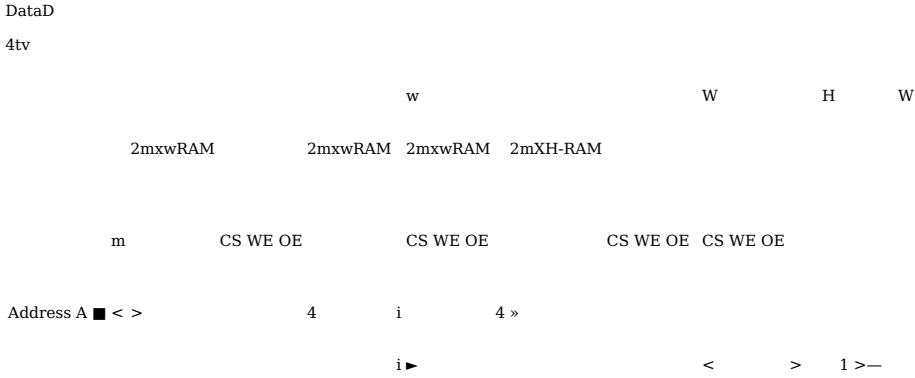
Control lines CS WE OE

Figure 6.11

Increasing the word size of a RAM by a factor of four.
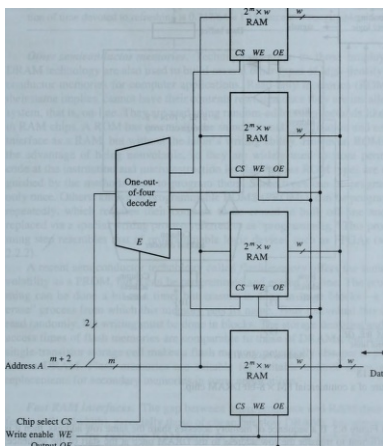
413

CHAPTER 6

Memory

Organization

DataD

Chip select CS

Write enable WE

Output OE

Figure 6.12

Increasing the number of words stored in a RAM by a factor of four.

6.11, and 6.12. First the row address is transferred to the DRAM by the external (mas-ter) device, which places the row address on the 8El's address bus and activates RAS. The master then places the column address on the address bus, and activates CAS. CAS also serves to indicate that a data word is ready on the data bus (write operation) or that external bus is ready to receive a data word (read operation). WE and OE are the write-enable and output-enable lines, respectively. As the overbars in their names indi-cate, all the control lines are active in the 0 state.

The read access time fA, which is 50 ns in faster versions of the 8E1. includes the time needed to transfer the row and column addresses to the DRAM and the time to read out a data word. The read cycle time tM with respect to a "random" address stream is 90 ns, since every such access is followed by an internal restoring write, as depicted

414

SECTION 6.1

Memory Technology

Timing and

refreshing

control logic

Data bus DQ\:DQ%

Internalcontrolsignals

Data buffer

Rowaddressdecoder

4>

->-

8192x1024x8storage cell array

A

Columnaddress decoder

Rowaddressbuffer

13

,'

RASCAS WE OEControl lines

10

Columnaddressbuffer

13,-Address bus ,40:412

Figure 6.13

Structure of a commercial 8M x 8-bit DRAM chip.

in Figure 6.5. If a sequence of memory' accesses share the same row address, then it issufficient to transfer the row address to the DRAM once at the start of the sequence.This transfer causes an entire row of data, referred to as a page, to be read out and heldin an internal buffer. A subsequent memory access to the same page needs to transferonly a column address, thus reducing the effective memory cycle time fM. This time isfurther reduced by the fact that there is no need to write back and restore the page dataevery time a word from it is accessed. A fast access method of this type is called pagemode, and in the 8E1 case it reduces tM to 30 ns. The memory address space of the8E1 consists of 8192 rows or pages, each containing 1024 locations. Both RAS andCAS are normally deactivated before the start of a new read or write cycle. Page modeis established by activating RAS to load the row address and then maintaining it activefor the duration of a sequence of column-address transfers in which CAS is toggled inthe normal way.

To ensure that the stored data does not decay, every cell in the memory must beread to refresh it at least once every 64 ms, which is the specified refresh period tREF.An internal restoring write that performs the refreshing accompanies each such read, asin Figure 6.5. The 2-D addressing structure makes it possible to read and restore thecontents of an entire row of storage locations in a single read cycle. Hence the refresh

controller need only sweep through all the row addresses in a sequence of internal read 415cycles to implement the refreshing. If a one-row read operation takes 90 ns, then thetotal time needed to refresh the DRAM once is 90 x 8192 ns = 0.737 ms. Thus the frac- CHAPTER 6tion of time devoted to refreshing is 0.737/64 = 1.15 percent—a negligible amount. emory Organization

Other semiconductor memories. Techniques similar to those employed inDRAM technology are also used to build several other types of high-density semi-conductor memories for computer applications. Read-only memories (ROMs), astheir name implies, cannot have their contents rewritten once they are installed in asystem, that is, on-line. They are read using random-addressing methods like thosein RAM chips. A ROM has essentially the same internal organization and externalinterface as a RAM, but without the latter's writing ability. However, ROMs havethe advantage of being nonvolatile, so they are widely used to store permanentcode at the instruction and microinstruction levels. Various ROM types are distin-guished by the methods used to program them. Some types can be programmedonly once. Others, known as programmable ROMS (PROMs), can be programmedrepeatedly, which requires their contents to be erased in bulk off-line and thenreplaced via a special writing process referred to as "programming." This program-ming step resembles that of programmable logic devices such as FPGAs (section2.2.2).

A recent semiconductor technology called flash memory offers the same non-volatility as a PROM, but it can be programmed and erased on-line. The program-ming can be done a bit at a time, but erasure is done in large blocks—a "flasherase" process from which this memory gets its name. Thus individual bits can beread randomly, but writing must be done in blocks. The storage densities and read-access times of flash memories are comparable to those of DRAMs, but a simplersingle-transistor storage cell makes a flash memory potentially cheaper to producethan a DRAM. Flash memories are suitable for writable control stores and asreplacements for secondary memories in some applications.

Fast RAM interfaces. The gap between microprocessor and RAM data-trans-fer rates (bandwidth), especially those of cheap but slow DRAMs. has given rise tonovel methods for enabling RAM units to communicate at higher-than-normalspeeds. The use of multiple memory types, serial as well as random access, in amemory hierarchy is a separate speedup issue that we examine later. Here we arejust concerned with one level in the hierarchy—main memory, for example.

Suppose a particular RAM technology must supply a faster external processorwith individually addressable /7-bit words. There are two basic ways we can in-crease the data-transfer rate across its external interface by a factor of 5:

• Use a bigger memory word. We can design the RAM with an internal memoryword size of w = Sn bits. This size permits Sn bits to be accessed as a unit in onememory cycle time TM. We then need fast circuits inside the RAM that, in thecase of a read operation, can access an Sn-bil word, break it into S parts, and out-put them to the processor, all within the period TM. During write operations,these circuits must accept up to S n-b'n words from the processor, assemble theminto an nS-bit word, and store the result, again within the period 7*M.

• Access more than one word at a time. We can partition the RAM into S separatebanks MQ,M} Ms_,, each covering part of the memory address space and each

416 provided with its own addressing circuitry. Then it is possible to carry out S inde-

, pendent accesses simultaneously in one memory clock period TM. Once more,

Memory Technology

we need fast circuits inside the RAM unit to assemble and disassemble the wordsbeing accessed.

Both approaches increase the memory bandwidth«by increasing the amount ofparallelism in memory accesses, and both require fast parallel-to-serial and serial-to-parallel circuits at the processor-memory interface. Hence an interface technol-ogy different from that of the RAM itself may be necessary; therefore, theseapproaches may not be suitable for single-chip RAM designs. The special interfacecircuits also add substantially to the overall cost of the memory system.

The S words produced or consumed by the processor in each memory cyclenormally have consecutive memory addresses, so we must consider how theseaddresses are implemented inside the RAM, particularly when S independentmemory banks are used. Let Xh,Xk+l,Xh+2>--- be words that are expected to beaccessed in sequence by the processor, for example, consecutive instruction wordsin a program. They will normally be mapped to consecutive physical addresses/4I,/4I + 1,y4,+2.-- in the RAM. The following rule is employed to distribute theseaddresses among 5 memory banks:

Interleaving rule: Assign address A, to bank M} if j = / (modulo 5).

Thus Aq, As, A25,... are assigned to M0; A,, As+1, A^+p... are assigned to A/,; andso on. This way of distributing addresses among memory banks is address inter-leaving. The interleaving of addresses among S banks according to the above ruleis S-way interleaving. It is convenient to make 5, the number of banks, a power oftwo, say, S = 2P. Then the least significant p bits of a memory address immediatelyidentify the bank to which the address belongs.

The appropriate number of memory banks S is determined by comparing thecycle time of the RAM technology to the data requirements of its host processor.Consider the case of the Cray-1, an influential supercomputer of the mid-1970s,which uses address interleaving in its main memory M. The CPU cycle time is12.5 ns, and the semiconductor main memory has a cycle time of ?M= 50 ns and aword size of w = 64 bits. (The Cray-1 has no cache, however.) Although the num-ber of memory accesses associated with each CPU cycle varies from cycle tocycle, a reasonable estimate is that when operating at maximum speed, oneinstruction word and two input operand words are read from M and one resultword is written into M. Hence a memory bandwidth of four 64-bit words per CPUcycle, or 16 words per memory cycle, is required. Consequently, the Cray-1 has 16memory banks and uses 16-way address interleaving.

The efficiency of an interleaved memory system is highly dependent on theorder in which memory addresses are generated; this order is determined by theprograms being executed. If two or more addresses require simultaneous access tothe same module, then memory interference or contention occurs. The memoryaccesses in question cannot be executed simultaneously. In the worst case, if alladdresses refer to the same module, the advantages of interleaving are entirely lost.

Various high-performance interfacing techniques have been devised for RAMs[Kumanoya, Ogaywa, and Inoue 1995]. As discussed in Example 6.1, a 2-D RAMorganization with multiplexed row and column addresses facilitates page address-ing, in which the row or page address remains fixed while the processor supplies a

stream of column addresses. This technique, which exists in several variations, hasthe effect of approximately doubling the data-transfer rate compared with pure"random" addressing. Another DRAM design style called synchronous DRAM(SDRAM) achieves a speed doubling by pipelining its internal operations and byimplementing two-way address interleaving. To facilitate this internal architecture,the timing relationships among the SDRAM's control signals (WE, CS, and so on)are

streamlined so that the SDRAM presents a synchronous (clocked) interface to the outside world. The so-called cached DRAMs (CDRAMs) feature an on-chip cache realized by a small, fast SRAM that acts as a high-speed buffer or front-end memory for the main DRAM. A common characteristic of the preceding RAM styles is that they can have a fast burst mode of operation, where an initial slow access is followed by a sequence or burst of much faster accesses.

EXAMPLE 6.2 THE RAMBUS DRAM AND INTERFACE [PRINCE 1996]. First announced in 1992, this is a proprietary DRAM design with a supporting processor-memory interface that aims to transfer memory data at very high rates over a narrow processor-memory link. It employs several speedup techniques, including a synchro-nous interface, address interleaving and caching inside the DRAM units, very fast sig-nal timing, and stringent electrical design rules. The Rambus data bus is 8 or 9 bits wide, with the 9th bit typically serving as a parity check. The peak data transfer rate is 500 MB/s which, however, is achievable only in burst mode.

Figure 6.14 depicts the overall Rambus organization. As we will see in Chapter 7, it is closer in style to that of a typical IO interface than a traditional memory interface. Rambus DRAM units are attached to this shared interface—the Rambus channel—which consists of a nine-line data bus D and a small set of control lines. Access is con-trolled either by the host CPU or a special Rambus controller chip acting as the master control unit. Each Rambus DRAM unit covers part of the memory-address space and acts as an independent slave device that communicates with the master via the Ram-bus channel. The Sm/S0M lines that link the DRAM units in daisy-chain fashion are used for initialization. They enable the master to visit each DRAM unit in turn to load a configuration register that determines the range of addresses to which that unit responds.

Normal Rambus operation is as follows. The master transmits an initial "packet" of information on the Rambus channel: this packet contains a target memory address

RambusDRAM R a   RambusDRAM/?

s,,,   1   L_ Hr

c

Master

(CPU)

RambusDRAM Rk

h~

                        9,

            5.

        8'

< »

Data bus DBus controlBus enableReceive clockTransmit clockPower (VDD)GroundVref

Figure 6.14

The Rambus DRAM interface.

418 and the desired access (read or write) operation. Each Rambus DRAM chip examines

the address, and the DRAM unit /?, containing that address returns either a "ready" or a "busy" control signal to the master. If /?, is ready, the master then proceeds to transfer

Memory Technology tQ ^ a data packet of up to 250 bytes (write case) or /?,- sends the master a data packet

(read case). This data transmission takes place in burst mode at speeds up to 500 MB/s, which implies accessing and transferring up to 1 byte a very 2 ns. If Rt is busy with an earlier operation when an access request arrives, the master must try again later and a significant delay in response time occurs.

6.1.3 Serial-Access Memories

The data in a serial-access memory must be accessed in a predetermined order via read-write circuitry that is shared by different storage locations. Large serial mem-ories typically store information in a fixed set of tracks, each consisting of a sequence of 1-bit storage cells. A track has one or more access points at which a read-write "head" can transfer information to or from the track. A stored item is accessed by moving either the stored information or the read-write heads or both. Functionally, a storage track in a serial memory resembles a shift register, so data transfer to and from a track is essentially serial.

Serial-access memories find their main application as secondary computer memories because of their low cost per bit and relatively long access times. Low cost is achieved by using very simple and small storage cells. Long access time is due to several factors:

• The read-write head positioning time.

• The relatively slow speed at which the tracks move.

• The fact that data transfer to and from the memory is serial rather than parallel.

Because access speed is so important, we now consider this factor in detail.

Access methods. Serial memories such as magnetic hard disks can be divided into those where each track has one or more fixed read-write heads and those whose read-write heads are shared among different tracks. In memories that share read-write heads, the need to move the heads between tracks introduces a delay. The average time to move a head from one track to another is the seek time ts of the memory. Once the head is in position, the desired cell may be in the wrong part of the moving storage track. Some time is required for this cell to reach the read-write head so that data transfer can begin. The average time for this movement to take place is the latency tL of the memory. In memories where information rotates around a closed track, tL is called the rotational latency.

Each storage cell in a track stores a single bit. A w-bit word may be stored in two different ways. It can consist of w consecutive bits along a single track. Alter-natively, w

tracks may be used to store the word, with each track storing a different bit. By synchronizing the w tracks and providing a separate read-write head for each track, all w bits can be accessed simultaneously. In either case it is inefficient to read or write just one word per serial access, since the seek time and the rotational latency consume so much time. Words are therefore grouped into larger units called blocks. All the words in a block are stored in consecutive locations so that the time to access an entire block includes only one seek and one latency time.

Once the read-write head is positioned at the start of the requested word or block, data is transferred at a rate that depends on two factors: the speed of the stored information relative to the read-write head and the storage density along the track. The speed at which data can be transferred continuously to or from the track under these circumstances is the data-transfer rate. If a track has a storage density of T bits/cm and moves at a velocity of V cm/s past the read-write head, then the data-transfer rate is TV bits/s.

The time tB needed to access a block of data in a serial-access memory can be estimated as follows. Assume that the memory has closed, rotating storage tracks of the type shown in Figure 6.4. Let each track have a fixed (average) capacity of A/words and rotate at r revolutions per second. Let n be the number of words per block. The data-transfer rate of the memory is then rN words/s. Once the read-write head is positioned at the start of the desired block, its data can be transferred in approximately n/{rN) seconds. The average latency is l/(2r) seconds, which is the time needed for half a revolution. If ts is the average seek time, then an appropriate

formula for tB is

$$t_B = t_D + \frac{1}{2r} + \frac{n}{rN}$$

(6.1)

419

CHAPTER 6

Memory

Organization

Memory organization. Figure 6.15 shows the overall organization of a serial-memory unit. Assume that each word is stored along a single track and that each access results in the transfer of a block of words. The address of the data to be accessed is applied to the address decoder, whose output determines the track to be

V

I

Read-write head

Address decoder

Address bus

V

Storage tracks

Read-write head

Timing and control logic

Data buffers

V

I

Read-write head

TX

t i

Control bus Data bus
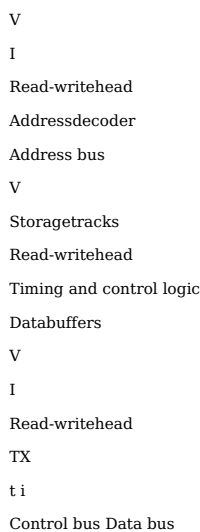
Figure 6.15

Organization of a serial-access memory unit.

420 used (the track address) and the location of the desired block of information within

section 6 ^e tra°k ^e block address). The track address determines the particular read-

Memo - Technology WI*ite head to be selected. Then, if necessary, the selected head is moved into posi-tion to transfer data to or from the target track. The desired block cannot be accessed until it coincides with the selected head. To determine when this condi-tion occurs, a track-position indicator generates the address of the block that is cur-rently passing the read-write head. The generated address is compared with the block address produced by the address decoder. When they match, the selected head is enabled and data transfer between the storage track and the memory data buffer registers begins. The read-write head is disabled when a complete block of information has been transferred.

The number of different types of storage media and access mechanisms used to construct serial memories is quite large. In many such memories the read-write heads or the storage locations are moved through space by electromechanical devices, such as electric motors, in order to perform an access. The most widely used group of secondary memory devices, magnetic-disk and -tape units, fall into this category, as do many optical memories. Some optical memories—CD-ROMs are an example—employ laser beams with electromechanical focusing as their read-write heads. Only a few serial memories have no moving parts, for example, the so-called solid-state disks, which use semiconductor RAM technology to simu-late the behavior of disk memories in applications that need unusually fast (and therefore expensive) secondary memory. Magnetic memories with electromechani-cal access have had many years of development. The storage media (magnetic disks and tape cartridges) are inexpensive and portable. Electromechanical equip-ment is less reliable than electronic equipment, however, and is a common source of computer system failure.

Magnetic-surface recording. Magnetic-disk and -tape memories store infor-mation on the surface of tracks coated with a magnetic medium such as ferric oxide. Each cell of a track has two stable magnetic states that represent logical 0 and 1. These magnetic states are defined by the direction or magnitude of the cell's magnetic flux in the cell. Electric currents alter and sense the magnetic states, for example, via an inductive read-write head of the type shown in Figure 6.16. The read and write signals pass through coils around a ring of soft magnetic material. A very narrow gap separates the ring from a cell on the storage track so that their respective magnetic fields can interact. This interaction permits information trans-fer between the read-write head and the storage medium.

To write data, the addressed cell is moved under the read-write gap. A pulse of current is then transmitted through the write coil, which alters the magnetic field at the ring gap; this in turn alters the magnetization state of the cell under the gap. The direction or magnitude of the write current determines the resulting state. To read a cell, it is moved past the read-write head, causing the magnetic field of the cell to induce a magnetic field in the core material of the read-write head. Since the cell is in motion, this magnetic field varies and so induces an electric voltage pulse in the read coil. This voltage pulse, which is then fed to a sense amplifier, identifies the state of the cell. The readout process is nondestructive; in addition, magnetic-surface storage is nonvolatile.

Electromechanically accessed magnetic memories are distinguished by the shapes of the surfaces in which the storage tracks are embedded. In disk memories

the tracks form concentric circles on the surface of a plastic or metal disk. In tape memories the tracks form parallel lines on the surface of a long, narrow plastic tape.

Magnetic-disk memories. A magnetic-disk unit employs storage media con-sisting of thin disks with a coating of magnetic material on which data can be recorded. One or both surfaces of a disk contain thousands of recording tracks arranged in concentric circles as shown in Figure 6.17a. Several disks can be attached to a common spindle; the four disks in Figure 6.11b provide up to eight recording surfaces.

During operation of the memory, the disks are rotated at a constant speed by a disk drive unit. Each recording surface is supplied with at least one read-write head. The read-write heads can be connected to form a read-write arm, as shown in Figure 6.17ft, so that all heads move in unison. This arm moves back and forth to

421

CHAPTER 6

Memory

Organization

Read-writehead

Data in

->

c

c

6"

=> c

c

CTU

>-

M<

Data out

Track surface

Motion

ttt

Ul

ttt

u*

Magneticmedium

Substrate

Figure 6.16

Magnetic-surface recording mechanism.

Magnetic disk

Sector
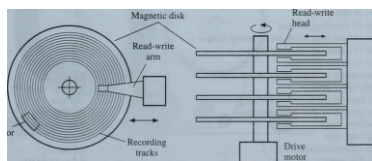
(a)

Figure 6.17

(a) Top view and (b) side view of a magnetic-disk drive unit.

(*)

422 select a particular set of tracks for reading or writing. The recording surface is

~~~7~,. , divided into sectors so that the part of a track within a sector stores a fixed amount

SECTION 6.1 „. _ .. i •,.,,»,

M t h l ° in'ormatlon corresponding to the memory unit s block size. Memory control is

simplified if all tracks store the same amount of data, in which case the track den-sity (bits stored per cm) on the outer tracks is less than the maximum possible.

Since their introduction in the 1950s by IBM, mdgnetic-disk memories haveundergone steady evolution characterized by decreasing physical size and increas-ing storage density. Small flexible magnetic disks referred to as floppy disks form acompact, inexpensive, and portable medium for off-line storage of small amountsof data, for instance, 1.4 MB. They are contrasted with hard disks, which are oftensealed into their drive units and have much higher storage capacity and reliability.

EXAMPLE 6.3 A COMMERCIAL MAGNETIC HARD-DISK MEMORY UNIT [QUAN-TUM corp. 1996]. The XP39100 is a 9.3 GB hard-disk memory in die Atlas II seriesmanufactured by Quantum Corp. and introduced in the mid-1990s. It is housed in arectangular box whose dimensions are approximately 14.6 x 10.2 x 4.14 cm. It con-tains ten 3.5 (8.89 cm) diameter disks, supplying a total of 20 recording surfaces,each with its own read-write head. Figure 6.18 summarizes the main features of misdevice. The cited capacity of 9.1 GB is for a formatted disk, which stores a directoryand other control information needed to make the disk drive ready for use. The numberof sectors along a track varies from 108 to 180, and each sector within a track accom-modates a 512-byte block. While the sector size is fixed, the number of sectors pertrack varies due to the fact that the inner tracks are smaller and can therefore store lessinformation at the maximum recording density of the magnetic medium. The averageblock access time given by Equation (6.1) with the data from Figure 6.18 is

where ts = 7.9 ms, r = 0.120 revs/ms, n = 8, and we take (108 + 180)/2 = 144 to be theaverage number of sectors per track, implying mat A/= 144 x 512 = 73,728 bytes/track.Observe that the seek time is the major factor in rB. The data-transfer rate rN = 120 x

Parameter Size

Disk diameter (form factor) 3.5 in (8.89 cm)

Number of disks 10

Number of recording surfaces 20

Number of read-write heads per recording surface 1

Number of tracks per recording surface 5964

Number of sectors per track 108 to 180

Storage capacity per track sector (block size) 512 bytes

Track-recording density 110,000 bits/in

Storage capacity per recording surface (formatted) 445 MB

Storage capacity of disk drive (formatted) 9.1 GB

Disk-rotation speed 7200 rev/min

Average seek time 7.9 ms

Average rotational latency 4.2 ms

Internal data-transfer rate 8.7 to 13.8 MB/s

External (buffered) data-transfer rate 20 to 40 MB/s

Figure 6.18

Characteristics of the Quantum Atlas II model XP391000 magnetic hard-disk memory unit.

73,728 = 8.85 MB/s, which is consistent with Figure 6.18. Because of factors such asdata buffering in the hard-disk unit and the format of its external interface, the usermay see a different and higher effective data-transfer rate.

Other noteworthy features of the XP39100 hard disk are a built-in 1 MB cache tobuffer data transfers and an error-correcting code that is applied on the fly to data beingstored in the XP39100. The system's reliability is measured by its MTBF, which themanufacturer projects to be 1 million hours.

423

CHAPTER 6

MemoryOrganization

Magnetic-tape memories. The magnetic-tape unit is one of the oldest andcheapest forms of mass memory. Its main use today is to provide backup storagefor a computer system in the event of failure of its hard disk subsystem. Magnetic-tape memories resemble domestic tape recorders, but instead of storing analogsound, they store binary digital information. The storage medium has as its sub-strate a flexible plastic tape, usually packaged in a small cassette or cartridge. Fig-ure 6.19 shows a standard memory of the data-cartridge type containing a magnetictape, which is 0.25 in (6.35 mm) wide and about 200 m long.

Data is stored on a tape in parallel, longitudinal tracks. Older tapes employednine such tracks designed to store one data byte and a parity bit across the tape;newer tapes have as many as several hundred tracks. A read-write head can simul-taneously access all tracks. Data transfer takes place when the tape is moving atconstant velocity relative to a read-write head; hence the maximum data-transferrate depends largely on the storage density along the tape and the tape's speed. Forexample, if an 80-track tape has a per-track storage density of 110 Kb/in and the

Drive belt

Read-write headaccess door

Figure 6.19

A magnetic tape cartridge.

Drive capstan

Recording tape

424 tape speed is 50 in/s, the maximum data-transfer rate d is 110,000 x 80/8 x 50 = 55

cc^nv., MB/s. A 200 m tape of this type can store about 55/50 x 200/0.0254 = 8.661 GB, a

SECTION 6.1 ..jji/-- • m- .,

Memory Technology number that is reduced by formatting requirements. The time to scan or rewind anentire tape is about a minute.

Information stored on magnetic tapes is organized into blocks, usually of fixedlength. A relatively large gap is inserted at the end of each block to permit the tapeto start and stop between blocks. If the block length is bl and the interblock gaplength is gl, then the tape's (space) utilization u is measured by

(6.2)

bl + gl

For example, if gl = 0.6 in, the storage density s = 3200 b/in, and a block storesbs = 4KB of data, then bl = 4096/3200 = 1.28 in. Equation (6.2) implies that u =1.28/1.88 = 0.68.

Because of the interblock gaps and the time needed to start and stop the tapebetween accesses, the effective data-transfer rate def{ seen by the user is less thanthe quoted, maximum rate d. Let tD denote the time to scan a data block, let tG bethe time to scan an interblock gap, and let /ss be the time to start and stop the tape.Then

eff " 'd + 'g + 'ss

If the block and gap sizes in bytes are bs and gs, respectively, then tD = bs/d, andtG = gs/d, so this equation becomes

bsd

de{f = - - (6.3)

bs + gs + tss ■ d

and the effective block access time tB is l/deff. For example, with bs = 4096 bytes;gl = 0.6 in, corresponding to gs = 1.92 bytes; d = 100,000 bytes/s; and /ss = 2 ms,Equation (6.3) yields de{{ = 65,894 bytes/s, a reduction of 34 percent from the max-imum data-transfer rate.

Optical memories. Optical or light-based techniques for data storage havebeen the subject of intensive research for many years. Such memories usuallyemploy optical disks, which resemble magnetic disks in that they store binaryinformation in concentric tracks (or a spiral track in the CD-ROM case) on an elec-tromechanically rotated disk. The information is read or written optically, how-ever, with a laser replacing the read-write arm of a magnetic-disk drive. Opticalmemories offer extremely high storage capacities, but their access rates are gener-ally less than those of magnetic disks. Read-only optical memories are well devel-oped, but low-cost read-write memories have proven difficult to build.

The CD-ROM is a well-established read-only optical memory. CD-ROMs arean offshoot of the audio compact disks (CDs) introduced in the 1980s. They aremanufactured in the same 12 cm format and can be mass-produced at very low costper disk by injection molding. Binary data is stored in the form of 0.1 urn widepi'fsand lands (nonpitted areas) in circular tracks on a plastic substrate; see Figure 6.20.A laser beam scans the tracks and is reflected differently by the pits and lands. Amirror-and-lens system forms a read arm that can move back and forth across thetracks. The mirror can also be tilted slightly to provide fine tracking adjustments.

Mirror

Light beamsplitter

CD-ROM disk Data out

Figure 6.20

Optical readout mechanism for a CD-ROM.

The reflected light from the laser is picked up by a sensor and decoded to extract the stored information, which is then converted to electronic form for further processing. A standard 12 cm CD-ROM has a capacity of around 600 MB. which is enough to store some 240.000 pages of printed text—a large encyclopedia, for instance. Access time is about 100 ms, and data is transferred from the disk at a rate of 3.6 MB/s (in so-called 24-speed CD-ROM drives). Low-cost CD drives are available that allow computer users to create their own CD ROMs under such names as CD-recordable (CD-R) and CD-rewritable (CD-RW). They employ a laser to create (bum) pits on the surface of blank disks. A much denser type of CD called a digital video disk (DVD) has recently been introduced in both read-only and read-write forms. With two recording surfaces and one or two storage layers per surface, a DVD can have a capacity as high as 16 GB.

A few types of secondary memory devices combine magnetic and optical recording methods. A magneto-optical disk memory uses rotating disks that store information in magnetic form but are accessed by a laser beam similar to that in a CD-ROM drive. Like a magnetic disk, a magneto-optical disk has a magnetizable surface coating whose direction of magnetization can be polarized (up or down corresponding to 0 or 1) as depicted in Figure 6.16. A cell is read by bouncing a laser beam off it. The beam's angle of polarization is affected by the cell's magne-tization direction, a phenomenon known as the Kerr effect. The slight change in the polarization angle of the reflected laser beam is sensed and decoded by the read mechanism. Writing is accomplished by using the laser beam to briefly heat a cho-sen cell above a specific temperature (the Curie temperature of the magnetic medium), at which point the cell's magnetic coercivity becomes zero, making the cell sensitive to external magnetic fields. An electromagnetic coil placed below the rotating disk then supplies a magnetic field of the required direction. The heated cell captures the magnetic field's direction which is retaineo after the cell cools below its Curie temperature.

426 6.2

MEMORY SYSTEMS

SECTION 6.2 Memory Systems

This section examines the general characteristics of memory systems that have a multilevel, hierarchical organization. Two key design issues are considered in detail: automatic translation of addresses and dynamic relocation of data.

6.2.1 Multilevel Memories

A computer's memory units form a hierarchy of different memory types in which each member is in some sense subordinate to the next-highest member of the hier-archy. The object of this organization is to achieve a good trade-off between cost, storage capacity, and performance for the memory system as a whole.

General characteristics. Consider a general n-level system of n memory types $(M_1, M_2, ..., M_n)$. Figure 6.21 shows some examples with n = 2, 3, and 4. Typ-ical technologies used in these hierarchies are semiconductor SRAMs for cache memory, semiconductor DRAMs for main memory, and magnetic-disk units for secondary memory. The two-level hierarchy of Figure 6.21a is typical of early computers. Figure 6.2\b adds a cache of a type called a split cache, since it has separate areas for storing instructions (the I-cache) and data (the D-cache). The third example (Figure 6.21c) has two cache levels, both of the nonsplit or unified type. Embedded microcontrollers also use the various hierarchical organizations depicted in the figure, but often lack the secondary or the cache levels.

The following relations normally hold between adjacent memory levels M, and Mi+1 in a memory hierarchy:

Cost per bit $c_i > c_{i+1}$

Access time $t_{A_i} < t_{A_{i+1}}$

Storage capacity $S_i < S_{i+1}$

The differences in cost, access time, and capacity between M, and M,+ 1 can be sev-eral orders of magnitude. Considerable system resources are devoted to shielding the CPU from these differences, so it almost always sees a very large and inexpen-sive memory space and rarely sees an access time greater than that of M,, the first (highest) level of the memory hierarchy.

The CPU and other processors can communicate directly with M_1 only, M_1 can communicate with M_2, and so on. Consequently, for the CPU to read informa-tion held in some memory level M, requires a sequence of i data transfers of the form

$M_{i-1} := M_i$; $M_{i-2} := M_{i-1}$; $M_{i-3} := M_{i-2}$; ... $M_1 := M_2$; CPU $:= M_1$.

An exception is allowed in the case of caches; the CPU is designed to bypass the cache level(s) and go directly to main memory, as we will see later. In general, all the information stored in M, at any time is also stored in M,+ 1, but not vice versa.

During program execution the CPU produces a steady stream of memory addresses. At any time these addresses are distributed in some fashion throughout the memory hierarchy. If an address is generated that is currently assigned only to

I: Instruction flow   I   Secondary memory
                                    M_3

D: Data flow   I   Main memory
                             M_2

                CPU

        ■ «—► D D

(«)

I

Secondarymemory

M3

Mainmemory
M2

I-cache

CPU

D-cache

1/   h

ib)

I   Secondarymemory

Mainmemory
M3

Level 2cache
M2

Level 1cache
If,

CPU

D   D   D

(c)

Figure 6.21

Common memory hierarchies with (a) two, (b) three, and (c) four levels.

M, where i * 1, the address must be reassigned to M,, the level of the memory hier-archy that the CPU can access directly. This relocation of addresses involves thetransfer of data between levels M, and M,—a relatively slow process. For a mem-ory hierarchy to work efficiently, the addresses generated by the CPU should befound in M, as often as possible. This approach requires that future addresses be tosome extent predictable so that information can be transferred to M, before it isactually referenced by the CPU. If the desired data cannot be found in M,, then theprogram originating the memory request must be suspended until an appropriatereallocation of storage is made.

Cache and virtual memory. The various parts of a memory hierarchy are con-trolled in very different fashions. Cache and main memory form a distinct subhier-archy whose design objective is to support CPU accesses with a minimum of delay.Hence hardware controllers that are transparent to both user and system programs

428

SECTION 6.2Memory Systems

usually manage this subhierarchy. Much more than the rest of the memory system,the cache and main memory resemble a single memory M to the software beingexecuted.

Main and secondary memory form another distinct two-level subhierarchy.This interaction is managed by the operating system, however, and so is not trans-parent to system software, although it is somewhat transparent to user code. Theterm virtual memory is applied when the main and secondary memories appear to auser program like a single, large, and directly addressable memory. Traditionally,there are three reasons for using virtual memory:

• To free user programs from the need to carry out storage allocation and to permitefficient sharing of the available memory space among different users.

• To make programs independent of the configuration and capacity of the physicalmemory present for their execution; for example, to allow seamless overflowinto secondary memory when the capacity of main memory is exceeded.

• To achieve the very low access time and cost per bit that are possible with amemory hierarchy.

A memory system is addressed by a set V of logical or virtual addressesderived from identifiers explicitly or implicitly specified in an object program. Aset of physical or real addresses R identifies the fixed physical storage locations ineach memory unit M,. An efficient and flexible mechanism to implement addressmappings of the form/: V —> R is the key to successful design of a multilevel mem-ory.

Locality of reference. The predictability of memory addresses depends on acharacteristic of computer programs called locality of reference, which says thatover the short term, the addresses generated by a program tend to be localized andare therefore predictable.

One reason for locality of reference is that instructions and, to a lesser extent,data are specified and subsequently stored in a memory unit in approximately theorder in which they are accessed during program execution. Suppose a request ismade for a one-word instruction / stored at address A, but this address is currentlyassigned to M, ^ M,. The instruction most likely to be required next by the CPU isthe one immediately following / whose address is A + 1. Figure 6.22, which showspart of the 680X0 program for vector addition discussed in Example 3.8 (section

Machine Language

Assembly Language

Location Instruction ; 680X0 program for vector addition

0100	2078 07D1	MOVE.L	A+1000,A0
0104	2278 0BB9	MOVE.L	B+1000.A1
0108	2478 0FA1	MOVE.L	C+1000,A2
010C	C308	START ABCD	-(A0),-(A1)
010E	1511	MOVE.B	(A1),-<A2)
0110	B0F8 03E9	CMPA	A,A0
0114	66F6	BNE	START

;Set pointer beyond end of A;Set pointer beyond end of B;Set pointer beyond end of C.Decrement pointers and add; Store result in C;Test for termination;Branch to START if Z * 1

Figure 6.22

Code fragment illustrating locality of reference.

3.3.3), illustrates this tendency. The first (4 byte) instruction fetched has address 010016, the next has address 010416, the next 010816, and so on. This type of local-ity is called spatial because it implies that consecutive memory references are toaddresses that are close to one another in the memory-address space.

Instead of simply transferring / to M, when it is referenced, it is more efficientto transfer a block of consecutive words containing /. A common way to automatethis process is to subdivide the information stored in M, into pages, each containinga fixed number SP of consecutive words. Information is then transferred one pageor SP words at a time between levels M, and M,_,. Thus if the CPU requests word /in level M(, the page of the length SP in M, containing / is transferred to MM,then the page of length SP , containing / is transferred to M,_2, and so on. Finally,the page P of length SP containing / reaches M,, where the CPU can directlyaccess it. Subsequent memory references are likely to refer to other addresses in P,so the single transfer to M, anticipates future memory requests by the CPU.

A second factor in locality of reference is the presence of loops in programs.Instructions in a loop, even when they are far apart in spatial terms, are executedrepeatedly, resulting in a high frequency of reference to their addresses. This char-acteristic is referred to as temporal locality. When a loop is being executed, it isdesirable to store the entire loop in M, if possible. For example, in the small, four-instruction program loop shown in boldface in Figure 6.22, the BNE branchinstruction with address 011416 is usually followed by the instruction with the non-consecutive address $010C16$ (START).

The items of information whose addresses are referenced during the time inter-val from t - 7" to t, denoted (t - T, t), constitute the current working set W(t, T) of aprogram. W(t, T) tends to change rather slowly; hence by maintaining all of W(t, T)in the fastest level of memory M1? the number of references to M, can be made fargreater than the number of references to other levels of the memory hierarchy.

429

CHAPTER 6

Memory

Organization

Cost and performance. The overall goal in memory-hierarchy design is toachieve a performance close to that of the fastest device M, and a cost per bit closeto that of the cheapest device M,,. The performance of a memory system dependson various related factors, the more important of which are the following:

• The address-reference statistics, that is, the order and frequency of the logicaladdresses generated by programs that use the memory hierarchy.

• The access time /A of each level M, relative to the CPU.

• The storage capacity 5, of each level.

• The size Sp of the blocks (pages) transferred between adjacent levels.

• The allocation algorithm used to determine the regions of memory to whichblocks are transferred by the block-swapping process.

These factors interact in complex ways, which are by no means fully understood.Simulation of a multilevel memory using realistic address traces is often the bestway to determine suitable values for /A, 5,, SP, and other important design param-eters. A few analytic models indicate how these factors are related. Some usefulmodels of this kind are discussed next.

For simplicity we restrict our attention to a generic two-level memory hierarchydenoted by $(M^N^)$, which can be interpreted as (cache, main memory) or (mainmemory, secondary memory). It is not difficult to generalize our analysis front two-

430 level to H-level hierarchies. The average cost per bit of memory is given by

SECTION 6.2 ClSl + C2S2

Memory Systems C = (6.4)

5, + S2

c

where c, denotes the cost per bit of M, and 5, denotes the storage capacity in bits ofM,. To reach the goal of making c approach c2, 5, must be much smaller than S2.

The performance of a two-level memory is often measured in terms of the hitratio H, which is defined as the probability that a virtual address generated by theCPU refers to information currently stored in the faster memory M,. Since refer-ences to Mj (hits) can be satisfied much more quickly than references to M2(misses), it is desirable to make H as close to one as possible. Hit ratios are gener-ally determined experimentally as follows. A set of representative programs is exe-cuted or simulated. The number of address references satisfied by M1 and M2,denoted by iV, and N2, respectively, are recorded. H is calculated from the equation

and is highly program dependent. The quantity 1 - H is called the miss ratio.

Let tA] and tAl be the access times of Mj and M2, respectively, relative to theCPU. The average time tA for the CPU to access a word in the two-level memory isgiven by

tA = HtAi +(l-//)/A2 (6.6)

In most two-level hierarchies, a request for a word not in the fast level Mj causes ablock of information containing the requested word to be transferred to M, fromM2. When the block transfer has been completed, the requested word is available inMj. The time fB required for the block transfer is called the block-access or block-transfer time. Hence we can write t* = tB + tA . Substituting into Equation (6.6)yields

tA= tAt + (l-H)tB (6.7)

In many cases tA » tA ; therefore, rA ~ rB. For example, a block transfer fromsecondary to main memory requires a relatively slow IO operation, making rAi andtB much greater than tA .

Let r = tA/tA denote the access-time ratio of the two levels of memory. Lete = tA/tA, which is the factor by which tA differs from its minimum possiblevalue; e is called the access efficiency of the two-level memory. From Equation(6.6) we obtain

1

$$e = \frac{1}{r+(1-r)H}$$

Figure 6.23 plots e as a function of H for various values of r. This graph shows the importance of achieving high values of H in order to make e ~ 1; that is, tA ~ tA. For example, suppose that r = 100. In order to make e > 0.9, we must have H > 0.998.

Memory capacity is limited by cost considerations; therefore, we do not want to waste memory space. The efficiency with which space is being used at any time can be defined as the ratio of the memory space Su occupied by "active" or "useful"

0.4 0.6

Hit ratio H

Figure 6.23

Access efficiency e = tA /rA of a two-level memory as a function of hit ratio H for vari-

user programs and data to the total amount of memory space available S. We call this the space utilization u and write

$$U = \frac{S_u}{S}$$

Since memory space is more valuable in M, than in M2, it is useful to restrict u to measuring M,'s space utilization. In that case the S - 5U words of M, that repre-sent "wasted" space can be attributed to several sources.

• Empty regions. The blocks of instructions and data occupying Ml at any time are generely of different lengths. As the contents of M, are changed, unoccupied regions or holes of various sizes tend to appear between successive blocks. This phenomenon is called fragmentation.

• Inactive regions. Data may be transferred to M,, for example, as part of a page, and may be subsequently transferred back to M, without ever being referenced by a processor. Some superfluous transfers of this kind are unavoidable, since address references are not fully predictable.

• System regions. These regions are occupied by the memory-management soft-ware.

A central issue in managing (M,,M2), or any multilevel memory, is to make it appear to its users like a single, fast memory of high capacity. This goal can be achieved in a way that is largely transparent to the users by providing a memory management system that automatically performs the following tasks:

• Translation of memory addresses from the virtual addresses encountered in pro-gram execution to the real addresses that identify physical storage locations.

• Dynamic (re)allocation or swapping of information among the different memory

._,..., . levels so that stored items reside in the fastest level before they are needed.

These issues are explored individually in the following sections.

c

6.2.2 Address Translation

The set of abstract locations that a program Q can reference is Q 's virtual address space V. Such addresses can be explicitly or implicitly named by identifiers that a programmer assigns to data variables, instruction labels, and so forth. The addresses can also be constructed or modified by the system software that controls Q. To execute Q on a particular computer, its virtual addresses must be mapped onto the real address space R, defined by the addressable (external) memory M that is physically present in the computer. This process is called address transla-tion or address mapping. The real address space R is a linear sequence of numbers

0. 1, 2, ..., n - 1 corresponding to the addressable word locations in M. It is conve-nient to identify M with main memory, while noting that R is usually distributed over several levels of the memory hierarchy, including the cache and the level labeled "main" memory. V is a loose collection of lists, multidimensional arrays, and other nonlinear structures, so it is much more complex than R.

Address translation can be viewed abstractly as a function /: V —> R. This function is not easily characterized, since address assignment and translation is car-ried out at various stages in the life of a program, specifically:

1. By the programmer while writing the program.

2. By the compiler during program compilation.

3. By the loader at initial program-load time.

4. By run-time memory management hardware and/or software.

Explicit specification of real addresses by the programmer was necessary in early computers, which had neither hardware nor software support for memory management. With modern computers, however, programmers normally deal only with virtual addresses. Specialized hardware and software within the computer automatically determine the real addresses required for program execution.

A compiler transforms the symbolic identifiers of a program into binary addresses. If the program is sufficiently simple, the compiler can completely map virtual addresses to real addresses. Address translation can also be completed when the program is first loaded for execution. This process is called static translation, since the real address space of the program is fixed for the duration of its execu-tion. It is often desirable to vary the virtual space of a program dynamically during execution; this process is dynamic translation. For example, a recursive proce-dure—one that calls itself—is typically controlled by a stack containing the linkage between successive calls. The size of this stack cannot be predicted in advance because it depends on the number of times the procedure is called; therefore, it is desirable to allocate stack addresses on the fly. Hardware-implemented memory management units (MMUs) have come into widespread use for run-time address translation.

Base addressing. An executable program comprises a set of instruction and data blocks each of which is a sequence of words to be stored in consecutive mem-

Base address

	Effective address
Displacement D	<4eff
0	B
1	B + 1

m-\

B + i

B + m-l

Figure 6.24

Block of m words with (base)address B.

ory locations during execution. A word W within a block has its own effectiveaddress Ae({, which the CPU must know to access W. (For the moment, we willignore the distinction between the real and virtual address spaces.) W is also speci-fied by the address B, called the base address, of the block that contains it, alongwith Ws relative address or displacement D (also called an offset or index) withinthe block, as shown in Figure 6.24. Clearly,

$$A_{eff} = B + D \quad (6.8)$$

Often the address is designed so that B supplies the high-order bits of Aeff while Dsupplies the low-order bits thus:

$$A_{eff} = B.D \quad (6.9)$$

Now Aeff is formed simply by concatenating B and D, a process that does not sig-nificantly increase the time for address generation.

A simple way to implement static and dynamic address mapping is to put baseaddresses in a memory map or memory address table controlled by the memorymanagement system. The table can be stored in memory, in CPU registers, or inboth. The address-generation logic of the CPU computes an effective address y4effby combining the displacement D with the corresponding base address 5, accord-ing to (6.8) or (6.9).

Blocks are easily relocated in memory by manipulating their base addresses.Figure 6.25 illustrates block relocation using base-address modification. Supposethat two blocks are allocated to main memory M as shown in Figure 6.25a. It isdesired to load a third block K3 into M; however, a contiguous empty space, or"hole," of sufficient size is unavailable. A solution to this problem is to move blockK2, as shown in Figure 6.25b, by assigning it a new base address B,2 and reloadingit into memory. This creates a gap into which block AT3 can be loaded by assigningto it an appropriate base address.

With dynamic memory allocation, we must control the references made by ablock to locations outside the memory area currently assigned to it. The block canbe permitted to read from certain locations, but writing outside its assigned areamust be prevented. A common way of doing this is by specifying the highestaddress L(, called the limit address, that the block can access. Equivalently, the sizeof the block may be specified. The base address fl, and the limit address L, are

Block A-,

Block K2

*1

i;

Block A,

, Block A:2

*

Block K3   -*

(a) (fe)

Figure 6.25

Relocation of blocks in memory using base and limit addresses.

stored in the memory map. Every real address Ar generated by the block is com-pared to Bj and L,; the memory access is completed if and only if the condition

B,<A<L;

is satisfied.

Translation look-aside buffer. Figure 6.26 shows how various parts of a mul-tilevel memory management typically realize the address-translation ideas just dis-cussed. The input address Av is a virtual address consisting of a (virtual) baseaddress Bv concatenated with a displacement D. Av contains an effective addresscomputed in accordance with some program-defined addressing mode (direct, indi-rect, indexed, and so on) for the memory item being accessed. It also can contain

Translationlook-aside buffer TLB

containing (part of)the memory map

Real base   al base
(block)   i   address Bv
address BR   Virtu(bloc

               i i   >   Virtual address Av

                                Displacen(offset) D tent

                                r   Real address AR >■ To memory system

Figure 6.26

Structure of a dynamic address-translation system.

system-specific control information—a segment address, for example—as we willsee later. The real address BR = f(Bv) assigned to Bv is stored in a memory mapsomewhere in the memory system; this map can be quite large. To speed up themapping process, part (or occasionally all) of the memory map is placed in a smallhigh-speed memory in the CPU called a translation look-aside buffer (TLB). TheTLB's input is thus the base-address part Bvof Av; its output is the correspondingreal base address BR. This address is then concatenated with the D part of Av toobtain the full physical address AR.

If the virtual address Bv is not currently assigned to the TLB, then the part ofthe memory map that contains Bv is first transferred from the external memory intothe TLB. Hence the TLB itself forms a cachelike level within a multilevel address-storage system for memory maps. For this reason, the TLB is sometimes referred toas an address cache.

435

CHAPTER 6

Memory

Organization

EXAMPLE 6.4 MEMORY ADDRESS TRANSLATION IN THE MIPS R2/3000

[Kane 1988]. The MIPS R2/30O0 microprocessor, whose main features were intro-duced earlier CExamples 3.5 and 3.7), employs an on-chip MMU. The MMU's primaryfunction is to map 32-bit virtual addresses to 32-bit real addresses. (Later members ofthe RXOOO family like the R10000 support 64-bit addresses.) A 32-bit address allowsthe R2/3O00 to have a virtual address space of 232 bytes, or 4 GB. Both address spacesare composed of 4KB pages, which are convenient block sizes for information transferwithin a conventional memory hierarchy comprising a cache (of the split kind), mainmemory, and secondary memory. The 4GB virtual-address space is further partitionedinto four parts called segments, three of which form the system region (or "kernelregion"' in MIPS parlance) devoted to operating system functions, while the other is theuser region, where application programs, data, and control stacks are stored.

The format of an R2/3000 virtual address appears in Figure 6.27. It consists of a20-bit virtual page address, referred to as the virtual page name VPN. and a 12-bit dis-placement D, which specifies the address of a byte within the virtual page. The high-order 3 bits 31:29 of VPN form a type of tag that identifies the segment beingaddressed. Bit 31 of VPN is 0 for a user segment and 1 for a supervisor segment: it thusdistinguishes the user and supervisor (privileged) control states of the CPU. The usersegment is kuseg and occupies half the virtual address space. The supervisor region isdivided into three segments. ksegO. ksegl, and kseg. each of which has differentaccess characteristics.

• kuseg: This 2GB segment is designed to store all user code and data. Addresses inthis region make full use of the cache and are mapped to real addresses via the TLB.

• ksegO: This 512MB system segment is cached and unmapped; mat is, virtualaddresses within ksegO are mapped directly into the first 512 MB of the real addressspace, which includes the cache, but no virtual address translation takes place. Thissegment typically stores active parts of the operating system.

• ksegl: This is also a 512MB segment, but is both uncached and unmapped. It isintended for such purposes as storing boot-up code (which cannot be cached) and forother instructions and data—high-speed 10 data, for instance—that might seriouslyslow down cache operation.

• ksegl: This is a 1GB segment which, like kuseg, is both cached and mapped.

The MMU contains a TLB to provide fast virtual-to-real address translation. TheTLB stores a 64-entry portion of the memory map (page table) assigned to each processby the operating system. The current virtual page address WW is used to access a 64-bit entry in the TLB. which, as shown in Figure 6.27, contains among other items', a 20-

436

SECTION 6.2

Memory Systems

bit page frame number PFN. This real page address is fetched from the TLB andappended to the displacement D to obtain the desired 32-bit real address. An R2/3000-based system often has less than 4 GB of physical memory, in which case not all theavailable real address combinations are used.

Observe that the VPN itself is also part of the TLB entry because a fast accessmethod called associative addressing is used; see seetion 6.3.2. Another major itemstored in each TLB entry is a 6-bit process identification field PID. This field distin-guishes each active program (process); hence up to 64 processes can share the availablevirtual page numbers without interference. There are also 4 control bits denotedNDVG, which define the types of memory accesses permitted for the correspondingTLB entry. For example, N denotes noncachable; when set to 1, it causes the CPU to godirectly to main memory, instead of first accessing the cache. D is a write-protection(read-only) bit; an attempt to write when D = 0 causes a CPU interrupt or trap.

The MMU has some features not shown in Figure 6.27, which are designed to traperror conditions that are collectively referred to as address translation exceptions.When a trap occurs, relevant information about the exception is stored in MMU regis-ters, which can be examined and modified by certain privileged instructions. A commonaddress translation exception is a TLB miss, which occurs when there is no (valid) entryin the TLB that matches the current VPN. The operating system responds to a TLB missby accessing the current process's page table, which is stored in a known location inkseg1, and copying the missing entry to the TLB. Another address-translation exceptiontype is an illegal access—for instance, a write operation addressed to a page with D = 0(read only) in its TLB entry.

7 0

63 43   37 31   12

VPN | PID | 000000 | PFN |,v|d|v|g| oooooooo

Real pageaddress PFN i

i

Translationlook-aside buffer TLB

Virtual pageaddress VPN

31   0

i:

1 1 1 VPN   0;~ t: Virtual address Av

s

Displacer(offset) D

Odd kuse1 0 0 ksegi1 0 1 kseg1 1 d kseg 31   i   iient

I   0

31   12

PFN   Offset   Real address AR

-*■ To memory system

Figure 6.27

Memory address mapping in the MMU of the MIPS R2/3000.

Segments. The basic unit of information for swapping purposes in a multi-level memory is a fixed-size block called a page. Pages are allocated to page-sized storage regions (page frames), whose fixed size and address formats make paging systems easy to implement. Pages are convenient blocks for the physical partition-ing and swapping of the information stored in a multilevel memory. It is often desirable to have higher-level information blocks, termed segments, that corre-spond to logical entities such as programs or data sets. Segments facilitate the map-ping of individual programs, as well as the assignment and checking of different storage properties. For example, write operations may not be permitted into certain regions of the virtual address space in order to protect critical items. It is easier to protect the information in question by making it a read-only segment S, rather than assigning access restrictions to the possibly large number of pages that compose S.

Formally, a segment is a set of logically related, contiguous words; it is there-fore a special type of block in the sense used in section 6.2.1. A word in a segment is referred to by specifying a base address—the segment address—and a displace-ment within the segment. A program and its data can be viewed as a collection of linked segments. The links arise from the fact that a program segment uses, or calls, other segments. Some computers have a memory management technique that allocates main memory by M_x segments alone. When a segment not currently resi-dent in M_j is required, the entire segment is transferred from secondary memory M_2. The physical addresses assigned to the segments are kept in a memory map called a segment table (which can itself be a relocatable segment).

Segmentation was implemented in this general form in the Burroughs B6500/7500 series [Hauck and Dent 1968]. Each program has a segment called its pro-gram reference table (PRT), which serves as its segment table. All segments asso-ciated with the program are defined by special words called segment descriptors in the corresponding PRT. As shown in Figure 6.28, a B6500/7500 segment descrip-tor contains the following information:

• A presence bit P that indicates whether the segment is currently assigned to M_j.

• A copy bit C that specifies whether this is the original (master) copy of the descriptor.

• A 20-bit size field Z that specifies the number of words in the segment.

• A 20-bit address field S that is the segment's real address in M_j (when $P = 1$) or M_2 (when $P = 0$).

A program refers to a word within a segment by specifying the segment descriptor word W in its PRT and the displacement D. The CPU fetches and examines W. If the presence bit $P = 0$, an interrupt occurs and execution of the requesting program

437

CHAPTER 6

Memory

Organization

50	45	39	19	0

0 1 1   i i i i i i i i iii   i i i i i i i i i i i i i i i i

Tag ' '.t Copy b itC Segment size Z   Real address S in M| or M2

Presence bit P

Figure 6.28

Segment descriptor of the Burroughs B6500/7500.

438 is suspended while the operating system transfers the required segment from M-, to

M_j. When $P = 1$, the CPU compares D to the segment size field Z in the descriptor.

Mm s stems If $D > Z$, then D is invalid and an interrupt occurs. If $D < Z$, the address field S

from the descriptor is added to the displacement D. The result $S + D$ is the real address of the required word in M_t, which can then be accessed.

The main advantage of segmentation is that segment boundaries correspond to natural program and data boundaries. Consequently, information that is shared among different users is often organized into segments. Because of their logical independence, a program segment can be changed or recompiled at any time with-out affecting other segments. Certain properties of programs such as the scope (range of definition) of a variable and access rights are naturally specified by seg-ment. These properties require that accesses to segments be checked to protect against unauthorized use; this protection is most easily implemented when the units of allocation are segments. Certain segment types—stacks and queues, for instance—vary in length during program execution. Segmentation varies the region assigned to such a segment as it expands and contracts, thus efficiently using the available memory space. On the other hand, the fact that segments can be of different lengths requires a relatively complex allocation method to avoid exces-sive fragmentation of main-memory space. This problem is alleviated by combin-ing segmentation with paging, as discussed later.

Some computers implement a more specialized form of segmentation. The MIPS R2/3000 divides the virtual address space into four large regions that are treated as segments; see Example 6.4. Just 3 bits of the virtual address define the current segment. Microprocessors in the Intel 80X86 series, including the Pentium, have four 16-bit segment registers forming a segment table that supports a very large number of segments.

Pages. A page is a fixed-length block that can be assigned to fixed regions of physical memory called page frames. The chief advantage of paging is that data transfer between memory levels is simplified: an incoming page can be assigned to any available page frame. In a pure paging system, each virtual address consists of two parts: a page address and a displacement. The memory map, now referred to as a page table, typically contains the information shown in Figure 6.29. Each (vir-tual) page address has a corresponding (real) address of a page frame in main or secondary memory. When the presence bit $P = 1$, the page in question is present in main memory, and the page table contains the base address of the page frame to which the page has been assigned. If $P = 0$, a page fault occurs and a page swap ensues. The change bit C indicates whether or not the page has been changed since it was last loaded into main memory. If a change has occurred ($C = 1$), the page

Page address Page frame Presence bit P Change bit C Access rights

A 0000000 1 0 R, X

C D6C7F9 0 d R. W, X

E 0000024 1 1 R. W. X

F 0000016 1 0 R

Figure 6.29

Representative organization of a page table.

must be copied onto secondary memory when it is preempted. The page table can also contain memory protection data that specifies the access rights of the current program to read from, write into, or execute the page in question. Page tables differ from segment tables primarily in the fact that they contain no block size informa-tion.

As noted earlier, pages require a simpler memory allocation system than seg-ments, since block size is not a factor in paging. On the other hand, pages have no logical significance, as they do not represent program elements. Paging and seg-mentation can also be compared in terms of memory fragmentation. In systems with segmentation, holes of different sizes tend to proliferate throughout main memory; they can be eliminated by the time-consuming process of memory com-paction. Unusable space between occupied regions is called external fragmenta-tion. Since page frames are contiguous, no external fragmentation occurs in paged systems. However, if a k-word block is divided into p n-word pages, and k is not a multiple of n, the last page frame to which the block is assigned will not be filled. Unusable space within a partially filled page frame is called internal fragmentation.

Paging and segmentation can be combined in an attempt to gain the advantages of both. The great advantage of breaking a segment into pages is that it eliminates the

need to store the segment in a contiguous region of main memory. Instead, allthat is required is a number of page frames equal to the number of pages into whichthe segment has been broken. Since these page frames need not be contiguous, thetask of placing a large segment in main memory is eased.

When segmentation is used with paging, a virtual address has three compo-nents: a segment index SI, a page index PI, and a displacement (offset) D. Thememory map then consists of one or more segment tables and page tables. For fastaddress translation, two TLBs can be used as shown in Figure 6.30, one for seg-ment tables and one for page tables. As discussed earlier, the TLBs serve as fastcaches for the memory maps. Every virtual address Av generated by a programgoes through a two-stage translation process. First, the segment index SI is used toread the current segment table to obtain the base address PB of the required pagetable. This base address is combined with the base index PI (which is just a dis-placement within the page table) to produce a page address, which is then used toaccess a page table. The result is a real page address, that is, a page frame number,which can be combined with the displacement part D of Av to give the final (real)address AR. This system, as depicted in Figure 6.30. is very flexible. All the variousmemory maps can be treated as paged segments and can be relocated anywhere inthe physical memory space.

439

EXAMPLE 6.5 MEMORY ADDRESS TRANSLATION IN THE INTEL PENTIUM

[INTEL 19941. The Pentium is a 32-bit microprocessor introduced in 1993 that pro-vides direct hardware support for both segmentation and paging. It is a member ofIntel's 80X86 microprocessor family and maintains some degree of compatibility at theobject-code level with its predecessors back to the original, 1976-vintage 8086 CPU.Most of the memory-addressing features discussed here originated with the 80386,introduced in 1985.

Like the MIPS R2/3000 (Example 6.4), the Pentium"s real address space can be aslarge as 4 GB (232 bytes); however, the virtual address space can be an extremely large64 TB (64 terabytes = 246 bytes). An on-chip MMU has a segmentation unit that per-forms address translation for segments ranging in size from 1 to 232 bytes. A separatepaging unit handles address translation for pages of size 4 KB or 4 MB. Any one of the

440

Virtual address Av

Real address AR

To memorysystem M

Figure 6.30

Two-stage address translation with segments and pages.

following four memory access methods can be selected under program control: unseg-mented and unpaged, segmented and unpaged, unsegmented and paged, and segmentedand paged. The output of the paging unit is a 32-bit real address, while that of the seg-mentation unit is a 32-bit word called a linear address. If both segmentation and pagingare used, every memory address generated by a program goes through a two-stagetranslation process

Virtual address Av —> linear address TV —> real address AR

as depicted in Figure 6.31. Without segmentation Av = N, while without paging N =AR. The segmentation and paging units both contain TLBs to store the active portionsof the various memory maps needed for address translation, so the delay of the transla-tion process is small. This delay is further diminished by overlapping (pipelining) theformation of the virtual, linear, and real addresses, as well as by overlapping memoryaddressing and fetching, so the next real address is ready by the time the current mem-ory cycle is completed.

An active process controlled by the Pentium has several segments associated withit, such as the object program code, a program control stack, and one or more data sets.Each segment can be thought of as a virtual memory of size 4 GB, which has the linear

Segmentationunit

Segment table

Segmenttable base

J?

Segment \ 14index Ls

46-bit virtualaddress A v

Segmentregister

32-'

32

Le ,'14

Effectiveaddress

32/

32-bit linear -

address N >■

10

' -'10

Paging page ^Me

unit

Page tableindex Nn

Page directory

'32

20

Effective address unit

32-bit real address AR

Page frame

Page

directory

base

Page directory index Nd

■ ' -'12

Page Displacement frame

Figure 6.31

Address translation with segmentation and paging in the Intel Pentium.

address organization of main memory. The CPU contains six segment registers that store pointers to the segments in current use. For example, the segment registers CS and SS address a code (program) and stack segment, respectively. These registers are typically used in a manner that is transparent to the application programmer. For instance, when an instruction fetch is initiated, a 32-bit (effective) address obtained from the program counter PC is appended to a 14-bit segment index Ls obtained from the CS register to form a 46-bit virtual address L. As Figure 6.31 indicates, Ls serves as a relative address for an 8-byte segment descriptor stored in one of many possible seg-ment tables. The descriptor specifies the base address and length of the segment 5 referred to by Ls. It also indicates 5"s type and access rights, and whether 5 is present in main memory. The linear address TV is constructed by adding the base address obtained from the segment descriptor to the program-derived effective address.

Figure 6.31 also shows how the paging unit processes the linear address N to pro-duce a real address AR, assuming a page size of 4 KB. A two-step table lookup process is employed to obtain AR from N. The right-most 12 bits of N form a displacement within the page containing the desired information; they therefore supply the right-most 12 bits of AR. The remaining 20 bits of N yield a real page address as follows. First a page directory is accessed, which contains entries defining up to 1024 page tables. The left-most 10 bits Nd of N form the relative address of a 32-bit entry E in the page table directory. E contains the 20-bit base address of a page table T, as well as such standard information as a presence bit, a change bit (indicating whether or not the page has been written into), and some protection information. Using the base address derived from £, the page table T is then accessed, and the word £", which is stored at the relative address pointed to by the 10-bit field N of the linear address N. is fetched. E', which has the same format as £, provides the 20-bit page address (page frame num-ber) of the desired real address AR.

442 Page size. The page size Sp has a big impact on both storage utilization and

section 6 2 the effective memory data-transfer rate. Consider first the influence of Sp on the

Memory Systems space-utilization factor u defined earlier. If Sp is too large, excessive internal frag-

mentation results; if it is too small, the page tables become very large and tend to reduce space utilization. A good value of Sp should achieve a balance between these two extremes. Let Ss denote the average segment size in words. If Ss » S„, the last page assigned to a segment should contains about 5/2 words. The size of the page table associated with each segment is approximately S/Sp words, assum-ing each entry in the table is a word. Hence the memory space overhead associated with each segment is

2+Sp

The space utilization u is

" = FTT = ~2 — (6-10)

Ss + S S2p + 2SS(\+Sp)

OPT

The optimum page size Sp can be defined as the value of Sp that maximizes u or, equivalently, that minimizes 5. Differentiating S with respect to 5 , we obtain

dS_ _ 15,

JSp'i'sl

S is a minimum when dS/dS - 0, from which it follows that

OFT /

S„ = JlSs (6.11)

p

The optimum space utilization is

i + V27s;

Figure 6.32 shows the space utilization u defined by Equation (6.10) plotted against Ss for some representative values of S .

The influence of page size on hit ratio is complex, depending on the program reference stream and the amount of space available in Mv Let the virtual address space of a program be a sequence of numbers Aq, Alt..., AL_X. Let A, be the virtual address referenced at some point in time, and let Ai+d be the next address gener-ated, where d is the "distance" between A, and A, + d. For example, if both addresses point to instructions, A{ + d points to the (d + l)st instruction either preceding or fol-lowing the instruction whose virtual address is A,. Let Sp be the page size and sup-pose that an efficient replacement policy such as LRU is being used. The probability of Ai + d being in M, is high if one of the following conditions is satis-fied:

• d is small compared with Sp, so A, and A, + d are in the same page P. The probabil-ity of these addresses both being in P increases with the page size.

• d is large relative to S but A, + d is associated with a set of words that are fre-quently referenced. A, + d is therefore likely to be in a page P' * P, which is also

3 0.8

2 3 4

Segment size Ss, in K-words

Figure 6.32

Influence of page size Sp5 and segment size Ss onspace utilization u.

in M,. This likelihood tends to increase with the number of pages stored in M,; ittherefore tends to decrease with the size of Sp.

Thus H is influenced by two opposing forces as S is varied. When S is small,H increases with Sp. However, when Sp exceeds a certain value, H begins todecrease. Figure 6.33 shows some typical curves relating H and S for variousmain-memory capacities. Simulation studies indicate that in large systems, the val-ues of 5 yielding the maximum hit ratios can be greater than the "optimum" pagesize given by Equation (6.11). Since high H is important in achieving small tA (dueto the relatively slow rates at which page swapping takes place), values of Sp thatmaximize H are preferred. The first computer with a paging system (the Universityof Manchester's Atlas computer) had a 512-word page, while the Pentium dis-cussed in Example 6.5 supports page sizes of 4 KB and 4 MB.

5.6.1 Memory Allocation

As we have seen, the various levels of a memory system are divided into sets ofcontiguous locations, variously called regions, segments, or pages, which store

Page size 5_

Figure 6.33

Influence of page size Spon hit ratio H.

444 blocks of data. Blocks are swapped automatically among the levels in order to min-

sfction imize the access time seen by the processor. Swapping generally occurs in

M s t m response to processor requests (demand swapping). However, to avoid making a

processor wait while a requested item is being moved to the fastest level of mem-ory Mj, some kind of anticipatory swapping must be implemented, which impliestransferring blocks to M, in anticipation that they will be required soon. Goodshort-range prediction of access-request patterns is possible because of locality ofreference.

The placement of blocks of information in a memory system is called memoryallocation and is the topic of this section. The method of selecting the part of M, inwhich an incoming block K is to be placed is the replacement policy. Simplereplacement policies assign K to M, only when an unoccupied or inactive region ofsufficient size is available. More aggressive policies preempt occupied blocks tomake room for K. In general, successful memory allocation methods result in a highhit ratio and a low average access time. If the hit ratio is low, an excessive amountof swapping between memory levels occurs, a phenomenon known as thrashing.Good memory allocation also minimizes the amount of unused or underused spacein M,.

The information needed for allocation within a two-level hierarchy (M,,M-,)—unless otherwise stated, we will assume the main-secondary-memory hierarchy—can be held in a memory map that contains the following information:

• Occupied space list for Mx. Each entry of this list specifies a block name, the(base) address of the region it occupies, and, if variable, the block size. In sys-tems using preemptive allocation, additional information is associated with eachblock to determine when and how it can be preempted.

• Available space list for Mx. Each entry of this list specifies the address of anunoccupied region and, if necessary, its size.

• Directory for M2. This list specifies the unit(s) that contain the directories for allthe blocks associated with the current programs. These directories, in turn, definethe regions of the M2 space to which each block is assigned.

When a block is transferred from M2 to Mx, the memory management systemmakes an appropriate entry in the occupied space list. When the block is no longerrequired in Mx, it is deallocated and the region it occupies is transferred from theoccupied space list to the available space list. A block is deallocated when a pro-gram using it terminates execution or when the block is replaced to make room forone with higher priority.

Many preemptive and nonpreemptive algorithms have been developed fordynamic memory allocation. Accurate analysis of their performance is difficult; asa result, simulation is the most widely used evaluation tool. The performance of anallocation algorithm can be estimated by the various parameters introduced in sec-tion 6.2.1, such as the hit ratio H, the access time tA, and the space utilization u.

Nonpreemptive allocation. Suppose a block K: of nl words is to be transferredfrom M2 to M,. If none of the blocks already occupying M, can be preempted(overwritten or moved) by K{, then it is necessary to find or create an "available"region of «, or more words to accommodate Kt This process is termed nonpreemp-tive allocation. The problem is more easily solved in a paging system where all

blocks (pages) have size 5. words and Mj is divided into fixed S^-word regions(page frames). The memory map (page table) is searched for an available pageframe; if one is found, it is assigned to the incoming block Kt. This easy allocationmethod is the principal reason for the widespread use of paging. If memory spaceis divisible into regions of variable length, however, then it becomes more difficultto allocate incoming blocks efficiently.

Two widely used algorithms for nonpreemptive allocation of variable-sizedblocks—unpaged segments, for example—are first fit and best fit. The first-fitmethod scans the memory map sequentially until an available region Rj of ni ormore words is found, where ni is the size of the incoming block Kt. It then allocatesK{ to Rj. The best-fit approach requires searching the memory map completely andassigning Kt to a region of rij > n, words such that rij - n, is minimized.

Suppose, for example, that at some point in time M, stores three blocks, as inFigure 6.34a. There are three available (shaded) regions, and the available spacelist has the form:

Region address

Size (words)

0

300

800

50

400

200

Further, suppose that two new blocks K4 and K5 whose sizes are 100 and 250words, respectively, are to be assigned to Mj. Figures 6.34b and c show the resultsobtained using the first-fit and best-fit methods, respectively, when the memoryscan begins at address 0.

The first-fit algorithm has the advantage of needing less time to execute thanthe best-fit approach. If the best-fitting available region can be found by scanning k

o ➤

300 ➤

800 ➤*■

Block K

Block K,

Block K

0 ➤*■

650 ➤

800 ➤*■

Block K

Block KA

Block K5

Block K7

Block K

0—*•

550—*•

900—*>

Block K

Block K<

Block K,

Block tf4

Block K,

(a)

(b)

{c)

Figure 6.34

(a) Initial memory state; (b) allocation of K4 and K5 by first fit; (c) allocation of KA and K5by best fit.

SECTION 6.2Memory Systems

entries of the available space list, the first fit can always be found by scanning k orfewer entries. The relative efficiency of the two techniques has long been a subjectof debate, since both have been implemented with satisfactory results [Knuth 1973;Shore 1975]. The performance obtained in a particular environment depends on thedistribution of the block sizes to be allocated. Simulation studies suggest that, inpractice, first fit tends to outperform best fit.

Preemptive allocation. Nonpreemptive allocation cannot make efficient useof memory in all situations. Memory overflow, that is, rejection of a memory allo-cation request due to insufficient space, can be expected to occur with M, only par-tially full. Much more efficient use of the available memory space is possible if theoccupied space can be reallocated to make room for incoming blocks. Reallocationmay be done in two ways:

• The blocks already in M[can be relocated within M, to create a gap large enoughfor the incoming block.

• One or more occupied regions can be made available by deallocating the blocksthey contain. This method requires a rule—a replacement policy—for selectingblocks to be deallocated and replaced.

Deallocation requires that a distinction be made between "dirty" blocks, whichhave been modified since being loaded into M,, and "clean" blocks, which havenot been modified. Blocks of instructions remain clean, whereas blocks of data canbecome dirty. To replace a clean block, the memory management system can sim-ply overwrite it with the new block and update its entry in the memory map. Beforea dirty block is overwritten, it should be copied to M2, which involves a slow blocktransfer.

Relocation of the blocks already occupying M[can be done by a method calledcompaction, which is illustrated in Figure 6.35. The blocks currently in memoryare compressed into a single contiguous group at one end of the memory. This cre-ates an available region of maximum size. Once the memory is compacted, incom-ing blocks are assigned to contiguous regions at the unoccupied end. The memory

Block A-

Block K2

Block AT

Block K

Block K-,

Block AT,

(a)

(b)

Figure 6.35

Memory allocation (a) before and (b) aftercompaction.

is viewed as having a single available region; new available regions due to freedblocks are ignored. When the gap at the end of the memory is eventually filled,compaction is carried out again. The advantages of this scheme are its simplicityand the fact that it eliminates the task of selecting an available region; its drawbackis the long compaction time.

Replacement policies. The second major approach to preemptive allocationinvolves preempting a region R occupied by block K and allocating it to an incom-ing block K'. The criteria for selecting K as the block to be replaced constitute thereplacement policy. The main goal in choosing a replacement policy is to maxi-mize the hit ratio of the faster memory M, or, equivalently. minimize the numberof times a referenced block is not in Mh a condition called a memory fault or miss.

It is generally accepted that the hit ratio tends to a maximum if the time inter-vals between successive memory faults are maximized. An optimal replacementstrategy would therefore at time ti determine the time t. > t, at which the next refer-ence to block K is to occur; the K to be replaced is the one for which t- - r, has themaximum value tK. This ideal strategy has been called OPT [Mattson et al. 1970;Stone 1993]. In principle. OPT can be implemented by making two passes throughthe executing program. The first is a simulation run to determine the sequence 5Bof distinct virtual block addresses generated by the program; the sequence is calledthe block address trace. The values of tK at each point in time can be computedfrom 5B and used to construct the optimal sequence 5b PT of blocks to be replaced.The second run is the execution run, which uses 5b to specify the blocks to bereplaced. OPT is not a practical replacement policy because of the cost of the sim-ulation runs and the fact that 5B can be extremely long, making 5bPT too expen-sive to compute. A practical replacement policy attempts to estimate tK usingstatistics it gathers on the past references to all blocks currently in Mj.

Two useful replacement policies are first-in first-out (FIFO) and least recently-used (LRU). FIFO selects for replacement the block least recently loaded into M,.FIFO has

the advantage that it is very easy to implement. A loading-sequence num-ber is associated with each block in the occupied space list. Each time a block istransferred to or from M,, the loading-sequence numbers are updated. By inspect-ing these numbers, the memory manager can easily determine the oldest (first-in)block. FIFO has the defect, however, that a frequently used block, for instance, onecontaining a program loop, may be replaced simply because it is the oldest block.

The LRU policy selects for replacement the block that was least recentlyaccessed by the processor. This policy is based on the reasonable assumption thatthe least recently used block is the one least likely to be referenced in the future.LRU avoids the replacement of old but frequently used blocks, as occurs withFIFO. LRU is slightly more difficult to implement than FIFO, however, since thememory manager must maintain data on the times of references to all blocks inmain memory. LRU is implemented by associating a hardware or software counter,called an age register, with every block in M,. Whenever a block is referenced, itsage register is set to a predetermined positive number. At fixed intervals of time,the age registers of all the blocks are decremented by a fixed amount. The leastrecently used block at any time is the one whose age register contains the smallestnumber.

The performance of a replacement policy in a given memory organization canbe analyzed using the block address stream generated by a set of representative

computations. Let N\ and N*2 denote the number of references to Mj and M2,respectively, in the block address stream. The block hit ratio H* is defined by

N,

H =

which is analogous to the (word) hit ratio H defined by Equation (6.5). Let n*denote the average number of consecutive word address references within eachblock. H can be estimated from H* using the following relation:

H = \-

\-H

In a paging system, H* is the page-hit ratio. 1 - H*, the page-miss ratio, is alsocalled the page fault probability.

EXAMPLE 6.6 COMPARISON OF SEVERAL REPLACEMENT POLICIES. Consider

a paging system in which M, has a capacity of three pages. The execution of a programQ requires reference to five distinct pages Pt, where i = 1, 2, 3, 4, 5, and / is the pageaddress. The page address stream formed by executing Q is

232152453252

which means that the first page referenced is P2, the second is P3, and so on. Figure6.36 shows the manner in which the pages are assigned to M, using FIFO, LRU, andthe ideal OPT replacement policies. The next block to be selected for replacement ismarked by an asterisk in the FIFO and LRU cases. It will be observed that LRU recog-nizes that P2 and P5 are referenced more frequently than other pages, whereas FIFO

Time   1 2 3 4 5 6 7 8 9 10 11 12

Address trace 2 3 2 1 5 2 4 5 3 2   5   2

FIFO
LRU
OPT
Hit
Hit

2*   3   2*3 2*31   5   52   5*24 5*24
                            3*1   1*

Hit Hit

2*   2*3 2   2   2   2   2*54
                3*1 2*51 5 5   5*4
                          1*

Hit Hit Hit

2   23   2   231 235 23 43   435
                 3   5   5

Hit

Figure 6.36

Action of three replacement policies on a common address trace.

2*

Hit

Hit

3   3 3*

2*   5 5

4   4* 2

Hit

3   3* 3*

5*   5 5

2   2 2

Hit Hit

2   2 2

3   3 3

5   5 5

Hit

does not. Thus FIFO replaces P2 twice, but LRU does so only once. The highest page-hit ratio is achieved by OPT, the lowest by FIFO. The page-hit ratio of LRU is quiteclose to that of OPT, a property that seems to hold generally.

Stack replacement policies. As discussed in section 6.2.1, the cost and perfor-mance of a memory hierarchy can be measured by average cost per bit c and averageaccess time tA. Equations (6.4) and (6.7) repeated here are convenient expressionsfor c and t A:

c =

t 1 Ji + C-y-J'j

5, + S-,

$tA= tA[+(l-H)tB$

The quantities c, tA, and tB are determined primarily by the memory technologiesused for M[and M2. Once these technologies have been chosen, the hit ratio H canbe computed for various possible system configurations. The major variables onwhich H depends are

• The address streams encountered.

• The average block size.

• The capacity of M,.

• The replacement policy.

Simulation is perhaps the most practical technique used for evaluating differ-ent memory system designs. H is determined for representative address traces,memory technologies, block sizes, memory capacities, and replacement policies.Figure 6.36 shows a sample point in this simulation process. Here the addresstrace, block size, and memory capacity are fixed, and three different replacementstrategies are being tested.

Due to the many alternatives that exist, the amount of simulation required tooptimize the design of a multilevel memory system can be huge. A number of ana-lytic models for optimizing memory design have been proposed. Notable amongthese is a technique called stack processing, which is applicable to paging systemsthat use a class of replacement algorithms called stack algorithms [Stone 1993].Let AT be any page address trace of length L to be processed using a replacementpolicy RP. Let t denote the point in time when the first t pages of A T have been pro-cessed. Let n be a variable denoting the page capacity of M,. Bt{n) denotes the setof pages in M, at time r, and L, denotes the number of distinct pages that have beenencountered at time /. Policy RP is called a stack algorithm if it has the followinginclusion property:

Bt(n)cBt(n+ 1) if n<Lt

Bt(n) = Bt(n+ 1) if n>L,

LRU retains in M, the n most recently used pages. Since these are alwaysincluded in the n + 1 most recently used pages, it can be seen right away that LRUis a stack algorithm. Some other replacement policies are also of this type. FIFO isa notable exception, however. Consider the following page address stream:

449

CHAPTER 6

MemoryOrganization

1234125 12345

450

SECTION 6.2Memory Systems

TimeAddress trace

w = 3

• j ■•:

2

* i *

2

* 4

2 2*

3 3*

* 1*2 2? 3

4

4 4*

3 2

1* 1*

2 2

3 3

4 4

n = 4

Hit Hit

Figure 6.37

FIFO replacement with two different memory capacities.

5   5

1* 1*

2   2

Hit

5   5

2* 1

3   3*

4   4

5   5

1* 3

2   2*

Hit

51 5*1

2   2

4* 3

5-* 5*

3   3

4   4

Hit

4   4

I* 5

2   2*

3   3

Figure 6.37 shows how this address stream is processed using FIFO and memorycapacities of three and four pages. It can be seen that at various points of time theconditions for the inclusion property are not satisfied. For example, when t = l,n =5, fl7(3) = {1, 2, 5}, andfl7(4) = {2, 3, 4, 5}. Hence fl7(3) 2 fl7(4), so FIFO is nota stack algorithm.

The usefulness of stack replacement algorithms lies in the fact that the hitratios for different Mx capacities can be determined by processing the addressstream once and by representing M1 by a list or "stack." The stack 5, at time t is anordered set of L, distinct pages 5,(1), 5,(2),..., St(Lt), with 5,(1) referred to as the topof the stack at time t. The inclusion property of stack algorithms implies that thestack can always be generated so that

B,(n) = {5,(1), 5,(2),..., 5,(n)} for n<LtBt(n) = {5,(1), 5,(2),..., St(Lt)} for n > L,

In other words, the behavior of a system in which Mj has capacity n is determinedby the top n entries of the stack. By scanning 5,, we can easily see whether a hitoccurs for all possible values of n. This type of analysis permits the simultaneousdetermination of hit ratios for various capacities of M1.

The procedures for updating the stack depend on the particular stack algorithmRP being used. There may be little resemblance between the order of the elementsin 5, and 5,+ 1; the stack should not be confused with simple LIFO stacks. The fol-lowing example describes the stack-updating process for LRU replacement.

EXAMPLE 6.7 DETERMINATION OF HIT RATIOS WITH LRU REPLACE-MENT. Let S, = {5,(1), 5,(2),..., St(k)} denote the stack contents at time t. Stack pro-cessing requires placing the most recently used page addresses in the top of stack sothat the least recently used page gets pushed to the bottom. More formally, let x be thenew page reference at time t. If x i Sr x is pushed into the stack so that x becomes5r+1(l), 5,(1) becomes 5,+ 1(2), and so on. If xe St, x is removed from 5, and then

Timer 1 2 3 4 5 6 7 8 9 10 11 12 451

Address trace 232152453252   CHAPTER 6

Memory

5,(1)   2   3 2   1 5 2   4 5   ?   2   5   2   Organization

5,(2)   2 3   2 1 5   2 4   5   J   2   5

5,(3)	3 2 1	5 2	4	5	3	3
5,(4)	3 3	1 1	2	4	4	4
5,(5)	3 3	1	1	1	1

72=1

n = 2 Hit Hit

7i = 3 Hit Hit Hit Hit Hit

n = 4 Hit Hit Hit Hit Hit Hit

71 = 5 Hit Hit Hit Hit Hit Hit Hi!

Figure 6.38

Stack processing of a page address trace using LRU.

pushed into the top of the stack to form S,+1. Figure 6.38 illustrates this process for the address stream used in Figure 6.36. To determine whether a hit occurs at time t for memory page capacity n, it is necessary only to check whether the new page reference x is one of the top n entries of 5,; if it is, a hit occurs. The hit occurrences for all values of n < 5 also appear in Figure 6.38. The values for the various page-hit ratios H* are as follows:

n= 1   2   3   4   5   >5

H* = 0.00 0.17 0.42 0.50 0.58 0.58

It follows from the inclusion property of stack replacement algorithms that the hit ratio increases with the available capacity n. If the next page address x is in B,(n), it must also be in Bt(n + 1) because Bt{n} cz Bt{n + 1}. Hence if a hit occurs with capacity n, a hit also occurs when the capacity is increased to n + 1. It might be expected that this inclusion property holds for all replacement policies, but it does not. The example in Figure 6.37 shows that increasing n from three to four pages in a system with FIFO replacement actually reduces the page-hit ratio in this case from 0.25 to 0.17. This phenomenon seems to be relatively rare, not occurring for most address traces.

Other replacement policies. A few other replacement algorithms are used in multilevel memories. As discussed later (in section 6.3), caches often have a low-cost replacement policy called direct mapping, where each incoming block from M2 is assigned to a fixed region of M, determined by the block's low-order address bits.

Another interesting and low-cost technique is random replacement, where the replaced block is chosen in an apparently random fashion. An example is found in the TLB in the MIPS R2/3000 of Example 6.4—recall that although pan of the MMU, the TLB is itself a special type of cache. The block to be replaced (a page

452 entry in the case of the R2/3000's TLB) is selected by a fast process that approxi-

sfction mates truly random selection and, unlike LRU, does not use memory-reference

data. The R2/3000's MMU contains a 6-bit register called RANDOM, which is dec-remented in each CPU clock cycle. RANDOM therefore continually loops through the numbers 8 through 63, each of which can act as an entry point or index to the 64-entry TLB. (RANDOM skips the numbers zero (hough seven so that the first eight entries of the TLB can be reserved for critical parts of the operating system.) When a TLB miss exception is being serviced, the MMU replaces the TLB entry whose index is the current value of RANDOM. Hence the randomness of the times at which TLB miss exceptions occur determines the randomness of this register's contents. There is no delay and very little hardware overhead associated with this replacement policy. Although less efficient than LRU, this policy appears to work quite well in practice.

•i.3CACHES

The term cache refers to a fast intermediate memory within a larger memory system [Smith 1982; Handy 1993]. Although caches appeared as early as 1968 in the IBM System/360 Model 91, they did not come into wide use until the appearance of low-cost, high-density RAM and microprocessor ICs in the 1980s. Caches directly address the von Neumann bottleneck by providing the CPU with fast, single-cycle access to its external memory. They also provide an efficient way to place a small portion of memory on the same chip as a microprocessor. If an additional off-chip cache is used that employs, say, fast SRAM technology—and the continuing dispar-ity between processor and DRAM speeds makes that desirable—a two-level cache organization results (refer to Figure 6.21c).

A cache serves as a buffer between a CPU and its main memory; in this section we focus on caches used in this way. However, caches appear as buffer memories in several other contexts. We saw in section 6.2 that the translation look-aside buff-ers (TLBs) used within a memory management system are specialized caches that permit very fast translation of memory addresses. Data buffers built into high-speed secondary memory devices such as hard disk drives are also called caches.

6.3.1 Main Features

The cache and main memory form a two-level subhierarchy (M,,M2) that differs in important ways from the main-secondary system (M2,M3); Figure 6.39 summa-rizes these differences. Because it is higher in the memory hierarchy, the pair(M[,M2) functions at much higher speed than (M2, M3). The access time ratio tAJtA is around 5/1, while tA/tA_> is about 1000/1. These speed differences require (M,,M2) to be managed by high-speed hardware circuits rather than by software routines; (M2,M3), on the other hand, is controlled mainly by the operating system. Thus while the (M2,M3) hierarchy is transparent to the application programmer but visible to the system programmer, (Mj,M2) is largely transparent to both. Another difference lies in the block size used. Communication within (M,,M2) is by pages,

| Two-level hierarchy | Cache-main memory | Main-secondary memory | 453 |
| (M,--i,M,) | (Mi, Mj) | (M2,M3) | |

CHAPTER 6

Typical access time ratios rA/rAi._,	5/1	1000/1	MemoryOrganization
Memory management system	Mainly implemented by hardware	Mainly implemented by software	
Typical page size	8B	4KB	

Access of processor to second level M, | Processor has direct access to M2 | All access to M3 is via M2

Figure 6.39

Major differences between cache-main and main-secondary-memory hierarchies.

but the page size is much smaller than that used in (M2,M3). Finally, we note that the CPU generally has direct access to both Mj and M2, whereas it does not have direct access to M3.

Cache organization. Figure 6.40 shows the principal components of a cache. Memory words are stored in a cache data memory and are grouped into small pages called cache blocks or lines. The contents of the cache's data memory are thus copies of a set of main-memory blocks. Each cache block is marked with its block address, referred to as a tag, so the cache knows to what part of the memory space the block belongs. The collection of tag addresses currently assigned to the cache, which can be

noncontinuous, is stored in a special memory, the cache tag memory or directory. For example, if block Bj containing data entries Dj is assigned to M,, then 5 is in the cache's tag memory and Dj is in the cache's data memory.

Obviously for a cache to improve the performance of a computer, the time required to check tag addresses and access the cache's data memory must be less than the time required to access main memory. Thus if main memory is imple-mented with a DRAM technology having an access time t_{A1} = 50 ns, the cache's data memory might be implemented with an SRAM technology having an access time of t_A =10 ns. A basic issue in cache design, which we examine in section 6.3.2, is how to make the matching of tag addresses extremely fast.

Two general ways of introducing a cache into a computer appear in Figure 6.41. In the look-aside design of Figure 6.41a, the cache and the main memory are directly connected to the system bus. In this design the CPU initiates a memory-access by placing a (real) address A, on the memory address bus at the start of a

Cache M,   Hit

Cache

data

memory

Cache tag

memory

(directory)

.   i

Address Control Data

Figure 6.40

Basic structure of a cache.

454

SECTION 6.3 Caches

System bus

Cache access

J

Main-memory access

t

Block replacement

Main

memory

M,

J

(a)

CPU

Cache A A access

Block replacement

Cache controller

Main-memory controller

Main-memory

System bus

r

Main

memory

M2

(*)

Figure 6.41

Two system organizations for caches: (a) look-aside and (b) look-through.

read (load) or write (store) cycle. The cache M, immediately compares A, to the tag addresses currently residing in its tag memory. If a match is found in M,, that is, a cache hit occurs, the access is completed by a read or write operation executed in the cache; main memory M2 is not involved. If no match with A, is found in the cache, that is, a cache miss occurs, then the desired access is completed by a read or write operation directed to M2. In response to a cache miss, a block (line) of data Bj that includes the target address A, is transferred from M2 to M,. This transfer is fast, taking advantage of the small block size and fast RAM access methods, such as page mode (section 6.1.2), which allow the cache block to be filled in a single short burst. The cache implements some replacement policy such as LRU to deter-mine where to place an incoming block. When necessary, the cache block replaced by Bj in M, is saved in M2. Note that cache misses, even though they are infrequent, result in block transfers between M, and M2 that tie up the system bus, making it unavailable for other uses like IO operations.

A faster, but more costly organization called a look-through cache appears inFigure 6.4lb. The CPU communicates with the cache via a separate (local) busthat is isolated from the main system bus. The system bus is available for use byother units, such as IO controllers, to communicate with main memory. Hencecache accesses and main-memory accesses not involving the CPU can proceedconcurrently. Unlike the look-aside case, with a look-through cache the CPUdoes not automatically send all memory requests to main memory; it does so onlyafter a cache miss. A look-through cache allows the local bus linking M, and M2to be wider than the system bus, thus speeding up cache-main-memory transfers.For example, if the system data bus is 32 bits wide and the cache block size is 128bits =16 bytes (a typical value), a 128-bit data bus might be provided to link M,and M2, which would allow a cache block to be replaced in as little as a singleclock cycle. The main disadvantage of the look-through design, besides its highercomplexity and cost, is that it takes longer for M2 to respond to the CPU when amiss occurs.

455

Cache operation. Figure 6.42 shows a small cache system that illustrates therelationship between the data stored in the cache Mt and the data stored in mainmemory M2. Here a cache block (line) size of 4 bytes is assumed. Each memoryaddress is 12 bits long, so the 10 high-order bits form the tag or block address, andthe 2 low-order bits define a displacement address within the block. When a blockis assigned to M,'s data memory, its tag is also placed in M,'s tag memory. Figure6.42 shows the contents of two blocks assigned to the cache data memory; note thelocations of the same blocks in main memory. To read the shaded word, its addressAj = 101111000110 is sent to Mj, which compares A,'s tag part to its stored tagsand finds a match (hit). The stored tag pinpoints the corresponding block in Mj's

Cache tagmemory

1011110001

101111001

Tagcomparison

Cache data memory11 10 01 00

F4

AB

X

FF

55

00

F0

1A

FF

Dataselection

Address in 1011110001 10

Figure 6.42

Cache execution of a read operation.

Data out FF

101111010001101111010000101111001111101111001101011100110110111100110010111100101110111100101010111001001101111001000101111000111101111

Main

C3

67

AB

55

PO

FF

00

87

B4

10

F4

FF

(X)

1A

C4

99

Main

memory memoryaddresses data

456

SECTION 6.3Caches

data memory, and the 2-bit displacement is used to output the target word to theCPU.

A cache write operation employs the same addressing technique. As shown inFigure 6.43, the tag part of the target address A, is again presented to M,, alongwith the data word to be stored. When a hit occurs, the new data, in this case, 88, is

c

stored at the location pointed to by A, in the data memory of M,, thereby overwrit-ing the old data FF. Now a new problem arises: The data in M, with address /4, dif-fers from the data in M2 with the same address. A temporary inconsistency of thissort is acceptable as long as no device—another processor, for instance—attemptsto read the old or stale data. Preventing the improper use of stale data is the cachecoherence or cache consistency problem. It is a basic design issue in multiproces-sors where several CPUs share access to the same main memory but each has itsown cache. This issue also arises in single-CPU systems when an 10 controller orprocessor is present that has direct access to main memory, independent of theCPU; see problem 6.38. We can minimize cache-related inconsistencies by imple-menting a policy that systematically updates the data in M2 in response to changesmade to the corresponding data in M,. (We will discuss some general solutions tothe cache coherence problem in Chapter 7.)

Most of the methods used by virtual memory systems to update secondarymemory can be adapted for use with the cache-main-memory subhierarchy. Eachcache block in the data memory of M₁ can have a change bit C attached to it, whichis set to 0 when the block is first placed in M₁. Any subsequent write operationaddressed to that block sets C to 1. When a block with C = 1 is replaced, its datacontents are then written back to main memory M2. This technique is referred to aswrite-back or copy-back. It has the disadvantage that M₁ and M2 can be tempo-rarily inconsistent, that is, have different data associated with the same physical

Cache tagmemory

Cache data memoryli 10 01 00

1011110001

1011110011

F4

AB

Tagcomparison

88

55

00

FO

1A

FF

Dataselection

7

Address in 1011110001 10

Data in 88

10111101000110111101000010111001111101111001101011100110110111100110010111001011101111001010101110010011011110010001011110001111101111

Mainmemoryaddresses

C3

6^

AB

55

F0

FF

00

87

B4

F4

FF

00

1A

C4

99

data

Figure 6.43

Cache execution of a write operation.

address. Difficulties arise if several processors with independent caches are sharingM, because their data can become inconsistent. The write-back technique alsocomplicates recovery from system failures.

Direct communication links between the CPU and main memory, which are notpresent in the virtual-memory case, permit some novel write policies for caches. Analternative to write-back is to transfer the data word to both M₁ and M2 during everymemory write cycle, even when the target address is already assigned to the cache.This policy, called write-through, is easy to implement, and it guarantees that M2never contains stale information. On the other hand, write-through results in morewrite cycles to M-, than write-back does. Since the time needed for each write is thenthe slower (write) access time of M2, system performance may suffer. However,only a small fraction, perhaps 1/10, of all memory accesses are writes. Some pro-cessors support both write-back and write-through so that a user can select the pol-icy that best suits a particular program's memory-access behavior.

457

6.3.2 Address Mapping

When a tag address is presented to the cache, it must be quickly compared to thestored tags to determine whether a matching tag is currently assigned to the cache.The obvious approach of scanning all the tags in sequence is unacceptably slow.The fastest technique for implementing tag comparison is associative or contentaddressing, which permits the input tag to be compared simultaneously to all tagsin the cache-tag memory. Pure associative memories are very expensive, however,so it is only feasible to use them in small caches and TLBs (see Example 6.4). Var-ious less costly techniques have been developed to solve this problem, some ofwhich make limited use of associative addressing.

Associative addressing. In an associative memory any stored item can beaccessed by using the contents of the item in question, generally some specified sub-field, as an address. Associative memories are also commonly known as content-addressable memories (CAMs). The subfield chosen to address the memory iscalled the key. Items stored in an associative memory can viewed as having the two-field format

KEY, DATA

where KEY is the stored address and DATA is the information to be accessed. Forexample, if a page table of the kind shown in Figure 6.29 is placed in an associativememory, the page address can be selected as the key, while the page frame, pres-ence bit, change bit, and access rights form the data. Such a memory can then beaccessed with a request such as: Read the page frame number corresponding topage address E. However, we could equally well choose the page frame as the key.which would permit queries such as: Write 1 in the presence-bit field of page frameD6C7F9.

An associative cache employs a tag, that is, a block address, as the key. At thestart of a memory access, the incoming tag is compared simultaneously to all thetags stored in the cache's tag memory. If a match (cache hit) occurs, a match-indicating signal triggers the cache to service the requested memory access. A

458

no-match signal identifies a cache miss, and the memory access requested is for-warded to main memory for service. A cache block containing the target address is then sent from main memory to the cache, and at the same time, a data word is sent to the CPU or transferred from the CPU to the cache, in response to the origi-nal access request.

Associative memory. Figure 6.44 shows the general structure of an associative memory. Each unit of stored information is a fixed-length word. Any subfield of the word can be chosen as the key. Here the desired key is specified by a mask reg-ister, whose contents identify the bit positions (which need not be adjacent) that define the key. The current key is compared simultaneously with all stored words; those that match the key output a match signal, which enters a select circuit, which enables the data field to be accessed. If several entries have the same key, then the select circuit determines which data field is to be read out. It can, for example, read out all matching entries in some predetermined order. Since all words in the mem-ory are required to compare their keys with the input key simultaneously, each needs its own match circuit. The match and select circuits make associative memo-ries much more complex and expensive than conventional memories. Although VLSI techniques have made associative memories economically feasible, cost con-siderations still limit them to applications in which a relatively small amount of information must be accessed very rapidly, such as address mapping for caches.

The logic circuit for a 1-bit associative memory cell appears in Figure 6.45 [Triebel and Chu 1982]. The cell comprises a D flip-flop for data storage, a match circuit (the EXCLUSIVE-NOR gate) for comparing the flip-flop's contents to an external data bit D, and circuits for reading from and writing into the cell. The results of a comparison appear on the match output M, where M = 1 denotes a match and M = 0 denotes no match. The cell is selected or addressed for both read and write operations by setting the select line S to 1. New data is written into the cell by setting the write enable line WE to 1, which in turn enables the D flip-flop's

Input

1   r   Match

Input register

          r

'

Mask register

                              Key
          i
                              r

1

Storagecellarray Selectlogic

Select

1   i

Output register   Figure 6.44

                   Structure of an associative (contentaddressable) memory.

Ou   1
          put

clock input CK. The stored data is read out via the Q line. The mask control line MK is activated {MK = 1) to force the match line M to 0 independently of the data stored in the D flip-flop; MK also disables the input circuits of the flip-flop by forc-ing CK to 0. A cell like that of Figure 6.45 can be realized with about 10 transis-tors—far more than the single transistor required for a dynamic RAM cell (refer to Figure 6.9b). This high hardware cost is the main reason that large associative memories are rarely used outside caches.

Associative cells of the preceding type can be combined into word-organized associative memory arrays. Figure 6.46 shows a 16-bit associative memory that stores four words (columns) of 4 bits each. The words are individually addressable via their S lines. All words share a common set of data and mask lines for each bit position. Consequently, an external data bit D{ can be compared simultaneously to the z'th stored bit of every word in the memory. The output lines of the cells are designed so that they can be connected to form wired OR or AND gates, as indi-cated in the figure.

A small associative cache is found in Data General Corp.'s ECLIPSE, a 16-bit computer from the 1970s. This computer has a modular memory design in which each 8K-word main-memory module M2 is paired with a cache Mj that stores six-teen 16-bit words forming four 4-word blocks. M2 is constructed from MOS RAM chips with a 700 ns cycle time, while the cache M, uses bipolar RAMs with a cycle time of 200 ns. The memory (tag) addresses of the blocks stored in the cache are placed in an associative memory CAM. When the CPU generates a memory address A, it is sent to the CAM, which compares it to all tags currently in the cache. If the CAM indicates a match, Mx responds to the memory request directly by either reading or writing the corresponding data M(A). If A is not currently assigned to Mh then A is processed by the main memory M2, which responds to the original CPU request by executing a 700 ns read or write cycle. At the same time, M2 sends a four-word block containing M(A) to Mp which uses the new block to replace the least recently used cache block. The cache's LRU block replacement policy is implemented by special hardware that constantly monitors cache usage.

459

CHAPTER 6

Memory

Organization

Mask MK

Figure 6.45

Associative memory cell: (a) logic circuit and (b) symbol.

Outputdata Q

Match M

(b)

Direct mapping. An alternative, and simpler, address-mapping technique for caches is known as direct mapping. Let M_1 be divided into $s_1 = 2^s$ regions $M_1(0), M_1(l),..., M_1(s_1 - 1)$ called sets, each of which stores a block of n consecutive words. Main memory M2 is similarly divided into one-block regions $M2(0), M2(l),..., M2(s_2 - 1)$. With direct mapping, each block $M2(i)$ in M2 is mapped into

WE

Word 0

Word\

Word!

Word 3

Figure 6.46

A 4 x 4-bit associative memory array.

$M_1(0) M_1(l)$

$M2(0) M2(l) M2(2) M2(3)$ Main   memory M2
$M2(62) M2(63)$

Cache M[

Figure 6.47

Direct-mapped cache with block capacity of two.

one specific set $M_j(y')$ in M_1. The set address j is determined from i by the rule

$j = i \pmod{s_1}$

For example, if $s_1 = 2$ as in Figure 6.47, every even-address (unshaded) block in M2 is mapped into $M_1(0)$ and every odd-address (shaded) block in M2 is mapped into $M_1(l)$.

The hardware needed to implement direct mapping is fairly simple. The low-order s bits of each block address A form a set address that identifies the unique cache set that can store the block in question. The remaining t high-order bits of A now constitute the tag, and only these bits need be stored in the cache's tag memory. Consequently, the cache tag memory can be an ordinary RAM that is addressed by the 5-bit set-address part of an incoming memory address A. If there are 2^d words per set, then the low-order d bits of A form the displacement address of the word in question within its block. Thus an incoming address has three parts: a t-bit tag, an 5-bit set address, and a d-bit displacement.

The main drawback of direct mapping is that the cache's hit ratio drops sharply if two or more frequently used blocks happen to map onto the same region in the cache. This possibility is minimized by the fact that such blocks are relatively far apart in the memory-address space. For example, if $s_1 = 2^6 = 64$, then only the blocks with addresses i, i + 64, i + 128, i + 192,... can be mapped into the same cache set $M_1(j(i))$.

EXAMPLE 6.8 DESIGN OF A DIRECT-MAPPED CACHE [INTEGRATED

device technology 1994]. In this example we will use off-the-shelf ICs to design an add-on direct-mapped cache memory for a high-end microprocessor, such as the PowerPC. If the CPU has a built-in (level 1) cache, as is frequently the case, this design applies to a level 2 (secondary) cache. The CPU is linked to a byte-addressable external memory via a 32-bit address bus and a 64-bit bidirectional data bus using the the look-aside design style of Figure 6.41a. The desired cache capacity is 256 KB, and the cache block (set) size is assumed to be 32 bytes (32 B). Hence the cache must store $8K = 2^{13}$ blocks, implying that we need a cache tag memory of capacity 8K x r bits to store tags of length i. We also need a cache data memory of capacity 32K x 64 bits, where the 64-bit word size is determined by the system data bus. As shown in Figure 6.48, a 32-bit address generated by the CPU contains a 5-bit displacement to address a

Set address Displacement

Memory address

I?

n

CPU

Address bus

~~\ S

14,' Tag 13

Setaddress

15 /

8KX14

cache tag

RAM

(2X71B74)

D

MATCH

Data

word address

32KX64cache data

RAM(8X71256)

D

Memory

read/writecontrol logic

64 /

Data bus

Mainmemory

Figure 6.48

A 256KB direct-mapped cache for a microprocessor.

byte within a 32B block and a 13-bit set address to address the 8K blocks in the datamemory. Hence the remaining t = 32 — (13 + 5) = 14 high-order address bits form thetag. Since the cache's data RAM is accessed one 8-byte word at a time, it requires a 15-bit address consisting of the 13-bit set address plus 2 bits (the two high-order bits of thedisplacement) to select one-quarter of the current set.

The components selected for this design are the Integrated Device Technology(IDT) 71256, a 32K x 8-bit SRAM chip, which has an access and cycle time of 12 ns,and the IDT 71B74 chip, which is called a cache-tag RAM. The 71B74 contains ahigh-speed 64Kb memory organized as an 8K x 8-bit RAM. The cache-tag RAM isdistinguished from an ordinary SRAM by the fact that it has a built-in 8-bit comparatorto compare the addressed data (a stored tag) to a word placed on the 71B74's input databus. A MATCH output signal is set to 1 if the stored and applied data words match, andto 0 otherwise; matching can be done by the 71B74 in 8 ns. The MATCH signal is sup-plied to a small control circuit that then issues the memory access control signals (WE,CS, etc.) either to the cache data RAM (MATCH = 1) or to main memory (MATCH =0). To accommodate 14-bit tags, we need two 71B74s. We also need eight 71256s tostore the cached data. The final design of this cache unit appears in Figure 6.48.

Set-associative addressing. A more general address mapping method forcaches, called set associative, includes pure associative and direct mapping as spe-cial cases. As in direct mapping, blocks in main memory M2 are grouped intoequivalence classes determined by their addresses. M2(0 and M20') are in the sameequivalence class E if i = / (modulo $,). The cache is divided into sl multiblock

Main memory M2

Cache M,

SetM,(0) I

r

SetM,(l) <

M2(0)

M2n;

M2(2)M2(3)

M2(62)M2(63)

Figure 6.49

Cache with two-way set-associative addressing.

463

CHAPTER 6

Memory

Organization

regions M,(0), M,(l),…, Ml(sl - 1) called sets, each of which accommodates k =2h blocks. A block M20) in M2 is mapped into the set M,(/2), satisfying the condi-tion i = h (modulo Sj).

Each set My(h) in the cache is effectively a small associative memory, soaddress mapping within each set is associative. This k-wdy set-associative mappingpermits up to k members of the same equivalence class E to be stored in the cachesimultaneously, which is not possible with direct mapping. Figure 6.49 illustratesset-associative mapping with cache size 5, = 2 sets and set size k = 2. This mappingis therefore two-way set-associative and allows every shaded (unshaded) page inM2 to be mapped into either of the two shaded (unshaded) page frames in M,. Set-associative mapping reduces to direct mapping when k = 1; it reduces to fully asso-ciative mapping when s, = 1, implying that k equals the block capacity of thecache. Intermediate values of k lead to address-mapping methods requiring anintermediate amount of associative hardware. Only small values of k, such as k = 2or 4, are used in practice, which makes it feasible to use low-cost RAMs, ratherthan special associative memories like that of Figure 6.46, to store the tags, as thenext example illustrates.

EXAMPLE 6.9 DESIGN OF A TWO-WAY SET-ASSOCIATIVE CACHE. We con-sider an 8KB cache with two-way set-associative addressing, which is intended for a32-bit processor. A single 8KB two-way set-associative cache is used by the VAX-11/780, an influential minicomputer introduced by Digital Equipment Corp. in 1978[Clark 1983]. The on-chip I- and D-caches of the PowerPC model 603 introduced in1993 are also of the 8KB two-way set-associative type [Burgess et al. 1994; Heath1994]. The 11/780's cache block (line) contains 8 bytes, whereas the PowerPC 603'scaches have 32B blocks. We use the smaller 8B block size here, as in Example 6.8.

The organization of the cache appears in Figure 6.50. The 32-bit address A is inter-preted as follows. The low-order 3-bit displacement identifies a byte within an 8Bcache block. There are 2^9 = 512 sets, each containing two 8B blocks, so the next 9 bitsof A form the set address. The remaining 20 bits of A constitute the tag. (The number oftag bits needed depends on the size of the real address space actually used; we a&sumethe maximum size.) An incoming tag Atag that matches a stored tag can be associated

464

SECTION 6.3Caches

with either block in the matching set M,(i). The tag memory is therefore implementedby two 512 x 20-bit RAMs T0 and T,, each of which stores the tag for one block fromevery set M,(/). In addition, two 512 x 64-bit RAMs D0 and Dx form the cache's datamemory. One of the 64-bit data blocks of M,(0 is stored at address i in D0 (tagged byT0), while the other is at the same address i in Dj (tagged by T,). Consequently, the set-address field i of A is used as the address to access both the tag and the data memories.At the start of a memory access, the 9-bit set part of the address A is used as theaddress to read T0 and T, simultaneously, and the resulting output data (two storedtags) are compared simultaneously with A^. If a match occurs, one of two MATCHsignals, say, from T,, is asserted and used to initiate a memory access from the corre-sponding data memory D,. In a read operation D, outputs its stored data to the systemdata bus; in a write operation D, inputs a data word from the data bus. A 64-bit-widedata bus is assumed in Figure 6.50, which allows a cache block transfer in a singleclock cycle. If a smaller data bus is used, then a block must be transferred in severalcycles. The data memory can also use the 3-bit displacement field of A to select a part

Tag

Set address Displacement

Memory address

20 9 3

CPU

20

Address bus 32

-*-

Set address

Tag

Cache tag memory

512x20tag RAM T,

512x20tag RAM T

D —i

20-bit tagcomparator C,

Cache data memory

512x64data RAM Dn D

CS

512X64data RAM D,

CS

T

20-bit tagcomparator C0

Memory

read/write

control logic

MATCH

MATCH

Data bus

64

Mainmemory

Figure 6.50

An 8 KB two-way set-associative cache for a microprocessor.

of the block, down to a single byte. If a miss occurs, indicated by a no-match outcomefrom both tag comparisons, the memory controller initiates an 8-byte swap with mainmemory to bring the desired data into the cache. The block to be replaced is thenselected according to some replacement policy from the two available candidates. TheVAX-ll/780's cache uses a random replacement policy, whereas the PowerPC 603uses LRU. The 11/780 cache has the write-through memory updating policy, whereasthe 603 implements write-back.

465

CHAPTER 6

Memory

Organization

6.3.3 Structure versus Performance

We next examine some additional aspects of cache design: the types of informationto store in the cache, the cache's dimensions and control methods, and the impactof the cache's design on its performance.

Cache types. Caches are distinguished by the kinds of information they store.An instruction or I-cache stores instructions only, while a data or D-cache storesdata only. Separating the stored data in this way recognizes the different accessbehavior patterns of instructions and data. For example, programs tend to involvefew write accesses, and they often exhibit more temporal and spatial locality thanthe data they process. A cache that stores both instructions and data is referred to asunified. A split cache, on the other hand, consists of two associated but largelyindependent units: an I-cache for instructions and a D-cache for data. While a uni-fied cache is simpler, a split cache makes it possible to access programs and dataconcurrently. A split cache can also be designed to manage its I- and D-cache com-ponents differently.

Caches are also classified by the level they occupy in the memory hierarchy.Early computers employed a single, multichip cache that occupied one level of thehierarchy between the CPU and main memory. Two developments made it desir-able to introduce two or more cache levels in high-performance systems: the feasi-bility of including part of the real memory space on a microprocessor chip andgrowth in the size (but not the speed) of main memory in typical computers. Alevel 1 (LI) or primary cache is an efficient way to implement an on-chip memory.An additional memory level can be introduced via an off-chip, level 2 (L2) or sec-ondary cache. The desirability of an L2

cache increases with the size of main mem-ory, assuming that the size of the on-chip, LI cache is fixed. As main-memory sizeincreases further, even more cache levels may be desirable.

The PowerPC microprocessor family illustrates some of the diversity of com-mercial cache types. The caches for the four original members of the series aresummarized in Figure 6.51. These models are classified as low end (601), mid-range (603 and 604), and high end (620) in terms of performance. The 601 differsfrom the others in large part because it is a "bridge" design with architectural fea-tures of both the PowerPC and the earlier IBM POWER series; the other listedmodels are "pure" PowerPC machines. Each model is a single-chip microprocessorwith an on-chip level 1 cache. An external level 2 cache is easily added, as dis-cussed in Example 6.8. All PowerPC models have an LRU block (line) replace-ment policy, and the line size is either 32 or 64 bytes. The normal write policy oncache misses is write-back, but there is software support for write-through. With

SECTION 6.3Caches

Model	General type	Cache size Sx	Associativity k	Line size/7]
601	Unified	32KB	Eight way	64B
603	D-cache	8KB	Two way	32 B
	I-cache	8KB	Two way	32 B
604	D-cache	16KB	Four way	32 B
	I-cache	16KB	Four way	32 B
620	D-cache	32KB	Eight way	64 B
	I-cache	32KB	Eight way	64B

Figure 6.51

Cache features of some members of the PowerPC family.

the exception of the 601, all models have split caches, with identical I-cache and D-cache capacities. As indicated in the figure, the cache size Sx and the degree ofassociativity k double as we move from the 603 to each more powerful model.

Performance. The cache is the fastest component in the memory hierarchy,so it is desirable to make the average memory access time t_A seen by the CPU asclose as possible to access time t_A of the cache. To achieve this goal, M, shouldsatisfy a very high percentage of all memory references; that is, the cache hit ratioH should be almost one. A high hit ratio is possible because of the locality-of-reference property discussed earlier. From (6.7) we have $t_A = t_A + (1 - H)t_B$,where t_B is the block-transfer time from M2 to M,. The block size is small enoughthat, with a sufficiently wide M2-to-M1 data bus, a block can be loaded into thecache in a single main-memory read operation, making $t_B = t_A$ the main-memoryaccess time. Hence we can roughly estimate cache performance with the equation

$$t_A = t_{Ai} + (l-H)t_{A2} \quad (6.12)$$

A formula similar to (6.12) holds for the average cycle time.

Suppose that M2 is six times slower than M,.A reduction in H from 99 percentto 95 percent—approximately a 4 percent drop in the cache-hit rate—changes $t_A$from $r_{Ai} + (1 - 0.99)6r_A| = 1.06r_A|$ to $t_{Ai} + (1 - 0.95)6r_{A\_} = 1.30t_A$- that is, theaccess time increases by about 23 percent. Hence a small decrease in the cache's hitratio H has a disproportionately large impact on system performance. Consequently,considerable design effort is devoted to making H as close to one as possible. Thisproblem is often restated as that of making the cache-miss ratio 1 - H as close to zeroas possible.

Consider a &-way set-associative cache Mj defined by the following parame-ters: the number of sets sx, the number of blocks (lines) per set k, and the number ofbytes per block (also called the line size) px. Recall that the cache is fully associa-tive when sx = 1 and is direct-mapped when k= \. The number of bytes stored inthe cache's data memory, usually referred to as the cache size Sx, is given by thefollowing formula:

$$S_x = k s_x P_x \quad (6.13)$$

or, in words,

Cache size = number of blocks (lines) per set x number of setsx number of bytes per block

Although other factors, such as the tag memory (directory) size, influence the over-all cost C] of the cache, it is generally assumed that Cx is proportional to the datacapacity Sx; that is, Cx = cxSx.

Design process. The parameters in Equation (6.13), as well as factors like theblock replacement and write policies, influence the cache's hit ratio H in ways thatare hard to quantify because they depend on the workloads used with the cache.Such workloads, in turn, are application dependent. As a result, potential cachedesigns are evaluated by extensive trace-driven simulation experiments withaddress traces derived from representative programs or benchmarks for the targetapplications. Experiments involving billions of simulated address references areoften carried out in the design of the caches for a new microprocessor.

Increasing k, sx, px, or Sx, individually or collectively, tends to increase H. Thesize of an on-chip cache is often limited by area considerations. For example, thedesigners of the PowerPC 604 found that its 16KB caches were adequate for exe-cuting the SPEC benchmarks (see Example 2.8), but not the Transaction-ProcessingPerformance Council (TPC) benchmarks, which consist of programs that manipu-late huge databases in real time and so have large memory requirements. The hitratios for the TPC benchmarks running on the 604 continued to increase signifi-cantly, when the cache size was increased to 32 KB and beyond, a fact that influ-enced the larger cache size of the 620 [Ewedemi, Todd, and Yen 1994].

A general approach to the design of the cache's main size parameters k, sx, pxfollows [Stone 1993].

1. Select a block (line) size px. This value is typically the same as the width w ofthe data path between the CPU and main memory, or it is a small multiple of w.

2. Select the programs for the representative workloads and estimate the number ofaddress references to be simulated. Particular care should be taken to ensure thatthe cache is initially filled before H is measured.

3. Simulate the possible designs for each set size sx and associativity degree k ofacceptable cost. Methods similar to stack processing (section 6.2.3) can be usedto simulate several cache configurations in a single pass.

4. Plot the resulting data and determine a satisfactory trade-off between perfor-mance and cost.

The cache size Sx seems to dominate all other design factors affecting both hitrate and overall performance [Przybylski 1990]. 5! is usually a power of two,hence a basic design question is: How does increasing or decreasing the size by afactor of two affect H? It has been found that, in many cases, doubling the cachesize from S, to 25, increases H by about 30 percent [Stone 1993]. This 30 percentrule is depicted graphically in Figure 6.52, where both the horizontal and verticalscales are normalized quantities. These graphs suggest that beyond a certain point,the improvement in performance measured by the increase in H does not justify thesteadily increasing cost.

In general, k-way set-associative caches with values of k limited to two, four,or eight by cost considerations are preferred. However, it can be argued that for asingle-level cache of moderate size, set-associative addressing seldom performsbetter than direct-mapped addressing [Przybylski 1990]. The most popular blockreplacement policy is LRU, reflecting its tendency to yield lower miss rates* than

CHAPTER 6

Memory

Organization

Normalized cache size S

Figure 6.52

16 Influence of cache size on hitratio and cost.

other replacement policies. Write-back and write-through have both been widelyimplemented in commercial designs. They offer a trade-off between the amount ofmemory traffic generated (less with write-back) and the amount of temporaryinconsistency between the cache and main memory (less with write-through).

EXAMPLE 6.10 CACHE DESIGN FOR THE POWERPC 620 [EWEDEMI. TODD.

and yen 1994]. Figure 6.53 outlines the organization of a system based on thePowerPC Model 620, which is a 64-bit superscalar microprocessor intended to be ofuse in high-performance workstations and multiprocessors. As noted earlier (refer toFigure 6.51), the 620 was designed with a split level 1 cache consisting of an I-cacheand a D-cache each of size 5, = 32 KB, set-associative addressing with k = 8, andblock (line) size p{ = 64 bytes. The 620 also has a separate interface with its own 128-bit data bus to support an off-chip level 2 cache of up to 128 MB. The size parametersof the caches are the result of simulations carried out with various standard workloads.Although it was determined that some important workloads such as the TPC bench-marks would have benefited from larger caches, the 32KB values for the on-chipcaches were selected because chip-area considerations circa 1993 made larger cachesuneconomical.

We now retrace some of the original decisions affecting the design of the 620'slevel 1 cache [Ewedemi, Todd, and Yen 1994]. The block size /^was chosen to be64 bytes based on the need to balance the time spent loading a block into the cache—it is excessive if/?, is too large—with the number of such loads—it is excessive if p,is too small. Another factor influencing the choice of p} was the width of the systemdata bus, which is 16 bytes. With px - 64 bytes, a cache block can be refilled or writ-ten back to the next level of memory in four clock cycles.

The architectural specifications for the PowerPC require a minimum main-memorypage size of 4 KB. The low-order 12 bits of the 620's 64-bit memory address word aretherefore reserved for a displacement address within a page. From the cache perspec-tive, the high-order half of this 12-bit field forms a convenient set address, while thelow-order half can be used to address a byte within a 64B cache block. Six set-addressbits imply that each cache can have 26 = 64 sets. With S, = 32 KB and px = 16 B, a cachecan contain a total of 512 blocks (lines). Hence 512/64 = 8 lines can be placed in a set,which suggests the use of eight-way set-associativity—as was eventually decided. Thisrelatively large degree of associativity also gave good performance with the SPEC andTPC benchmark suites. For instance, simulation with the TPC-A benchmark yielded the

Program controlunit

Fetch unit •

Branch-processing unit

Dispatchunit

3 integerexecution units

General-purposeregister file

Floating-pointexecution unit

Load/storeunit

Data MMU

32KB

data cache

Floating-pointregister file

Instruction MMU

32KBinstruction cache

Bus interface units

40Address

128,'fData

36 ,

128,

System bus

Tag address Data

Level 2 cache

Figure 6.53

Organization of the PowerPC model 620.

following data for D-cache performance:

Cache size Sj	Associativity k	Relative miss rate
8KB	Four way	1.78
16KB	Four way	1.36
16KB	Eight way	1.29
32KB	Eight way	1.00

469

Implementation of a fc-way cache in the traditional manner illustrated by Figure6.50 imposes a speed penalty that increases rapidly with k. When k = 8, eight tags must

be compared simultaneously; the tag size can be up to 28 bits, depending on the size ofthe address space. Such comparisons can be quite slow. The 620 has an unusual imple-mentation of eight-way set-associative addressing, which uses several small CAMarrays like that of Figure 6.46 to speed up accesses within a set.

The preceding techniques for designing a single-level cache can be adapted inmany ways to add more cache levels to a computer. This task is of particular inter-est when designing around a single-chip microprocessor that already contains anLI cache; an off-chip L2 cache is a natural way to increase memory performance.The look-aside design of Figure 6.41a can, in principle, easily accommodate addi-tional cache levels, as Figure 6.54a suggests. Here the system bus carries all thememory traffic due to misses that must be processed by M2 (the L2 cache) and M3(main memory), as well as 10 data transfers. Figure 6.54b shows a version of thefaster, look-through organization (Figure 6.4lb) that is used in the PowerPC 620(Example 6.10) and the MIPS R10000 (Example 5.8). In each case the processorcontains a controller for a two-level cache and a special external bus, separatefrom the system bus, to which an L2 cache can be connected. Various controlmethods, some very complex, have been developed to maximize the memory sys-

Sinmicro   gle-chipprocessor

L2 cache   Main

M2   memory

                  M3

CPU   LI cacheM,

System bu s

(a)

Single-chip   LI cache

micro-processor M,

          Main

          memory

          M3

          1

                              L2 cacheM2

CPU   Cachecontroller

          System bu s

(b)

Figure 6.54

Two ways of adding an L2 cache to a microprocessor with an on-chip LI cache (a) look-aside and (b) look-through.

tern's performance and to ensure the consistency of the information stored in the 471three memory levels.

6.4SUMMARY

No one technology can supply all the memory needs of a computer. Fast memoriesare expensive: cost per bit increases as access time decreases. Consequently, sev-eral memory types with very different physical properties can be found in a typicalcomputer system. Besides cost per bit and access time, other important characteris-tics of memory devices are data-transfer rate, alterability, and compatibility withprocessor technologies.

Main memory is of the random-access type where the access time of everylocation is constant. RAMs are organized as two-dimensional arrays to reduce thecost of their access circuitry and facilitate manufacture. The dominant technologiesfor this application are semiconductor ICs, especially dynamic RAMs (DRAMs)based on single-transistor cells. Secondary memories require a lower cost per bitand a higher storage density. We can achieve these goals by using serial-accessmemory technologies that share access mechanisms and have access times thatvary with location. Serial-access memories store information on tracks that behavesomewhat like shift registers. The most widely used technologies in this group aremagnetic-surface memories with electromechanical access mechanisms, for exam-ple, magnetic-disk and -tape units. Also popular are serial memories that employoptical-recording techniques.

The memory units of a computer are organized as a multilevel hierarchy (M,,M2, ..., M„) in which M, is connected to the CPU, M2 is connected to M,, and soon. M, has less capacity, higher cost, but shorter access time than M, + 1. The goalof a memory hierarchy is to obtain a cost per bit close to that of the least expensivememory M,, and an access time close to that of the fastest memory M,. Such amemory system can be managed by hardware (a memory management unit) orsoftware (an operating system) to behave like a single large memory. This behav-ior is achieved by automatically translating the virtual-memory addresses refer-enced by programs into real addresses in the physical-address space and byautomatically transferring blocks (pages) of information between the various lev-els of the hierarchy. Locality of reference ensures that data is generally in M, whenreferenced by the CPU. A basic measure of the performance of a hierarchicalmemory system is the hit ratio //, which is the fraction of all memory referencesthat are satisfied by M,.

Memory space is a limited resource of a computer and so must be shared bydifferent applications. Dynamic allocation means determining the regions of mem-ory assigned to programs while they are in execution. Nonpreemptive methodsassign space to incoming blocks only if an available region of sufficient size exists;best fit and first fit are two possible allocation methods of this type. Preemptivemethods assign incoming blocks to occupied regions of M, and thereby permitmore efficient use of memory space. Blocks to be preempted are selected accord-ing to some replacement policy. Least recently used (LRU) is one of the mostwidely used replacement policies. The block types used to allocate memory Spacealso affect performance. Segments are blocks of variable size that correspond to

CHAPTER 6

Memory

Organization

472 logical units of a program. Pages are fixed-sized blocks with no logical signifi-

6 s cance. Memory space can be allocated by segments, pages, or a combination of

Problems k°tn (Pa8e(* segments). The use of fixed-size pages greatly simplifies memory

management.

To reduce the speed disparity between CPU and, main memory, one or moreintermediate memories called caches are used. A cache may be split into an I-cacheand a D-cache that store instructions and data, respectively; a unified cache storesboth. Information is stored in a cache's data memory in page-style blocks (lines).Each block is marked by a tag address held in a special tag memory (directory).When the CPU outputs a memory address, the cache compares it to the contents ofits tag memory. If a match (hit) occurs, the memory access is completed by thecache; otherwise, a block that includes the addressed item is transferred from mainmemory to the cache. The tag memory of a £-way set-associative cache is dividedinto k sets, each of which can be searched rapidly via an expensive technique calledassociative, or content, addressing. A lower-cost direct-mapped cache has only oneblock per set. The more powerful microprocessor chips incorporate an LI cacheand provide support for attaching a larger but slower L2 cache.

6.5PROBLEMS

6.1. List the main physical differences between the following memory technologies:SRAMs, flash memories, magnetic floppy disks, optical hard disks, and CD-ROMs.

6.2. When a CPU and its main memory M operate at similar speeds, a one-word load orstore can be completed in a single CPU clock cycle. The CPU is often designed to func-tion properly with slower memory technologies. It does so by retaining control of thesystem bus for two or more clock cycles until a slow load or store is completed; theextra clock cycles, during which the CPU is idle, are known as wait states, (a) Whatchanges must be made to the memory's external signals given in Figure 6.10 to accom-modate wait states? (b) Suppose a slow RAM requiring k > 1 wait states is used with afast CPU in a computer that achieves a performance level of p MIPS while executinga fixed workload at a CPU clock frequency of/MHz. Assuming that no other changesare made, describe in qualitative terms what happens top as/is steadily decreased tozero.

6.3. Consider the generic 1-D RAM organization depicted in Figure 6.7. Assume the stor-age cell unit is implemented by the DRAM cell of Figure 6.9b. Briefly describe threeways in which the RAM can be modified to double its data-transfer rate.

6.4. A 128MB RAM is to be designed from 2M x 4-bit RAM ICs. Assume that 1-out-of-2k decoder ICs are also available for k < 3, as well as ICs containing standard logicgates. The main design goal is to minimize the total number of ICs used, (a) Carryout the design assuming that each RAM chip has a single chip-select line CS and giveyour answer in the style of Figures 6.11 and 6.12. (b) Repeat the design assumingthat each RAM IC has two chip-select fines C5, and CS2 and is enabled if and only ifC5, = C52= 1.

6.5. Using the 64Mb DRAM of Example 6.1 as the basic component, design a 256M x 32-bit DRAM. Include in your answer a diagram in the style of Figures 6.11 and 6.12.

6.6. A 16Mb DRAM chip has a word size w = 8 bits. Like the 8E1 of Example 6.1, it has a2-D organization with multiplexed row-column addressing, (a) If the column addressis 10 bits, what is the size of the row address? (b) How many copies of this DRAM areneeded to make a 1G x 32-bit memory?

6.7. Occasionally, it is desirable to implement a small RAM using a single RAM IC of largecapacity. For example, DRAM manufacturers sometimes sell RAMs that are defectivebut contain sub-RAMs that are fully operational; these units are used in low-cost appli-cations such as toys. Describe how the 64Mb DRAM of Figure 6.13 can be used as a512Kx 4-bit DRAM.

473

CHAPTER 6

MemoryOrganization

6.8. For the 64Mb DRAM described in Example 6.1, calculate the minimum time requiredto read out the contents of every addressable location in the memory (a) if the addressesare generated in a random sequence and (b) if page mode is used.

6.9. A RAM is to be designed with a target capacity of 16 MB. Three DRAM ICs of thekind shown in Figure 6.10 are available to serve as components: (a) a 4M x 1-bitDRAM costing $22 per IC; (b) a 1M x 2-bit DRAM costing $10; and (c) a 256K x 8-bit DRAM costing $4.50. Access circuitry, including ICs and wiring, is estimated tocost $x + 10>\ where x is the number of RAM ICs used and y is the number of addressbits to be decoded externally. Determine which type of DRAM IC would minimize thecost of the memory.

6.10. Consider the three DRAM types a, b, and c defined in the preceding problem. We wantto build from one of these DRAM types a memory with a word size w = 4 bits. Thememory should have the largest possible storage capacity consistent with access cir-cuitry cost of Sx + lOy, as before, and a total system cost of at most $475. Determinethe DRAM type to use and the maximum capacity that can be achieved.

6.11. A RAM has N storage cells organized as Nx rows and Nv columns. The number of ad-dress drivers needed is Nx + N' (a) If N = M2, where M is an integer—that is, N is aperfect square-—show that the number of address drivers needed is a minimum if andonly if Nx = Ny = M. (b) If N is not a perfect square, provide an algorithm for determin-ing values of Nx and Nv that minimize the number of address drivers.

6.12. A certain 1M x 16-bit RAM has four-way address interleaving with four memorybanks M0, M,, M2. and M3. (a) Identify the bank to which each of the following hex-encoded addresses is assigned: 01234, ABCDE. 91272. and FFFFF. (b) If one of thememory banks is busy when a new read request arrives at the memory, what is theprobability that the request will be delayed due to memory contention?

6.13. List and discuss briefly three advantages and three disadvantages of the Rambusmethod (Example 6.2) for interfacing main memory to a very high performance work-station.

6.14. A moving-arm disk-storage device has the following specifications:

Number of tracks per recording surface 200

Disk-rotation speed 2400 rev/min

Track-storage capacity 62,500 bits

Estimate the average latency and the data-transfer rate of this device.

6.15. A certain magnetic hard disk drive has the following specifications in its data sheet:

Number of disks (recording surfaces) 14 (27)

Number of tracks per recording surface 4925

Number of sectors on all recording surfaces 17.755,614

474 Storage capacity (formatted) of disk drive 9.09 GB

Disk-rotation speed 5400 rev/min

SECTION 6.5 Average seek time 11.5 ms

Problems Internal data-transfer rate 44 to 65 MB/s

Calculate the block size B and the average block access time rB.

6.16. The seek time of a magnetic-disk memory depends on how fast the read-write head canmove between tracks. Suppose there are N tracks numbered 0 through N- I, and theread-write head takes time Dt to move from track i to track i ± D, that is, across Dtracks. Hence if an access addressed to read track / is followed by an access to trackj = i ± D, the seek time of the second access is Dt. The best-case seek time is 0 and theworst case is Nt. The question then arises: What is the average seek time ts as a functionof N and tl Assuming that the tracks are accessed in a random fashion, demonstrate thatfs = M/3; that is, the average seek time is approximately the time to move the read-writehead across one-third of the tracks. [Hint: Enumerate the seek times for all the possible(j'j) combinations for a small case such as N - 8 and then attempt to derive a generalexpression for the average seek time.]

6.17. A magnetic-tape system accommodates 2400 ft reels of standard nine-track tape. Thetape is moved past the recording head at a rate of 200 in/s. (a) What must the lineartape-recording density be in order to achieve a data-transfer rate of 107 bits/s? (b) Sup-pose that the data on the tape is organized into blocks each containing 32K bytes. A gapof 0.3 in separates the blocks. How many bytes can be stored on the tape?

6.18. A nine-track magnetic tape has fixed block and interblock gap sizes. The gap length is0.6 in, and the storage density is 1600 B/in. (a) If the space utilization u is 707c, whatis the block size in bytes? (b) Let the start-stop time be 1 ms and let the measured (ef-fective) data-transfer rate be 55 KB/s to read a single block. What is the maximum pos-sible data-transfer rate?

6.19. The data-transfer rate deff of a magnetic-tape memory with respect to a single blocktransfer is given by Equation (6.3). It is possible to increase deff by accessing more thanone block at a time, which spreads the start-stop fss over all the accessed blocks.Suppose that ?ss = 1.5 ms, the block size bs = 2048 B, the gap length gl = 0.25 in, andthe storage density s = 1600 B/in. If def{ = 95,000 B/s, how many blocks must be ac-cessed simultaneously in order to increase deffto at least 100,000 B/s?

6.20. Another medium for secondary memories is digital audio tape or DAT, which is asmall magnetic-tape cartridge adapted from videotape technology. High storage capac-ity and high data-transfer rates are achieved by storing the data in short, multitrack di-agonal strips along the tape and by wrapping the tape (which moves

relatively slowly)around a spinning set of one or more read-write heads. This design produces a veryhigh head-to-tape speed. A certain DAT unit has the following specifications: Thelength of the tape is 90 m. The tape moves at 0.7 in/s (1.79 cm/s), but the head-to-tapespeed is 270 in/s (68.58 cm/s). (a) If the DAT's storage capacity is 2 GB, estimate theeffective normal data-transfer rate in KB/s. (b) The DAT drive has a special search andrewind speed, which is 200 times the normal read-write speed. Estimate how long ittakes to fully rewind the tape.

6.21. The data sheet of a commercial magneto-optical disk drive includes the following spec-ifications:

Formatted storage capacity of unit with 1024-byte sectors 650 GB

Formatted storage capacity of unit with 512-byte sectors 600 GB

Read data-transfer rate with 1024-byte sectors 0.87 MB/s

Read data-transfer rate with 512-bvte sectors 0.79 MB/s

Write data-transfer rate with 1024-byte sectorsWrite data-transfer rate with 512-byte sectors

0.29 MB/s0.26 MB/s

(a) The larger (1024 byte) sector provides greater storage capacity and higher data-transfer rates than the smaller (512 byte) sector. Explain why. (b) The larger sectorsize appears to have all the advantages, so why is the smaller size ever used? (c) Whyis writing slower than reading?

6.22. The storage hierarchy of the IBM System/390 mainframe family of high-performancecomputers has been described as a pyramid with nine levels, with the internal CPU reg-isters forming the highest level and magnetic-tape storage forming the lowest (ninth)level. Suggest the memory types that define the remaining seven levels and their posi-tions in the hierarchy.

6.23. A computer has a two-level virtual-memory system. The main memory Mt and the sec-ondary memory M? have average access times of 10"6 and 10~3 s, respectively. Weknow that the average access time for the memory hierarchy is 10~* s, which is consid-ered unacceptably high. Describe two ways in which this memory access time could bereduced from 10-1 to 10"5 s and discuss the hardware and software costs involved.

6.24. A two-level memory (M,. M->) has the access times rA = 10"8s and f» = 10~3s. Whatmust the hit ratio H be in order for the access efficiency to be at least 65 percent ofits maximum possible value?

6.25. In an «-level memory, the hit ratio Ht associated with the memory M, at level i maybe defined as the probability that the information requested by the CPU has been as-signed to Mj. Assuming that all information assigned to M, also appears inM/+1, thenffj < H2 < ... < Ha = 1. Using this definition of //,-. generalize the expression for rAgiven in Equation (6.6) to an ?i-level memory hierarchy.

6.26. A certain memory configuration has four levels M,, M2, M3, and M4 with hit ratios of0.8.0.95,0.99. and 1.0. respectively. A program Q makes 3000 references to this mem-ory system. Calculate the exact number of references /?,- made by Q that are satisfied byan access to level M,.

6.27. The residual-hit ratio RH, of a level M, in a hierarchical memory system has been de-fined as the ratio of the number of access requests that actually reach M, to the numberof such requests that M, can satisfy. Clearly, RHl < Hr the hit ratio, because M, can sat-isfy any access request that is satisfied by a higher, faster level of the hierarchy. Calcu-late RH, for each level of the four-level memory and the program Q defined in Problem6.26.

6.28. A high-speed computer has a two-level paged virtual memory. Main memory has a ca-pacity of 64 MB and a cycle time of 50 ns. Secondary memory consists of magnetic-disk units with the following specifications: an average seek time of 7 ms; an averagerotational latency of 3 ms; and an internal data rate of 100,000 B/s. Essentially all diskaccesses result from page faults, very few of which require a page from main memoryto be copied back to disk. We know that main memory has a hit ratio of 0.9999998 andthat the average time to access memory as a whole is 60 ns. Estimate the page size P.showing all your calculations.

6.29. Let p, denote the fraction of memory-access requests that result in an access to levelM, in the three-level memory of Figure 6.55. When a miss occurs in M(. a page swapalways takes place between M, and M,-+1; the average time for this page swap is fB .(a) Calculate the average time fA for the processor to read one word from the memorysystem, (b) We want to make /A < 1.1 x 10"7 s In other words, fA should not exceedthe access time of M, by more than 10 percent. We can achieve this speedupvby re-placing M3 with a faster memory technology that reduces /B to a new value rB .

475

CHAPTER 6

Memory

Organization

4/0   Page transfertime tB (s)

SECTION 6.5 Level i time t^ (s) probability p,

Problems	M,	IO"7	0.999990	0.0005
	M2	10-6	0.000009	0.01
	M3	KT4	0.000001	

.Figure 6.55

Data for problem 6.29.

Memory	Capacity	Cost ($/B)	Access time (s)	Hit ratio
Cache 1	16 KB	io-3	10 ns	0.990000
Cache 2	256 KB	io-5	20 ns	0.999900
Main memory	32 MB	10"6	100 ns	0.999999
Disk memory	8 GB	IO"9	10 ms	1.000000

Figure 6.56

Data for problem 6.30.

What should r^, be? (c) Suggest and justify a more cost-effective way of satisfyingthe above requirement on r^than reducing t'By

6.30. (a) What are the average cost per bit and the access time of the four-level memory sys-tem specified in Figure 6.56? (b) Suppose that, as a cost-saving measure, the second-level cache is eliminated from the system. Determine the resulting percentage changesin the system's cost and access time, showing all your calculations.

6.31. A memory reference by the PowerPC microprocessor generates a 32-bit effective ad-dress Aeff that contains a 16-bit virtual address to a page of size 4 KB. Address Aeff alsocontains a pointer to a small set of segment registers that store segment descriptors, (a)How many segment registers does the PowerPC have? (b) Each segment descriptor in-cludes a 24-bit segment address, called the virtual segment identifier VSID. How bigis the PowerPC's virtual-address space? (c) As discussed in the text, the Pentium hasfour memory-address-translation modes, depending on whether or not segmentation orpaging are enabled. The PowerPC also has several address-translation modes, one ofwhich, called real addressing, is defined as the mode in which the effective and phys-ical addresses are the same. To which Pentium mode does real addressing correspond?

6.32. Assuming page size to be a function of average segment size only, determine the pagesize 2* that maximizes memory space utilization when the average segment

size is 5000words and k must be an integer.

6.33. The available space list of a 16KB memory has the following entries at some time t:

Region (base) hex address Size (bytes)

0000 2K

1000 1 K

2000 512

31FF 3K

The following sequence of allocation and deallocation requests then occurs:

Time   t + 1 t + 2 t + 3   t + 4

 Size of block to be allocatedAddress of block to be deallocatedSize of block to be deallocated IK   2K   2DFFIK IK

6.34.

Determine the available space list after all these requests have been serviced using(a) best-fit and (b) first-fit allocation. Assume that the memory is searched in ascending address sequence.

Consider the following page-address trace generated by a two-level cache-main-memory scheme that uses demand paging and has a cache capacity of four pages.

1

45143212146741317

Assume a "hot" start, in which the cache initially has pages 1, 2, 3, and 4 allocated toit. Which of the page-replacement policies FIFO or LRU is more suitable in this case? Show your calculations, and give a short intuitive justification of your answer.

6.35. Computers such as the MIPS R3000 have caches that use a random page-replacementpolicy that we referred to as RANDOM. The page to be replaced is selected by a fastprocess that approximates truly random selection and does not use any data on thepage's reference history. State whether or not RANDOM is a stack replacement algo-rithm and justify your answer.

6.36. A variation of the LRU replacement policy, which we call simplified LRU (SLRU), hasbeen used in some virtual-memory systems. Every page P, in an SLRU page table hasa reference bit Ri associated with it. Whenever P, is accessed, its reference bit P, is setto 1. If the access request for P, causes a page fault, then P, is reset to 0 for all j * i andP, is brought into main memory M,. When a page in M, must be selected for replace-ment, the SLRU algorithm scans all the P,'s in a fixed order. The first page encounteredwith a reference bit of 0 is replaced. If all the reference bits are 1, then the page withthe smallest (logical) address is replaced, (a) For the following page-address trace, de-termine the page-hit ratio under both SLRU and LRU, assuming that M, has a capacityof three pages and is initially empty.

242351341256

(b) Is SLRU a stack replacement policy? Justify your answer.

6.37. We want to build a small word-organized associative memory using the 4 x 4-bit mem-ory circuit of Figure 6.46 as the basic building block. The memory is to store ten 8-bitwords having the format shown in Figure 6.57. Any one of the fields A, B, and C maybe selected as the key. Assume that all stored keys are unique. When a match occurs,the entire matching word is to be fetched (read operation) or replaced (write operation).Draw a logic diagram for the memory including all access circuitry.

6.38. Suppose an 10 processor (IOP) is attached to the system bus of Figure 6.416. The IOPcan transfer data to or from the main memory M2 without interacting with the CPU.

A B C

1 i i i

Figure 6.57

Word format for problem 6.37.

SECTION 6.6References

478 while the CPU transfers data to and from the cache M,. Assume that a cache write-

through policy is implemented, as well as memory-mapped 10. Devise a realistic situ-ation where the IOP's interactions with M2 can cause the CPU to see stale memorydata, resulting in a system crash.

6.39. Suppose that a 2KB cache has set-associative address mapping. There are 16 sets, eachcontaining four cache blocks (lines). The memory-address size is 32 bits, and the small-est addressable unit is the byte, (a) To what set of the cache is the address 000010AF]6assigned? (b) If the addresses 000010AF16 and FFPF7xy^16 can be simultaneously as-signed to the same cache set, what values can the address digits xyz have?

6.40. (a) Suppose the system in Figure 6.48 has its address lines labeled A0:A31, where A0 isthe high-order address bit. Identify the 15 lines used to address the cache's data RAM.(b) Assume that a single-word transfer over the system bus takes 15 ns. Estimate howlong it takes the system to fully respond to a memory access when a cache miss occurs.

6.41. (a) Construct a register-level diagram for the IDT 71B74 cache-tag RAM IC used inExample 6.8. (b) Cache-tag RAMs such as the 71B74 have a reset input that clears allthe cache-tag RAM's tag-storage locations. Ordinary RAM ICs have no such reset con-trol line. Why?

6.42. An eight-way set-associative cache is used in a computer in which the real memory sizeis 232 bytes. The line size is 16 bytes, and there are 210 lines per set. Calculate the cachesize and tag length.

6.43. Redesign the direct-mapped cache of Example 6.8 with the following changes: the ca-pacity of the cache is to be reduced to 64 KB, and the cache block size and the widthof the system data bus are both to be 32 bits.

6.44. Design a four-way set-associative cache in the style of Example 6.9 with the followingparameters: the capacity of the cache is 64 KB; the cache block size is 32 B; and thewidth of the system data bus is 32 bits.

6.45. Discuss in qualitative terms the impact of the following design decisions on cache per-formance: (a) selection of a cache block (line) size p] that is too small; (b) selection ofa cache block size that is too big; (c) selection of an associativity level k that is toosmall.

6.6REFERENCES

1. Burgess, B. et al. "The PowerPC Microprocessor." Communications of the ACM, vol. 37(June 1994) pp. 34-42.

2. Clark, D. W. "Cache Performance in the VAX-11/780." ACM Transactions on Com-puter Systems, vol. 1 (February 1983) pp. 24-37.

3. Cook, B. M. and N. H. White. Computer Peripherals. 3rd ed. London: Edward Arnold.1994.

4. Ewedemi. S., D. Todd, and J.-T. Yen. "Design Issues of the High Performance PowerPC620 Microprocessor." unpublished MS. December 1994.

5. Handy, J. The Cache Memory Book. Boston: Academic Press. 1993.

6. Hauck, E. A. and B. A. Dent. "Burroughs' B6500/7500 Stack Mechanism." ProceedingsSpring Joint Computer Conference, pp. 245-51. [Reprinted in Siewiorek. D. P., C.

G.Bell, and A. Newell. Computer Structures: Readings and Examples. New York:McGraw-Hill, 1982, pp. 244-250.]

7. Heath, S. PowerPC: A Practical Companion. Oxford, UK.: Butterworth-Heine-mann, 1994.

8. Integrated Device Technology Inc. PowerPC Secondary Burst Cache Design, Applica-tion Brief AB-02. Santa Clara, CA, 1994.

9. Intel Corp. Pentium Processor Family User's Manual, vol. 3, Architecture and Pro-gramming Manual. Santa Clara, CA, 1994.

10. Kane, G. MIPS RISC Architecture. Englewood Cliffs, NJ: Prentice-Hall, 1988.

11. Knuth, D.E. The Art of Computer Programming, vol. 1, Fundamental Algorithms. 2nded. Reading, MA: Addison-Wesley, 1973.

12. Kumanoya, M, T. Ogaywa, and K. Inoue. "Advances in DRAM Interfaces." IEEE Mi-cro, vol. 15 (December 1995) pp. 30-36.

13. Mattson, R. L. et al. "Evaluation Techniques for Storage Hierarchies." IBM System Jour-nal, vol.9 (1970) pp. 78-117.

14. Micron Technology Inc. 8 Meg x 8 FPM DRAM, data sheet. Boise, ID, 1997.

15. Prince, B. High-Performance Memories. Chichester, UK: John Wiley & Sons, 1996.

16. Przybylski, S. A. Cache and Memory Hierarchy Design. 2nd ed. San Mateo, CA: Mor-gan Kaufmann, 1990.

17. Quantum Corp. ATLAS II Hard Disk Drives, data sheet. Milpitas, CA, 1996.

18. Shore, J. E. "On the External Fragmentation Produced by First-Fit and Best-Fit Alloca-tion Strategies." Communications of the ACM, vol. 18 (August 1975), pp. 433-440.

19. Smith, A. J. "Cache Memories." Computing Surveys, vol. 14 (September 1982),pp. 473-530.

20. Stone, H. S. High-Performance Computer Architecture. 3rd ed. Reading, MA: Addison-Wesley, 1993.

21. Triebel, W. A. and A. E. Chu. Handbook of Semiconductor and Bubble Memories. En-glewood Cliffs, NJ: Prentice-Hall, 1982.

22. Weste, N. and K. Eshraghian. Principles of CMOS VLSI Design. 2nd ed. Reading, MA:Addison-Wesley, 1992.

479

CHAPTER 7

System Organization

This chapter considers how computers and their major components are intercon-nected and managed at the processor or system level. It examines the methods usedfor internal and external communication, as well as the design of input-output (10)systems. The final topic is the use of multiple processors to achieve high perfor-mance, fault tolerance, or both.

7.1

COMMUNICATION METHODS

In recent years computing has become intimately associated with communication.A computer's internal or local communication methods significantly affect its flex-ibility and performance. External, long-distance communication allows computersto be linked together, for example, via the global Internet network. This sectionexamines the general nature of the local and long-distance communication mecha-nisms used with computer systems.

7.1.1 Basic Concepts

The difficulty in transferring information among the units of a computer largelydepends on the physical distances separating them. We distinguish two cases:intrasystem communication, which occurs within a single computer system andinvolves information transfer over distances of less than a meter; and intersystemcommunication, which can involve communication over much longer distances.Intrasystem communication is primarily implemented by groups of electrical wirescalled buses, which support parallel, that is, word-by-word, data transmission.Intersystem communication, on the other hand, is realized by a variety of physicalmedia, including electrical cables, optical fibers, and wireless (radio) links. Serial

480

(bit by bit) rather than parallel data transmission is preferred for communicationover longer distances. Serial links cost less, are more reliable, and are also easier tocontrol than parallel links. A set of computers and other system components thatare linked together over relatively long distances constitute a computer network.

Buses. The various processor-level components, CPU, caches, main memory,and IO (peripheral) devices within a computer system communicate via buses. Theterm bus in this context covers not only the physical links among the components,but also the mechanisms for controlling the exchange of signals over the bus.

Figure 7.1 depicts the most basic computer bus structure. Here a single bus, thesystem bus, handles all intrasystem communication. All units share the system bus.therefore at any time only two units can communicate with each other. A typicalsystem bus transaction is a memory read (load) operation that involves the transferof one or more data words over the system bus from the memory (cache or main)M to the CPU. A memory write (store) operation transfers data over the system busin the opposite direction. Input-output operations normally involve data transfersbetween an IO device and M. In all the preceding operations M is a passive or slavedevice with respect to system bus transactions, whereas the CPU can actively con-trol the system bus, that is, serve as a bus master. IO devices are normally thoughtof as slave units, but they can be made into bus masters via control units suchas specialized IO controllers or general-purpose IO processors.

As Figure 7.1 indicates, the system bus consists of three main groups of lines:address, data, and control. (Not shown are the lines that distribute electrical powerto the bus units.) The address lines, typically 8 to 32 in number, transmit theaddresses of data items stored in the system's main memory or IO address space.The data lines, typically 16 to 128 in number, transmit data words over the bus.Finally, the control lines perform such functions as identifying the transaction type(memory read, memory write, IO interrupt, and so forth) and synchronizing com-munication between fast and slow units.

The characteristics of a system bus tend to closely match those of its host CPUand vary widely between different microprocessor families and even betweenmembers of the same family. The evolution of CPUs in speed and word size hasbeen matched by a corresponding evolution in their system buses. For example, thefirst member of Intel's 80X86 family, the 8086 microprocessor, had internal dataand (real) address word sizes of 16 and 20 bits, respectively. The CPU data wordsize became 32 bits, and the address size 24 bits, with the 80286 microprocessor;

481

1 1

;   i i i t   i i i

CZ   1

    , i

rz   1

'   i   r   'r ^   i ii 'r i ir\r ■ r ' r i ir

CPU   Cache/mainmemory M   IO
device]   IOdevice n

Systembus

Figure 7.1

Communication within a computer via a single shared bus.

482

SECTION 7.1

Communication

Methods

PowerPC603micro-processor

54

— Power (3.3 V) and ground

— Control signals

-*• Data and address parity

32—/■—►• Address bus A

64

-*■ Data bus D

Figure 7.2

System bus of the PowerPC 603

microprocessor.

both became 32 bits with the 80386. The data and address sizes used inside theCPU are often, but not always, the same as those found in the external system bus.The 16-bit 8088, a variant of the 8086 used in the first IBM PC, has an 8-bit exter-nal data bus. On the other hand, the Pentium's external data bus is 64 bits wide.

Figure 7.2 outlines the system bus of the PowerPC 603 microprocessor, whichis typical of personal computers. It has 64 data-transfer lines D which are bidirec-tional; that is, they act either as inputs or outputs of the CPU—but not simulta-neously. The system bus can transfer from 1 to 8 bytes at a time. Its 32 addresslines A allow 2^{32} = 4G memory or IO locations to be specified. Twelve paritycheck lines, one for each byte of D and A, provide error detection. A large set ofcontrol lines supports data transfers, exchange of bus control, interrupt processing,and other bus functions.

The principal use of the system bus is high-speed data transfer between theCPU and M. Most IO devices are slower than the CPU or M and present an exter-nal interface that is different from that of the system bus. For example, magnetic-disk units and other secondary memories transfer data serially. Therefore, theyneed to be connected to the system bus via interface circuits called IO controllersthat perform series-to-parallel and parallel-to-series format conversions and othercontrol functions. A single IO controller can interface many IO devices to the sys-tem bus. This leads to the structure shown in Figure 7.3 in which the IO devices areconnected to a separate bus called an IO bus.

Computer manufacturers and standards organizations have standardized vari-ous IO bus types. For example, the Small Computer System Interface known as theSCSI (pronounced "scuzzy") bus was adopted as a standard by the American

CPU

Cache/mainmemory M

1 Svstem bus

IObuscontroller

IO (local) bus

IO

device 1

IO

device n

Figure 7.3

Computer with separate systemand IO buses.

SCSI

k   controller

'   10

m '

8

n ^

Control

*■ Data/address

SCSI10 bus

Figure 7.4

The Small Computer System Interface (SCSI) 10 bus.

National Standards Institute (ANSI) in 1986. This bus connects 10 devices such ashard disk units and printers to personal computers. SCSI was originally designed totransfer data a byte at a time at rates up to 5 MB/s. As can be seen from Figure 7.4,the SCSI bus is smaller and simpler than a system bus like the PowerPC's. Its datasubbus is only 8 bits wide and is also used to transfer addresses. Ten additionallines provide all the necessary control functions. Recent extensions to the originalSCSI standard have wider data buses (16 and 32 bits), more control features, andhigher data-transfer rates.

Another bus with a role similar to SCSI is the so-called Industry StandardArchitecture (ISA) bus originally developed by Intel for the IBM PC. Since itallows extra main-memory units as well as 10 devices to be added to a computer, itis often referred to as a local or expansion bus, rather than an IO bus. A morerecent bus standard that we examine in detail later is the Peripheral ComponentInterconnect (PCI) bus, which can transmit 4- or 8-byte words at rates of 500 MB/sor more.

483

CHAPTER 7

System

Organization

Long-distance communication. There are several important differences be-tween intra- and intersystem communication methods. Whereas intrasystem com-munication is serial by word, intersystem communication is usually serial by bitbecause of the difficulty of synchronizing data bits sent in parallel over long dis-tances. Serial transfers also reduce the cost of the communication equipment.Every long-distance data transfer requires a substantial amount of time to establishthe communication path to be used, for instance, the time associated with enteringa telephone number. To reduce this overhead, a sequence of many bits called amessage, which corresponds to the concept of block or page in memory systems, istransmitted at one time.

Intrasystem communication is implemented by transmitting digital signals inthe form of discrete 0 and 1 pulses over multiline buses. As they are transmitted,the pulses are distorted by variations in the bus's electrical properties, interferencebetween adjacent lines (crosstalk), and similar phenomena collectively known asnoise. The distortion caused by noise increases with the number of lines in the busand the signal transmission frequency; it is also affected by the quality of the trans-mission medium. Beyond some point the pulses become unrecognizable and trans-mission errors result. Over long distances, therefore, it is more cost-effective toembed the data in analog signals that are transmitted serially, in much the sameway as voice traffic has long been sent over telephone lines. Continuous analogsignals called carriers are generated and varied (modulated) in some manner toproduce distinct signal types that denote 0 and 1. A device called a modulator-

484

SECTION 7.1

Communication

Methods

Vol I101 lo|i|o|

Computer1

Modem   iwmm

•►

UlolH

mmm

Telephone line

Modem

_TL_TL

Computer

Sender

Receiver

Figure 7.5

Long-distance data transmission using frequency-modulated (FM) signals.

demodulator, or modem, converts data between the modulated analog form usedfor long-distance communication and the pulse form used inside the computer.

Figure 7.5 illustrates the modulation method called frequency modulation(FM) used by modems that connect a computer to a low-speed, "voice grade" tele-phone line. The carrier is a sine wave whose frequency f can be shifted slightly tocreate two distinct frequency levels: f_0 denoting 0 and f_1 denoting 1. Such signalsare heard as beeps of different pitch. Since the 1980s, complex signal-processingtechniques have been developed to increase the data-transfer rates over telephonelines from 300 bits/s—bits per second is often denoted bps in this context—to56,000 bits/s. which is close to the maximum possible. These techniques includethe assignment of multiple carrier frequencies to the sender and receiver, error-correcting codes that mask noise-induced errors, and data compression that detectsand eliminates redundant information in the data being transmitted.

Digital communication networks, that is, networks designed expressly fortransmitting information in digital form, can achieve much higher data-transferrates. An example is the integrated services digital network (ISDN), an interna-tional standard for transmitting audio, video, and other data in digital form.Although ISDN was originally proposed around 1960. it has only recently beendeployed worldwide. ISDN takes advantage of fiber-optic technology and fastcommunication methods to achieve data-transfer rates of 600 Mb/s or more. Wire-less (radio) transmission using orbiting satellites to relay messages can also achievevery high data rates.

Computer networks. Digital communication networks designed to link manyindependent computers are called computer networks. Their rationale is to permitsharing of computing resources (hardware, software, or data) that are widely dis-persed. For communication over distances of a few kilometers or so—within a sin-gle office building, for instance—local-area networks (LANs) are used. A LAN isa computer network employing data-transmission links that are private to the net-work in question. Computer networks spread over large geographical areas, that is,wide-area networks (WANs), use data-transmission facilities supplied by telecom-munications companies, which in many countries are government-owned or -regu-lated organizations.

Various techniques exist for sharing the links of a computer network that aimat reducing communication costs. One such technique is message switching, whichuses intermediate switching centers (servers) on long communication paths to storemessages and subsequently forward them toward the final destination; this processis called store and forward. Messages are collected by each server, where they areorganized (grouped into batches) to make efficient use of the data paths connectedto that server. Complete message transmission is thus accomplished by a sequence

Header (5 bytes)   Data (48 bytes)

r   r   >

Address | i i i I I I I i i 1

            Checksum

Con   trol

Figure 7.6

Format of a data packet(cell) used in asynchronoustransfer mode (ATM)networks.

of hops through a variable number of servers. Message switching utilizes the avail-able communication links far more efficiently than circuit switching.

Messages vary greatly in length so that short messages can be delayed whilelonger messages are being transmitted. This problem is reduced by dividing mes-sages into packets of fixed length and format and then transmitting packets fromlong messages interspersed with packets from short messages. The store-and-forward servers are then responsible for sorting the packets from the various mes-sages and transmitting them to their proper next destinations. Different packagescan be sent by different routes dictated by network traffic conditions. At the finaldestination a message must be reassembled from its constituent packets. This formof communication is called packet switching and is used for fast communication oflarge amounts of data. A type of packet switching called asynchronous transfermode (ATM) combines voice and data communication using short packets that canbe transmitted very fast. An ATM packet called a cell consists of a 5-byte headercontaining the destination address and certain control information, followed by a48-byte data field, as depicted in Figure 7.6.

Although the goal of a universal or open computer network to which any man-ufacturer" s computers can be attached remains elusive, the International StandardsOrganization (ISO) has developed a set of guidelines that provides a common basisfor computer network design. These guidelines are known as the ISO ReferenceModel for Open Systems Interconnection (OSI) and define seven functional levelsor layers through which users exchange messages in a computer network; see Fig-ure 7.7. Each layer is associated with certain network services—error control, forinstance—and different computers in a network can be thought of as exchanginginformation between corresponding layers. Consequently, a distinct set of commu-nication rules or protocol can be defined for each layer. In general, layers 1 to 3 ofthe OSI Reference Model involve services associated with data communicationsfunctions close to the network hardware, while layers 5 to 7 involve software(operating systems) functions close to the network user. The intermediate transportlayer (layer 4) serves to interface the network's hardware and software.

EXAMPLE 7.1 THE ETHERNET NETWORK ACCESS METHOD [SIMONDS

1994). Ethernet is a popular bus-oriented architecture for LANs. Its specificationinvolves only the physical and data-link layers, so it is seen as primarily an accessmethod for LANs. Computer-specific hardware (Ethernet controllers) and software(Ethernet drivers) implement the remaining layers of network control. At the physicallevel an Ethernet LAN has the structure shown in Figure 7.8. Up to 1024 nodes (com-puters) can be connected via coaxial cable: their maximum separation is limited to2.8 km. At the data-link level, communication is by messages or frames that contain

Layer

Associated services

1. Physical Electrical and mechanical hardware interfacing to the physical communication

medium.

2. Data link Message setup, transmission, and error control. *

3. Network Establishing message paths in the network (message routing and flow control).

4. Transport Interfacing network-independent messages with the specific network being used.

5. Session Creation and management of communication channels between the communicating

applications programs.

6. Presentation Data-transformation services such as character-code translation or encryption.

7. Application Providing network support functions such as file-transfer routines to application

programs (network users).

Figure 7.7

The protocol layers of the Open Systems Interconnection (OSI) Reference Model.

the address, control, and check bits, as well as a variable-length data field. Total mes-sage length can range from 64 to 1518 bytes.

A technique called carrier sense multiple access with collision detection (CSMAJCD) controls access to the Ethernet and some other LAN types. A node wishing to senda message over the Ethernet first senses (listens to) the main coaxial cable via a tap unitand transmits the message only if it detects no carrier signal, in which case the networkis not currently in use. Each message is broadcast throughout the network, and its des-tination address is examined by all nodes as it reaches them. Only the node whoseaddress matches that of the message header actually reads the message. Since all com-puters on the network have equal access to the main cable, it is possible for two nodesto begin message transmission at the same time. Consequently, as it transmits a mes-sage, a node monitors the actual signals on the cable and compares them with the sig-nals that the node itself is transmitting. If the transmitted and detected signals differ,

Computer(node)

k.

Communicationcontrol circuits

Computer(node)

7^

-CZr

Server(node)

Cabletap

Twisted-paircable

CZr-r—H

□

CZ>

Coaxial cable segment

Figure 7.8

Structure of an Ethernet-based LAN.

which will be the case if another computer is transmitting a message at the same time,then a collision is said to have occurred. On detecting a collision, an Ethernet nodeceases transmission and tries to transmit the same message again later. The time ofretransmission is randomly selected so that the chance of another collision is slight,although repeated collisions do occur.

Measurements of Ethernet performance show that the CSMA/CD access schemeis fair in that if n nodes request continuous access to the network over some period oftime T, each node gains access to the network for a period very close to T/n. The band-width loss due to collisions, even under heavy traffic conditions, is modest—typicallyless than 10 percent.

487

CHAPTER 7

SystemOrganization

Besides the CSMA/CD method used by Ethernet, another common way ofcontrolling access to a LAN is token passing, where each node in turn receives andpasses on the right to access the network; this right is represented by a special shortmessage called a token. The node that possesses the token has exclusive use of thenetwork for transmitting a message, after which it transmits the token to another(fixed) node. Token passing is often used in ring-structured networks (token rings),but is also used for bus-structured LANs (token buses). When a token ring is notpassing normal messages, the token circulates from node to node around the net-work. A node having a message to transmit waits until the token reaches it. It thenholds the token while it transmits its message. In a ring network, a (nontoken) mes-sage is usually passed in one direction from node to node until it reaches the desti-nation node; it can then be returned to the source node to confirm its receipt. Aftertransmitting one message, a node puts the token back into circulation so that allnodes get roughly equal access to the network.

The Internet. As discussed in section 1.3.3, the Internet is a huge, worldwidepacket-switched computer network descended from the ARPANET, which pio-neered the use of packet switching in the 1970s. Each ARPANET site had acomputer called an interface message processor (IMP), which performed the store -and-forward functions required for packet switching and connected one or morehost computers to the ARPANET. Since many types of computers could be hosts,the IMP acted as a standard interface controller between hosts on its local networkand a set of remote network servers. To ensure some degree of fault tolerance, thenodes and internode links were chosen so that at least two disjoint communicationpaths existed between every pair of IMPs.

The Transmission Control Protocol/Internet Protocol (TCP/IP) developed forthe ARPANET is used by every Internet server. The main function of the IP proto-col is to handle the routing of data packets over the Internet; it corresponds to layer3 (the network layer) of the OSI Reference Model. In particular. IP breaks mes-sages into packets of about 200 bytes each for transmission to another host. AnInternet address is 4 bytes long, implying a total of more than 4 billion distinctaddresses. It is normally represented by a four-part "dotted'" symbolic form likeserver1.net2.university3.edu. Because this address format is hierarchical a nodeneeds only limited routing information, for example, the possible paths to all thenetworks, but not the individual Internet servers, in the domain edu assigned toeducational institutions.

An Internet packet is transmitted with a header containing its most recentsource address and its final destination address HD, as well as a sequence number

488

SECTION 7.1

Communication

Methods

indicating its position in the original message. The packets leave the first server 5Swith consecutive sequence numbers 1,2,3,4,...; however, they can travel by differ-ent routes to the server 5D of the final destination HD and arrive there at differenttimes, not necessarily in the original order. An Internet server that is not on thelocal network containing HD retransmits each packet i* receives to another server towhich it is directly connected, following a routing algorithm that aims to find thefastest path to the ultimate destination. The actual path can vary with network traf-fic conditions. For example, having sent a packet to server S,, the current servermay decide to send the same package to a different server S to avoid network con-gestion, faulty links, or the like.

An Internet package can pass through dozens of servers before reaching thetarget server 5D. The TCP program on SD, which operates within the OSI transportlayer, is responsible for assembling packets in their proper sequence and checkingto see if any are missing or contain errors. If necessary, TCP can send a message toa remote server requesting it to resend a missing or erroneous package. When all amessage's packages have been received in satisfactory condition, TCP mergesthem to reconstruct the original message, which it forwards over the local networkto HD. A higher-level protocol called the hypertext transport protocol (http)enables the Internet to transfer multimedia files easily and efficiently and is thebasis for the World Wide Web.

Interconnection structures. A system's interconnection structure can be de-fined by a graph whose nodes denote components such as computers, memories,communications controllers, and so forth, and whose edges denote communicationpaths such as buses. A path designed to link only two devices is said to be dedi-cated. A path used to transfer information between different sets of devices at dif-ferent times is said to be (time)shared or multiplexed.

A conceptually simple interconnection method is to place dedicated busesbetween all pairs of components that need to communicate. The general case inwhich n units must be connected in all possible ways needs n(n - l)/2 dedicatedbuses. Figure 7.9 shows such a system when n = 4. Dedicated buses allow very fastinformation transfer: All n devices can send or receive data simultaneously, andthere is no delay due to busy connections. Furthermore, systems with dedicatedlinks are inherently reliable because a link failure affects only the two units con-nected to that link. These units may still be able to communicate if they can senddata to each other via other units. For example, if the bus linking U] and U4 in Fig-ure 7.9 fails, {/, and U4 can possibly communicate via U2 or t/3. The main draw-

^1   Dj

z ■——, "~7 Dedicated/ bus

L 3   L 4

Figure 7.9

System of four units connected by sixdedicated buses.

back of dedicated buses is their high cost. The number of buses needed increases asthe square of the number of units. Adding a unit to the system is difficult, as thenew unit must be physically attached to each existing unit.

At the other end of the spectrum, a single shared bus can provide all communi-cations among n units, as illustrated by Figure 7.1. At any time only two units cancommunicate with each other via the bus; the remaining units are effectively dis-connected from one another. A control method (protocol) is required to supervisesharing of the bus among the n devices. Bus control can be centralized in a specialbus-master unit, which can be one of the n communicating units Ur for example, aCPU. Alternatively, several units can be designed to act as bus masters at differenttimes (decentralized control).

In general, connection to a shared bus is established in two different ways:

• A unit Uj capable of acting as bus master initiates the connection of two units tothe bus, perhaps in response to an instruction in a program being executed by £/,.

• A slave unit sends a request to the current bus master for access to the sharedbus. The bus master then connects the requesting unit to the link if it is not in use.If the bus is busy, the requesting unit must wait until the bus becomes available.If several conflicting requests are received, the bus master uses some arbitrationscheme to decide which request to grant first.

The shared bus is one of the most widely used connection methods in computersystems. Its main advantage is low cost. It is also flexible in that new units can eas-ily be introduced without altering the system's overall structure or the connectionsto the old units. However, shared buses are relatively slow, since units are forced towait when the bus is busy. The system is also sensitive to failure of the shared con-trol circuits.

Between the extremes of a set of dedicated buses and a single shared bus lievarious interconnection structures that involve some sharing of links, but permitmore than one word to be transferred at a time. An example is the crossbar net-work shown in Figure 7.10. A crossbar connects two groups of units G, -{UX,U2,-.-, £/,„} and G2 = {U\,U'2,..., U'n) so that any unit of G, can be connectedto any unit of G2, but two units in the same group need never be connected. Forexample, G, can be a set of memory

banks and G-, a set of processors. Crossbar net-works have also been used to connect 10 processors to IO devices. As Figure 7.10 shows, each unit in G, (G2) is attached to a shared, horizontal (vertical) bus. The horizontal and vertical buses are in turn connected via a set of n x m controllers called crosspoint switches, which can logically connect any horizontal bus to any vertical bus. At any time only one crosspoint can be active in each row and column. If k = min{m, n}, then k units in G, can be simultaneously connected to k units in G2. Hence the crossbar network allows up to k data transfers to take place simulta-neously. Access conflicts and delays occur when two units in G| attempt to com-municate with the same unit in G2, or vice versa, at the same time.

Many structures employing shared or nonshared buses have been proposed for intra- and intersystem communication in computer systems. More links increase communication speed, but they also increase cost in terms of the buses themselves and their interface circuits. In practice, direct, dedicated connections are provided among only a subset of the communicating units. Units not directly connected must communicate indirectly via intermediate units that relay data in store-and-forward fashion until the final destination is reached. Indirect communication of this type is

490

G,

SECTION 7.1

Communication

Methods

^i u2   K

Vi   ~ Y-'

^2

Crossbar

•   switching

unit

f«

Crosspoint switch

Figure 7.10

Crossbar connection of two groups of units.

slow, and if used extensively, can significantly reduce performance. The amount of such communication occurring depends both on the system's structure and its com-munication needs. Interconnection structures are therefore selected to balance hardware costs against communication delays for some broad class of applications.

Figure 7.11 shows graphs that abstractly represent some important computer interconnection structures [Feng 1981; Quinn 1994], a few of which we encoun-tered earlier. Here the nodes denote computers or processor-level components such as IO controllers, while the edges denote shared or nonshared buses. The linear or one-dimensional array structure of Figure 7.1 la models the basic system-bus based structure of Figure 7.1, provided the buses are shared. The mesh (two-dimensional array) structure (Figure 1 A\b) occurs in the systolic multiplier of Figure 4.59. The ring structure of Figure 7.11c adds an extra link to the six-node linear structure, thereby cutting in half the length of the longest path between any two units. It also introduces some tolerance of bus failures by providing two, rather than one, com-munication paths between each unit pair. The graph of Figure 7.1 \d is called a star for obvious reasons and has a central or root node connected to all n - 1 other nodes. The linear and star graphs are special cases of a tree, which is a graph with no cycles. The three-dimensional hypercube is depicted in Figure 1 .We, while the complete graph for n = 6 nodes appears in Figure 7.11/. The ring, hypercube, and complete graphs are considered to be homogeneous because all nodes have pre-cisely the same type of connections, making them interchangeable. For instance, each node x has the same number d(x) of neighbors, where d(x) is called the degree of x and is a rough indication of the cost of its bus interface. The other examples in Figure 7.11 are not homogeneous, because all nodes do not have the same degree.

Figure 7.12 summarizes some pertinent properties of the preceding intercon-nection structures. The number of edges and the maximum node degree serve as a

measure of the hardware cost of the structure. The distance between two nodes is the number of edges along a shortest path in the graph from one node to the other. The maximum of these distances, called the diameter of the graph, is an indication of the worst-case communication delays that can occur. In the examples of Figure 7.12, the total number of connecting edges ranges from approximately n2/2 (for large n) in the complete-graph case to the minimum possible value of n - 1 for the linear and star graphs. The complete graph and the star share the largest node degrees, while the linear structure has the largest diameter. The other structures exhibit various compromises between hardware cost and delay. Of particular inter-est is the hypercube, which achieves a reasonable balance between all three param-eters. Therefore, it has been used as the interconnection network in several massively parallel computers.

7.1.2 Bus Control

This section examines the methods to establish and control intrasystem communi-cation via a shared bus [Thurber et al. 1972; Gustavson 1984]. Two key issues are the timing of transfers over the bus and the process by which a unit gains access to the bus. We assume the general structure of Figure 7.1, which applies to most sys-tem and IO buses. We also assume that one particular unit acts as the bus master

O—O—O—O

(a)

(b)

(c)

(d)

(e)

Figure 7.11

Interconnection structures: (a) linear; (b) mesh; (c) ring; (d) star; (e) hypercube; (/) complete.

492

Interconnection structure	Number of edges (buses)	Maximum node degree	Maximum internode distance
Linear	$n-1$		
Ring	n		
Mesh	$(n^{0.5} \times n^{0.5})$	$2(n-n^{0.5})$	
Star	$n-1$		
Hypercube $(n = 2^x)$	$<n/2) \log_2 n$		
Complete	$n(n-1)/2$		

Figure 7.12

Comparison of the interconnection structures of Figure 7.11 assuming each

contains n nodes.

1	$n-1$
2	nil
4	$2(n^{0.5}-1)$
$n-1$	2
$\log_2 n$	$\log_2 n$
$n-1$	1

and supervises the use of the bus by the other units, the bus slaves. In many cases the CPU is the bus master, while the memory and IO interface circuits are slaves; IO controllers also serve as bus masters, however. Only a master can initiate data transfers, although slaves can request them. Both master and slave participate equally in the data-transfer process after it is initiated.

Basic features. Buses are distinguished by the way in which data transfer over the bus are timed. In synchronous buses each item is transferred during a timeslot (clock cycle) known to both the source and destination units. Therefore, the bus interface circuits of both units are in step, or synchronized. Synchronization can be achieved by connecting both units to a common clock source, which is fea-sible only over very short distances. The rising or falling edge of the clock signal, which is one of the bus's control signals, determines when other bus signals attain stable (valid) states. Alternatively, each bus unit can be driven by separate clock signals of approximately the same frequency. Synchronization signals must then be transmitted periodically between the communicating devices in order to keep their clocks in step with each other.

Synchronous communication has the disadvantage that data-transfer rates are largely determined by the slowest units in the system, so some devices may not be able to communicate at their maximum rate. An alternative approach widely used in both local and (especially) long-distance communications is asynchronous tim-ing, in which each item being transferred is accompanied by a control signal that indicates its presence to the destination unit. The destination can respond with another control signal to acknowledge receipt of the item. Because each device can generate bus-control signals at its own rate, data-transfer speed varies with the inherent speed of the communicating devices. This flexibility is achieved at the cost of more complex bus-control circuitry. In local communication where a clock signal is present, data transmission can be asynchronous in the sense that the num-ber of clock periods between bus events (signal changes) can be indeterminate, while the events themselves are synchronized by the clock.

A unit is selected for connection to the main bus in two ways. The bus master can initiate the selection of a slave unit U in response to an instruction in a program or a condition occurring in the system that requires the services of U. Alternatively,

U itself can request access to the shared bus by sending a bus-request signal to the bus master. In each case the master unit must perform a specific sequence of actions to establish a logical connection between U and the bus. If several units can generate requests for bus access simultaneously, the bus master needs a way to select one of the units; this selection process is called bus arbitration. The CSMA/CD collision avoidance technique used by the Ethernet (Example 6.1) is an exam-ple of an arbitration process for LANs.

Bus lines fall into three functional groups: data, address, and control lines. The data lines transmit all bits of an n-bit word in parallel. They consist of either two sets of n unidirectional lines or a single set of n bidirectional lines. The data-bus width n is usually a multiple of eight, with n = 8, 16, 32, or 64 being common values. The address lines identify a unit to participate in a data transfer. Sometimes the same lines transfer addresses as well as data, a method termed data-address multiplexing. This method decreases the cost of the bus, along with the number of external connections (pins) of the units attached to the bus. A computer's system bus usually contains separate

address and data lines, but its 10 buses often do not; see Figure 7.4, for instance. This difference stems from the fact that an address accompanies each data-word transfer over the system bus, whereas data transfers via an 10 bus tend to involve long blocks of consecutive words and need only the starting address of the block. That address can be sent at the start of the data trans-fer. The control lines convey timing signals and status information about the units attached to the bus; they also identify the type of information present on the data-address lines.

Bus interfacing. A significant contributor to the cost of a bus is the number and type of circuits required to transfer signals to and from the bus. A bus line rep-resents a signal path with potentially very large fan-in and fan-out. Consequently, buffer circuits called bus drivers and receivers are needed to transfer signals to and from the bus, respectively.

A special transistor circuit technology called tristate logic is often used in bus design. It is characterized by the presence of three signal values 0, 1, and Z, where the third value Z is the high-impedance state. The binary values 0 and 1 have their usual interpretation, and correspond to two specific electrical states of a line, such as 0 volts and 3.3 volts. The high-impedance state Z, on the other hand, denotes the state of a line that is electrically disconnected from all voltage sources, that is, an open-circuited or floating line. Figures 7.13a and b define a tristate buffer, which serves as a bus-line driver. The inputs x and e are ordinary binary signals that take the values 0 and 1; the output z, however, can take all three values 0, 1, and Z. The tristate buffer (and every other tristate device) has a special input line e called out-put enable, which when set to 0 disables the output line z by changing it to the high-impedance state Z. When e = 1, the circuit becomes an ordinary noninverting buffer with z = x. Figures 7.13c and d show equivalent circuits corresponding to the buffer in the enabled and disabled states.

Tristate logic circuits have two big advantages in the design of shared buses:

• They greatly increase the fan-in and fan-out limits of bus lines, permitting very large numbers of devices to be attached to the same line.

• They support bidirectional transmission over the bus by allowing the same bus connection to serve as an input port and as an output port at different times.-

(a)

(c)

Inputs x e Outputs

0 1	
	0
1 1	
	1
0 0	
	zz
1 0	

e = 0

(b)

id)

Figure 7.13

, _ 2 Tristate buffer: (a) logic symbol;

(b) truth table; (c) equivalent circuit when enabled; (d) equivalent circuit when disabled.

Figure 7.14 shows how we use tristate logic to interface two units (/, and U2 to a set of bidirectional bus lines. If ex = 1 and e2 = 0, then Ul controls or drives the bus lines in question; information is transferred over the bus from Ux to U2. in effect making x2i = Zi, for all i. Conversely, if ex = 0 and e2 = 1, then U2 drives the bus and information is transferred in the opposite direction from U2 to Ux, making xXi = z2ii for all i. If ex = e2 = 0, then the outputs of both Ul and U-, are logically dis-connected from the bus and impose only a minuscule electrical load on it. The combination ex = e2 - 1 is invalid, because it applies two different signals simulta-neously to each bus line making the line's state indeterminate. Proper operation of the bus requires that at most one driver connected to each bus line be enabled at any time.

The bus lines that can be driven by a particular bus unit (/, that is, used by U to send data to other units, depend on U's function in the system. Bus masters have the ability to drive most bus lines, including certain lines that slave units cannot

Bidirectional

bus lines i   ...

Driver ,

1 n   V

\§L   ^ '"<7L   f
ft ^2.0   .0

Zj

M   \   <   Tl   h Transceiver

Receiver

Inputs

Outputs	t	1	e \		e2
T*1.0				x2.1 .7	.7
>	.0 z	.1 ;	,7 x2.\		j

Figure 7.14

Use of tristat

; logii   1: for bus ii Jnit t/,nerfacing.   Unit U2

drive. For example, a CPU can drive all data, address, and most control lines of a system bus. A main-memory unit, which is a bus slave, can drive the data lines but not the address lines, since it only needs to receive information from the address lines.

Timing. The details of some typical data transfers over a bus are shown in Figure 7.15 by means of timing diagrams. The CLOCK signal of period T serves as a timing

reference, making this type of transfer synchronous. In this example, the 0-to-1 transition of CLOCK, that is, its rising edge, determines when other bus sig-nals are recognized. All active signals must be set up with their new values before the CLOCK signal rises. Signal changes are expected to propagate along the bus to their destinations before the next 0-to-l transition of CLOCK.

Figure 7.15 also illustrates some typical signal exchanges between slave and master units; these exchanges follow certain ordering rules called the bus protocol. Consider the read operation depicted in Figure 7.15a. Communication begins when

CLOCK

Address

Control

Data

M

Address

(from master)

Read   Status

(from master) (from slave)

Data

(from slave)

(a)

CLOCK

Address

Control

Data

(from master)

lb)

Figure 7.15

Synchronous data transfers: (a) read and (b) write.

the bus master places one or more predetermined signals on the control lines speci-fying the desired bus transaction, for instance, read from memory (load) or read from 10 device (input). At the same time, the master places the address of the desired (part of the) slave on the bus's address lines. All potential slave units then examine the active control and address signals. The slave with an address matching that on the bus responds in the next clock cycle by placing the requested data word on the bus's data lines; it can also optionally place status information, for example, (no) error occurred, on certain control lines. A synchronous write operation is sim-ilar except that the bus master rather than the slave is the data source; see Figure 7.15b. Note that both edges of CLOCK can be used as reference points in a bus transaction, and the read or write transactions of Figure 7.15 can be designed to take place during one clock cycle of period 2T.

The requirement that the slave respond immediately (in the next clock cycle) to the bus master is lifted by providing a control signal called an acknowledge sig-nal ACK, as shown in Figure 7.16 for a read bus transaction. ACK is controlled by the slave unit and is not activated until the slave has completed its part of the data transfer. The master therefore waits until it has received the ACK signal for the cur-rent data-word transfer before initiating a new one. Thus an acknowledge signal allows a delay of one or more bus cycles, called wait states, to be inserted in a bus transaction to accommodate slow devices. Although ACK may be activated in any cycle, its changes are synchronized with those of CLOCK. This type of communi-cation is often used between main memory and a CPU. By inserting a variable number of wait states and signaling with ACK when it is ready, a memory of essen-tially any speed can communicate with a faster CPU.

Purely asynchronous timing eliminates the bus's clock signal and replaces it with timing control signals like ACK, which are generated by the communicating units. These units are thus self-timed, and units with quite different data-transfer

CLOCK

Address

Control

Data

Acknowledge (control)

Address

(from master)

Read

(from master)

n-cycle delay(n wait states)

Status

(from slave)

Data

(from slave)

ACK

(from slave)

Figure 7.16

Synchronous data transfer (read) with wait states.

rates can communicate asynchronously. We distinguish two cases:

• One-way control in which one of the two communicating devices supplies alltiming signals.

• Two-way, or interlocked, control in which both devices generate timing signals.

If one-way control is employed, a single signal controls each address or datatransfer. This signal can be activated by the source and destination unit, either oneof which can be the bus master. Figure 7.17a shows a source-initiated data transferof this sort. The source places the data word on the data bus. After a brief delay thesource activates the control line with the generic name DATA READY. The delay isto prevent the DATA READY signal from reaching the destination before the dataword. Alternatively, the source can activate DATA READY and place data on thedata bus at the same time. The destination unit must then insert a delay between itsreceipt of DATA READY and its reading of the data bus. The data lines and theDATA READY control line must remain in the active state long enough to allow thedestination unit to copy the data from the data bus. Figure 7. lib shows a data trans-fer initiated by the destination unit. In this case the destination begins the datatransfer by activating the control line DATA REQUEST. The source responds byplacing the required word on the data lines. Again the data must remain active longenough for the destination unit to read it.

Often the DATA READY/REQUEST signals are used to load the data from thesource unit to the bus or from the bus to the destination unit. Such control signalsare called strobe signals and are said to strobe data to or from the bus. For example,the source may generate a data word asynchronously and place it in a buffer regis-ter connected to the bus data lines. A signal on DATA REQUEST activates theclock input line of the buffer, thereby "strobing" the data onto the bus; Figure 7.18illustrates this process.

The disadvantage of one-way control is that it does not verify that the datatransfer has been successfully completed. For example, in a source-initiated datatransfer, the source unit receives no indication that the destination unit has actu-ally received the data transmitted to it. If the destination unit is unexpectedly slowin responding to a DATA READY signal, the data may be lost. This problem iseliminated by introducing a second control line that allows the destination unit tosend a reply signal to the source when it receives a DATA READY signal. Thiscontrol line has the generic name DATA ACKNOWLEDGE or ACK. Figure 1 A9a

497

Data   Data

DATA

DATA

'EADY   REQUEST

(a)

(*)

Figure 7.17

One-way asynchronous data-transfer timing: (a) source initiated and (b) destination

initiated.

498

Data
Sourceunit   n Data bus   Destinationunit
          suffer

DATA REQUEST

.1

Figure 7.18

Use of a DATA REQUEST line to strobe data.

shows the exchange of signals, often called handshaking, that accompanies asource-controlled transfer in this case. The source unit maintains the data on thebus until it receives the ACK signal. The destination activates ACK after copyingthe data from the bus. This sequence allows delays of arbitrary length to occurduring the data transfer. Figure 7.1% depicts a similar technique for destination-initiated communication. The source unit activates ACK to indicate that therequested data is available on the bus's data lines. The source maintains the dataon the bus until the destination unit deactivates DATA REQUEST, an action thatconfirms successful receipt of the data at its destination. As Figure 7.19 demon-strates, a pair of control lines can perform the ready, request, and acknowledgefunctions for all types of asynchronous bus communications.

Bus arbitration. The possibility exists that several master or slave units con-nected to a shared bus will request access to the bus at the same time. A selectionmechanism called bus arbitration is therefore required to enable the current mas-

Data

DATA READY

Data

ACK

(a)

Data

DATA REQUEST

ACK

Data

Figure 7.19

Asynchronous data transfer (handshaking): (a) source initiated and (b) destination initiated.

ter, which we will refer to as the bus controller, to decide among such competing requests. We discuss three representative arbitration schemes: daisy chaining, poll-ing, and independent requesting. These methods differ in the number of controllines they require and in the speed with which the bus controller can respond tobus-access requests of different priorities. Some bus systems combine several dis-tinct arbitration techniques.

Figure 7.20 illustrates daisy-chaining arbitration. This method involves threecontrol signals to which we assign the generic names BUS REQUEST, BUSGRANT, and BUS BUSY. All the bus units are connected to the BUS REQUESTline. When activated, it merely serves to indicate that one or more units are request-ing use of the bus. The bus controller responds to a BUS REQUEST signal only ifBUS BUSY is inactive. This response takes the form of a signal placed on the BUSGRANT line. On receiving the BUS GRANT signal, a requesting unit enables itsphysical bus connections and activates BUS BUSY for the duration of its new busactivity.

The main distinguishing feature of daisy chaining is the way the BUS GRANTsignal is distributed; it is connected serially from unit to unit as shown in Figure7.20. When the first unit requesting access to the bus receives BUS GRANT, itblocks further propagation of that signal, activates BUS BUSY, and begins to usethe bus. When a nonrequesting unit receives the BUS GRANT signal, it forwardsthe signal to the next unit. Thus if two units simultaneously request bus access, theone closer to the bus controller, that is, the one that receives BUS GRANT first,gains access to the bus. Selection priority is therefore determined by the order inwhich the units are linked (chained) by the BUS GRANT lines.

Daisy chaining requires very few control lines and embodies a simple, fixedarbitration scheme. It can be used with an essentially unlimited number of busunits. Since priority is wired in, a unit's priority cannot be changed under programcontrol. If it generates bus requests at a sufficiently high rate, a high-priority unitlike U{ can lock out a low-priority device like £/„. A further difficulty with daisychaining is its susceptibility to failures involving the BUS GRANT lines and theirassociated circuitry. If unit (/, is unable to propagate the BUS GRANT signal, thenno Uj where j > i can gain access to the bus.

The bus-arbitration scheme called polling replaces the BUS GRANT line of thedaisy-chain method with a set of poll-count lines that are connected directly to allunits on the bus, as depicted in Figure 7.21. As before, the units request access tothe bus via a common BUS REQUEST line. In response to a signal on BUS

499

CHAPTER 7

System

Organization

Buscontroller   BUS GRANT   11,   U,   -* -* r/.

BUS REQUEST   1   :

BUS BUSY

Bus

Figure 7.20

Bus arbitration using daisy chaining.

500

SECTION 7.1

Communication

Methods

ut   u2 Un

i

Poll Jcount]

I"" If

Buscontroller

1   €

BUS REQUEST

BUS BUSY

> Bi

Figure 7.21

Bus arbitration using polling.

REQUEST, the bus controller proceeds to generate a sequence of numbers on thepoll-count lines. Each unit compares these numbers, which may be thought of asunit

addresses, to a unique address assigned to that unit. When a requesting unit Ul finds that its address matches the number on the poll-count lines, Ui activates BUS BUSY. The bus controller responds by terminating the polling process, and Ul con-nects to the bus.

The priority of a bus unit is determined by the position of its address in the polling sequence. This sequence can be programmed if the poll-count lines are connected to a programmable register; hence selection priority can be altered under software control. A further advantage of polling over daisy chaining is that in poll-ing a failure in one unit need not affect the other units. This flexibility is achieved at the cost of more control lines (k poll-count lines instead of one BUS GRANT line). Also, the number of units that can share the bus is limited by the addressing capability of the poll-count lines.

The third arbitration technique, independent requesting, has separate BUS REQUEST and BUS GRANT lines for every unit sharing the bus. This approach,

u,   Ui   v.

Bus controller   BUS GRANTl ♦   J \ - 11

^BUS REQUEST]

BUS GRANT!

^BUS REQUEST!

BUS GRANTn

^BUS REQUESTn

BUS BUSY

*

Bus

Figure 7.22

Bus arbitration using independent requesting.

which is depicted in Figure 7.22, provides the bus controller with immediate iden-tification of all requesting units and enables it to respond rapidly to requests for bus access. The bus-control unit determines priority, which is programmable. The main drawback of bus control by independent requesting is the fact that In BUS REQUEST and BUS GRANT lines must be connected to the bus controller in order to control n devices. In contrast, daisy chaining requires two such lines, while poll-ing requires approximately log2 n lines.

501

CHAPTER 7

System Organization

EXAMPLE 7.2 THE PERIPHERAL COMPONENT INTERCONNECT (PCI) BUS

[SHANLEY and Anderson 1995]. The PCI bus, often referred to as a "local" bus, was developed by Intel in the early 1990s and has since become a widely adopted stan-dard for microprocessor-based computer products such as single-board microcomput-ers. Unlike some earlier standard buses, the PCI bus is designed to be easily interfaced with different microprocessor families, main memory, and a very wide range of 10 devices. Many of the PCI bus's lines are optional, so it can be attached to bus units with as few as 47 pins and as many as 100. It can support either 32-bit or 64-bit data trans-fers. In version 2.1, the maximum clock rate is 66 MHz, which allows a data-transfer rate of up to 524 MB/s.

The PCI bus is basically intended for attaching 10 devices to a computer, but it has many of the characteristics of a high-performance system bus. It can be configured as an 10 bus as in Figure 7.3 so that the microprocessor can communicate with memory via its system bus while the PCI bus controller communicates independently with 10 devices via the PCI bus. Figure 7.23 shows a different configuration in which the PCI bus has a more central role. Here the PCI bus is linked to the host CPU's "system" bus via a memory controller referred to as a bridge, which gives it direct access to the host's main memory. This arrangement, unlike that of Figure 7.3, allows CPU-cache

Micro-processor (CPU)

Microprocessor (system) bus

Memory controller (bridge)

Main memory

Graphics

video terminal

SCSI controller

PCI local bus

Ethernet controller

Local-area network

] SCSI 10 bus

Hard disk units

Figure 7.23

Computer system organized around a PCI bus.

502

SECTION 7.1

Communication

Methods

and IO-memory transfers to take place simultaneously. High-speed devices such as video terminals and fast network controllers that have little need of the CPU are con-nected directly to the PCI bus. IO devices intended to conform with other bus standards such as SCSI or ISA can also be interfaced to the PCI bus via appropriate IO control-lers, such as the SCSI bus controller in Figure 7.23.

Each PCI device is required to implement a set of registers called its configuration registers, whose format is defined in the PCI bus specification. When the system is

firstpowered up, all such registers are accessed by the system control software to determinewhich IO devices are currently attached to the PCI bus and their basic communicationrequirements.

Figure 7.24 summarizes the 100 lines that make up the PCI bus. On the left are thesignals required to support basic data transfers using 32-bit or smaller words. On theright are optional lines that support 64-bit data transfers, interrupt control, and other,less-common functions. To reduce pin counts and the size of the connectors needed byPCI-compatible units, addresses and data are multiplexed over a common set of linesdenoted AD. A typical bus transaction involves two phases: In the first phase, anaddress is sent over AD; in the second phase, one or more data words are sent over AD.The remaining lines of the bus perform various control functions, which are outlinedbelow. All bus operations are timed by a clock signal, so the PCI bus is considered tobe synchronous; however, ready and acknowledge signals are provided to allow slowdevices to insert wait states. Most lines are tristate and are considered inactive in the Zand 0 states, unless they have an overbar, in which case they are inactive in the Z and 1states.

The command/byte-enable lines perform different functions at different times.During the address-transmission phase, ClBE defines a bus command that the bus mas-ter uses to tell the bus slave the type of transaction required. The possible commandsinclude memory read, memory write, IO read, IO write, interrupt acknowledge, and afew others. During the data-transmission phase, ClBE indicates which bytes of ADcarry valid data. PAR specifies the parity of the 36 bits AD[3l:0] and ClBE [3:0]; itserves the usual function of single-bit error detection. The lines designated basic inter-face control include FRAME, which delimits a data-transfer transaction and therefore

Required bus lines

Optional bus lines

Address-data AD [31:0]

Command/byte-enable C/fi£[3:0]

Parity PAR

Basic interface control

Clock CLK

Reset RST

Error reporting

Arbitration (master unit only)

32   PCI-compatibleunit   32

4   4

6, .   4

„ 2' -

2

4

1,

5,

'

Address-data AD [63:32]Command/byte-enable ClBE [7:4]Parity PAR64Basic interface control

64-bit

Lock controlInterrupt requestCache supportTesting support

Specialfunctions

Figure 7.24

Signals of the PCI standard bus.

is active for the duration of the entire transaction; a pair of data-ready/acknowledgelines, IRDY (initiator ready) and TRDY (target ready), for use by the master and slave,respectively; and a STOP line that the slave uses to ask the master to halt the currenttransaction. The system clock signal CLK is responsible for synchronizing all bus trans-actions, while RST resets all bus-control registers attached to the PCI bus. The twoerror-reporting lines indicate parity errors and related problems.

A pair of lines REQ (bus request) and GNT (bus grant) control bus arbitration.The bus-arbitration method is not part of the PCI bus's specification, which requiresonly that the central bus controller receive a single request at a time on the REQ lineand that all attached bus masters receive their fair share of access to the bus. The daisy-chaining method discussed earlier is easily implemented. Independent requesting canalso be implemented without difficulty by means of priority-encoding logic that selectsone of several active requests to forward to the PCI bus's REQ line.

Figure 7.25 shows a representative three-word data transfer from slave to mastervia the PCI bus; for example, an IO read operation. The transaction begins when theinitiating master unit (which is assumed to already be in control of the bus) activatesFRAME by setting it to 0 in clock cycle 1; as its name suggests, FRAME frames theentire data transfer sequence. The master then places an address and command word(IO read in our example) on the AD and C/BE lines, respectively; this informationshould be valid when clock cycle 2 begins. During cycle 2 all the units attached to thebus try to decode the address and command. In this instance an IO unit containing thecurrent address will be successful and will prepare to communicate with the master. Inthe next cycle the master relinquishes control of AD and places valid byte-enable infor-mation on the C/BE lines for the remainder of the transaction. To avoid conflictswhen the master stops driving the AD bus (and certain control lines) and the slavebegins to do so, an idle, turnaround cycle—cycle 3 in Figure 7.25—must follow theaddress phase. The slave can transmit a sequence of data words via AD, beginning incycle 4 at the maximum rate of one word per clock cycle. The two communicatingunits control the actual transfer rate via the IRDY and TRDY lines, which permit anynumber of wait states to be inserted after each data-transfer cycle.

503

CHAPTER 7

System

Organization

Clock CLR

FRAME

C/BE

Address-data AD

IRDY

TRDY

DEVSEL

1   2   3 4   5 6 " 8

1

1

1 IO read Byte enable

Address   Data 1   Data 2 Data 3

1

1   Wait 1

Wan

AD Data Wait Data Wait Data

turnaround transfer 1 state transfer 2 state transfer 3

Figure 7.25

Data-transfer transaction (memory read) via the PCI bus.

504

Data transfer cannot begin until the master activates IRDY to indicate that it isready to receive data; this occurs in cycle 2. The slave makes the data word 1 availableand signals this fact by making TRDY - 0 in cycle 3; the data transfer takes place incycle 4. In this example the slave immediately deactivates its ready line (TRDY = 1)making cycle 5 into a wait state; it then reactivates TRDY and places data word 2 onAD for transmission in cycle 6. The slave places data w€ord 3 on AD for transmission incycle 7. This time, however, the master decides to insert a wait state by making IRDY =1 for one clock cycle. Consequently, data word 3's transfer is delayed until cycle 8. Themaster deactivates FRAME in cycle 7 to signal that the following cycle marks the endof the bus transaction. The last control line DEVSEL (device select) shown in the fig-ure is activated by the slave device in cycle 2 to indicate that the slave has successfullydecoded the address and is the target of the current bus transaction. No data transfercan occur until DEVSEL is active, so this line serves to tell the master when a bustransaction cannot be completed due to a missing or faulty slave unit.

A write transaction (where the master is the data source rather than the slave) isvery similar to that of Figure 7.25. No turnaround cycle is needed after the address-transfer phase, because the master continues to drive AD throughout the transaction.

7.2

IO AND SYSTEM CONTROL

The main data-processing functions of a computer involve its CPU and external(cache-main) memory M. The CPU fetches instructions and data from M, pro-cesses them, and eventually stores the results back in M. The other system compo-nents—secondary memory, user interface devices, and so on—constitute theinput-output (10) system. In this section we discuss the hardware and softwareneeded to implement 10 operations. We also discuss operating systems—thesupervisory programs that manage a system's major resources including the CPU,main memory, and 10 subsystems.

10 control methods. Input-output operations are distinguished by the extentto which the CPU is involved in their execution. (Unless otherwise stated, 10 oper-ation refers to a data transfer between an 10 device and M, or between an 10device and the CPU.) If such operations are completely controlled by the CPU, thatis, the CPU executes programs that initiate, direct, and terminate the 10 operations,the computer is said to be using programmed 10. This type of 10 control can beimplemented with little or no special hardware, but causes the CPU to spend a lotof time performing relatively trivial IO-related functions. One such function is test-ing the status of 10 devices to determine if they require servicing by the CPU.

A modest increase in hardware enables an 10 device to transfer a block ofinformation to or from M without CPU intervention. This task requires the 10device to generate memory addresses and transfer data to or from the bus (systemor local) connecting it to M via its interface controller; in other words, the 10device must be able to act as a bus master. The CPU is still responsible for initiat-ing each block transfer. The 10 device interface controller can then carry out thetransfer without further program execution by the CPU. The CPU and 10 controllerinteract only when the CPU must yield control of the memory bus to the 10 con-troller in response to requests from the latter. This level of 10 control is called

direct memory access (DMA), and the 10 device interface control circuit is called aDMA controller.

The DMA controller can also be provided with circuits enabling it to requestservice from the CPU, that is, execution of a specific program to service an 10device. This type of request is called an interrupt, and it frees the CPU from thetask of periodically testing the status of 10 devices. Unlike a DMA request, whichmerely requests temporary access to the system bus, an interrupt request causes theCPU to switch programs by saving its previous program state and transferring con-trol to a new interrupt-handling program. After the interrupt has been serviced, theCPU can resume execution of the interrupted program. Most computers have DMAand interrupt facilities, which are supported by special DMA and interrupt controlunits.

A DMA controller has partial control of IO operations. Essentially completecontrol of 10 operations can be relinquished by the CPU if an 10 processor (IOP)is introduced. Like a DMA controller, an IOP has direct access to main memoryand can interrupt the CPU; however, an IOP can also execute programs directly.These programs, called 10 programs, may employ an instruction set different fromthe CPU's—one that is oriented toward 10 operations. It is common for larger sys-tems to use general-purpose microprocessors as IOPs. An IOP can perform severalindependent data transfers between main memory and one or more 10 deviceswithout recourse to the CPU. Usually the IOP is connected to the devices it con-trols by a separate bus system, the IO bus, as illustrated in Figure 7.3.

505

CHAPTER 7

SystemOrganization

7.2.1 Programmed IO

First we examine programmed 10, a method included in every computer for con-trolling IO operations. It is most useful in small, low-speed systems where hard-ware costs must be minimized. Programmed 10 requires that all IO operations beexecuted under the direct control of the CPU; in other words, every data-transferoperation involving an IO device requires the execution of an instruction by theCPU. Typically the transfer is between two programmable registers: one a CPUregister and the other attached to the 10 device. The 10 device does not have directaccess to main memory M. A data transfer from an IO device to M requires theCPU to execute several instructions, including an input instruction to transfer aword from the IO device to the CPU and a store instruction to transfer the wordfrom the CPU to M. One or two additional instructions may be needed for addresscomputation and data-word counting.

10 addressing. In systems employing programmed 10, the CPU, M, and 10devices usually communicate via the system bus. The address lines of the systembus that are used to select memory locations can also be used to select 10 devices.An 10 device is connected to the bus via an 10 port, which, from the CPU's per-spective, is an addressable data register, thus making it little different from a main-memory location.

A technique used in many machines, such as the Motorola 680X0 series, is to assign a part of the main-memory address space to IO ports. This technique is called memory-mapped 10. A memory-referencing instruction that causes data to

Data —T

Address

READ

WRITE

Mainmemory

CPU

IO port

IO port 2

IO port 3

IOdevice A

IO

device B

Figure 7.26

Programmed IO with shared memory and IO address space (memory-mapped IO).

be fetched from or stored at address X automatically becomes an IO instruction if X is made the address of an IO port. The usual memory load and store instructions are used to transfer data words to or from IO ports; no special IO instructions are needed. Figure 7.26 shows the essential structure of a computer with this type of IO addressing. The control lines READ and WRITE, which are activated by the CPU when processing a memory reference instruction, are used to initiate either a mem-ory access cycle or an IO transfer.

In the organization shown in Figure 7.27, sometimes called IO-mapped IO, the memory and IO address spaces are separate. This scheme is used, for example, in the Intel 80X86 microprocessor series. A memory-referencing instruction activates the READ M or WRITE M control line which does not affect the IO devices. The CPU must execute separate IO instructions to activate the READ IO and WRITE IO lines, which cause a word to be transferred between the addressed IO port and the CPU. An IO device and a memory location can have the same address bit pattern without conflict. A minor modification of the circuit of Figure 7.27 can merge the memory and IO address spaces, if desired.

DataAddress

—<   } 1   1 1 i 1   ►<   >

           i   READM   /   READ IO

           ''   WRITE M 1   I i WRITE IO

           ''   r   i   U ,i   T 1' ♦ T r i r

Mainmemory   CPU   IO port 1   IO port 2   IO port 3

           IOdevice A   IOdevice B

Figure 7.27

Programmed IO with separate memory and IO address spaces (IO-mapped IO).

10 instructions. As few as two 10 instructions can implement programmed 10. For example, members of the Intel 80X86 series have two IO instructions called IN and OUT. The instruction IN X causes a word to be transferred from IO port X to the 80X86' s accumulator register A. The instruction OUT X transfers a word from the A register to IO port X. The CPU assigns no special meaning to the words transferred to IO devices, but the programmer can do so. Some words may indicate IO device status and others may be control information (commands) for the IO device.

When the CPU executes an IO instruction such as IN or OUT, the addressed IO port is expected to be ready to respond to the instruction. Therefore, the IO device must transfer data to or from the CPU-IO data bus within a specified period. To prevent loss of information or an indefinitely long IO instruction execu-tion time, the CPU must know the IO device's status so that the transfer is carried out only when the device is ready. With programmed IO the CPU can be pro-grammed to test the IO device's status before initiating an IO data transfer. Often the status is specified by a single bit of information that the IO device makes avail-able on a continuous basis, for example, by setting a flip-flop connected to the data lines at some IO port.

The CPU must perform the following steps to determine the status of an IO device:

1. Read the IO device's status bit.

2. Test the status bit to determine if the device is ready to begin transferring data.

3. If not ready, return to step 1; otherwise, proceed with the data transfer.

Figure 7.28 shows an 80X86-style program to transfer a data word from an IO device to the CPU's A register. It is assumed that the device is connected to ports 1 and 2 like device A in Figure 7.27. The IO device's status is assumed to be contin-uously available at port 1, while the required data is available at port 2 when the status word has the value READY.

If programmed IO is the primary method of input-output control in a com-puter, additional IO instructions can be provided to augment the IN and OUT instructions discussed so far. For example, the Digital PDP-8, an early minicom-puter, has an IO instruction called TSK that tests the status of the IO device and modifies the CPU program counter based on the test outcome. TSK, which means "test IO device status flag and skip the next instruction if the status flag is set," can be implemented by two control lines linking the CPU and the IO device, as shown in Figure 7.29. On executing TSK, the CPU sends a signal called TEST STATUS

Instruction

Comment

WAIT:

INCPI

JNZIN

1READY

WAIT2

Read IO device status into A register

Compare immediate word READY to A, if equal, set flag Z = 1.

otherwise set Z = 0

If Z *■ 1 (IO device not ready), jump to WAIT

Read data word into A register

Figure 7.28

Program to read one word from an IO device.

508

SECTION 7.210 and SystemControl

to the 10 device. If the device status flag is set, a return pulse is sent on the SKIPline, which increments the program counter, thereby skipping the next instruction.Given an instruction of this type, the 10 program of Figure 7.28 can be reduced tothe following:

WAIT:

TSK 1JMP WAITIN 2

A common 10 programming task is the transfer of a block of words betweenan 10 device and a contiguous region of memory. Figure 7.30 shows an inputblock-transfer program written in assembly code for the Intel 8085 microproces-sor. (The 8085 is described in problems 1.31 and 1.32.) We assume here that theinput device generates data at the rate required by the CPU, so no status testing isneeded. The Zilog Z80, another early microprocessor that is software compatiblewith the 8085, has a single instruction INIR (input, index, and repeat) that per-forms all the functions specified by the last five instructions in Figure 7.30. INIRinputs a word from the 10 port addressed by the C register and transfers it to thememory location addressed by the HL address register. INIR then increments HL;decrements B (which is used as a word-count register); and repeats the transfer,increment, and decrement steps until B = 0. Thus, ignoring minor differencesbetween the 8085 and Z80 instruction names, the program of Figure 7.30 reduces

TEST STATUS

Instructionregister TSK   v D   10 device

SKIP I

n

v.   1   StatusliD-flor

Programcounter

INC   10

CPU   port

Figure 7.29

Implementation of the test status and skip (TSK) 10 instruction.

Instruction

Comment

LOOP:

LXI   H,10

MVI   B.100

IN   7

MOV M,A

INX   H

DCR B

JNZ   LOOP

Load memory address register H.L with 10Load (move immediate) register B with 100Read word from input port 7 into register AStore contents of A in memory location M(H.L)Increment memory address register H.LDecrement register B (used as a byte counter)If B*0. jump to LOOP

Figure 7.30

Program to input a block of data from an 10 device.

to the following Z80 program:

LXI   H, 10

MVI B, 100

MVI C,7

INIR

It is interesting to compare these instructions to the INPUT and OUTPUT instruc-tions of the IAS computer mentioned in section 1.2.2.

10 interface circuits. The task of connecting an 10 device to a computer sys-tem is greatly eased by the use of standard ICs variously known as 10 interface cir-cuits, peripheral interface adapters, and the like. These circuits allow 10 devices ofwidely different characteristics to be connected to a standard bus with a minimumof special-purpose hardware or software. The simplest interface circuit is a one-word, addressable register that serves as an 10 port. The major microprocessorfamilies contain various general-purpose and special-purpose 10 interface circuits.They are called programmable if they can be modified under program control tomatch the characteristics of different IO devices.

Among the most basic 10 interface circuits are programmable circuits intendedto act as serial or parallel ports. Serial ports accommodate many types of slowperipheral devices ranging from secondary memory units to network connections.Parallel ports are designed to interface with 10 devices employing multibit, bidi-rectional data paths. A small interface circuit of the parallel type is discussed in thenext example.

EXAMPLE 7.3 THE INTEL 8255 PROGRAMMABLE PERIPHERAL INTER-FACE circuit [Intel 1993]. This IC, whose structure is shown in Figure 7.31,was designed for interfacing 10 devices with the Intel 8085 and other small micropro-cessors. It is housed in a 40-pin package: 8 pins connect the 8255 to an 8-bit bidirec-tional CPU data bus; 24 IO pins can be attached to several 10 devices. These 10 pinsare programmable in that the functions they perform are determined by a control wordissued by a CPU instruction and stored internally in the 8255. This control word canspecify a variety of operating modes involving either synchronous or asynchronousdata transfers.

The 24 pins on the IO side of the 8255 are divided into 8-bit groups designated A,B, and C, each of which can act as an independent 10 port. The C lines are further sub-divided into two 4-bit groups CA and CB. They are commonly used as status or hand-shaking lines in conjunction with the A and B ports. Two address lines A0 and A, selectone of the three ports A, B, and C for use in an IO operation. The fourth address combi-nation is used in conjunction with an output instruction of the form OUT CW to storean 8-bit user-specified control word CW in the 8255's internal control. This controlword has two principal functions:

• It specifies whether the A, B, and C ports are to act as input, as output or, in thecase of A and B only, as bidirectional 10 ports.

• It programs certain C lines to generate handshaking and interrupt signals auto-matically in response to actions by an IO device.

Figure 7.32a shows one of the many possible configurations in which the -4, B,and C lines are programmed as simple 10 ports with no handshaking or interrupt capa-bility. Figure 7.32b shows another configuration in which the A port is programmed to

509

CHAPTER 7

System

Organization

510

SECTION 7.2IO and SystemControl

ToCPU

Databus

READWRITE

Databuffer

8-bit

internal

bus

f A   Controllogic ■«—

J A0

1 4,

I '

Controlregister CR

8255

-^ A "\

^—C=

B >

ToIOdevices

Figure 7.31

The 8255 programmable peripheral interface circuit.

be an input port with asynchronous timing signals generated by the C lines. The linecalled DATA READY is used by the IO device to strobe a word into the buffer registerat port A. The 8255 then automatically generates a response signal on another C line,which can be sent to the IO device as an ACK signal if the IO device requires two-waycontrol. A third C line generates an interrupt signal, which is sent to the CPU to indi-cate the presence of data at IO port A.

The Intel 8256 IC, defined as a multifunction microprocessor support control-ler, combines a number of useful IO interfacing functions in a single IC [Intel1993]. As Figure 7.33 shows, the 8256 contains two parallel 8-bit IO ports 1 and 2,which we can program for synchronous or asynchronous data transfers in the same

(a)

Inputport A

CPU gdata —?-bus

READWRITEINTERRUPTREQUEST'

8255

IO data bus

DATA READYDATA ACK

(b)

Figure 7.32

Two possible configurations of the 8255 programmable peripheral interface circuit.

way as we program ports A and C, respectively, of the 8255. The universal asyn-chronous receiver-transmitter (UART) module controls a communications portthat supports serially transmitted data with various character lengths and transmis-sion speeds, such as a modem might need. An interrupt controller handles up toeight interrupt requests. We can program a set of five 8-bit counters called timersto realize some useful timing functions. For example, timer 5 is designed to operateas a watchdog timer, which means that we can program it to generate an interruptto the CPU if a particular IO event fails to occur within a specified time T. Thistimer is (re)loaded with value T whenever a specific input line on the 8256's IOinterface is activated. The system clock then decrements the timer automaticallyuntil it reaches zero, at which point the timer automatically sends an interruptrequest to the CPU. Hence as long as the IO device in question triggers the reload-ing of timer 5 within the specified period T, no interrupt is generated.

511

CHAPTER 7

System

Organization

7.2.2 DMA and Interrupts

The programmed IO method discussed in the preceding section has two limita-tions:

• The speed with which the CPU can test and service IO devices limits IO data-transfer rates.

''Address- ^,data bus

CPU >

Bus

control

Interrupt —control ■+-

8256

Address-data buffer

8-bit

internal

bus

Nine controlregisters

Controllogic

" t t

Interruptcontroller

Fivecounter-timers

Parallel

IOport 1

Port2A

Port2B

UART

-* Data 1 ^

Data 2A

Data 2B

ToIO

(devices

;i

Serialdata

Control

Interruptrequest J

Figure 7.33

The Intel 8256 multifunction IO interface circuit.

512

SECTION 7.210 and SystemControl

• The time that the CPU spends testing 10 device status and executing 10 datatransfers can often be better spent on other tasks.

The influence of the CPU on 10 transfer rates is twofold. First, a delay occurswhile an 10 device needing service waits to be tested by the CPU. If there are many10 devices in the system, each device may be tested infrequently. Second, pro-grammed 10 transmits data through the CPU rather than allowing it to be passeddirectly from main memory to the 10 device, and vice versa.

DMA and interrupt circuits increase the speed of 10 operations by eliminatingmost of the role played by the CPU in such operations. In each case special controllines, to which we assign the generic names DMA REQUEST and INTERRUPTREQUEST, connect the 10 devices to the CPU. Signals on these lines cause theCPU to suspend its current activities at appropriate breakpoints and attend to theDMA or interrupt request. Thus these special request lines eliminate the need forthe CPU to execute routines that determine 10 device status. DMA further allows10 data transfers to take place without the execution of 10 instructions by the CPU.

A DMA request by an IO device only requires the CPU to grant control of thememory (system) bus to the requesting device. The CPU can yield control at theend of any transactions involving the use of this bus. Figure 7.34 shows a typicalsequence of CPU actions during execution of a single instruction. The instructioncycle is composed of a number of CPU cycles, several of which require use of thesystem bus. A common technique is to allow the machine to respond to a DMArequest at the end of any CPU clock cycle. Thus during the instruction cycle of Fig-ure 7.34 there are five points in time (breakpoints) when the CPU can respond to aDMA request. When such a request is received by the CPU, it waits until the nextbreakpoint, releases the system bus, and signals the requesting 10 device by acti-vating a DMA ACKNOWLEDGE control line.

Interrupts are requested and acknowledged in much the same way as DMArequests. However, an interrupt is not a request for bus control; rather, it asks theCPU to begin executing an interrupt service program. The interrupt program per-forms tasks such as initiating an 10 operation or responding to an error encounteredby the 10 device. The CPU transfers control to this program in essentially the sameway it transfers control to a subroutine. The CPU responds to interrupts onlybetween instruction cycles, as indicated in Figure 7.34.

CPU

Instruction cycle ►

-«— clock—»■cycle

Fetchinstruction   Decodeinstruction Fetchoperand   Executeinstruction Storeresult

Interruptbreakpoint

Figure 7.34

DMA and interrupt breakpoints during instruction processing.

Direct memory access. The hardware needed to implement DMA is shown inFigure 7.35. assuming that all access to main memory is via a shared system bus.The 10 device is connected to the system bus via a special interface circuit, a DM4controller, which contains a data buffer register IODR. as in the programmed 10case; it also controls an address register IOAR and a data count register DC. Theseregisters enable the DMA controller to transfer data to or from a contiguous regionof memory. IOAR stores the address of the next word to be transferred. It is auto-matically incremented or decremented after each word transfer. The data counterDC stores the number of words that remain to be transferred. It is automaticallydecremented after each transfer and tested for zero. When the data count reacheszero, the DMA transfer halts. The DMA controller is normally provided with aninterrupt capability, in which case it sends an interrupt to the CPU to signal the endof the 10 data transfer. The logic necessary to control DMA can easily be placed ina single IC with other 10 control circuits. A DMA controller can be designed tosupervise DMA transfers involving several 10 devices, each with a different prior-ity of access to the system bus.

Data can be transferred in several different ways under DMA control. In aDMA block transfer a data-word sequence of arbitrary length is transferred in a sin-gle burst while the DMA controller is master of the memory bus. This DMA modeis needed by secondary memories like disk drives, where data transmission cannotbe stopped or slowed without loss of data, and block transfers are the norm. BlockDMA transfer supports the fastest 10 data-transfer rates, but it can make the CPUinactive for relatively long periods by tying up the system bus. An alternative tech-nique called cycle stealing allows the DMA controller to use the system bus to

513

CHAPTER 7

SystemOrganization

Mainmemory

Data

AddressControl

Addressregister AR

Register

file

CPU

Control

unit

DMA REQUEST

DMA

ACKNOWLEDGE

Datacounter DC

Addrewregister IOAR

Systembus

Dataregister IODR

Controlunit

DMA

controller

Figure 7.35

Circuitry required for direct memory access (DMA).

514

SECTION 7.2

lO and SystemControl

transfer one data word, after which it must return control of the bus to the CPU.Consequently, long blocks of 10 data are transferred by a sequence of DMA bustransactions interspersed with CPU bus transactions. Cycle stealing reduces themaximum IO transfer rate, but it also reduces the interference by the DMA control-ler in the CPU's memory access. It is possible to eliminate this interference com-pletely by designing the DMA interface so that bus cycles are stolen only when theCPU is not actually using the system bus; this is transparent DMA. Thus varyingdegrees of overlap between CPU and DMA operations are possible to accommo-date the many different data-transfer characteristics of IO devices.

DMA transfers proceed as follows for the system depicted in Figure 7.35.

1. The CPU executes two 10 instructions, which load the DMA registers IOARand DC with their initial values. IOAR should contain the base address of thememory region to be used in the data transfer. DC should contain the number ofwords to be transferred to or from that region.

2. When the DMA controller is ready to transmit or receive data, it activates theDMA REQUEST line to the CPU. The CPU waits for the next DMA breakpoint.It then relinquishes control of the data and address lines and activates DMAACKNOWLEDGE. Note that DMA REQUEST and DMA ACKNOWLEDGE areessentially BUS REQUEST and BUS GRANT lines for control of the system bus.Simultaneous DMA requests from several DMA controllers are resolved by oneof the bus-priority control techniques discussed earlier.

3. The DMA controller now transfers data directly to or from main memory. Aftera word is transferred, IOAR and DC are updated.

4. If DC has not yet reached zero but the 10 device is not ready to send or receivethe next batch of data, the DMA controller releases the system bus to the CPUby deactivating the DMA REQUEST line. The CPU responds by deactivatingDMA ACKNOWLEDGE and resuming control of the system bus.

5. If DC is decremented to zero, the DMA controller again relinquishes control ofthe system bus; it may also send an interrupt request signal to the CPU. TheCPU responds by halting the 10 device or by initiating a new DMA transfer.

DMA can be subsumed under a general method for system-bus arbitration. InMotorola 680X0-series computers, for example, the system bus accommodatesvarious types of bus masters including DMA controllers and certain coprocessorsdesignated DMA coprocessors. Three control lines are provided for bus arbitration:bus request BR, bus grant BG, and bus grant acknowledge BGACK. The BR lineis an input control lineto the CPU and is wire-ORed to all other potential bus mas-ters. It is activated (BR = 0) when one of those devices U—a DMA controller, forinstance—requires control of the system bus. The CPU responds by activating BGand relinquishing control of the system bus at the end of its current bus cycle,which it does by driving the data, address, and certain control lines to the high-impedance state Z. The requesting unit U detects the end of the bus cycle by moni-toring these control lines, at which point U activates BGACK and deactivates itsBR signal. The CPU responds to BGACK by deactivating BG. This step com-pletes bus arbitration. U is the new bus master and can carry out any number ofDMA read or write operations. It returns the system bus to the CPU by deactivatingBGACK.

CPUs such as the 680X0 have no internal mechanisms for resolving multipleDMA requests; this must be done by external logic. Passing the DMA (bus) grant

signal from the CPU through appropriate priority logic to the potential bus masterscontrols bus-access priority. External logic may also be needed to implement cyclestealing by forcing the requesting device to deactivate its DMA request signal aftersome number of bus cycles. In 680X0-based computers these and other DMA control functions are implemented by the Motorola 68450 DMA controller IC, whichsupports up to four independent and concurrent DMA operations via the 68000system bus. The 68450 contains four copies of the basic DMA controller logic ofFigure 7.35, each constituting a separate DM4 channel. Other registers in eachDMA channel store the priority assigned to the channel and the data-transfermodes to be used. A "chaining" mode of operation is supported that allows a chan-nel to reinitialize its address register IOAR and data-count register DC automati-cally at the end of the current block transfer. This approach enables the 68450 tocarry out a sequence of DMA block transfers without reference to the CPU. Whenits current data count reaches zero, a DMA channel that has been programmed forchained DMA fetches new values of DC and IOAR from a memory region MR thatstores a set of DC-IOAR pairs. An address register in each DMA channel holds thebase address of MR.

By reducing the CPU's need to access main memory, a cache can greatlyreduce conflicts between CPU and IO data transfers. High-performance micropro-cessors often have separate cache-CPU and IO-main-memory access paths, whichmeans that a DMA transfer involving main memory can proceed in parallel withCPU-cache operations. In the system of Figure 7.23, for instance. DMA operationsuse the PCI local bus. while the CPU communicates with the cache via the systembus. Only when the CPU needs access to main memory—in response to a cachemiss, for example—does it come into conflict with DMA controllers; such con-flicts are resolved by the PCI bridge unit.

Interrupts. The word interrupt is used in a broad sense for any infrequent orexceptional event that causes a CPU to temporarily transfer control from its currentprogram to another program—an interrupt handler—that services the event inquestion. Interrupts are the primary means by which IO devices obtain the servicesof the CPU. They significantly improve a computer's IO performance by giving IOdevices direct and rapid access to the CPU and by freeing the CPU from the need tocheck the status of its IO devices.

Various sources internal and external to the CPU can generate interrupts. IOinterrupts are external requests to the CPU to initiate or terminate an IO operation,such as a data transfer with a hard disk. We include in this category interruptscaused by a main-memory miss in a virtual memory system, which requires amain-secondary memory page swap involving one or more IO operations. Inter-rupts are also produced by hardware or software error-detection circuits that invokeerror-handling routines within the operating system. A power-supply failure, forinstance, can generate an interrupt that requests execution of an interrupt handlerdesigned to save critical data about the system's state. An attempt by an instructionto divide by zero, or to execute a privileged instruction when not in the priv ilegedstate, are examples of software-generated interrupts. An operating system will alsodeliberately interrupt a user program that has exceeded its allotted time.

The basic method of interrupting the CPU is by activ ating a control line withthe generic name INTERRUPT REQUEST that connects the interrupt source to theCPU. An interrupt indicator is then stored in a CPU register that the CPU tests

periodically, usually at the end of every instruction cycle. On recognizing the pres-ence of the interrupt, the CPU executes a specific interrupt-handling program.Normally, each interrupt source requires execution of a different program, so theCPU must determine or be given the address of the interrupt program to be used.The presence of two or more interrupt requests at the same time causes a furtherproblem. Priorities must be assigned to the interrupts/and the one with the highestpriority selected for handling.

The CPU responds to an interrupt request by a transfer of control to an inter-rupt handler in a manner similar to a subroutine call. The following steps are taken:

1. The CPU identifies the source of the interrupt, for example, by polling 10devices.

2. The CPU obtains the memory address of the required interrupt handler. Thisaddress can be provided by the interrupting device along with its interruptrequest.

3. The program counter PC and other CPU status information are saved as in asubroutine call.

4. The PC is loaded with the address of the interrupt handler. Execution proceeduntil a return instruction is encountered, which transfers control back to theinterrupted program.

Instruction sets usually include instructions to selectively disable or maskinterrupt requests, thereby causing the CPU to ignore certain interrupts. Withoutsuch control, an 10 device that generates interrupts rapidly might require too muchof the CPU's time and interfere with the CPU's other tasks. When a high-priorityinterrupt is being serviced, it is desirable that all interrupts of lower priority be dis-abled. An interrupt enable instruction must subsequently be executed to give thelower-priority interrupts access to the CPU.

Interrupt selection. The problem of selecting one 10 device to service fromseveral that have generated interrupts strongly resembles the arbitration process forbus control discussed in section 7.1.2. Indeed, some interrupt methods require thatthe interrupting device be given control of the system bus. The techniquesemployed for bus arbitration—daisy chaining, polling, and independent request-ing—can all be readily adapted to interrupt handling and can be realized by soft-ware, hardware, or a combination of both.

The interrupt selection method requiring the least hardware is the single-linemethod that appears in Figure 7.36. All 10 ports share a single INTERRUPT

Interruptflip-flop   1STERRUPT REQUEST

                                          1

CPU   IOportO   10 port 1   10 port 2 IOport3

        >,   1   1   J

        V

        10 devices

Figure 7.   36
Single-lin   e int   srrupt   system.

REQUEST line. On responding to an interrupt request, the CPU must scan all the10 devices to determine the source of the interrupt. This procedure requires activat-ing an INTERRUPT ACKNOWLEDGE line (corresponding to BUS GRANT) that isconnected in daisy-chain fashion to all 10 devices. The connection sequence of thisline determines the interrupt priority of each device. Alternatively, the CPU canexecute a program that polls each 10 device in turn requesting interrupt statusinformation. Polling has the advantage of allowing the interrupt priority to be pro-grammed.

Figure 7.37 depicts another common interrupt selection method called multiple-line or multilevel interrupts, which amounts to independent requesting of interruptservice. Each interrupt request line is assigned a unique priority. The source of theinterrupt is immediately known to the CPU, thus eliminating the need for a hard-ware or software scan of the 10 ports. Unless further measures are taken, the CPUmay still have to execute a program that fetches the address of the interrupt-serviceprogram to be used. This step can be eliminated by another technique called vec-toring of interrupts.

Vectored interrupts. The most flexible response to interrupts is obtainedwhen an interrupt request from a particular device causes a direct, hardware-imple-mented transition to the correct interrupt-handling program. The interruptingdevice must then supply the CPU with the starting address or interrupt vector ofthat program.

Figure 7.38 shows a basic way to derive interrupt vectors from multiple inter-rupt request lines. Each interrupt request line generates a unique fixed address,which is

used to modify the CPU's program counter PC. Interrupt requests are stored on receipt in an interrupt register. The interrupt mask register can disable any or all of the interrupt request lines under program control. By setting bit i of this register to 1 (0), interrupt request line i is disabled (enabled). The k masked interrupt signals are fed into a priority encoder that produces a $[\log_2 k]$-bit address, which is then inserted into PC.

To see how program control is transferred using this type of vectored inter-rupt, suppose that three devices are connected to four 10 ports as shown in Figure 7.39a. Assume that when an interrupt request from 10 port / is accepted, the 2-bit address i is generated by the priority encoder and inserted into the program counter

Interruptregister   INTREQ3

                INTREQ2

                1NTREQX

                INTREQO

                    10 port 0   10 port 1 IO port 2   10 port 3

                ^   T"   y

CPU   10 c V   ices

Figure 7.37

Multiple-line interrupt system.

518

SECTION 7.210 and SystemControl

Interruptrequest

Address 2^bits to PC ''

INPUTACTIVE

Priorityencoder

INTREQ 3

INTREQ 2

INTREQ 1

INTREQO

10devices

Interrupt

mask

register

Figure 7.38

A vectored interrupt scheme.

PC. For example, if memory M is addressed by byte and addresses are 4 bytes (one word) long, then i might be placed in bits 3:2 of PC and the remaining 30 bits of PC (bits 31:4 and 1:0) can be set to 0. This results in assigning the first four word-storage locations of M to interrupt vectors, as shown in Figure 139b. The contents of these locations are the user-assigned start addresses of the interrupt-handling

CPU

Data bus

INTREQ 3

INTREQ 2

INTREQ

INTREQO

10 port 0 | | 10 port 1 | 110 port 2 | | IP port 3

Inputdevice A

Inputdevice B

Outputdevice C

(a)

0   z Address 16

4   z Address 28

H   z Address 40

12 z Address 240

16 Device Ainput routine

28 Device Aoutput routine

4(1 Device Bservice routine

40 Device C
service routine

Figure 7.39

(a) A system with vectored IO interrupts and (b) location of the interrupt han-dlers in memory.

Interrupt-handler address

519

CPU <

INTREQn-\

Prioritycontrolcircuit   Organization

INTERRUPT   INTACKn-\

REQUEST   INTREQX   1'

INTERRUPT   INT ACK]

ACKNOWLEDGE INTREQO   r

■«
INTACKO

1   '

IO port 0   IO port 1   IO port n - 1

v.   1   V

IO devices

Figure 7.40

Another implementation of vectored interrupts.

routines. The routines themselves are of arbitrary length and can be located any-where in M.

The foregoing scheme has a one-to-one correspondence between interruptrequest lines and interrupt handlers. Hence if an IO device requires the services ofk distinct programs, it needs k distinct interrupt request lines. Figure 7.40 showsanother, more general, vectored interrupt scheme that does not have this restriction:Each IO port can request the services of many different programs. Again multipleinterrupt request lines are used, but each IO port now has its own interruptacknowledge line. When the CPU activates an acknowledge line in response to aninterrupt request, the IO port in question places the address of the desired interrupthandler on the main data bus, which transfers the address to the CPU, where itmodifies the program counter. This approach requires the interrupting IO port to beable to generate at least partial memory addresses and to act as a bus master.

Another possibility is for an IO device to send the CPU an interrupt vector inthe form of a CPU instruction. The CPU removes this instruction from the data busand executes it in the normal manner. Thus if the IO device sends the instructionCALL PROG to the CPU, execution of this instruction saves essential CPU infor-mation, such as the program counter, and transfers control to an interrupt-handlingroutine named PROG. 8085-based microcomputers use this technique to imple-ment vectored interrupts.

To reduce the number of external connections to the CPU—an important con-sideration in the case of microcontrollers—the interrupt-priority control logic canbe external to the CPU as in Figure 7.40. An interrupt request's priority is deter-mined by the priority circuit input line to which it is connected. An interruptacknowledge signal from the CPU is transmitted to the highest-priority IO portwith an active interrupt request.

PCI interrupts. The PCI local bus discussed in Example 7.1 provides generalsupport for interrupt handling; details such as the vectoring method used are archi-tecture specific and depend on the particular devices using the bus. The PCI bus

520

SECTION 7.210 and SystemControl

has four interrupt request lines named INTA:D among its optional lines (refer toFigure 7.24). A single-function 10 device with interrupt capability must use INTAas its interrupt request line; multifunction 10 devices can use all four lines. A par-ticular pattern on the PCI bus's command lines denotes interrupt acknowledge.Together, the INTx interrupt request lines and the interrupt acknowledge com-mand can implement the request-acknowledge signal exchange needed during aninterrupt transaction over the PCI bus.

Every PCI-compatible device must have a standard set of addressable configu-ration registers CR that identify the device and its communication needs. When thesystem is powered up. the system controller (operating system) reads the CR regis-ters to determine, among other things, the device's interrupt connections. Its 8-bit"interrupt pin" register in CR tells the system controller which interrupt requestline INTx the 10 device is using. A second 8-bit register in CR called the "inter-rupt line" register specifies the system controller's input line that is connected toINTx so that the routing of the interrupt request lines is programmable. The systemcontroller can use this fact to determine the 10 device's interrupt-request priorityto access its interrupt vectors. The CR registers form a small address space thatis separate from the main-memory and 10 address spaces, as indicated by the exist-ence of configuration read and configuration write in the command set specifiedfor the PCI bus.

EXAMPLE 7.4 INTERRUPT CONTROL IN THE MOTOROLA 680X0 [TRIEBEL

and singh 1991]. Interrupts in 680X0-series computers are referred to as excep-tions and include program-generated traps and hardware-induced errors, as well asexternal IO interrupts. Each exception has an associated 8-bit vector N, which points toa main-memory location M(4A/) that stores the address (the exception vector) of a ser-vice program for that exception. Memory locations 0:1023 form an interrupt vectortable storing 256 thirty-two-bit addresses used for interrupt processing. (Figure 739bhas a four-member vector table of this type.) Most of the 680X0's vector table(addresses 256:1023) is reserved for up to 192 user-supplied interrupt vectors; theremaining locations are preassigned by Motorola to specific interrupt types. For exam-ple, on encountering a divide-by-zero instruction, the 680X0 executes a trap sequencethat transfers control to the program whose start address is stored in locationsM(20:23), corresponding to exception vector N = 5. Two types (modes) of vectoredinterrupts are supported: a general mode in which the interrupting device supplies an8-bit vector number referring to an entry in the exception vector table and a simpler"autovector" mode that allows the 10 device to request any of seven fixed exceptionvectors whose addresses are generated internally by the CPU.

Interrupts are processed in the following way in 680X0-based computers: At theend of each instruction cycle the CPU checks to see whether any interrupt request ispending and tests its priority as described below. If the CPU accepts the request, it sus-pends normal instruction processing and enters an interrupt-response sequence.

TheCPU first saves the old contents of the status register SR in a temporary register andthen sets the system state to the supervisor mode. It then either reads a vector N pro-vided by the interrupt source (general interrupt mode) or generates N internally(autovector mode), as specified by control signals from the interrupt source. The CPUproceeds to save the contents (return address) of the program counter PC, the old con-tents of SR, and certain internal information by pushing them into the supervisor stack,one of two stacks maintained by 680X0 CPUs in main memory. Next, using 4A7 as theaddress, the CPU executes a memory read to fetch the exception vector M(4A0 which itloads into PC; normal instruction processing is then resumed.

Figure 7.41 shows a representative hardware interface used for 680X0 10 inter-rupts. Three control lines called IPL (interrupt priority level) serve both for makinginterrupt requests and indicating their priority level. IPL = 0 means that there is nointerrupt request, while IPL = i, where i ranges from one to seven, means that an inter-rupt of priority level i is being requested. On receiving an interrupt request (IPL & 0).the CPU compares the number IPL with three interrupt mask bits I stored in its statusregister SR. If IPL > I, the CPU responds to the interrupt request at the end of its cur-rent instruction cycle; if IPL < I, the interrupt request is ignored. Since SR can bealtered by certain privileged instructions, whether or not the CPU responds to inter-rupts is under software control. Setting the interrupt mask I to zero enables all interruptrequests. If I is set to seven, all interrupts are rejected except those of highest priority(IPL = 7). which are nonmaskable. Interrupt sources can thus use up to 192 vectors,each of which can be assigned to any of seven priority levels.

The CPU acknowledges an interrupt request by setting each of its FC (functioncode) output lines to one to form a 3-bit signal denoting interrupt acknowledgment. Italso places the priority level of the interrupt being acknowledged on address linesAl:3. In the general interrupt mode, the interrupt controller responds by placing aninterrupt vector number N on data lines D0:7. In the circuit of Figure 7.41 with the

521

CHAPTER 7

System

Organization

68000 CPU

IPL

FC

A 1:3

D0:7

VPA

Interruptrequest/priority

Interruptvector /V

3-bit latch

4 4 4

8-bit latch \

Priorityencoder

I-*

Interrupt

selection

logic

Interruptcontroller

1 2

VPA

1 2

192

Interrupt requests(autovector mode)

Interrupt requests(general mode)

Figure 7.41

Interfacing interrupts to the Motorola 68000 CPU.

522

SECTION 7.2IO and SystemControl

68000 CPU, the FC signals are used directly to strobe the interrupt vector N onto thedata bus. To indicate the autovector mode, the interrupt controller responds to FC = 7by activating a special control line (VPA for the 68000 and A VEC for the 68020).causing the CPU to generate N internally according to the formula N = 24 + 1PL.

Pipeline interrupts. After an interrupt occurs, the controlling CPU must beable to identify the interrupting instruction and the register contents needed forany corrective actions. This is not a problem when instructions are executed insequence and only one is active at any time. However in a pipelined processor withseveral instructions in process concurrently, it is possible for instructions to finishout of sequence; that is, an instruction can finish sooner than another instructionthat was issued earlier. This condition is illustrated in Figure 7.42, where threefloating-point instructions, a 7-cycle multiply and two 4-cycle adds, are being pro-cessed by one or more pipelined units. Figure 7.42a shows a situation that corre-sponds to maximum throughput and where the first add instruction addl iscompleted before the multiply instruction, even though the latter was issued onecycle earlier. Assuming no hazards occur due to data dependencies, this comple-tion order is acceptable as far as the main computation is concerned.

Suppose, however, that addl generates an interrupt due to, say, a result (sum)that overflows in its execution stage EX corresponding to cycle 4. Control mustthen be transferred to an interrupt handler designed to service adder overflow. It ispossible that the ongoing multiply instruction will generate another interrupt, say,in cycle 6. This second interrupt can change the CPU's state in ways that preventproper processing of the first interrupt. In particular, registers affected by addl canbe further modified by the multiply interrupt so that proper recovery from the addlinterrupt may not be possible. In this situation the CPU state is said to have becomeimprecise.

We define a precise interrupt to be one where the system state information orcontext needed both for correct transfer of control to the interrupt handler and forcorrect return to the interrupting program is always preserved. A more restricteddefinition requires the system state when the interrupt occurs to be the same as thatin a nonpipelined CPU that executes instructions in sequential order. In that case aninterrupt occurring during the execution of an instruction / is precise if the follow-

Multiply IF RD EX1 EX2 EX3 EX4 WB   IF RD |eXi|eX2||eX3| EX4||wB

Addl IF RD EX WB |F 3ULD0H

Add2 IF RD EX WB   H0DEDE -

Clock cycle 12 3 4 5 6 7(a)   1 23456789

Figure 7.42

Instruction processing in a pipeline with (a) out-of-order completion and (b) in-order completion.

ing conditions are met [Moudgill and Vassilliadis 1996]:

• All instructions issued prior to / have completed their execution.

• No instruction has been issued after /.

• The program counter PC contains /'s address.

We can solve the imprecise-interrupt problem illustrated by Figure 7.42a in several ways. The most direct is to make all interrupts precise by forcing all instructions to complete in the order in which they are issued. This approach is illustrated in Figure 7A2b, where the add instructions are delayed so that they com-plete after the multiply instruction. The undesirable result of this forced, in-order execution method is that the combined processing time for the three instructions increases from seven to nine cycles, so some of the performance benefit of pipelin-ing is lost.

An alternative solution is to allow the state to become imprecise as in Figure 7.42a, that is, allow out-of-order completion but provide a mechanism to recover the precise state or context of the processor at the time of the interrupt. A small reg-ister set, sometimes known as a history buffer HB, is introduced to store tempo-rarily the initial state of every register that is overwritten by each executing instruction /. Hence if an interrupt occurs during /'s execution, the corresponding precise CPU state can be recovered from the values stored in HB, even if a second, conflicting interrupt is generated by a still-completing instruction.

523

CHAPTER 7

SystemOrganization

7.2.3 IO Processors

The IO processor (IOP) is a logical extension of the IO control methods considered so far. In systems with programmed IO, peripheral devices are controlled directly by the CPU. The DMA concept extends limited control over data transfers to IO devices. An IOP has the ability to execute instructions, which gives it fairly com-plete control over IO operations. Like a CPU, an IOP is an instruction-set proces-sor, but it has a more restricted instruction set. IOPs are primarily communication control units designed to link IO devices to a computer. They have also been called peripheral processing units (PPUs) to emphasize their subsidiary role with respect to the central processing unit (CPU).

IO instruction types. In a computer with an IOP, the CPU does not normally execute IO data-transfer instructions. Such instructions are contained in IO pro-grams that are stored in M and are fetched and executed by the IOP. The CPU does execute a few IO instructions that allow it to initiate and terminate the execution of IO programs via the IOP and also to test the status of the IO system. The IO instructions executed by the IOP are primarily associated with data-transfer opera-tions. A typical IOP instruction has the form: READ (WRITE) a block of n words from (to) device X to (from) memory region Y. The IOP is provided with direct access to M (DMA) and so can control the memory bus when the CPU does not require that bus. Like the more sophisticated DMA controllers examined in the pre-ceding section, an IOP can execute a sequence of data-transfer operations involv-ing different regions of M and different IO devices without CPU intervention. Other instruction types such as arithmetic, logical, and branch are included in the

524

SECTION 7.210 and SystemControl

IOP's instruction set to facilitate the calculation of addresses, 10 device priorities, and so on. A third category of 10 instructions are those executed by 10 devices. These instructions control functions such as REWIND (for a magnetic-tape unit), SEEK ADDRESS (for a hard disk unit), or PRINT PAGE (for a printer). Instruc-tions of this type are fetched by the IOP as data and passed on to the appropriate 10 device for execution. '

Figure 7.43 shows the formats used for IO instructions in the IBM System/360 series and its successors, which have IOPs that are referred to as channels [IBM1974]. The CPU supervises 10 operations by means of a small set of privileged instructions with the format of Figure 7.43a. The address field (bits 16:31) speci-fies a base register B and a displacement (offset) D, which identify both the IO device to be used and the IOP to which it is attached. There are three major instruc-tions of this type: START IO, HALT IO, and TEST IO. The START 10 instruction initiates an IO operation. It provides the IOP it names with the memory address of the 10 program to be executed by the IOP. The instruction HALT 10 causes the IOP to terminate IO program execution, while TEST 10 allows the CPU to deter-mine the status of the named 10 device and IOP. Status conditions of interest include available, busy, not operational, and (masked) interrupt pending.

The instructions executed by the IOP are called channel command words (CCWs) and have the format shown in Figure 7.43b. They are of three types:

• Data-transfer instructions. These include input (read), output (write), and sense (read status). They cause the number of bytes in the data count field to be trans-ferred between the specified memory region and the previously selected 10 device.

• Branch instructions. These cause the IOP to fetch the next CCW from the speci-fied memory address rather than from the next sequential location.

• 10 device control instructions. These are transmitted to the 10 device and specify functions peculiar to that device.

The opcode of a data-transfer instruction can be transmitted directly to the 10 device as the "command" byte while the 10 operation is being set up. If the 10 device requires more control information, it is supplied via an output data transfer.

16 20

31

Opcode B D

VIOP/IO device address

(a)

32 37

4b

63

Opcode Memory address Flags Data count (bytes)

(*)

Figure 7.43

Formats of System/360 IO instructions executed (a) by a CPU and (b) by an IOP (channel).

Instruction	Comments
CCW X'07', , X'40',	Rewind tape
CCW X'37', , X'40',	Skip first record
ccw x'or, BUFFER1 ,X'40', 100	Write second record from BUFFER 1
CCW X'lF, , X'40',	Write tape mark
CCW X'07', , X'OO',	Rewind tape and stop

Figure 7.44

A System/360 IO program to write a record on a magnetic tape.

The flags field of the CCW modifies the operation specified by the opcode. For example, a program control flag PCI can be set to instruct the IOP to generate an IO interrupt and make the current IOP status available to the CPU. Another flag specifies command chaining, which means that the current CCW is followed by another CCW that is to be executed immediately. If this flag is not set, the IOP ceases IO program execution after executing the current CCW.

Figure 7.44 lists a small IO program written in System/360 assembly language that writes a 100-byte record on a magnetic tape. The tape is assumed to already contain two records, the second of which is being replaced. Every CCW contains four fields separated by commas, which correspond to the opcode, memory address, flags, and data count fields of Figure 7.43b. This program contains only one data-transfer instruction, which transfers 100 bytes to the tape from the memory region called BUFFER 1. The other CCWs control operations that are peculiar to magnetic tapes and do not use the memory address or data count fields. In all CCWs the opcode and flags have been defined by hexadecimal numbers indicated by the pre-fix X. The flag field X'40' causes the command chaining flag to be set. In the last CCW no flags are set, so the IOP stops after execution of this CCW.

IOP organization. The essential structure of a system containing an IOP appears in Figure 7.45a. The IOP and CPU share access to a common memory M via the system bus. M stores separate programs for execution by the CPU and the IOP; it also contains a communication region IOCR for passing information in the form of messages between the two processors. The CPU can place there the param-eters of an IO task, for example, the addresses of the IO programs to be executed, and the identity of the IO devices to be used. The CPU and IOP also communicate with each other directly via control lines. Standard DMA or bus grant/acknowledge lines are used for arbitration of the system bus between the two processors, as dis-cussed earlier. The CPU can attract the IOP's attention, for instance, when execut-ing an IO instruction like START IO, by activating the ATTENTION line. In response the IOP begins execution of an IOP program whose specifications have been placed in the IOCR communication area. Similarly the IOP attracts the CPU's attention by activating an INTERRUPT REQUEST line, causing the CPU to exe-cute an interrupt handler that typically responds to the IOP by identifying a new IO program for the IOP to execute. Figure 7.45b summarizes the overall behavior of the IOP and its interaction with the CPU.

EXAMPLE 7.5 THE INTEL 8089 IO PROCESSOR [EL-AYAT 1979]. The 8089

is a one-chip IOP for use in systems based on the Intel 8086 microprocessor. As sr^own

Main memory

IOCR

CPU programs

IOP programs

CPU-IOP communication region

WAIT:

SETUP:

SEND:

EXIT:

System bus

CPU

DMA REQUEST

DMA ACKNOWLEDGE

INTERR UPT REQ UEST

ATTENTION

IOP

IO bus

if ATTENTION = 1 then begin

Fetch parameters from IOCR;

Set up DMA control registers;

Begin IO program execution;

Send gommand(s) to I/O device;

Transmit data word;

if transmission error then go to EXIT;

if not end of data then go to SEND;

if not end of IO program then go to SETUP;

Place termination status in IOCR;

end:

go to WAIT;

(b)

IO devices

(a)

Figure 7.45

Computer containing an IOP: (a) system organization and (£>) CPU-IOP interaction.

in Figure 7.46, it has two "DMA channels," each of which can control an independent IO operation. In addition to the usual address and data-count registers found in DMA controllers, the 8089's DMA channels have their own program counters and other cir-cuits necessary to execute an instruction set that is specialized toward IO operations. Thus the 8089 can execute two unrelated IO programs concurrently and logically appears to the CPU like two independent IOPs. The DMA channels share a 20-bit ALU intended mainly for processing memory addresses. They also share bus interface cir-cuits for communication with memory and IO devices. Partly because of pin con-straints—the 8089 is packaged in a 40-pin package—the channels also share a 20-bit bidirectional external bus that is used to multiplex data and address transfers to or from IO devices; the same lines also transfer addresses and data between the IOP and mem-ory. Both 8- and 16-bit data words can be transmitted and received by the 8089, which contains the necessary assembly-disassembly circuits for conversion between these two data formats. If desired, external circuits can be used to create separate system and IO buses, as in Figure 7.45. If the 8089 is configured with a local IO bus, then its IO programs can be placed in a private memory attached to that bus. thus reducing the instruction traffic on the shared system bus.

The CPU and IOP communicate via several message regions in memory, whichare illustrated in Figure 7.47. Each DMA channel has an associated parameter blockPB containing a pointer to the channel's current IO program, that is, a channel addressword. PB also contains application-specific input parameters for the IO program, as

TohostCPU

Miscellaneouscontrol

Bus control

Data-address bus

Attention CA

Channelselect SEL

IOP

control

unit

v To 10I devices

20

Bus

interface

unit

Assembly-disassemblyunit

20-bit
ALU

Interruptrequest

PC

IOAR

DC

1	1		PC
1	i		IOAR
1	1		DC

Miscellaneousregisters   Miscellaneousregisters

DMA cfaa mnel 1   DMA channel 2

Figure 7.46

Structure of the Intel 8089 IOP.

well as output parameters for variables that the channel is to return to the CPU. Theseparameters identify 10 buffer regions in main memory, 10 device names, dataaddresses in secondary memory devices, and so on. The locations of the two PBs arestored in a channel control block CB. which is created by the CPU when the system ispowered up or reset. CB stores status information and a command from the CPU foreach channel. These 1-byte commands fill essentially the same role as the START.TEST, and HALT 10 instructions of the System/360-370 series. The CPU also usesthem to enable, disable, or deactivate the channel's interrupt request line. Thus theCPU supervises each IOP channel by writing into its PB region and into its portion ofCB. Once it has set up the necessary control information in main memory, the CPUdispatches a DMA channel, that is, it initiates an 10 operation, by executing a data-transfer instruction such as OUT or MOVE that activates the 8089's channel-attentionline CA and a second line SEL that indicates which of the two channels is to be dis-patched. The selected channel then proceeds to read its command word from CB, forexample, '"start IO program execution." which makes the channel load the 10 programpointer from PB into its program counter, thereby launching execution of the 10 pro-gram. The channel then executes the program in much the same way as a CPU. The8089 uses DMA to fetch 10 instructions from main memory and. of course, for mem-ory data transfers. Each DMA channel has a programmable channel control (CC) reg-ister that defines the type of DMA transfer to be used.

Channel 1 PB pointer

Channel 1 status

Channel 1 command

Channel 2 PB pointer

Channel 2 status

Channel 2 command

10 program pointer

Parameters for10 program

10 program pointer

Parameters forIO program

10

program

10program

Channel control block CB

Parameter block PB for IOP channel 1

Parameter block PB for IOP channel 2

Figure 7.47 Memory organization for the 8089 IOP.

The 8089's instruction set and the corresponding assembly language (which are quite different from those of the host CPU) contain about 50 different instruction types. The instructions are broadly similar to those of a general-purpose CPU but have only a few simple data and address types and limited data-processing and program-control capabilities. For example, the arithmetic instructions consist only of add, increment, and decrement with unsigned or twos-complement fixed-point operands. No provision is made for overflow detection in signed arithmetic operations. The major instruction types are data-transfer instructions that move data or address words between the 8089's internal registers and its external memory-IO bus. Note that in addition to IO opera-tions, the 8089 can execute memory-to-memory block transfers very efficiently. The 8089's specialized IO control instructions include WID (set bus width), which defines the word size for data transfers as either 8 or 16 bits; XFER (transfer), which prepares a channel for a DMA transfer; and SINTR (set interrupt), which activates the channel's interrupt request line, thus enabling an IO program to interrupt the CPU.

Microcontrollers as IOPs. Developments in IC technology in recent years have made it attractive to use general-purpose microprocessors as IOPs by equip-ping them with specialized IO interface circuits and support software. An example is the Intel i960 RP input-output processor introduced in 1995 as a single-chip

"intelligent IO subsystem" [Intel 1996]. Figure 7.48 indicates the complexity of this IC. The IOP is built around the 80960 microprocessor, a member of the i960 family of pipelined 32-bit RISCs. The 80960*s instruction set is noteworthy for its fast implementation of call and return instructions, its high-performance integer ALU, and its large register file. The core processor also contains a 4KB two-way set-associative I-cache and a 2KB direct-mapped D-cache. To provide quick response to interrupts, the i960 RP allows the programmer to permanently lock critical IO routines such as interrupt handlers in its I-cache.

The i960 RP IOP supports a pair of 32-bit PCI buses: a primary bus for con-nection to the host CPU and a secondary bus for IO devices. It also has a 32-bit internal "local" bus—the 80960's system bus—to which IO devices can be attached, as well as some specialized IO buses such as the I"C (inter-integrated cir-cuit) bus, a serial IO bus developed by Philips Semiconductor. Not surprisingly, this single-chip device has a very large number of IO pins—352 in all. The i960 RP IOP has controllers for three independent DMA channels, two dedicated to the pri-mary PCI bus and one to the secondary PCI bus. It also has flexible controllers to support vectored interrupts, including the advanced programmable interrupt con-trol (APIC) interface used by the Pentium and other Intel microprocessors.

529

CHAPTER 7

System

Organization

7.2.4 Operating Systems

Except when it is dedicated to a single task, a computer is usually managed by a supervisory program called an operating system, which provides a uniform soft-ware interface for other system programs and for applications programs. In mul-tiuser environments the operating system controls such shared resources as CPU time, memory space, IO devices, utility programs, and databases [Silberschatz 1994].

Local memory

Memory controller

Local bus (80960 C system bus)

Two-channel

DMA

controller

To host CPU

80960 core processor

I-cache

D-cache

Advanced programmable

interrupt control (APIC)

Local

bus

arbiter

Primary address translator

Message unit

PCI bus arbiter

Primary PCI bus

One-channel

DMA

controller

PCI-to-PCI bridge

I-C bus interface

Secondary address translator

Secondary PCI bus

Figure 7.48

Structure of the Intel i960 RP input-output processor.

APIC interrupt bus

PC serial IO bus

~}+ *• To IO devices

T* ». To IO devices

530

SECTION 7.2 IO and System Control

Introduction. Programs use a computer's resources in various, and often unpredictable, ways. Resource requirements also change dynamically during the execution of a single program. For example, programs often alternate between computations that use the CPU and IO operations that use IOPs and peripheral devices but do not require the CPU. If several programs are available for execution at the same time, then the computer's performance as measured by overall through-put can be improved by assigning one program to the CPU while others are assigned for execution by IOPs. The scheduling of CPU and IO processing is a typ-ical function of an operating system.

Another important shared resource is mem-ory, both main and secondary, whose management is also typically an operatingsystem task.

Several types of operating systems have evolved over the years. The earliestsystem control programs (batch monitors and spooling systems) were mainly con-cerned with reducing the time required for IO operations involving user programs.Modern operating systems attempt to manage a wide range of computer resourcesefficiently—not just IO devices. They provide textual or graphical interfaces thatallow users to interact directly with the operating system by specifying theresources needed for a particular job. Current operating systems have their originsin several influential systems developed in the 1960s, such as IBM's OS/360,which became a de facto standard for mainframe computers. Early work atManchester University (Atlas), MIT (Multics), and elsewhere led to the UNIXoperating system, which was developed at Bell Laboratories in the mid-1970s andis now in wide use, especially in workstations.

Processes. The basic unit of computing managed by an operating system is aprocess, which is loosely defined as a program module in the course of execution.The resources needed by a process, including processors and memory space, areallocated to it dynamically during execution. Examples of processes are a proce-dure executed by a CPU and an IO program executed by an IOP. A process can becreated in response to a user command to the operating system. Processes can alsobe created by other processes, for example, in response to interrupts. When nolonger needed, a process (but not the underlying program) is deleted by the operat-ing system, and the resources currently allocated to the process are released. Whilein existence, a process has three major states: ready, running, and blocked, asdepicted in Figure 7.49a. In the ready state a process is waiting, perhaps in a queuewith other processes, for the resources that it needs to enter the running or activestate. A blocked process is waiting for some event to occur, such as completion ofanother process that provides it with input data. A transition from one process stateto another is triggered by conditions such as interrupts and user instructions to theoperating system.

Figure 7.49b shows the state behavior of a typical user process P in a systemwith an independent IOP. It is assumed that P runs on the CPU until an IO instruc-tion is encountered, at which point the operating system intervenes and changes Pfrom running to blocked. P can also be terminated by a timer-generated interrupt,which the operating system uses to limit the amount of time that any one process isassigned to the IOP. In this case P is returned to the ready state, where it remainsuntil rescheduled for execution by the operating system via the CPU. A new pro-cess P' can now be created to run on an IOP and carry out the required IO opera-

tion. Completion of P1 results in an IO interrupt that causes the CPU to transfer Pfrom blocked to ready. At this point f can be deleted if it is no longer needed by Por other active processes. As soon as the CPU is available to execute P, that is.when there are no CPU processes of higher priority ready for execution, P is trans-ferred once more to the running state. It continues running until it encountersanother IO instruction, exceeds its allocated time, or completes execution. In thelast case a call is made to the operating system, which can then delete P.

531

CHAPTER 7

System

Organization

Kernel. An operating system comprises many resource management pro-grams, including processor scheduling routines, virtual memory management rou-tines, and IO device control programs (device drivers). Common utility programs,such as compilers, text editors, and the like, often form part of the operating sys-tem. Thus operating systems tend to contain more software than can fit comfort-ably in main memory. The part of an operating system that resides more or lesscontinuously in main memory and consists of its most frequently used parts istermed the kernel or nucleus. The other, less frequently used parts, such as filemanagement routines, reside in secondary (disk) memory and are transferred tomain memory when needed.

The kernel of an operating system is responsible for the creation, deletion, andstate switching of the many processes that define a computer's behavior. The ker-nel performs its tasks by quickly responding to a steady flow of interrupt requests.These requests have many sources such as user-generated requests for operatingsystem services; CPU-generated process time-outs; memory faults; IO operations:and hardware or software errors. The kernel achieves rapid response by brieflydisabling other interrupts while responding to the current one and then dispatchingor, if necessary, creating a system process to execute the appropriate interrupt-handling routine. The performance and reliability of the kernel can be improvedby implementing its more basic functions in hardware or firmware.

Trap on IOinstruction(START IO)

(a)

Figure 7.49

Process behavior: (a) general case and (b) CPU process in system with iOP.

(b)

532

SECTION 7.210 and SystemControl

The kernel keeps track of each process by means of a data segment called aprocess control block PCB, which defines the most recent execution state or con-text of the process. The PCB typically contains all the programmable registersassociated with a process, including its program counter, stack pointers, status reg-ister, and general-purpose data and address registers. The PCB normally resides inmain memory. When the process is about to be executed, its PCB is transferred tothe corresponding processor registers. The transfer of control from one process toanother {context switching) is implemented by saving the context of the old processin its PCB in memory and loading the PCB of the new process into the processor inits place.

Figure 7.50 shows the PCB used by the VMS operating system for the DigitalVAX computer series. This PCB contains several stack pointers used by the oper-ating system, the CPU's general registers, the program counter PC, and the pro-gram status word PSW. The latter stores CPU status (flag) bits and the interruptpriority level of the process. The last entries in the PCB specify the base addressand length of two page tables: one for the user program and one for the user stack.Page tables play an essential role in the firmware-implemented address mappingthat manages the VAX's virtual memory. Two VAX instructions SVPCTX (saveprocess context) and LDPCTX (load process context) support context switching bytransferring the complete PCB to and from memory, respectively.

An operating system supervises a potentially large set of processes that func-tion asynchronously and concurrently. Many of the more subtle problems indesigning an operating system are due to attempts by concurrent processes to useshared resources in undesirable or improperly synchronized ways. Next we con-sider two basic problems in concurrency control—mutual exclusion and dead-lock—and their solutions.

31

Operating system

and user

stack pointers

General registersR0:13

Program counter PC

Program status word PSW

Page table baseand length registers foruser program and stack

Figure 7.50

Process control block PCB for the VMS

operating system.

Mutual exclusion. Suppose that two concurrent processes Px and P2 shareread and write access to a data region R in main memory. It is generally necessaryto prevent one process from writing into R while the other process is reading fromit. Unless precautions are taken, P2 can modify a variable X of R immediately afterPx has read its old value; in this case subsequent processing decisions will be basedon a wrong value of X. This problem is solved by enforcing certain rules for mutualexclusion so that, in the present instance, Px has exclusive access to R for as longas it needs it, without interference from other processes. Shared resources like R thatrequire mutual exclusion are termed critical.

A software solution to the mutual exclusion problem is to associate with eachcritical resource R a control variable S called a flag that indicates when R is busy.Before attempting to take control of R, a process P first reads R's flag 5. If S = 1(busy), indicating that R is already in use, P does not attempt to use it. If, on theother hand, P finds that 5 = 0, implying that R is available, P immediately sets 5 to1 (busy) and proceeds to access R. When it has finished with R, the process P resetsS to 0 so that other processes can use R.

For the foregoing control mechanism to work, mutual exclusion must beenforced for accessing the flag S to determine the state of R. Some processors pro-vide a test-and-set instruction to implement flag control in the kernel of the operat-ing system. To guarantee mutual exclusion, this instruction is designed to beindivisible in the sense that all the steps of its instruction cycle must be completedwithout interference by other instructions. The 8089 IOP (Example 7.5) has suchan instruction called TSL (test and set while locked). In the following 8089 assem-bly-language code fragment TSL causes the flag 5 to be read from memory andcompared with 0.

R:

TSL S, LWAIT

MOV S, 0

(7.1)

ENDR

If S = 0, TSL writes the specified 1 value into 5 and control is transferred to theroutine R, which uses the resource protected by S. If TSL finds that S * 0, then ittransfers control to the branch address WAIT. To ensure mutual exclusion, TSLactivates a special output signal called LOCK on the 8089 chip. This signal drives abus-lock line of the bus to which the memory storing S is attached; the PCI bus(Example 7.1) has such a line called LOCK. Activating the lock signal preventsother instruction from using the bus while the TSL instruction is being executed;consequently, TSL has the required exclusive access to 5. The final move instruc-tion in the preceding 8089 code implements S := 0 to reset the flag. The routine R isan example of a critical section of an assembly-language program, which is pro-tected by the flag S. If the initial TSL statement is replaced by

WAIT: TSL S, 1,WAIT

(7.2)

then the test-and-set operation is executed repeatedly until 5 becomes available. Ineffect, the process requesting R waits until 5 changes from busy to not busy.

The preceding flag-control mechanism has several deficiencies. It uses a busyform of waiting in which a processor spends a great deal of time simply testing the

533

CHAPTER 7

System

Organization

534

SECTION 7.2IO and SystemControl

flag 5. Moreover, a particular process P may never find 5 = 0 and gain access to Rbecause of competition from other processes. These problems are addressed by aspecial resource control variable S called a semaphore, which is a nonnegativeinteger serving as the control flag for a resource R. It has two indivisible proce-dures, WAIT(S) and SIGNAL(S), which can be defined as follows, where P is theprocess calling WAIT or SIGNAL:

WAIT(S):SIGNAL(S):

if 5> 0 then S := S - 1else suspend P and place it in queue Q for R:

if Q is nonempty then dispatch one process from Qelse S := S + 1;

(7.3)

(7.4)

The semaphore S is used to encapsulate the code R for a critical resource withWAYT(S) and SIGNAL(S) operations thus:

WAIT(S)

R

SIGNAL(S)

(7.5)

and initializing 5 to 1. The first requesting process gains access to R and sets S to 0.Subsequent processes attempting to enter R are queued. Hence only one processcan use the critical region R, thereby ensuring that mutual exclusion is preserved.By initializing 5 to a larger value k > 1, the number of processes in the criticalregion can be limited to k. Although (7.1) and (7.5) are superficially similar, thesemaphore in (7.5) avoids busy waiting; the queueing by WAIT and releasing bySIGNAL of requests for R ensure that all requesting processes eventually get to useR in some sequence—for instance, FIFO—determined by the queueing disciplinefor blocked processes.

Deadlock. Another common synchronization problem in system managementis deadlock; that is, a process is waiting for an event such as the release of a sharedresource, but the event in question never occurs. Suppose that processes Px and P2both require the use of two resources R} and R2 that can only be controlled by oneprocess at a time. Let /?, be allocated to P,, which then requests R2 while stillretaining control of Rv At the same time, let P2 control R2 and be requesting con-trol of/?,. If neither process can continue until it obtains control of both processes,then a deadlock results in which each process ends up waiting for the other torelease a resource, a circular waiting situation that is characteristic of deadlocks. Asingle process can also become deadlocked while waiting for an external eventsuch as an acknowledgment signal that fails to appear in an IO bus transaction.Deadlocks can result from hardware faults as well as from hardware or softwaredesign errors.

Three basic ways to deal with deadlock are prevention, avoidance, and faulttolerance. The prevention approach tries to eliminate the possibility of a deadlockoccurring. Less stringent approaches do not completely eliminate the possibility ofa deadlock, but try to ensure that all potential deadlock situations are avoided. Thethird approach allows deadlocks to take place but provides mechanisms for detect-ing them and recovering from their effects. In practice, all these techniques areused in various parts of a typical operating system, with deadlock prevention tech-niques playing the major role.

For deadlock to exist, the processes and resources involved must meet severalconditions:

1. Mutual exclusion. Each process must have exclusive access to the resources itcontrols.

2. Resource waiting. A process can hold the resources already allocated to it whilewaiting for access to another.

3. Nonpreemption. A process cannot be preempted; it never releases its resourcesuntil it has completely finished with them.

4. Circularity. A circular chain of processes must exist; each process controls aresource that is being requested by the next process in the chain.

Deadlocks are avoided by designing the relevant parts of an operating systemso that one or more of the above conditions cannot occur. Condition 1 usually can'tbe eliminated without severely restricting resource sharing; however, the otherdeadlock conditions can be avoided in various ways. For example, no deadlock canoccur if a process P is blocked until all the resources it needs become available.Blocking circumvents condition 2 (resource waiting), but it leads to inefficient useof available resources. The partial set of resources tied up by P can be freed byrequiring P to release them and rerequest them later along with the other resourcesnot yet available. This latter step preempts P's resources, therefore denying condi-tion 3 (nonpreemption). Eliminating conditions 2 or 3 in this way can cause someprocess requests to be blocked indefinitely. The circularity condition can beremoved by assigning a unique number p(R) to each resource R and enforcing therule that if P holds R, it can only

request additional resources with numbers higherthan p(R). This technique works well if the normal order in which the processesrequest the resources closely matches the order in which they are numbered; other-wise, P may need to acquire and hold low-numbered resources long before it actu-ally uses them.

The detection of deadlock situations, either to avoid them or to eliminate themafter they occur, implies the ability to check for the circular wait condition definedabove. To do so, the system manager must maintain a list of all the resources heldby each process and, for each resource, the names of the processes waiting to use it.These resource assignments and requests can be represented by a resource alloca-tion graph, an example of which appears in Figure 7.51. Here the circles denoteprocesses {/',■}, and the squares denote resources {/?•}. An edge or arrow fromresource /?■ to process P, implies that R. has been allocated to P;, while an arrowfrom Pj to Rj means that P, is requesting Rj. The existence of a closed loop in whichall arrows go in the same direction, in this case, P2^> R4^> P5^> R6^> P4^> R{^>P2, indicates that the given allocation satisfies the circularity condition for a dead-lock. Note that the mutual exclusion condition is satisfied by requiring that onlyone arrow leave each resource in the resource allocation graph.

Figure 7.52 gives a recursive procedure CHECK(P,R) to test for the circularityconditions that lead to deadlock; in effect, it finds closed loops in a resource allo-cation graph. CHECK(P,R) is intended to be executed whenever a process Pmakes a request for resource R; it reports a deadlock if the requested allocationresults in a closed loop. Suppose that the procedure is applied to the system of Fig-ure 7.51 when P2 makes a new request for control of R5. Assume that processesand resources are scanned in ascending numerical order determined by the P and Rsubscripts. On entering CHECK(P2,R5), the resources allocated to P2, namely,

535

CHAPTER 7

System

Organization

536

SECTION 7.210 and SystemControl

Figure 7.51p6) Example of a resource allocationgraph.

procedure CHECK (P: process; R: resource);

begin

for all resources {/?,} allocated to P do

begin

for all processes {P,j} waiting for/?, do

if PQ holds R then REPORT (deadlock) else CHECK (Pp /?);

end;end;

Figure 7.52

Procedure for deadlock detection.

{/?lvr?2,/?3}, are scanned. Then the processes waiting for R{, namely, {P4}, areidentified. Since P4 does not have R5 allocated to it, CHECK(P4,R5) is nowinvoked. On reentering the CHECK procedure with P = P4 and R = R5, theresources {R6} held by P4 are identified. Then the processes {P5} waiting for R6are considered. It is found immediately that P5 holds R5, leading to the conclusionthat a deadlock exists. This deadlock corresponds to the loop P2^> R4~* P5—>R6^P4^R{^P2.

EXAMPLE 7.6 THE UNIX OPERATING SYSTEM [SOUTHERTON 1993]. The goal

of UNIX is to provide a relatively simple, interactive operating system aimed at ageneral-purpose time-shared environment. Simplicity is achieved by keeping the oper-ating system quite small so that it can easily be installed in small computers, especiallyworkstations. The kernel of UNIX consists of about 10,000 lines of source code writtenmainly in C, a programming language developed specifically to implement UNIX. Theuse of C as the source language, and the general availability of its source code, givesUNIX a high degree of portability among different computer types. The functions pro-vided by UNIX for managing processes, IO, and so on, are quite general, which keepsits kernel small and enables UNIX to address a wide range of operating system tasks.UNIX has associated with it a large set of general-purpose programs (utilities), includ-ing compilers, debuggers, and text editors. These utilities, most of which are also writ-ten in C, are considered an integral part of UNIX and have done much to enhance its

popularity. UNIX has a textual user interface called the shell, which provides a com-mand language for process management, as well as access to the UNIX utilities.

UNIX recognizes two main types of processes: system (supervisor) and user.Each active program or user-created task is treated as a user process. When such a pro-cess requires an operating system function because of an interrupt, a system process isinvoked and then becomes the running process. System processes execute in the hostprocessor's supervisor or privileged state, while user processes execute in the nonpriv-ileged user state. (Note that these two processor states have hardware support in manycomputers ranging from the System/360 to the 680X0.) The information associatedwith a process, termed an image in UNIX parlance, consists of the contents of thememory locations used by the process along with the processor status and registerinformation constituting a process control block. The process image is constructedfrom several dynamic segments for instruction, data, and control stack storage. A pro-cess's image resides in main memory while it is being executed, but can be swapped out of memory when the process is inactive oranother process needs the space. UNIX employs a FIFO algorithm to allocate bothmain and secondary memory space.

UNIX makes extensive use of the process concept and has many mechanisms formanipulating processes. The kernel deals with each new task by creating a process tohandle it so that at any time many processes are being executed concurrently. VariousUNIX operations invoked by shell commands exist for managing processes. Figure7.53 lists some representative commands available to the user for process control. Pro-cesses communicate and synchronize their activities by means of events, which typi-cally are control flags set by the occurrence of some specified condition. A process issuspended by instructing it to wait for an event to occur; it is subsequently dispatchedby signaling the occurrence of the event in question.

In a uniprocessing UNIX environment only one process can be executed at a time.Processes are executed in time-shared fashion with each process receiving a slice ofCPU time of no more than a second or so before it is suspended and a new process dis-patched. UNIX assigns a priority number to every process; the number determines theprocess to run next. System processes receive execution priorities based on theirexpected response needs. For example, processes to control disk transfers receive highpriority, while processes that service user terminals receive low priority. User pro-cesses have lower priority than the lowest system-process priority. To ensure reason-ably rapid response, user processes that have received relatively little processor timeare given higher priority than processes that have received a lot of processor time. Pro-cesses with the same priority are run in round-robin fashion. If a suspended process ofhigher priority wakes up, it preempts a running process of lower priority. To preventsome processes from being indefinitely suspended, UNIX increases the priority of pro-cesses that have been ignored for a long time.

A UNIX file is a one-dimensional array of characters (bytes) and is the basic unitfor information storage on secondary memory. Unlike the records found in other file

537

CHAPTER 7

System

Organization

Command

Function performed

fork Create a new (child) process

kill Destroy process

pause Suspend process until a specified event occurs

ps Print status information on active processes

sleep Suspend process execution for a specified time

wait Wait for a child process to terminate

wake Resume a suspended process

Figure 7.53

Some UNIX commands ■

for process management.

systems, UNIX files do not have internal structures. There are no restrictions on thelength or contents of a file as seen by the user. Files are stored physically in pages(blocks) of a fixed size, initially 512 bytes, but larger block sizes are used in laterUNIX versions. UNIX maintains a set of internal tables to keep track of the disk-fileusage.

The logical organization of UNIX files as seen'by a user is that of a tree-struc-tured hierarchy. This structure facilitates both file management and the protection offiles from unauthorized access. Special files called directories store related files; a useraccesses a file by naming the directory that contains it. A directory can contain otherdirectories, leading to the file organization depicted in Figure 7.54. The directory at thehighest level of the tree is known as the root and is denoted by the special name "/".The nondirectory files are at the lowest levels of the tree. The level below the root con-tains major system directories such as bin, which stores the UNIX utilities; dev, whichcontains files used to access 10 devices; and usr, which contains users' files. A file ordirectory is identified by specifying the sequence of directories that contain it, withdirectory names separated by a slash. For example, the file "mail" in Figure 7.54 isreferred to by lusrltomlmail, which is the file's path name. UNIX provides many opera-tions to manipulate files, for example, create, close, copy, open, read, and write.

An unusual feature of UNIX is its extension of the file concept to 10 manage-ment. 10 devices are treated as special types of files, with device-specific IO driverroutines serving to create a filelike interface to UNIX. Hence all IO operations can bemanipulated by file management operations such as open, close, read, and write,which implement START IO, HALT 10, INPUT, and OUTPUT, respectively. Thisapproach makes UNIX unusually independent of the characteristics of the 10 devicesattached to the host system and enhances this operating system's hardware indepen-dence. File concepts are also used for more general interprocess communication. Aprocess can send (write) information to one end of the special queuelike file called apipe, and the information can be received (read) from the other end by a second pro-cess.

Directory

/usr/lerrie

File/usr/lom/mail

Figure 7.54

Organization of the UNIX file system.

7.3

PARALLEL PROCESSING

Computer performance can be increased by executing many instructions simulta-neously or in parallel. This section examines processor-level parallelism in com-puters, focusing on the use of multiple CPUs to achieve very high throughput andfault tolerance.

7.3.1 Processor-Level Parallelism

Although computer performance has increased steadily thanks to faster hardwaretechnologies and processor designs, many important computational problemsremain beyond the capabilities of the fastest current machines [Hwang 1993].Some computer designers believe that processor and memory technologies areapproaching physical limits on their size and speed. Size reductions and speedincreases well beyond present levels are feasible, but their cost may not be accept-able. One way to address these issues is to exploit processor-level parallelism, forexample, by building computers containing large numbers—perhaps hundreds orthousands—of low-cost processors that can work in parallel on common tasks.Suppose that such a computer P(n) is constructed by combining n copies of a single(sequential) computer P(\). If a task T can be partitioned into n subtasks of similarcomplexity and Pin) can be programmed so that its n processors execute the n sub-tasks in parallel, then we would expect Pn to process T about n times faster thanP(l) can process it. In contrast, instruction-level parallelism (section 6.3) aims atspeeding up the single processor P(1) and can increase performance only by a fac-tor of 10 or so.

A further advantage of processor-level parallelism is tolerance of hardwareand software faults. While failure of its CPU is almost always fatal to a sequentialcomputer, a parallel computer can be designed to continue functioning, perhaps ata reduced performance level, in the presence of defective CPUs.

Illustration. Consider the application of parallel processing to the smallnumerical problem of computing the sum SUM of N numbers (constants) bx,b2,...,bN. A straightforward algorithm for solving this problem can be expressed asfollows:

SUM := 0:

for / = 1 to Ndo SUM := SUM + b[i]:

(7.6)

If this summation algorithm is implemented on a conventional computer, N con-secutive add operations, each taking time 7~add, are required. Certain other book-keeping operations are necessary, such as initializing SUM to zero, and theindexing operations implied by the for-do loop. These operations depend onimplementation details and so are often omitted when estimating the overall com-plexity of the computation. Thus N x Tadd serves as a rough indication of the time asequential computer needs to execute (7.6). We now consider in detail a parallelprocessing approach to this problem.

KXAMPLE 7.7 SUMMATION BY \ ONE-DIMENSIONAL ARRAY MlLTIPttO-CESSOR. Consider a hypothetical computer containing n identical processors Pr each

P,

Figure 7.55

One-dimensional (linear) array of n

processors.

of which is a small sequential computer with its own CPU and memory, for example, anetwork of n workstations. The n processors are assumed to be interconnected in thelinear (one dimensional) array configuration depicted in Figure 7.55. Each P: is con-nected by dedicated buses to its left and right neighbors, />,_, and P, + , (where theyexist), and communicates with them by means of two 10 operations called send andreceive. The command send(NEIGHBOR, MESSAGE) causes P, to output some datacalled MESSAGE, typically the result of a computational step, either to Pi_i (whenNEIGHBOR = LEFT) or else to Pi+, (when NEIGHBOR = RIGHT). When P, exe-cutes receive(NEIGHBOR, MESSAGE), it waits for MESSAGE to be sent to it fromthe designated neighbor; then Pi inputs MESSAGE into its local memory, send andreceive can be programmed by message-handling procedures whose implementationdetails are not of concern here. We assume that the processor array also has 10 facili-ties connecting it to the outside world via the right-most processor Pn, as suggested inFigure 7.55. By repeated execution of send and receive, data can be transferredbetween any processor in the array and external devices.

The summation (7.6) can be solved in parallel on this computer as follows: Sup-pose that N = kn, where k is an integer. The TV input numbers to be summed are dividedinto n sets of k numbers, and each set is loaded into the local memory of one of the navailable processors. Every processor is provided with a copy of a summation program,which it executes on its k numbers. Since all processors can operate in parallel, nk addi-tions resulting in n partial sums can be performed in the time required to do k add oper-ations. The partial sums must then be summed to give the final result. We assume thateach processor Pt transmits its result to its right neighbor Pi+i, which then adds thereceived sum to its own sum and transmits the new result to Pi+2- Thus after n - 1sequential summation and data-transfer operations, the final result is stored in Pn.

A program to implement the foregoing parallel summation scheme appears in Fig-ure 7.56. It is placed in the local memory of each processor P, and is executed usingthat processor's particular data set (k of the nk numbers to be summed). Processor Pi

{Each processor Pi computes the sum of its local numbers b[l:£]}SUM:=0

for i: = 1 to k do SUM := SUM + b[i];{Processor P, sends its local result SUM to P2)if INDEX =1 then

begin

if n > 1 then send(RIGHT, SUM);

end else{Every remaining P, waits to receive an external result from P,_,}

begin

receive(LEFT, LEFTSUM);

SUM := SUM + LEFTSUM;

{Each P, except Pn sends its new value of SUM to P,■ + ,}

if INDEX < n then send(RIGHT, SUM);

end,*

Figure 7.56

Parallel summation code forthe machine of Figure 7.55.

also stores a variable INDEX, which is P,'s own address i\ in other words, each P,"knows" its location within the array of processors. Similarly, the processor Pn at theend of the array knows that it has only one neighboring processor, and it interprets theprogram of Figure 7.56 accordingly. The communication between the processors issuch that, on encountering receive, P, waits until P,_, has completed transmission ofits result SUM, which P, then stores internally as LEFTSUM.

The time T(n) needed to execute the parallel summation algorithm on n processorshas two main components. There is a local computation time TL due primarily to thek = N/n sequential additions performed in parallel by each of the n processors. Thistime can be written KxN/n, where Kx is some constant depending on the time needed bythe add instructions and any associated bookkeeping operations. The second compo-nent Tc of T(n) is the communication time to send n - 1 intermediate results from leftto right and the time needed to perform the final n - 1 additions. Tc can be written asK2(n - 1), where K2 is a constant representing interprocessor communication delays.Thus, ignoring minor constant terms, the n-processor execution time is approximatedby

$T(n) = TL+TC = KxN/n + K2\{n - 1\}$

(7.7)

Since K2 measures the time for a slow message-passing IO operation, K2 is much largerthan Kx. Thus the reduction in computation time TL due to increasing the number ofprocessors n is offset by the increase in communication time Tc. Trade-offs of this kindbetween computation and communication times are common to parallel processingtasks. The time for a comparable sequential computer to solve the summation isobtained by setting n to one in Equation (7.7), yielding

$T(l) = TL = KxN$

(7.8)

As expected, the local processing time TL increases by a factor of n, and the interpro-cessor communication time Tc reduces to zero.

541

CHAPTER 7

System

Organization

A problem closely related to the foregoing one is to compute all N partial sumsdefined by the recurrence relation

$X: = x \_, + b$ for i = 1, 2,..., N

(7.9)

Comparing this to (7.6), we see that the latter is designed to compute only onenumber SUM = xN. However, with a small modification, (7.6) and the program ofFigure 7.56 can compute and store the ordered set or vector of N values denoted(*!, x2,..., xN) in place of the single, or scalar, value xN. The relation (7.9) can berewritten as a set of N equations thus:

-x, + x-,

-x2 + x3

= b2= b3

(7.10)

-x

.V-l

+ xv = bx

The solution of these equations is the required vector of N partial sums.

Now (7.10) is a special case of a set of linear equations, which have the follow-ing general form:

542

SECTION 7.3Parallel Processing

01,1*1 +a\,2x2+ ■■■ + al,mxm = b\a2,lx\+a2.2x2+-+a2,nrXm = b2 0711)

Here the a,7's and b,'s can denote either integer (fixed-point) or real (floating-point) numbers, and the x^'s are integer or real variables whose values are to becomputed. Equations (7.11) can be expressed more concisely as

AxX = B

where A denotes the two-dimensional matrix, the operator x denotes matrix multi-plication, and X and B denote (column) vectors. The matrix A can be decomposedinto a set of n row vectors or m column vectors so that the solution of sets of equa-tions like (7.10) and (7.11) is essentially a vector-processing task. Problems of theforegoing type occur frequently in scientific computation, and their regular struc-ture makes them well suited to solution by parallel processing.

Dependencies. The main benefit of parallel processing is faster computation.A price is paid, however, in the need for a significant amount of extra hardware.Roughly speaking, increasing the number of processors by a factor of n makes an/7-fold increase in computing performance possible. In practice, this maximumspeedup is rarely achieved because it is difficult to keep all members of a set ofparallel processors continually working at their maximum rates. Dependenciesamong subtasks can force a processor to wait until other processors supply resultsthat it needs. In the parallel summation algorithm for the linear processor array(Figure 7.56), for instance, the processors must wait for data from their left neigh-bors. The processors in a parallel computer often share resources such as memorybanks, 10 devices, or operating system routines, which can be used by only oneprocessor at a time. A major issue, therefore, in designing and programming paral-lel systems is to avoid conflicts in the use of shared resources. The extent to whichall processors can be kept busy depends on the computer architecture, the tasksbeing performed, and the way in which the tasks are programmed.

Parallel computers are far more difficult to program than sequential ones. Asillustrated by Figure 7.56, the parallel machines require special programming con-structs that allow processors to communicate with one another and to specify com-plex actions like vector operations. Because parallel programming is still poorlydeveloped, achieving an acceptable level of performance requires a costly softwaredevelopment effort, especially when programming for tasks that, unlike Example7.7, have little overt parallelism. Ordinary programs tend to contain significantnumbers of inherently sequential operations that cannot be processed in parallel.As discussed later, even a small percentage of sequential operations can have alarge negative effect on the performance of a parallel computer. Removal of thesesequential features is a major challenge in the design of algorithms, programminglanguages, and compilers for parallel processing.

Classification methods. A processor such as a CPU operates by fetchinginstructions and operands from memory M (main memory or cache), executing the

Processor P   Instruction Memory' M

              streamData

"   stream

Figure 7.57

Instruction and data streams in a sequential

computer.

instructions, and placing the final results in M. The instructions form an instructionstream flowing from M to the processor, while the operands form another stream,the data stream, flowing to and from the processor, as suggested in Figure 7.57.Michael J. Flynn has proposed a broad classification of processor-level parallelismbased on the number of simultaneous instruction and data streams seen by the pro-cessor during program execution [Flynn 1966]. Suppose that processor P is operat-ing at maximum capacity so that its full degree of parallelism is being exercised.Let m{ and mD denote the minimum number of instruction and data streams, respec-tively, that are active. ml and mD are termed the instruction- and data-stream multi-plicities of P and measure its degree of parallelism. Note that mx and mD are definedby the minimum, instead of by the maximum, number of streams flowing at anypoint, since the most limiting components of the system—its bottlenecks—deter-mine the overall parallel processing abilities.

Flynn's classification divides computers into four broad groups based on thevalues of mx and mD associated with their CPUs.

• Single instruction stream single data stream (SISD): mx= mD = 1. Conventionalmachines with a single CPU capable only of scalar arithmetic fall into this cate-gory. SISD computers and sequential computers are synonymous.

• Single instruction stream multiple data stream (SIMD): mx = 1, mD > 1. This cat-egory includes such early parallel computers as ILLIAC IV that have a singleprogram-control unit and many independent execution units.

• Multiple instruction stream single data stream (MISD): m{ > 1. mD = 1. Few par-allel computers fit well in this class. Fault-tolerant computers where severalCPUs process the same data using different programs are MISD.

• Multiple instruction stream multiple data stream (MIMD): m, > 1, mD > 1. Thiscategory covers multiprocessors, which are computers with more than one CPUand the ability to execute several programs simultaneously. An example exam-ined later in this chapter is the Symmetry multiprocessor from Sequent Com-puter Systems Inc.

The foregoing classification depends on a somewhat subjective distinctionbetween control (instructions) and data. It is also essentially behavioral in that itsays nothing about a computer's structure. We turn next to some ways of classify-ing parallel computers based on their interconnection structure. Every computer

consists of a set of n > 1 processors (CPUs) P, P2 P„, and m > 0 shared (main)

memory units A/,, A/,,...., Mm communicating via an interconnection network N, asillustrated in Figure 7.58. For simplicity, we do not consider IOPs or 10 devices inthe classification process. In a conventional SISD computer n - m - 1, and N is thesystem bus over which processor-memory communication takes place. The mem-ory units then constitute a global main memory that provides a convenient messagedepository for processor-to-processor communication. A system with this organi-zation is called a shared-memory- computer.

543

                        Processors   Memories

Pi   Pi Pn   W, M2 Mm

  Interconnection network A'

Figure 7.58

General structure of a computer with n processors and m

memory units.

A global shared memory can be a serious bottleneck, particularly when theprocessors share large amounts of information, since normally only one processorcan access a given memory module at a time. If the processors have their ownlocal memories, then the global memory can be reduced in size, or even elimi-nated completely. To separate the functions of processing (computation) andmemory, we will refer to a CPU or IOP with no associated main memory, but withother temporary storage units such as register files and caches, as a processing ele-ment or PE. A processor is then the combination of a PE and a local memory; itcan also include 10 facilities forming, in effect, a self-contained computer. In asystem with little or no global memory, the processors communicate via messagestransmitted between their local memories, as in the system of Figure 7.55. In thiscase the main memory is the sum of the local memories, and the system is referredto as a distributed-memory computer. The term message-passing computer is alsoused for such machines. Figure 7.59 illustrates the main structural differencesbetween shared-memory and distributed-memory computers.

The internal structure of the interconnection network N is also used to classifyparallel computers. A selection of interconnection topologies appears in Figure7.60. Because of the ease with which it can be designed and controlled, the singleshared bus (Figure 7.60a) is widely used in parallel as well as sequential comput-ers. When n, the number of PEs, and m. the number of memory units, are large,very fast buses are required, and special design precautions must be taken to mini-mize contention for access to the bus. Bus contention can be relieved (but not elim-inated completely) by providing several independent buses. The crossbarinterconnection network of Figure 1.60b is a special kind of multiple-bus system inwhich each PE has a (horizontal) bus linking it to all memories, or equivalently,each memory has a (vertical) bus linking it to all PEs. Annxm crossbar allows upto min{n,m} bus transactions to take place simultaneously. However, in the worstcase where all the processors attempt to access the

same memory unit Af, simulta-neously, the number of bus transactions drops to one. Although crossbar networkshave often been employed in computer systems, their hardware complexity quicklybecomes very high as m and n increase.

Figures 7.60c and 7.60a1 illustrate networks that use high-speed, dedicated con-nections (uni- or bidirectional buses) to link the system components, each of whichis an independent processor with its own memory and a small group of neighbors.The neighboring processors are physically close and cooperate in the processing ofcommon tasks. They communicate with one another via send and receive IO oper-ations of the type discussed in Example 7.7. While neighboring processors can

Processing elements

Processors with local memories

M,

PE,

PE?

Interconnectionnetwork N

CHAPTER 7

System

Organization

(a)

Figure 7.59

(a) Shared-memory and (b) distributed-memory computers.

(b)

communicate rapidly via their dedicated bus links, communication between non-neighboring processors is slower and requires intermediate processors to act asstore-and-forward message-transfer stations. For example, to transmit data D fromPooo to P011 in Figure 7.60c requires the following two steps: First send D fromPqoo to a neighbor processor such as P00l and store D there temporarily. Then sendD from PQOl to its neighbor P011.Various interconnection structures other thanthose of Figure 7.60 have been proposed for parallel computers, but few have beenimplemented commercially.

The computer structure in Figure 7.60c is an n-dimensional hypercube, alsocalled a {binary) n-cube. It contains 2" processors, each of which is connected to nimmediately adjacent (neighboring) processors. In the example n = 3, so eight pro-cessors are used, and the cubelike interconnection structure is clear. If each proces-sor is indexed by an n-bit binary address as shown in Figure 7.60c, then P, is aneighbor of P} if and only if their addresses i and j differ by one bit. The intercon-nection structure of Figure 7.60d is that of a tree, in this case a binary tree, becauseeach processor (except those in the bottom row) is connected to two processors—its "children"—in the row beneath. The name tree derives from the fanciful resem-blance of Figure 7.60<i to an upside-down tree in which processor P{ x is the "root,"and processors Ppy-PpjP~x ^Q the "leaves." This binary tree computer contains n =2P- 1 processors, so the number/? of levels of the tree is approximately log2«. Asin the hypercube case, communication between neighboring processors (a parentand a child) is fast, while communication between nonneighboring processors ismuch slower.

Like most multiprocessors with specialized interconnection structures,-treecomputers are well suited to certain kinds of parallel processing. Consider again

p,

(a)

SystemJ bus

M,   M2 • • • Mm

Pi

p,

...   ...

Pn

I ~r 0   K   \» T n te K onnection netwc r c

Processor

Dedicated

interprocessor

bus

(b)

(c)

Rootprocessor

Pu   Dedicated

interprocessorbus

P2.2

/

• • •

Pp-u   W

Leafprocessors

/   \   /   \

pp+   PP.2   V2 -1   pP,r

(d)

Figure 7.60

Interconnection network structures: (a) single bus; (b) crossbar; (c) hypercube (3-cube);(d) tree.

the summation problem

$$SUM = b_x + b_2 + \ldots + b_N$$

where $N = 2P\sim{}'$. It can be solved by the following tree-oriented parallel algorithm:Load the input operands $b_l, b_2, \ldots, b_N$ into the $2P\sim{}'$ leaf processors of the binary tree(Figure 1.60d). Then for each pair b_j and b_{J+}, stored in the children of some level-(p - 1) processor $P\{i$, transfer b_j and b_{j+} : to Pp_Xi, compute the sum $y_7 = b_j +b_{j+l}$, and store it in the parent processor $Pp.ij$. This reduces the number of operandsto be added in half, and all are now stored in level-(p - 1) processors. These N/2operands are then added in parallel by the processors in level p - 2, and so on.Eventually, the final result SUM is computed by, and stored in, the root node PlvThe entire summation process requires $p - 1 \sim \log_2 N$ addition times.

We can further distinguish computers on the basis of the unit-to-unit connec-tion paths provided by their interconnection networks. These paths can be static,that is, fixed and unchangeable, or dynamic, that is, reconfigurable under systemcontrol. The single-bus and crossbar interconnections of Figure 7.60 arc examplesof dynamic interconnection networks, whereas the hypercube and tree have staticinterconnections. The system bus (Figure 7.60a) allows any of the n processors toconnect to any of the m memories for one or more bus cycles, for example, to fetchan instruction. In a subsequent cycle some other processor-memory pair can usethe bus, so the communicating bus units vary dynamically. In contrast, each pro-cessor in the binary tree (Figure 7.60J) has dedicated buses to its nearest neighborsand must communicate with other processors indirectly.

It is clear from the preceding discussion that the same computer can often beclassified in several ways, depending on the aspects of its parallelism that are sin-gled out for attention. A computer in the nCUBE series, for example, can be calleda (distributed memory) multiprocessor, an MIMD computer, a hypercube com-puter, or a (massively) parallel computer.

CHAPTER 7

System

Organization

Performance. The performance of a parallel computer depends—often incomplex and hard-to-define ways—on the parallelism inherent in its architectureand the programs it executes. Several basic performance measures encounteredearlier in the context of pipelining (section 5.3.2) also apply to processor-level par-allelism. An example is the speedup S(n) defined by the ratio of total executiontime 7(1) on a sequential computer to the corresponding execution time T(n) on thecomputer whose degree of parallelism is n.

$$S(n) =$$

$$7X1)T(n)$$

(7.12)

In Example 7.7, where N numbers are summed by an ^-processor array, T(n) and7/(1) are defined by Equations (7.7) and (7.8), respectively, yielding the speedupformula:

$$S(n) =$$

$$k,n$$

$$K_lN/n + K_2(n- 1)\, l + Kn/k$$

Here $K = K_2/K_l$ is a system constant, and $k = N/n$. If the interprocessor communica-tion delays are ignored by setting K_2 to zero, then S(n) becomes /?, which is obvi-ously the maximum speedup achievable with n processors. On the other hand, if K_2

548 is large relative to /;, it is possible for S(n) to become less than one, in which case a

n . single sequential processor with no interprocessor communication requirements is

Parallel Processing faStef than an "-Pressor System!

A related performance measure expressed as a single number (a fraction or apercentage) is the efficiency E(n), which is the speedup per degree of parallelism,and is defined as follows:

$$E\{n\} = \sim Y \quad (7\text{-}13)$$

E{n} is also an indication of processor utilization and may be so named. In general,speedup and efficiency provide rough estimates of the performance changes thatcan be expected in a parallel processing system by increasing the parallelismdegree n—by adding more processors, for instance. These measures should be usedwith caution, however, since they depend on the programs being run and canchange dramatically from program to program, or from one part of a program toanother.

The influence of program parallelism—or the lack thereof—on performancecan be seen from the following analysis. Suppose that all computations of intereston a parallel processor are divided into two groups involving arithmetic operationsonly: vector operations employing vector operands of some fixed length TV and sca-lar operations where all operands are scalars ($N = 1$). Let F be the fraction of allfloating-point operations that are executed as scalar operations, and let 1 - F be thefraction executed as vector operations. Hence 1 - F is a measure of the degree ofparallelism in the programs being executed and varies from one, corresponding toall-vector operations, to zero (all-scalar operations). Suppose that vector and scalaroperations are performed at throughput rates of b_v and b_s, respectively. Let theaverage system throughput be b in suitable units such as MFLOPS (millions offloating-point operations per second). Then b, b_y, and b% are related by the follow-ing useful formula:

b bs bv

The execution time for a single TV-element vector operation is Tv = N/bv, while thatof a single scalar operation is Ts = \/bs. These parameters are related by

NT.

o

where T0 is some fixed setup time that is independent of vector length and n is thecomputer's parallelism degree. When TV is large, T0 can be ignored so that thisequation reduces to Tv = N TJn. Substitution into Equation (7.14) yields

b = nb> (7.15)

l + (/i-l)F

Since b%, the scalar throughput, and n, the processor parallelism, can be taken to beconstants, Equation (7.15) defines b as a function of F.

Suppose for example that bs = 10 MFLOPS and n = 100. Equation (7.15) thenbecomes b = 1000/(1 + 99f). The maximum performance of 1000 MFLOPS occurswhen F = 0, that is, when there are no scalar operations. When F = 0.01, in other

1000

Throughput

b(MFLOPS)

0.2 0.4 0.6 0.8

Fraction F of nonparallelizable operations

1.0

Figure 7.61

Illustration of Amdahl's law for n = 100.

CHAPTER 7

System

Organization

words, when 1 percent of the computations are scalar, b drops from 1000 toapproximately 500 MFLOPS, thus cutting the throughput in half. Increasing F to0.1 or 10 percent reduces b to less than 100 MFLOPS, an order of magnitude dropin performance; see Figure 7.61.

This analysis suggests that the performance of a highly parallel computer isvery sensitive even to small numbers of nonparallel (sequential) operations, a con-clusion that has been verified experimentally for many types of parallel machines.Hence it is often worthwhile to devote considerable effort to "parallelize" pro-grams for such machines to eliminate sequential operations. If we take the speedupS(n) to be bib , then (7.15) can be rewritten as

S(n) =

l + (n-l)F

(7.16)

With F interpreted broadly as the fraction of nonparallelizable operations orinstructions, then Equation (7.16) is often referred to as Amdahl's law, after GeneM. Amdahl, one of the architects of the IBM System/360.

Besides the presence of nonparallelizable code, there are several other reasonswhy a computer with n independent processors rarely achieves a speedup of n.These reasons include inefficiencies in task distribution (load balancing) amongthe available processors and contention for access to shared system resources,especially memory and interconnection networks. It has been conjectured that thespeedup typically achievable with n processors in a multiprocessor system rangesfrom log2 n to nf\oge n (see problem 7.30).

An indication of the influence of contention for shared memory on perfor-mance can be obtained by considering a system containing n processors P,,P2,-.,Pn connected to m shared memory units MxMz,---Mm via a crossbar or

SECTION 7.3Parallel Processing

n = 4

n = number of
processors ^^" --"^ « = 3

-

- B = 2

c

- B= 1

i i i 1 1

1 -

12 3 4 5 6

Number of memory units m

Figure 7.62

Performance of a shared-memory multiprocessor.

similar interconnection network, as in Figure 1.60b. All programs and data used by the processors are stored in the m-unit global memory and are accessed via the crossbar network. It is reasonable to assume that the instruction or data bandwidth b of a processor P is proportional to the rate at which P_i accesses memory. The latter is, in turn, proportional to the average number of busy memory units B. Suppose further that the probability of any processor P_i generating a request to Mi is $1/m$ in other words, the memory requests are distributed uniformly. Hence the probability that Mj is idle, and therefore free to respond to memory requests, is $(1 - 1/m)^n$ the probability p_i that A/, is busy is $1 - (1 - 1/m)^n$. If Mi is busy when a new request for access to it is received, that request is not serviced until A/, becomes free again. The average number of busy memory units B is therefore given by

$$B = \sum_{i=1}^{m} =$$

1

m

(7.17)

As might be expected, if m is fixed and n approaches infinity (n —> °°), then B —> m. Similarly, if n is fixed and m —> <», then Equation (7.17) implies B —> n; that is, all processors become busy. Figure 7.62 plots B against m for some small values of n. From this analysis we see that we can improve the performance of a multiprocessor by placing information that is frequently accessed by f- in a local memory assigned to P: while limiting the use of global memory to the storage of infrequently shared programs and data.

7.3.2 Multiprocessors

A multiprocessor is an MIMD computer containing two or more CPUs that cooperate on common computational tasks. Multiprocessors are distinguished from multicomputers and computer networks, which are systems with multiple CPU operating largely independently on separate tasks. The various processors making up a multiprocessor typically share resources such as communication facilities, 10 devices, program libraries and databases and are controlled by a common operating system.

Motivation. The main reasons for including multiple CPUs in a computer system are to improve performance and reliability. Performance is improved either by distributing the computation of a large task among several CPUs or by performing many small tasks in parallel using separate CPUs A multiprocessor with n identical processors can. in principle, provide n times the performance of a comparable SISD system or uniprocessor. A major goal, therefore, in designing an /i-CPU multiprocessor is to achieve a speedup Sin) as close to n as possible. By enabling such resources as secondary memory to be shared, a multiprocessor can reduce overall system costs. Many multiprocessors also have the advantage of scalability; that is. the system size can be increased incrementally by adding processors to meet growing computation needs. Scalability is facilitated by making all CPU identical and allowing each to execute either operating system (kernel; or user code: multiprocessors with these properties are said to be symmetric. Finally, system reliability is improved by the fact that the failure of one CPU need not cause the entire system to fail. The functions of the faulty CPU can be taken over by the other CPUs: consequently, multiprocessors enable fault tolerance to be incorporated into the system.

As discussed earlier, multiprocessors are classified by the organization of their memory systems (distributed memory and shared memory) and by their interconnection networks (dynamic or static;. Shared-memory and distributed-memory multiprocessors are sometimes referred to as tightly coupled and loosely coupled. respectively, reflecting the speed and ease with which they can interact on common tasks. Multiprocessors are also classified by the number of processors they contain: Massively parallel machines can contain thousands of processors. Most multiprocessors, however, are modestly parallel, containing from 2 to about 30 processors: such multiprocessors have existed since the 1960s. The relative success of multi-processors with a few CPUs stems from the difficulty of programming large numbers of CPUs to cooperate efficiently. The lack of standard, widely used languages and application packages for parallel programming has been a major obstacle to wider use of multiprocessors.

Shared-bus systems. Most commercial multiprocessors have been built around a single shared system bus B because of B's relative simplicity and low cost. The CPUs, memory, and 10 units are attached directly to B and time-share its communication facilities. Only one pair of units can use B at a time, either for CPU-memory or IO-memory communication. The memory units and 10 devices on B are global to all the processors: hence single-bus multiprocessors are of the shared-memory class. If the access time to the shared memory is the same for each processor, the multiprocessor is said to be of the uniform-memory access i I'M A > type.

The global bus B is clearly a communication bottleneck in shared-bus multiprocessors, leading to contention and delay whenever two or more units request access to main memory. In practice, memory contention limits to about 30 the number of CPUs that can be included in the system without an unacceptable degradation in performance. Figure 7.63 shows that a single-bus multiprocessor's performance can be improved by supplying each CPU with a local bus. The local bus is connected to a local memory unit that contains part of the shared address space: it can also support a local 10 subsystem, as illustrated in Figure 7.63. This system configuration removes most of the routine memory traffic from B so that it can be

Global memory

CPU,

CPU,

CPU„

Local memory Local 10

Local memory Local 10

Local memory Local IO

Global resources

Global (system) bus B

Processors

Local buses

Local resources

Figure 7.63

Shared-bus multiprocessor with global and local resources.

reserved primarily for interprocessor communication. Many microprocessor families can be configured as multiprocessors in this way. The Intel Pentium, for example, was designed for use in shared-bus multiprocessors, with standard buses like the PCI bus serving as local buses.

Despite its relative simplicity, the shared-bus architecture exhibits some of the basic synchronization problems common to all multiprocessors. Consider the situation in which two CPUs share a region R of global memory where mutual exclusion (section 7.2.3) applies; that is, only one processor should have access to the shared region at a time. Access to R is conveniently controlled by a semaphore (flag) F that indicates whether R is currently being used by some other process (F =1) or is available for use by a new process (F = 0). Before it attempts to access R, a CPU first reads F, which must be stored in global memory. If F = 0, the CPU then changes F to 1 and proceeds to use R. If it finds that F is already 1, then it does not attempt to use R. The mutual exclusion requirement can be violated if it is possible for two CPUs to independently access the semaphore at the same time and find F =0. This violation can occur if a second processor CPU2 can read F after the first processor CPU! has read it, but before CPU, has changed F to 1. The problem lies in the fact that semaphore flag test-and-set instructions issued by the CPUs can be broken down into interleaved bus cycles as

follows:

Global bus cycle Action

i   CPU, fetches semaphore F = 0.

i+l   CPU2 fetches semaphore F = 0.

i + 2   CPU, sets F to 1.

i + 3   CPU2setsFto 1.

At time i + 4, both CPU, and CPU2 assume they have exclusive control over thecritical region R, with potentially catastrophic consequences. A solution to this

problem, which is discussed in section 7.3.1, is to allow semaphore test-and-setinstructions to have exclusive control of the system bus while they are being exe-cuted. Such instructions lock the bus until their execution is complete, therebydelaying any test-and-set instructions awaiting execution by other CPUs until thefirst CPU has safely set the semaphore to busy.

553

CHAPTER 7

System

Organization

EXAMPLE 7.8 THE SEQUENT SYMMETRY SHARED-BUS MULTIPROCESSOR

[sequent 19 96]. The Symmetry multiprocessor series is built around a high-speedshared bus, multiple CPUs from the Intel 80X86/Pentium series, and the UNIX operat-ing system. Symmetry multiprocessors can be variously characterized as MEMD,shared memory, tightly coupled, scalable, symmetric, and UMA. They are typically-used in applications such as on-line transaction processing characterized by heavycomputation requirements and a need for high reliability.

The Symmetry 5000 system introduced in 1995 has the general organizationdepicted in Figure 7.64. It contains from 2 to 30 Pentium CPUs each with a 2MB cache:there are no other local memories. The CPUs are packaged two per circuit board; thesystem can be expanded to the maximum allowed by adding CPU boards. The main-memory system is also packaged in circuit boards that facilitate modular expansion. Thememory is interleaved (section 6.1.2) to increase performance, and an error-correctingcode improves reliability. The 10 subsystem includes one or more high-speed IO pro-cessors designed to communicate with magnetic disk and tape memories via high-speedSCSI buses. Additional, slower IO controllers support other IO devices, as well as var-ious standard external interfaces and communication protocols such as Ethernet.

A key component of the Symmetry 5000 is its proprietary system bus that linksall processors, memory units, and IO controllers. This Highly Scalable Bus (HSB)contains a 64-bit data-address bus designed to transmit (in multiplexed mode) 64-bit

Interleavedmemory banks(up to 3.5 GB)

Global

main

memon-

System bus(240 MB/s)

Two to 30Pentium-based CPUs

Secondarymemory

SCSI buses

IO

processors

IOcontrollers

YME bus

Ethernet

FDDI optical bus

Additional IO devices

Figure 7.64

Organization of the Sequent Symmetry 5000 multiprocessor.

554 data words and 32-bit addresses. It has an unusual "pipelined" data transmission mode

that supports a simplified form of package switching, which allows memory and 10data to be transmitted in bursts whose transmission can be overlapped. The HSB was

Parallel Processing designed for a maximum data bandwidth of 240 MB/s.

The Symmetry's operating system, DYNEX, is a version of UNIX with enhance-ments to support multiprocessing. Each CPU acts like a uniprocessor that is executingindependently under UNIX supervision; it executes processes from a list that all theCPUs share. Interrupt signals are generated at periodic intervals to force the CPUs toexamine the list of waiting processes and schedule a high-priority process for execu-tion. This approach forces all CPUs to share the system's workload. To avoid conflictsamong CPUs when executing kernel routines stored in the shared memory, a sema-phore mechanism of the kind discussed earlier enforces mutual exclusion.

Cache coherence. In shared-bus multiprocessors like the Symmetry, cachesplay a vital role in reducing the contention for the shared system bus. Withoutcaches, connecting more than two or three CPUs to the same bus might be imprac-tical. Typically, each CPU has a private one- or two-level cache, which forms alocal memory and allows the CPU to access data and instructions without using thesystem bus. With an independent cache in each CPU, the possibility exists for twoor more caches to contain different (inconsistent) versions of the same informationat the same time; this is the cache-coherence problem. This problem is alleviated,but not solved, by using write-through, which, as discussed in section 6.3, causesboth the cache and main (global) memory to be updated whenever a memory writeoperation occurs. Suppose, for example, that one CPU updates variable X in bothits cache and the global memory. If another CPU then changes X, the new value ofX will be written into main memory, but the two caches will contain different val-ues for X. Subsequent reads from these caches can lead to inconsistent results. Thusto ensure coherence we need a mechanism that informs each cache about changesto shared information stored in other caches.

We can solve the cache-coherence problem with either hardware or software.One software-based solution is to mark (tag) information during program compila-tion as either cacheable or noncacheable. All writable shared items are marked asnoncacheable, meaning they can be accessed directly only from main memory. Awrite-through policy that requires a processor to mark a shared cache item X asinvalid, or to be deallocated, whenever the processor writes into X can then ensurecache coherence. When the processor references X again, it is forced to bypass thecache and access main memory, thereby always acquiring the most recent versionof X. This approach can significantly degrade system performance, however. Inval-idation also forces the removal of needed data from the cache, thus increasing itsmiss ratio, which, in turn, increases the main-memory traffic.

Hardware-based methods of maintaining cache coherence offer the advantagesof higher speed and program transparency, but they tend to be expensive. One pos-sible

approach is for a processor to broadcast its write operations to all caches and the global memory via the shared bus. Every cache controller in the system then examines its assigned addresses to see if the broadcast item is presently allocated to it. If it is, the cache block (line) in question is either updated or marked as dirty (modified). The drawback of this technique is that every cache write forces all caches to check the broadcast data, making the caches unavailable for normal pro-cessing.

A related, but less costly, hardware-based method known as cache snooping 555 equips each CPU with circuitry to continuously monitor or "snoop" on system-bus activity in order to detect references by other processors to memory addresses cur-rently in its cache. The CPU can also signal other CPUs that it has a copy of the referenced item and, when necessary, modify or delay the other CPUs' main-mem-ory accesses. If CPU2 attempts to read (write) memory data with an address that is currently assigned to CPU,'s cache, CPU, detects this attempt in what is called a snoop read (write) hit by CPU,. On making a snoop hit, CPU, determines whether actual or potential incoherence exists and then takes appropriate steps to eliminate it. The following courses of action are typical:

• Suppose that CPU! makes a snoop read hit when its cache copy of the requested item is dirty and it has not yet updated main memory—this situation can occur only when the write-back policy is used. CPU, signals CPU2 to suspend its read request while CPU, updates main memory by writing back the block containing the requested word. Then CPU! signals CPU2 to complete its memory read oper-ation.

• If CPU, makes a snoop write hit, it knows that its own cache copy of the requested item is about to become dirty. It therefore marks that copy as dirty. Hence the next time CPU, tries to read the item in question, a cache miss occurs that forces CPU, to read a valid copy from main memory.

An alternative response to a snoop write hit by CPU, is for CPU, to capture the new data on the system bus as CPU2 writes it to global memory. CPU, can then use the captured data to update its cache.

EXAMPLE 7.9 THE MESI CACHE COHERENCY PROTOCOL [MOTOROLA

1994; anderson and shanley 1995]. To maintain consistency in a multipro-cessor, or in a uniprocessor with independent 10 processors, a cache controller must keep careful track of the state of each cache block (line) under its control. It does so by attaching a few state bits to every block stored in the cache data memory and process-ing the states according to some coherence algorithm or protocol, as it is often called. Microprocessors such as the Pentium and some PowerPC models employ a standard cache coherence protocol based on the following four states:

• M (modified): The block has been modified or "dirtied" by a recent write hit to the cache.

• E (exclusive): The block is "clean," that is, the same as the copy in main memory, and no other processor has a copy.

• 5 (shared): The block is clean, but other processors may have a copy.

• / (invalid): The data in the block is not valid.

A cache-control algorithm using these states is known as the MESI coherence protocol for obvious reasons. Figure 7.65 gives a slightly simplified version of the MESI proto-col, which shows how the states of a cache block change in response to various read and write conditions, assuming that a write-back policy and a cache-snooping mecha-nism are used. We also assume a one-level cache, although this protocol works equally well with multiple cache levels.

First consider the effect of read operations on the state of a cache block. Read hits to the block leave its state unchanged. Read misses, however, are not so simple. When a processor P, first tries to read the (empty) cache, the cache controller changes all block states to / (invalid) and forwards the read request to main memory. Thus / acts like a reset state that triggers a block transfer to the cache: the incoming block's state is

556

SECTION 7.3 Parallel Processing

Invalid

Snoop write hit

Shared

Read hit Snoop read hit

Modified

Exclusive

Figure 7.65

State-transition graph (simplified) for a cache block using the MESI coherence protocol.

set to E (exclusive) if no other processor has a copy of the same block. (An initial write also brings into the cache a block whose state is marked E.) If during P{s read opera-tion, a snooping processor P2 signals via the shared bus that its cache has a clean copy of the same block, in which case no incoherence exists, the state of the block in P,'s cache is set to 5 (shared) instead of E. If, on the other hand, P2 signals that its cache has a dirty (modified) copy of the same block, the caches are no longer coherent. To resolve this incoherence, the signal from P2 causes Px to postpone its memory read and to relinquish the system bus. P2 then assumes the role of bus master and writes its mod-ified block back to main memory. P2 also changes the state of its copy of the cache block from E to 5 because it now knows that the block in question is shared. This state change is specified by the transition from £ to 5 marked "snoop read hit" on the right side of Figure 7.65. Finally, the first processor Px repeats its main-memory read request and obtains a clean copy of the block, which it marks as 5.

Now consider the cache block's state when P, addresses a write hit to it. If the tar-get block is in either of the clean states 5 or E, the block's state changes to M (modifier or dirty). In the S case Px signals the other processors that it is writing to a shared block; they respond by marking their copies of the shared block / (invalid). The modi-fied cache block remains in the M state in Px during subsequent reads and writes to it, unless P,'s own snooping detects read or write hits addressed to the same block in other caches.

A write miss by P, triggers a memory read operation that replaces the target block in the cache, where it is eventually marked M. If some other processor P2 has a clean (S or E) copy of the same block, P2 changes the state of its copy to /. If P2 has a dirty (M) copy of the block in question, P2 sends a signal to this effect to Px, causing the latter to delay its memory read. P2 then takes control of the system bus and writes its modified block to main memory; P2 also changes the state of its cache copy from /, since it knows that the copy of the shared block in main memory is about to be changed by Pt. Control of the bus is then returned to />,, which completes its block transfer.

Message-passing computers. As developments in VLSI technology during the 1980s ushered in powerful one-chip microprocessors and memory (RAM) chips with capacities in the multimegabit range, it has become feasible to build massively parallel multiprocessors, with hundreds or thousands of processors. Multiprocessor architectures with distributed memory systems, where interproces-sor communication is by message-passing, avoid most of the contention problems inherent in the use of single shared memories and buses. Such computers can pro-vide extremely high performance, but they also pose problems in algorithm and program design that are far from being satisfactorily solved.

Various static and dynamic interconnection structures have been proposed for massively parallel multiprocessors. Static structures like hypercubes and trees are easier to build and control when many processors are involved. Dedicated buses or 10 communication lines typically serve as interprocessor links. Neighboring pro-cessors can then interact at the maximum possible rate, with little interference from other processors. Interconnection networks are selected to trade hardware cost for communication speed in some class of applications. The hypercube structure achieves a good balance between these parameters. Consequently, it has been used in several commercial computers of the massively parallel type [Hayes and Mudge 1989].

An n-dimensional hypercube computer is characterized by the presence of 2"nodes, each consisting of a processor and its local memory. Each processor P, has direct links to n other processors (its neighbors); these links form the edges of the hypercube. A set of 2" distinct n-bit binary addresses can be assigned to the pro-cessors in such a way that P,'s address differs from each of its neighbors in exactly 1 bit; Figure 7.60c illustrates hypercube addressing for n = 3. Hypercubes have several attractive features:

• A hypercube can be expanded or scaled up while maintaining a good balance between the number of nodes and the cost of internode communication. As n is incremented by one, the number of nodes doubles, but the node degree and the maximum internode distance both increase only by one (from n to n + 1).

• A hypercube is homogeneous in that the system appears the same when viewed from any of its nodes. This feature simplifies programming because all nodes can execute the same programs on different data when collaborating on a common task.

• We can embed other useful interconnection structures, such as rings and meshes, efficiently in the hypercube. We say that (graph) G is embeddable in H if and only if every node in G can be mapped into a distinct node in H such that all nodes that are neighbors in G are also neighbors in H. In other words, G is embeddable in H if we can find an exact (isomorphic) copy of G inside H.

• A large hypercube can support multiple concurrent users with each user program assigned to a private embedded hypercube or subcube that is disjoint from other users' subcubes. For example, in a four-dimensional hypercube (Figure 1.66b), four-node subcubes can be assigned to two users, and an eight-node subcube can be assigned to a third user.

Embeddability can be used to compare different interconnection structures for multiprocessors. Let Cx with (static) interconnection network Nx and C2 with interconnection network N2 be computers employing similar processors. If Nx is

557

CHAPTER 7

System

Organization

558

SECTION 7.3 Parallel Processing

0011

Q Q Q O

0001

0111

0101

mi

1101

1011

0 0 0 O

1001

O O O O

0000 0100 1100 1000

(a)

0010

0000

0001

1000

1001

(*)

Figure 7.66

(a) A 3 x 4 mesh and (b) embedding the mesh in a four-dimensional hypercube.

embeddable in a sufficiently large version of N2, then C2 will be able to embed C,. Therefore, any structure embeddable in Cx is also embeddable in C2, and C2 is at least as powerful as Cx from a static structural viewpoint. Referring to Figure 7.11, it is obvious that any /c-node system can be embedded in a system of k or more nodes with the structure of a complete graph (Figure 7.11/). A sufficiently big mesh-structured system can embed any path or ring. It cannot, however, embed a hypercube, since for n > 4, every node of an ^-dimensional hypercube has greater degree than every node of the mesh. A hypercube can embed both the ring and the star structures. Less obvious is the fact that a mesh can be embedded in a hyper-cube. An embedding of the 12-node 3x4 mesh into the 16-node four-dimensional hypercube appears in Figure 7.66. Heavy lines show the nodes and edges of the hypercube that correspond to those of the mesh.

EXAMPLE 7.10 THE nCUBE HYPERCUBE MULTIPROCESSOR [HAYES AND

mudge i989;nCUBE 1990]. Hypercube multiprocessors were proposed as early as 1962 at the University of Michigan, but the first working machine was not demon-strated until the completion of the six-dimensional (64-node) Cosmic Cube computer at Caltech in 1983. Influenced by this work, several commercial hypercube computers

were introduced in the mid-1980s, including Intel's iPSC series and the nCUBE (then written NCUBE) series developed by nCUBE Corp. The original nCUBE 1 family included hypercubes of various sizes up to a 10-dimensional (1024 node) machine. Subsequent nCUBE computers increased the number of nodes to $8192 = 2^{13}$.

An nCUBE processor node is equipped with a set of high-speed 10 channels, each consisting of a serial input line and a serial output line. One channel connects to a host or front-end computer; the remaining channels connect the node to its neighbors in the hypercube. Processor-to-processor communication is implemented by transmitting messages between buffer areas in the local memories of communicating nodes. Each interprocessor link has both an address register pointing to its message buffer area and a count register indicating the number of bytes to be sent or received. Once a processor initiates a message transfer, the processor can continue with other tasks while the inter-processor message transfer proceeds as a DMA operation between the memories of the communicating nodes. A broadcasting instruction is also supported that allows the same data to be transmitted to all processors in the hypercube; see problem 7.38.

First we consider interprocessor communication in an nCUBE 1 system. Assume that an n-dimensional subcube is assigned to the user and that the message source and destination nodes have the binary addresses S = ^,_,...s150 and D = dn_l...dld0, respectively. The EXCLUSIVE-OR function R = S © D = rn_v..rlr0, where r, = s, 0dt for i = 0,1,...,n - 1, controls the routing process. The values of i for which $r_t = 1$ indicate the dimensions of the hypercube to be traversed by a message en route from source to destination. The operating system kernel residing in each node that receives the message reads the destination address D (a field in the message header); computes R = P © D, where P is the address of the current node; and scans R from left to right until it encounters some r■ = 1. Node P then forwards the message to the neighboring node P" whose address differs from P' s in the _/'th bit. If R = 0, then P - D and P recognizes itself as the destination node and proceeds to process the message. Thus in a six-node subcube of the nCUBE 1, a message being sent from node 7 to node 45 passes through nodes with the following sequence of addresses:

5 = 000111 -» 100111 -» 101111 -» 101101 =D

This store-and-forward routing method sends each message along a shortest path so that the minimum number of intermediate nodes relay messages between the source and destination. In the nCUBE 2 computer, each node P contains a high-speed mes-sage-routing unit that allows messages for other units to pass though P without affect-ing P's ongoing operations; this approach largely eliminates the need to temporarily store messages in intermediate nodes.

A node of the nCUBE 2 consists of a full-custom 64-bit CPU on a single IC, plusa six-chip local memory. The CPU's architecture resembles that of the Digital VAXfamily; it has a CISC-style instruction set with fixed-point and floating-point arithmeticinstructions and all the logic necessary for memory management and IO control. Itsspeedup features include a four-stage instruction pipeline, an I-cache and a D-cache, aswell as the special message router noted already. The local memory size can range upto 64 MB per node, so an 8192-node system can have a distributed memory of 256 GB.With a modest clock rate of 20 MHz, each processor delivers about 2.4 MFLOPS(assuming 64-bit operations), implying a peak performance of around 2.4 x 8192 =19.7 GFLOPS, so the nCUBE 2 was classed as a massively parallel "supercomputer."

The structure of an nCUBE 2 system is outlined in Figure 7.67. The hypercubearray of processors H is packaged into printed-circuit boards, each of which contains a64-node hypercube forming a subcube of H. Each processor has 14 communicationchannels, one of which connects to an IO subsystem, such as a "farm" of IO disksforming the system's secondary memory. Many IO channels to the hypercube arrayenable a large number of peripherals to operate in parallel to satisfy the nCUBE's

Host(front end)computer

10

subsystem

Hypercube of processors

& & & 0^

cA

o^

<^

&-

&-

<^

<^

<^

<^

<^

<^

<^

^

<^-

c^

</

<^

<^

<^

& (^ (^ 6*

o*

<^

^

&

&

Figure 7.67

Organization of the nCUBE 2 hypercube multiprocessor.

massive computation ability. Each channel is controlled by the nCUBE processor usedin the hypercube array. Disk storage capacity can exceed a terabyte (240 bytes), mak-ing the nCUBE well suited to the management of very large databases.

The nCUBE operating system provides all the usual UNIX system managementand programmer support functions (see Example 7.8). It treats a hypercube of proces-sors as a device, which in the UNIX philosophy is a special type of file. Consequently,a hypercube of any size can be opened, closed, written into, and read from like anyother UNIX file. This feature permits the operating system to allocate independent sub-cubes to different users so that one or two large applications or many small applicationscan share the processor hypercube concurrently.

Multistage interconnection networks. Dynamic interconnection networksfor multiprocessors can be constructed from two-state switching elements of thekind depicted in Figure 7.68. Each switch 5 has a pair of input data buses Xj,X2; apair of output data buses ZUZ2; and some control logic (not shown). All four busesare identical and can function as processor-processor or processor-memory links. Shas two states determined by the control line c: a through or direct state T, as illus-trated in Figure 7.68fr where Z, = X, (Z, is connected to X[) and Z^ = X2, and across state X where Zx = X2 and Z2 = X, (Figure 7.68c).

Switch5

1 '" -

2

Control c(a)

X2 ►

c=l(b)

Figure 7.68

(a) Switching element; (b) through state T; (c) cross state X.

*- z,

•► z,

By using 5 as a building block, multistage interconnection networks (MINs) can be constructed for use in massively parallel computers [Siegel 1990]. Figure 7.69 shows a small MIN that has 12 switching elements arranged into three stages (columns) and is intended to provide dynamic connections among eight processors denoted i'ooo^in- By setting the control signals of the switching elements in vari-ous ways, many different interconnection patterns are possible. The processor-to-processor connections that are possible depend on the number of stages, the fixed connections linking the stages, and the settings of the switching elements. The par-ticular MIN in Figure 7.69 is called an 8 x 8 omega network. A large version of this MIN was used in the experimental Cedar multiprocessor designed at the University of Illinois in the 1980s. We now examine the major characteristics of some typical MINs, concentrating on those designed for processor-to-processor communication.

An N x N MIN 57V provides a flexible set of communication links between TV processors, which are the sources and destinations of SN. Since the processors are identified by n-bit binary addresses, it is convenient to make TV = 2". The processor-pairs that are connected to each other at any time by SN are determined by the states of the switching elements, each of which can be in either the through (T) or cross (X) state. Control logic associated with the MIN sets the switch states dynamically to satisfy interconnection requests from the processors. A particular MIN state is retained long enough to allow at least one package to be transferred through the network. The state then changes to match the source-destination requirements of the next set of packages, and so on. We assume that a processor can buffer or queue its outgoing packages until the MIN is ready to transfer them. The processors accept incoming packages as soon as they arrive.

A fundamental requirement of a MIN is that it be possible to connect every processor P, to every other processor P- using at least one configuration of the net-work; this feature is termed the full-access property. It is easy to show that the omega network of Figure 7.69 is a full-access network. Figure 7.70 shows the

561

CHAPTER 7

System

Organization

Stage 1 Stage 2 Stage 3

P k $i.i   5,.2   S,.3

UXX) *'ooi ~~\ / *

Figure 7.69

Three-stage 8x8 omega multistage interconnection network (MIN).

562

SECTION 7.3 Parallel Processing

Destination Stage 1 Stage 2 Stage 3

P„

5,, =T511=T

5u=T5U=X5,,=X5U=X5, , =X

512 = X

su=x

52.2 = TS2,2 = Tj-i •) = X

513 = X523 = T

52.3 = X

53.3=TS33 = X54.3 = T

54.3=X

Figure 7.70

Switch settings of the three-stage omega network of Figure 7.69 to connect P^ to each of the other processors.

seven unique switch configurations needed to connect Pqqq to each of the other pro-cessors; here S;J = T (X) indicates that switch / of stage j is set to the through (cross) state. A complete network configuration in which P^ is connected to P00l appears in Figure 7.71. In this state the network also connects P010, P100, and P110 to Pon, P101, and Pul, respectively, thus providing simultaneous communication among four processor-pairs. Reducing the number of stages from three to two eliminates the full-access property.

Another useful property of a MIN is the ability to establish a connection between any pair of processors that are not using the network, without altering the switch settings already established to link other processors; this is the nonblocking property. The three-stage omega MIN of Figure 7.69 does not have this property and is therefore a blocking network. For example, suppose that Pqqq is already con-nected to Pqqp this condition requires the top row of switches to be set to TTX, as specified in Figures 7.70 and 7.71. It is now impossible to connect P100 either to P010 or to Pou. The preexisting setting of S, j creates a path from PI00 through stage

Stage 1

Stage 2

Stage 3

Figure 7.71

One state of the three-stage omega network.

1 to S2 2- No links exist from S2 2 to S23, the third-stage switching element con-nected to P0lQ and Pon; hence S22 cannot be set to forward data to P010 or PQn.This type of blocking causes communication delays similar to those occurring in asingle-bus system when several processors attempt to use the system bus simulta-neously. Nonblocking MINs require an excessive number of switches for mostcomputer applications. An NxNcrossbar switch is an example of a nonblockingnetwork because it allows any idle row to be connected to any idle column. How-ever, it contains N2 complex crosspoint switches, whereas anNxNomega networkcontains only (N/2) log2 N simpler 2x2 switches.

A few basic interstage wiring patterns characterize the most common MINtypes proposed for multiprocessors. Each such pattern is a mapping vj/ from a set ofsources {S,} to a set of destinations {Dw{i} } for / = 0,1,...,N -1. Here 5, is the addressof an output port of a processor or switching element, and Dv(l) is the address of theinput port to which 5, is wired. The shuffle pattern is defined by the following map-ping:

0(0 = 2i+l(2i)/Nj (modulo AO

(7.18)

Here o is the shuffle function illustrated by Figure 7.72a for N = 8. The nameshuffle comes from the fact that the destination addresses 0, 1,2, 3, 4, 5, 6, 7 can bemapped into (connected to) the source addresses 0, 4, 1, 5, 2, 6, 3, 7 by interleavingthe first half 0, 1, 2, 3 of the address sequence with the second half 4, 5, 6, 7 in themanner of a perfectly shuffled deck of cards. Let each address i be represented bythe corresponding n-bit binary number bn_lbn_2...b0. An equivalent definition to(7.18)is

o(/) = bn_2bn

boK-

(7.19)

indicating that the shuffle function corresponds to rotating the source address 1 bitto the left to determine the destination address. By following a shuffle connectionwith N/2 switching elements, each of which can exchange (cross) a pair of buses,we obtain the single-stage shuffle-exchange network, shown in Figure 1.12b for thecase N = S. The omega network of Figure 7.69 is built from n = \og2N shuffle-exchange stages.

Another useful class of MINs is based on the butterfly connection depicted inFigure 7.73a. The 4x4 single-stage butterfly network appears in Figure 7.736;note that the butterfly connection is placed after, rather than before, the N/2 switch-ing elements. Consider an NxN multistage network with n stages 1,2, ..., n and Nport addresses i = 0, 1,terfly function p\ is defined as follows fork = 1, 2,..., n — 1:

., N - 1, where, as before, i = brl_lbn_2"bQ. The klh but-

PA,-i" ^+i^A-i 'Vo) = K-\ " bk+lbob

k-\

b]bk

Thus $k interchanges bits 0 and k of the source address to obtain the destinationaddress. For example, when k = 1 and jV = 4, we obtain

Pi (00) = 00P,(01)=10P,(10) = 01P,(H)=11corresponding to the interconnection pattern on Figure 7.73a.

                    000   »► 000   000

Parallel Processing 001 —v ,—•• 001 001

                    010 .   A—► 010 010

                    Oil > /   \ /—*" on   on

                    100 —') y ^-*- 100 100

                    101 —/ \^~** 101 101

                    110 —/ \-* 110   110

                    111   * 111   111

+■ 110

(a)

(b)

Figure 7.72

(a) Shuffle connection for N = 8 and (b) single-stage shuffle-exchange network.

The connection pattern defined by

$$er!(0 = V«-A-3 ~*i \quad (7-20)$$

is called the inverse shuffle function a-1. Equation (7.20) is the same as Equation(7.19), defining the shuffle function a with the direction of the address bit rotationreversed. Figure 7.74 shows a 16 x 16 version of a MIN called the indirect hyper-cube network, which in the N x N case consists of log2 N stages of N/2 switchingelements; the wiring patterns following the stages are defined by (3,, (32, ■■■> Pn-i>CT1. This MTN's name comes from the fact that it can easily simulate the connec-tions of a static hypercube interconnection network; see problem 7.39.

Indirect hypercube and shuffle-exchange MINs have similar properties. Sup-pose that the directions of all the arrows in an N x N shuffle-exchange network arereversed, implying that the shuffle connection a in each stage is replaced by a-1.The resulting Nx N inverse omega network and the N x Nindirect hypercube net-work are essentially the same MIN drawn in different ways. Consequently, for

(a)

(b)

Figure 7.73(a) Butterfly connectionfor N = 4 and (b) single-stage butterfly network.

each state of the indirect hypercube network, there is a state of the inverse omeganetwork that connects the N processors in exactly the same way, and vice versa.This equivalence is not obvious and explains the many names under which thisclass of MINs appears in the literature (inverse omega, indirect binary n-cube, but-terfly, and so forth).

Since an address contains n = \og2N bits, at least n = log2N stages must bepresent for an N x N MIN to have the full-access property. With this number ofstages, it is also easy to determine the switch settings needed to connect an arbi-trary pair of processors, since each stage controls 1 bit (dimension) of the addressspace. We illustrate this for the indirect hypercube MIN of Figure 7.74. Supposethat a source processor with binary address S = sn_lsn_2's0 is to be connected to adestination processor with address D = dn_xdn_2d§. As in the static hypercube

565

CHAPTER 7

System

Organization

Stage 1

Stage 2

Stage 3

Stage 4

Figure 7.74

16x16 indirect hypercube network.

566 routing algorithm (Example 7.10), we compute R = S®D= rn_lrn_2 rQ, and use R

section 7 3 t0 contr°l *e MIN's switch settings. If r, = 0, then all the switches in stage i + 1

Parallel Processing (assuming again that the stages are numbered 1, 2, ..., n) are set to the through (T)state; these switches are set to the cross (X) state if r, = 1. For example, Figure 7.74shows the switch settings to connect source S = 2 to destination D = 14. In this casei? = 0010 © 1110= 1100, requiring two T and two X switch settings as indicated.The heavy lines in Figure 7.74 mark the path along which packages travel from Sto D. If all switches are set to T, then S = D, so each processor is connected to itselfvia a path through log2/V switches. Changing the state of the switch in stage / + 1along this path from T to X connects the source processor to the destination proces-sor that differs from it in the ith address bit. It follows that there is only one paththrough each of the foregoing (log2AO-stage networks linking every source-desti-nation pair.

The routing of packages through a MIN can be managed by a centralized con-troller attached to the network that examines all source-destination address pairs S,D generated by processors and sets the appropriate switching elements to the statesspecified by R = S © D. An alternative is to attach R as a routing tag to each pack-age to be transmitted from 5 to D and to use R to set the switching element statesas the package passes through the MIN. When the package enters a switch Sji+1 instage / + 1, Sjl+l examines the routing tag R using control logic built into theswitch for this purpose. &jl+l then sets its own state to T if r, = 0, and to X if rx■ = 1.Thus the centralized controller can be replaced by decentralized control logic dis-tributed throughout the MIN. Each package determines its own path through theMIN and so can be viewed as self-routing. For example, to transmit a packagefrom S = 2 to D = 14 in the four-stage MIN of Figure 7.74, the routing tag R =r3r2riro = 1100 is appended to the package generated by the source processor P2.The switch 52 , attached to P2 in stage 1 inspects bit r0 of R. Since r0 = 0, switchS2i sets itself to the through state T. This setting causes the package to be sent tothe topmost switch Sl2 in stage 2, which also sets its state to T, since r, = 0. Thepackage proceeds to the final two stages, which set themselves to the cross state X,since r2 = r3 = 1.

The Butterfly computer developed by Bolt, Beranek and Newman Inc. around1980 [Crowther et al. 1985] and its successor the TC2000 introduced in 1989 areexamples of commercial multiprocessors based on MINs. They are shared-memoryMIMD computers in which the MIN connects N processors to N memory units thatform the shared memory. In the original Butterfly multiprocessor, the processorsare based on the Motorola 680X0 series, and TV ranges from 1 to 256. Every proces-sor contains a microprogrammed coprocessor to handle virtual memory manage-ment, package transfer to and from the MIN, and related functions.

The Butterfly's MIN has single-chip 4x4 switching elements, each of whichis obtained by cascading two copies of the basic butterfly network of Figure 7.72.Consequently, the processor-memory interconnection network is an TV x TV butter-fly MIN composed of log2/V stages of 2 x 2 switching elements. Data transmissionthrough the network is by bit-serial packages, which can be transmitted at a rate of32 Mb/s along any processor-memory path. Each package contains its destinationaddress and is made self-routing in the manner described earlier by employing2 bits of the destination address to determine the setting of each 4x4 switchthrough which the package passes. Should two packages attempt to use the samelink in the MIN simultaneously, one is allowed to proceed and the other is retrans-

mitted after a short delay. This type of application-dependent contention increases the execution time of a typical program by only a few percent.

Fault tolerance has been defined as "the ability of a system to execute specified algorithms correctly regardless of hardware failures and program errors" [Avizie-nis 1971]. It is of some concern in all computer systems, while in applications such as spacecraft control and telephone switching, fault tolerance is a major design goal [Siewiorek and Swarz 1992]. Most hardware failures have physical causes such as component wear or electromagnetic interference. The nature and frequency of these failures can be determined experimentally, which makes it possible to study the faults and their consequences using analytic or simulation models. Soft-ware faults are primarily due to algorithm or programming mistakes (design errors) and so are more difficult to deal with.

Redundancy. Fault tolerance is intimately associated with the concept of redundancy. When a component fails, its duties must be taken over by other, fault-free components of the system. If those components are intended to improve only the reliability of the system and do not significantly affect its computing perfor-mance, they are termed redundant. Redundancy can be introduced in several over-lapping ways:

• Hardware redundancy: Multiple copies of critical hardware units.

• Software redundancy. Multiple versions of programs for critical operations.

• Information redundancy: Error-correcting or error-detecting codes.

• Time redundancy: Repeating or retrying critical operations.

The goal of these redundant design features is to prevent failures due to physical faults or design mistakes from producing errors, that is, data values or operating modes that lead to system failure. Information redundancy via coding methods is discussed in section 3.2.1. In this section, we examine the use of redundant hard-ware to achieve fault tolerance.

Two broad approaches, static and dynamic redundancy, have been identified for designing fault-tolerant systems. Static redundancy refers to the use of redun-dant hardware or software components, which form a permanent part of the sys-tem, to mask the error signals generated by faults. One form of static redundancy replaces a critical unit that generates a word X with $n > 3$ copies of that unit, con-figured to generate n independent copies of X in parallel. If the unit in question is a processor, then the resulting system is a type of multiprocessor. The n versions of X are applied to a circuit called a voter, which is designed to output the value of X appearing on the majority of its n input buses. Thus errors produced by any of the replicated units are masked by the voter, provided more than half of the units pro-duce the correct X values at all times. A system of this type with n identical units and a voter is said to employ n-modular redundancy (nMR).

A frequently implemented version of nMR is triple modular redundancy (TMR), in which n = 3, as shown in Figure 7.75. In this case the behavior of the voter is defined by the logic equation

$$A = A j A 2 + A i A_{-i} + A 2 A 3$$

U   *2

U Voter

System

input   *3

U

A — A i Ai + A jAi + Atai

System output

Triplicated units

Figure 7.75

Example of triple modular redundancy (TMR).

where + denotes the (word) OR operation; this is the well-known majority func-tion. The voter's output X always has the correct value, assuming that no more than one of $X x ^2 ^3$ is incorrect and that the voter itself does not fail in a way that pro-duces an erroneous output. Thus a TMR system can tolerate faulty behavior by anyone of its triplicated units. Although static redundancy can be implemented at any complexity level, it is normally implemented at the processor level where the repli-cated units are CPUs, memory units, switching networks, or entire computers.

Dynamic redundancy tolerates faults by actively reorganizing the system so that the functions of the faulty unit are transferred to one or more fault-free units. The reorganization is usually achieved in three steps:

1. Fault diagnosis: Diagnostic procedures are carried out to detect the fault and isolate it to a replaceable or repairable unit.

2. Fault elimination: The fault is removed from the system either by repairing the faulty unit, replacing it by a spare, or logically reconfiguring the system around the fault.

3. Recovery: Procedures are executed to restore the system to a state that existed before the fault occurred. Normal operation is resumed from that point.

Although more complex to manage than static redundancy, dynamic redundancy has the advantage that faulty units can be rapidly eliminated from the system. In the static case faults can accumulate undetected until a total system failure occurs. Figure 7.76 shows an example of a fault-tolerant system employing dynamic redundancy. It is called a duplex system because it contains two identical (dupli-cated) copies of the basic nonredundant or simplex unit. The two units operate in tandem, performing the same operations on the same (or duplicated) data at the same time. A circuit called a match detector or equality checker does a continuous comparison of the results generated by the duplicated units. When the match detec-tor finds a mismatch indicating the occurrence of a fault, normal operation is sus-pended and a testing procedure is initiated to identify the faulty unit. Once identified, the faulty unit is disconnected from the system, logically if not physi-cally. The system can then be restarted in simplex mode using only the fault-free unit. The failed unit can be repaired off-line and eventually restored to the system.

U

u

Configuration control

System   Match detector   i ► System

input   output

n

U

Figure 7.76Example of a du plex systen 1.

Redundant disk arrays. Magnetic hard disks (section 6.1.3) are the principaltechnology employed for secondary memory systems in computers. While provid-ing large amounts of storage at low cost per bit, disk memories—both magneticand optical—have several drawbacks.

• They have relatively slow data-transfer rates.

• Their electromechanical construction makes them prone to both transient andcatastrophic failures.

A way to increase the data-transfer rate is to build a disk memory from an array ofsmall disk units, all capable of operating in parallel. With n such parallel units, theeffective data-transfer rate is n times that of a single unit. Furthermore, includingredundant disk units in the array can improve fault tolerance. In the late 1980sthese considerations led to a general approach to disk-memory design known asredundant array of inexpensive disks (RAID), which has since been widelyadopted by manufacturers of disk memories [Chen et al. 1994].

The idea behind RAID is to distribute the stored data over a set of disks config-ured to appear like a single large disk. The data can be distributed in various waysreferred to as RAID levels 0:6, or simply as RAID-0:6. The different RAID levels,all of which are illustrated in Figure 7.77, provide different performance-costtrade-offs. In RAID-0, the n disk units are intended to increase performance only.There is no redundancy for fault tolerance, and so the system is vulnerable to thefailure of a single disk. RAID-1 is a duplex design with In instead of n units, whereall data written onto one disk is duplicated on another. This high-cost approach haslong been used under the name disk mirroring in applications that must recoverinstantly from a fault.

The remaining five RAID organizations have less redundancy and rely on var-ious coding schemes to implement fault tolerance. RAID-2 employs error-correct-ing codes of the type found in RAMs and has extra disks to store check (parity) bitsfor all data words stored in the main disks. As discussed in section 3.2.1, to achievesingle-error correction, we need c check bits for every n data bits, where 2l > n +c + 1. Therefore, c ~ log2n redundant disk units are required to tolerate singleerrors. The n + c disks of RAID 2 can be thought of as storing (n + c)-bit wojds,with one particular bit position assigned to each disk in interleaved fashion. (Other

RAID-0

No redundancy

RAID-1

Duplex (mirroring)

RAID-2Error-correcting codes

RAID-3Bit-interleaved parity

RAID-4

Block-interleaved parity

RAID-5

Block-interleaveddistributed parity

RAID-6

P + Q redundancy

Figure 7.77

Redundant arrays of inexpensive disks (RAID); shaded blocks denote redundant data.

noninterleaved storage patterns are also allowed.) When an inconsistent check bitis detected during a read operation, the erroneous codeword identifies the errone-ous bit and hence the faulty disk that contains it.

It is not necessary to have RAID in order to detect an error in a disk unit,since the unit's controller can easily do so via its internal, conventional mecha-nisms for error detection. Hence it is enough to store a single parity bit in order tocorrect, and therefore tolerate, a single error in any word. This approach is thebasis of RAID-3, where each (n + l)-bit data word bin_\bin_2"biQpi'\s spreadover an (n + l)-unit disk array. One (redundant) disk stores all the parity bits {/?,},and its contents are computed on the fly via a parity equation of the form:

Pl = bin_x © bin_2 © © bu © • © bl0 (7.21)

If an error is detected in disk;', then the lost or damaged b,,;s in disk j can be recov-ered from the remaining n disks according to the following equation implied by(7.21).

b, t = bi „_> © b: „_■> ©

'i,n-\

i.n-2

©Vi®Vi® ®bi0@p,

Intuitively, the parity disk stores the "sum" of the data on the other disks. On a diskfailure the lost data is obtained by "subtracting" the data on the n - 1 good disks

from the contents of the parity disk. (Recall that the EXCLUSIVE-OR operation ©corresponds to sum or difference modulo 2).

The RAID-4 scheme is similar to RAID-3 except that blocks of arbitrary sizeare interleaved, rather than individual bits. Because the single parity disk tends toact as a bottleneck—it does not participate in read operations, for example—RAID-5, which distributes the parity bits evenly over all available disks, is pre-ferred to RAID-4. In RAID-4 and 5, write operations, especially short writes, arecomplicated and performance is reduced by the fact that it is necessary to read allthe disk units, including units not being written into, in order to compute the newparity bits. The final scheme, RAID-6, uses two redundant disk units and multibiterror-correcting codes to tolerate the failure of up to two disk units.

Reliability. The ability of a system to tolerate faults can be measured in sev-eral ways. One useful fault-tolerance measure is availability, defined as the frac-tion of its operating lifetime during which the system is not disabled by faults. The availability of the AT&T No. 1 Electronic Switching System (ESS), one of the ear-liest computer-controlled telephone exchanges (first deployed in the 1960s), was specified at two hours of downtime over an expected operating life of 40 years. This value is equivalent to an availability of 99.9994 percent.

A more common fault-tolerance measure is reliability R(t), defined as the probability of a unit or system surviving (functioning correctly) for a period of duration t. The reliability of a unit can be estimated from the failure statistics for a large number of samples of the unit. The failure rate is the fraction of the samples that fail per unit time. For most physical devices, the failure rate varies with time in the manner shown in Figure 7.78. During the early life of the unit (the burn-in period), a high failure rate is experienced that reflects faults occurring during man-ufacture or installation. A high failure rate is again encountered toward the end of the unit's life (the wear-out period). During most of the unit's working life, how-ever, failures can be expected to occur randomly at a fairly constant rate; this period corresponds to the flat central part of the "bathtub" curve of Figure 7.78.

Analytic approaches based on probability theory have long been successfully used to study the reliability of computer systems. Suppose that N(0) copies of a unit such as a CPU begin their operating life (after the burn-in period) at time t = 0. Let N(t) be the number of units surviving after time t so that the number of failed units Nj-(t) is N(0) - N(t). The reliability R(t) of the unit is given by the fraction of

Operating
Burn-in   life   Wear-out

Failure rate \j   »,4 ►

1

Time

Figure 7.78

Typical variation of failure rate with time.

572 surviving units at time t; that is

SECTION 7.3

Parallel Processing   "(*> ~ /V(0) (7.22)

which can be interpreted as the probability of any unit surviving to time t. Let A.denote the unit's failure rate, which, in accordance with Figure 7.78, is assumed to be constant. Therefore, the number of units dNf that fail during the small interval of time from no t + dt is given by

dNf=XN(t)dt (7.23)

Now N(t) = N(0) - Nf(t) and N(0) is independent of t; hence dN = -dNf. Substitut-ing into Equation (7.23), we obtain

dN=-XN(t)dt

Now (7.23) implies that dR = dN/dN(0); hence dR = -\N(t) dt/N(0). Using (7.23) again to replace N(t)/N(0) by R(t), we obtain

f = ~lR(t)

Integration with the boundary value R(0) = 1 yields

R(t) = e* (7.24)

This classical exponential law of failure is very often used to model the reliability of the components in a computer system.

From the reliability R(t) we can obtain a single number MTTF called the mean time to failure, which is a useful measure of the expected working life of a unit. Letting F(t) denote the unreliability 1 - R(t), MTTF can be defined as follows:

MTTF = j tf(t)dt where/(0 = ^^ (7.25)

o "t

The MTTF corresponding to the exponential reliability function (7.24) is

MTTF = J tXe~X'dt = \

so the expected working life of a unit with an exponentially distributed reliability is the reciprocal of its failure rate.

System reliability. Once the failure rates of its individual units are known or can be estimated, it becomes possible to calculate the reliability of the entire sys-tem. Two basic circuit system structures from a reliability point of view are the series and parallel configurations appearing in Figure 7.79. In a series system (Fig-ure 7.79a), it is assumed that if any component fails, the entire system fails. Hence the system reliability which, for brevity, we denote by R instead of R(t), is a prod-uct of the component reliabilities.

R = l\R,

i= 1

«i

R2

Rn

(a)

Figure 7.79

Two basic reliability structures: (a) series and (b) parallel.

(*)

573

CHAPTER 7

System

Organization

In a. parallel system (Figure 1.19b), on the other hand, all components must fail in order for the system to fail. Hence the system's unreliability F = 1 - R is the prod-uct of the component unreliabilities 1 - R{, from which it follows that

R = 1

no-*,-)

(7.26)

As these equations show, putting units in series decreases reliability, while putting units in parallel increases reliability. A parallel connection of n units is a basic fault-tolerant structure; we find it, for example, in duplex and TMR systems, where n = 2 and 3, respectively.

Systems can sometimes be decomposed into series and parallel subsystems, and their reliability can be calculated by repeated application of the preceding equations. For example, the series-parallel system S in Figure 7.80 consists of two subsystems S, and 52, which are connected in series; S{ and 52 are themselves par-allel systems. Assuming that each individual unit has reliability R, the system reli-ability R(S) is given by

R(S) = [l-(\-R)i][l-(l-R)2]

= 6R2 - 9R3 + 5R4-R5

Let us now apply the preceding equations to a TMR system like Figure 7.75. We can view it as three parallel copies of U in series with a voter V. Assume that

R

R

s

R

Figure 7.80

Example of a series-parallel system

574 each of the triplicated units has reliability R](t) = e~h and that the voter has reliabil-

SECTION 7 3 ^ ^v^ = e~XV'' ^et ^'^ ^e ^e Probability of any i of the triplicated units surviving

Parallel Processing t0 time L The system reliability R3(t) is then given by

R3(t) = [P2(t) + P3(t)]Rv(t) Now P2(t) = [2)(e~X'HI-e~X'), while P3(t) =' (e~X'y, hence

R3(t) = (3e~2X'-2e~iX')e~ v' (7.27)

The voter is usually much simpler than the functional units; consequently, its reli-ability is very high. If we assume /?v(r) = 1, that is, if we ignore the possibility of voter failure, then Equation (7.27) reduces to

R3(t) = 3e'2X'-2e'3X' (7.28)

Figure 7.81 plots this equation for X = 0.01. The reliability of a single unit /?,(/) = e-k is shown for comparison. For values of R less than about 0.7/X, the reliability of the TMR system is greater than that of the simplex system; beyond this point its reliability is less. In practice, TMR reliability can be higher than the foregoing analysis suggests, since the system may continue to function correctly even if two units fail. For example, if the two failed units never generate incorrect output sig-nals at the same time, then the voter still produces the correct output. The unreliability density function/(f) corresponding to (7.28) is

fit) = jt[\-R3(t)) = 6e-2X'-6e-3X' Substituting into (7.25) yields the mean time to failure MTTF3 for a TMR system.

MTTF3 = J t(6e~2X'-6e~3X')dt (7.29)

o

Integrating (7.29) by parts, we obtain

MTTF, = [t(-3e'2X' + 2e-3X')L-\ (-3e~2Xt + 2e~3X')dt = jr

Since the MTTF of the corresponding simplex system is l/X, the MTTF of the TMR system is the smaller of the two. These values are consistent with Figure 7.81, which shows that while the TMR system's initial reliability is high, it falls off more rapidly than the simplex reliability as the two systems age.

The foregoing reliability analysis considered only static systems in which there are no maintenance or repair activities. No matter how fault tolerant we make such a system, it can be expected that its reliability R(t) -» 0 as t -» °o. With repair, how-ever, it is possible to increase the chances of the system functioning correctly at time t beyond R(t) to a value termed the {instantaneous) availability A{t). In gen-eral, A(t) is the sum of R(t), the probability that no faults occurred up to time t, and the probability that the system failed before t but was repaired and continues to sur-vive. With regular repair we can make A{t) approach a nonzero steady-state value as t increases. The working life of a dynamic system that is always repaired after a failure occurs consists of an alternating sequence of periods of fault-free normal operation and periods during which the system is down for repairs. The system's

1.0

Reliability 0.5

W TMR

Vv R3(t) = 3<r° 02' _ 2e-O03<

XV   Simplex

1   Rt(t) = e-°-0i'

575

CHAPTER 7

System

Organization

100

200

300

Figure 7.81

Reliability comparison between TMR and simplex systems.

actual availability, therefore, over its entire lifetime L is the ratio of its total fault-free working life to L. If the repair process makes the system "as good as new,"then the expected (average) duration between the completion of a repair and theoccurrence of the next fault is the system's MTTF. Similarly, we may characterizethe duration of the repair process by the mean time to repair (MTTR), which is theexpected time between system failure and the completion of repair. The expectedavailability A of the system, which is usually what is meant by the termavailability, is therefore given by the following useful formula:

A =

MTTF

MTTF+MTTR

(7.30)

The denominator MTTF + MTTR is referred to as the mean time between failures(MTBF) and is approximately the same as MTTF when MTTR is very small.Equation (7.30) indicates that availability can be increased either by increasing thesystem's inherent reliability, as indicated by MTTF, or by reducing the timeneeded for repair after a fault occurs.

We conclude with an example of a commercial fault-tolerant multiprocessorseries, the Tandem NonStop, whose technology evolution reflects that of thecomputer industry in general [Kong 1994]. This series began in 1976 with theNonStop I, a small-scale multiprocessor based on bipolar (TTL) MSI integratedcircuit technology. Its CPU was a custom-designed 16-bit processor of the CISCtype, with a hardwired, stack-oriented organization. Operating at a clock frequencyof 10 MHz, CPU performance was about 0.7 MIP. The NonStop I had no cache, avirtual memory of 512 KB, and each system contained from 2 to 16 processors. Adecade and several models later, the Tandem VLX (1986) employed bipolar (ECL)gate-array ICs and a 32-bit microprogrammed architecture incorporating suchspeedup techniques as pipelining and a 64KB unified cache. The CPU performancehad increased to 3.0 MIPS at 12 MHz, and virtual memory had expanded to 1 GB.The Tandem Himalaya series, introduced in 1993, employs the MIPS R4400 64-bit

576 microprocessor, an off-the-shelf CMOS RISC. This superscalar microprocessor

supports a two-level cache and a virtual memory of 2M B; its performance is in the

Parallel Processing ^ MIPS range at 200 MHz. Tandem's Himalaya systems are designed in two- or

four-processor clusters built around a high-speed shared bus. Massively parallelsystems containing hundreds of processors can be constructed by linking clusterstogether via a large-scale interconnection network with a meshlike structure. Tan-dem's goal of high performance coupled with high hardware and software integrityhas increasingly become the concern of the entire computer industry.

EXAMPLE 7.11 THE TANDEM NONSTOP HIMALAYA MULTIPROCESSOR

[KONG 1994]. Starting in the mid-1970s, Tandem Computers Inc. was the first com-puter maker to focus on commercial applications with high availability as the principaldesign goal. An important example is on-line transaction processing (OLTP), such assecurities trading or on-line ticket reservation, where even a brief system shutdown canentail huge economic losses. Applications of this sort also tend to have very high per-formance requirements. Tandem's "NonStop" architectural approach was developedwith the following specific objectives:

• A system organization that prevents any one hardware fault—a single-pointfailure—from causing a crash or compromising the integrity of the system or appli-cations software

• Dynamic on-line detection of faults, removal of faulty units for repair, and return ofrepaired units to service while redundant components keep the system in operation

• Scalability that allows processor, memory, and 10 capacity to be increased withoutaffecting the application's software.

To meet these objectives and remain cost competitive with mainstream computermanufacturers, Tandem opted for a modular multiprocessor architecture in which themultiple processors provide much of the redundancy needed both for fault toleranceand for high performance. Components that are not naturally redundant such as thepower supply, system bus, and 10 controllers are duplicated to ensure that all theirsingle-point failures can be masked. For example, disk mirroring (RAID-1) is used toautomatically create backup copies for all data in secondary memory. Standard codingtechniques check for errors occurring in the major data paths and main memory. TheNonStop operating system kernel is built around duplex, distributed processes thatexchange messages for interprocess communication. Hard disks and other 10 devicesare connected to two 10 controllers, one of which "owns" each device. 10 deviceownership can be switched by the operating system at any time. Software control ofeach 10 device resides in a redundant primary/backup pair of processes. The primaryprocess manages the device but also sends "checkpoint" information to the backupprocess to keep it up-to-date in case it must take control of the 10 device. User pro-cesses are handled in a similar way; when a user process starts on one processor, abackup copy of the same process is automatically started on another processor.

Figure 7.82 shows the structure of a four-processor cluster or "section," which isthe basic hardware building block of every NonStop system. Each processor contains aCPU, a portion of main memory, and an 10 processor. The processors in a section com-municate with one another via a high-speed interprocessor bus, the Dynabus, which isduplicated. A set of IO buses (channels) links each processor to a set of 10 controllersso that every IO controller is connected to two processors. The processors in a sectionare tightly linked via the Dynabus. The sections, in turn, can communicate via a LAN-style network to form a loosely coupled system containing tens or hundreds of clusters.Such configurations are well suited to OLTP servers, which typically deal with hugenumbers of largely independent tasks.

Although, as noted above, the Tandem family has evolved steadily to embraceadvances in hardware technology, the overall design philosophy depicted in Figure7.82 has remained remarkably intact from one generation to the next. The originalNonstop I (1976) was based on a custom-designed 16-bit CISC processor; recent prod-ucts like the Himalaya series (1993) use off-the-shelf 64-bit RISC microprocessors.Each Himalaya CPU actually contains two R4400s operating in lockstep, with one pro-cessor (the slave) serving as a check on the other (the master). The Himalaya alsointroduces a novel type of interconnection network, referred to as TorusNet, whichuses fiber-optic cables to connect the clusters. Clusters (sections) can be linkedtogether in a ring network, and each cluster participates in separate H (horizontal) andV (vertical) rings. The TorusNet H and V rings accommodate up to 4 and 14 clusters,respectively, so 56 clusters or 224 processors can be connected in this way. As illus-trated by Figure 7.83, the interconnection network has a toroidal structure in whichevery cluster is directly linked to four others, and indirectly to all the clusters in thesystem. By providing many alternative paths among its processors, a large Himalayasystem can tolerate the simultaneous failure of several of its clusters and their intercon-nections.

577

CHAPTER 7

System

Organization

To TorusNet H

To TorusNet V

Netinterface

Dvnabus X

Processors

CPU

Memory

IOP

Dynabus Y

CPU

Memory

IOP

Diskcontroller

Diskmemory

Diskcontroller

Section

Commun.
controller

CPU

Memory

IOP

Diskcontroller

Diskmemory

Diskcontroller

Netinterface

CPU

Memorv

IOP

Tapecontroller

Tapememory'

Commun.controller

Figure 7.82

Processor cluster (section) in Tandem NonStop computers.

r\

Q

0

Q

r~\ r~\ r~\

TorusNet H

TorusNet V

Figure 7.83

Toroidal interconnection network of the Tandem Himalayacomputer.

7.4 SUMMARY

The communication methods used in a computer system depend on the physicaldistances involved. Intrasystem communication uses shared buses that transmitbinary signals a word at a time over short distances. Intersystem communication,on the other hand, is implemented using serial-by-bit data transmission. Manyinterconnection structures and transmission media are possible, and they offer var-ious trade-offs between bandwidth and cost. Data transfer over a shared bus can besynchronous with clock control or asynchronous with handshaking control signals.At any time only two units can be logically connected to the bus: a bus master,such as a CPU, an 10 processor (IOP), or a direct memory access (DMA) control-ler, and a bus slave such as a memory unit or an 10 port. Arbitration techniquessuch as daisy chaining or polling determine which of several requesting units gainsaccess to the bus. Buses are characterized by the numbers and types of data,address, and control lines they contain and by the conventions (protocols) they usefor signal selection, synchronization, and arbitration. Standard buses such as thePCI bus are widely used as system or 10 (local) buses.

A computer network is a connected set of computers and other system compo-nents separated by large physical distances. Various standards exist for computernetworks, with the seven-layer OSI Reference Model providing general guidelinesfor standardization. A representative standard architecture for local-area network(LANs) is Ethernet, which employs a shared cable link and CSMA/CD arbitration.

Input-output systems are distinguished by the extent of CPU involvement in10 operations. The use of CPU programs to control all phases of an 10 operation iscalled programmed IO. By providing IO devices with DMA and 10 interrupt con-trol, data transfers can be implemented independently of the CPU. Maximum speedand independence are achieved by providing IOPs capable of executing their ownprograms to manage 10 operations. Overall management of a computer is handledby an operating system, which is responsible for efficient sharing of a computer'scentral processing, memory, and IO resources, both hardware and software. Theoperating system supervises a set of concurrent processes, which implement sys-tem and user tasks. Among the more widely used operating systems are UNIX,used primarily in workstations, and Windows, used in personal computers.

The motivations for introducing parallelism into computer systems are higherperformance and reliability. Many methods have been proposed for classifyingcomputer parallelism. A distinction is made between shared-memory and distrib-uted-memory (message-passing) computers; parallel processors are also classifiedby their interconnection structures. Examples of static interconnections are meshesand hypercubes, while dynamic interconnections are exemplified by shared busesand multistage interconnection networks (MINs). The performance of a parallelprocessor depends on its architecture and the programs it executes. A basic perfor-mance measure is the speedup S(n), defined as the ratio of execution time on asequential computer to execution time on a comparable computer of parallelism n.The speedups achieved in practice are less than n due to such effects as memorycontention and the presence of nonparallelizable code.

A computer containing more than one CPU is a multiprocessor. The CPUs canbe tightly coupled via shared memory or loosely coupled via messages transmittedbetween the processors' local memories. Multiprocessors have been designedaround various interconnection networks of which the shared bus is the most com-mon. Advances in VLSI technology have made it feasible to construct massivelyparallel distributed-memory machines using such interconnection structures ashypercubes. Some large multiprocessors rely on MINs like the omega network forprocessor-memory or processor-processor communication. A few multiprocessorshave fault tolerance as a primary design goal, which they achieve via various formsof static or dynamic redundancy; for example, n-modular redundancy (nMR). Faulttolerance is measured by reliability, availability, and the mean times to failure(MTTF) and repair (MTTR).

7.5 PROBLEMS

7.1. Explain why the single shared bus is so widely used as an interconnection medium inboth sequential and parallel computers. What are its main disadvantages?

7.2. A useful characteristic of an interconnection network represented by an /i-node graphG is its bisection width, defined as the minimum number of edges that must be

removed to divide G into two parts, such that each part contains nil edges (if n is even) or one part contains one more edge than the other (if n is odd). A small bisection width can correspond to a dataflow bottleneck and so is undesirable. Calculate the bisection widths of the six graphs in Figure 7.11.

SECTION 7.5 Problems

(a)

00

Figure 7.84

Two proposed interconnection structures for computers: (a) pyramid and (b) cube-connected-cycles network.

7.3. A pyramid graph consists of a complete, quaternary (degree 4) rooted tree of k levels, with extra links to make every level into a two-dimensional mesh. With the apex(root) as level 1, each level k contains $4k~l$ processors forming a $21"' x 2*\_I$ mesh. A three-level pyramid appears in Figure 7.84a. (a) Calculate the number of nodes, the maximum node degree, and the maximum internode distance (diameter) in a &-level pyramid, (b) A pyramid tries to combine the advantages of mesh and tree networks. To what extent is it successful?

7.4. A cube-connected-cycles (CCC) graph is formed from a ^-dimensional hypercube by replacing each node jc, (which is of degree k) of the hypercube with a &-node ring or cycle C,. Each node of C, is connected to a distinct edge of the J-member set originally connected to xt. A three-dimensional CCC graph appears in Figure l.S4b. (a) Calculate the number of nodes, the maximum node degree, and the diameter of a fc-dimensional CCC graph, (b) To what extent is the CCC graph an improvement over the hypercube as a computer interconnection structure?

7.5. Define each of the following terms in the context of bus design: handshaking, lock sig-nal, master unit, skew, tristate, wait state.

7.6. Analyze the three bus-arbitration methods—daisy chaining, polling, and independent requesting—with respect to communication reliability in the event of hardware fail-ures.

7.7. Consider the timing diagram for a read operation over the PCI bus shown in Figure 7.25. (a) Draw a similar timing diagram to show a four-word read transfer occurring at the maximum possible rate (burst mode), (b) Repeat this problem for a four-word burst-mode write operation.

7.8. Intel designed the Multibus (IEEE Standard 796) as a standard system bus for micro-processor-based computers. It supports a heterogeneous set of 8- and 16-bit micropro-cessors in multiprocessing configurations. Figure 7.85 summarizes the 86 lines (excluding 20 for power and ground) that make up the Multibus, (a) How large a mem-ory address space is supported (without special logic)? (b) What types of IO addressing are supported?

Signal type	Buslines	Functions
Data and address	DAT0A5	Data bus (16 lines)
	ADR0.23	Address bus (24 lines)
Data-transfer control	MRDC	Memory read enable
and handshaking	IORC	IO read enable
	MWTC	Memory write enable
	I owe	IO write enable
	XACK	Acknowledge
Bus arbitration and	BREQ	Bus request
timing	CBRQ	Common bus request
	BUSY	Bus busy
	BCLK	Bus clock
	BPRN	Bus priority in
	BPRO	Bus priority out
Interrupt control	INTO:!	Interrupt request (8 lines)
	INTA	Interrupt acknowledge
Miscellaneous control	CCLK	Master clock
	INIT	System initialization
	BHEN	Byte high enable
	INH 1:2	Inhibit memory (2 lines)

LOCK   Lock bus

CHAPTER 7

System

Organization

Figure 7.85

Structure of the Multibus (IEEE 796) standard bus.

7.9. The Multibus (Figure 7.85) has a set of arbitration lines for transferring bus control among a set of potential master units. BUSY is activated (BUSY - 0) by the current bus master, and this line prevents any other unit from becoming master until it is deac-tivated. When BUSY = 1, a unit can gain control of the Multibus via the bus priority lines BPRN and BPRO, which can be daisy-chained as shown in Figure 7.86. A po-tential master then requests control of the Multibus by deactivating its BPRO line, which prevents all lower-priority units from accessing the bus. The requesting unit takes control of the bus if its own BPRN line has not been deactivated by a higher-priority unit. Design a faster, parallel method for arbitration of the Multibus that uses

Multibus

fe Bus

<   <:

5; master 2 ~

Highest priority

Figure 7.86

Some of the bus-arbitration logic in the Multibus.

Lowest priority

582 only the existing control lines and a small amount of extra logic. Assume that up to

eight potential bus masters can be present.

SECTION 7.5 Problems

7.10. Compare and contrast the CSMA/CD and token-passing network-arbitration tech-niques from the viewpoints of response time, fairness, and fault tolerance.

7.11. A computer network's reliability is sometimes measufed by its connectivity. The node connectivity $cN(G)$ of network G is defined as the smallest number of nodes whose re-moval disconnects G, that is, eliminates all paths between at least two nodes, or else reduces G to the trivial 1-node 0-edge graph GT. (a) What is the node connectivity of the ARPANET as it appears in Figure 1.31? (b) What is $cN(G)$ when $G = Kn$, the com-plete graph of n nodes?

7.12. Another measure of the reliability of a network G (see the preceding problem) is its edge connectivity $cE(G)$, defined as the smallest number of edges whose removal dis-connects G or reduces it to GT. If G has n nodes and m edges, then prove that $cE(G) <$

l(2m)/nj.

7.13. Define each of the following IO control methods: programmed 10, DMA controllers, IOPs. List the advantages and disadvantages of each method with respect to program-design complexity, 10 bandwidth, and interface hardware costs.

7.14. Consider a 32-bit microprocessor with 32-bit data and address buses. The CPU clock frequency is 50 MHz, and a memory load or store instruction cycle takes two clock cycles. Memory-mapped 10 is used, and the CPU supports both vectored interrupts and DMA block transfers with arbitrary block length. Typical interrupt response time is 15 CPU clock cycles. It is desired to add to the system a hard disk drive with a data-transfer rate of N bits/s. Estimate the maximum value that N can have for each of the following ways of controlling the disk drive: programmed IO and DMA. Show your calculations, and state all your assumptions.

7.15. (a) A typical CPU allows most interrupt requests to be enabled and disabled under soft-ware control. In contrast, no CPU provides facilities to disable DMA request signals. Explain why this is so. (b) Suppose you want to be able to occasionally delay a CPU's response to a DMA request until the end of the current instruction cycle. Design the necessary add-on logic to implement this type of delayed DMA request, assuming that a conventional one-chip CPU is being used whose internal hardware or instruction set cannot be modified. A pair of existing instructions should serve to turn on (enable) and turn off (disable) the DMA delay. State clearly all the assumptions underlying your de-sign.

7.16. A CISC computer consists of a CPU and an IO device D connected to main memory M via a one-word shared bus. The CPU can execute a maximum of IO6 instructions per second. An average instruction requires five machine cycles, three of which use the memory bus. A memory read or write operation uses one machine cycle. Suppose that the CPU is continuously executing "background" programs that require 90 percent of its instruction execution rate but no IO instructions. Now D is to be used to transfer very large blocks of data to and from M. (a) If programmed IO is used and each one-word IO transfer requires the CPU to execute two instructions, estimate the maximum IO data-transfer rate rMAX possible through D. (b) Estimate rMAX if DMA is used.

7.17. In addition to supporting memory-IO communication, some DMA controllers and IOPs also support block transfers from one region of main memory to another; that is, they perform memory-to-memory communication via DMA block transfers. (a) Ex-plain how a main-memory block transfer can be implemented by an IOP such as the Intel 8089. Describe also the IO instructions needed to set up this type of operation. (b) What are the advantages and disadvantages of this type of main-memory block transfer

compared with implementing the same data transfer by means of a BLOCK MOVE in-struction, such as is found in some CPU instruction sets?

7.18. Often a new model of a microprocessor has instructions not found in older members of the same microprocessor family. The older microprocessors can, however, be updated by providing them with programs that implement the new instructions in software, a process called emulation. (Note the resemblance to emulation of instructions via mi-croprograms.) Explain how an old microprocessor can use an interrupt mechanism to determine when a particular instruction should be emulated in this way, rather than be executed directly.

7.19. Consider the pipelined multiply and add instructions appearing in Figure 7.42. Suppose that the number of execution stages of multiply is increased from four to six (EX 1:6) and the number of execution stages of add is increased from one to two (EX1:2). Con-sider execution of the following three-instruction code segment.

rl := r4 x rO; r2 := r4 + r6; r3 := r2 x r5;

(7.31)

7.20.

(a) What is the minimum number of cycles to process this code with out-of-order completion allowed? (b) What is the minimum number of cycles to process this code with in-order completion? Include in your answers timing diagrams in the style of Figure 7.42.

Imprecise interrupts can be avoided without the performance penalty of in-order com-pletion if a check is made in advance for interrupt-causing conditions. For example, floating-point multiply instructions only generate interrupts due to overflow or under-flow. Overflow occurs only if the sum of the multiplier and multiplicand's exponents plus one exceeds the largest valid exponent value; a similar condition holds for under-flow. The Pentium's floating-point logic contains special hardware to test

for condi-tions of this sort. If the potential interrupt conditions are not present, fast, out-of-orderexecution is permitted in situations like that of Figure 7.42; otherwise, in-order execu-tion is enforced. Suppose that in the preceding problem, the multiplier completes a testfor potential interrupts in two clock cycles. What is the minimum number of cyclesneeded to process the code (7.31) when no potential interrupts are detected for multi-ply? Give a timing diagram for this case in the style of Figure 7.42.

7.21. Instructions such as store instructions that modify memory make it difficult to supportprecise interrupts in pipelined CPUs. Why is this so? Outline a design method to solvethis problem.

7.22. What are the advantages of defining two distinct classes of software processes for sys-tem management: system (supervisor) processes and user processes? Describe thehardware features typically provided in a CPU to support this process dichotomy.

7.23. The following three-instruction program written in 80X86 assembly language is pro-posed for implementing the wait or test-and-set function for a binary semaphore 5; allmajor actions of the instructions are specified by the comments. The CPU is connectedvia the Multibus (refer to Figure 7.85) to a global memory storing 5. The MultibusLOCK signal is not activated unless the prefix LOCK precedes an instruction to whichthe signal is applicable.

WAIT: TESTS,0

JNE WAIT

Fetch the variable S and compare to zero. Set the Z flagto 1 if S = 0 (not busy); otherwise, set the Z flag to 0.

Jump to WATT if Z = 1; otherwise, continue to nextinstruction.

583

CHAPTER 7

SystemOrganization

MOVS, 1 Set S to 1 (busy)

584 (a) Explain why this code fails to meet the mutual exclusion requirement for sema-

phore access, (b) Design a replacement program that solves this problem, using com-ments to explain your instructions. Indicate how exclusive access to S is ensured.

Problems

7.24. Consider the operating system state described by the resource allocation graph G ofFigure 7.51. Let resource R6 and the edges connected to it be removed from G to forma new graph G'. (a) Does G' contain a deadlock? {0) Suppose that P3 and P5 requestaccess to R3 in G'. Can these new requests lead to deadlock? (c) Suppose that P, andP2 request access to a new resource /?7 added to G'. Can this lead to deadlock?

7.25. (a) Identify and briefly compare the mechanisms available for interprocess communi-cation in the UNLX operating system, (b) What are the advantages and disadvantagesof treating all 10 devices as logical files in the manner of UNDC?

7.26. Redesign the parallel summation program of Figure 7.56 for execution by the binarytree computer whose structure appears in Figure 1.60d. Assume that N=2P~1 and thatthe N numbers to be added are stored in the leaf nodes initially. The final sum is to bestored in the topmost (root) node.

7.27. Classify under the headings (/') shared/distributed memory, and (if) SIMD/MIMD/MISD, the following computers mentioned in this chapter: nCUBE 2, Sequent Sym-metry, Tandem Himalaya. Identify each computer's interconnection structure type.

7.28. Let 30 be the degree of parallelism of a certain parallel computer C. Let/ be the frac-tion of the operations performed by C that are strictly scalar (cannot be processed inparallel). Assume that all other operations are processed at the maximum possible(vector) rate. Let 20 be the speedup achieved by C for the tasks under consideration.

(a) What is/? (b) By how much must/be changed to increase the speedup to 90 per-cent of the maximum possible?

7.29. Consider a vector supercomputer that processes vectors whose average length is N. Theaverage setup time for vector operations is T0, and the CPU (and pipeline) clock periodis rdock. Derive an expression for the efficiency E of the computer in terms of N, T0,

and Tdock-

7.30. It has been conjectured from observing real multiprocessors, that because of memoryand bus conflicts, algorithm inefficiencies, and so on, the actual speedup S(n) obtainedwhen n identical processors are used to execute a single large program Q lies betweenlog2 n and nl\o%e n. Show that if we assume the probability of being able to assign Q toi processors is l/i, for i = 1,2, ..., n, we obtain the upper bound nfloge n on S(n).

7.31. (a) Let s denote the fraction of time that must be spent on the serial parts of a pro-gram Q, and let p denote the fraction spent on the parallelizable parts of Q. Assumingthat s + p = 1, show that Amdahl's law for the speedup S(n) achievable by an «-pro-cessor computer executing Q can be reformulated as follows:

S(n) = —±—s + p/n

(b) Amdahl's law makes the implicit assumption that/? is independent of N. In prac-tice, the problem size tends to increase with n\ that is, problems expand to use theadditional processors. This situation suggests that the time for a serial processor toexecute Q should be represented by s + pn, given that it runs in time s + p - I on theparallel processor. With this assumption, derive an alternative expression for S(n).Comment on its implications concerning the performance of massively parallel com-puters.

7.32. A useful measure of communication delay in static multiprocessor interconnectionstructures is the average distance dav between all pairs of nodes (processors). Calculaterfav as a function of n for any three of the six structures listed in Figure 7.11.

7.33. A multiprocessor with two CPUs Px and P2 employs the shared-bus multiprocessor 585organization (Figure 7.63) and the MESI cache-coherence protocol (Figure 7.65). As-sume each local memory is an LI cache. List the actions of Px and P2 in response toeach of the following situations, giving the final states of all affected cache blocks:(a) Px reads a word Wx that is in its cache (a read hit). P2 also has a copy of Wx, andboth copies are marked 5 (shared), (b) Px writes to Wx in its cache (a write hit). AgainP2 has a copy of Wx, and both copies are marked 5. (c) />, writes to Wx, but now Wx isnot assigned to its cache (a write miss). However, P2 has a cache copy of Wx that ismarked E (exclusive), (d) Px reads Wx, but Wx is not assigned to its cache (a readmiss). Once more, P2 has a cache copy of Wx that is marked E (exclusive).

7.34. Consider the MESI cache-coherence protocol as defined in Figure 7.65. Some of theindicated state transitions caused by regular or snoop hits and misses force a blocktransfer from global to local (cache) memory. Identify three such cases and briefly ex-plain why the block transfer is needed.

7.35. The PowerPC Model 603, unlike the Model 601, employs a three-state cache coher-ence protocol called MEI, which is defined as a "coherent subset" of the MESI protocolthat omits the 5 (shared) state, (a) Since processors in a multiprocessor configura-tion of 603s still need to know whether their cache data is shared, suggest how the MEIprotocol handles this issue, (b) Construct a state diagram similar to Figure 7.65 for theMEI protocol.

7.36. With Figure 7.66 as a guide, devise a general labeling procedure to embed a two-dimensional mesh of size nx x n2 in an n-dimensional hypercube for some n. Illustrateyour method by using it to embed a 4 x 4 mesh in a four-dimensional hypercube.

7.37. (a) Show by construction that for sufficiently large n, it is possible to embed a &-nodecycle (ring), where k is even, in an ^-dimensional hypercube graph, (b) Show that nomatter how large n is, it is impossible to embed the pyramid graph of Figure 7.84a, orany larger pyramid, in a hypercube.

7.38. A multiprocessor node must sometimes send a message to more than one other proces-sor, a task referred to as broadcasting. Suppose that a node P0 in an n-dimensional hy-percube system has to broadcast a message to all 2" - 1 other processors. Thebroadcasting is subject to the constraints that the message can be forwarded (retrans-mitted) by a node only to a neighboring node and that each node can transmit only onemessage at a time. Assume that each message transmission between adjacent nodes re-quires one time unit. In a two-dimensional system, for example, P0 could broadcast amessage MESS as follows: At time / = 0, P0 sends MESS to Px. At t - 1, P0 sendsMESS to P2 and Px sends MESS to F3, thus completing the broadcast in two time units.Construct a general broadcasting algorithm for the ^-dimensional case that allows amessage to reach all nodes in n time units. Specify clearly the algorithm used by eachnode to determine the neighboring nodes to which it should forward an incoming mes-sage.

7.39. Figure 7.87 shows a three-stage version N of an indirect hypercube MIN. (a) Supposethe four switching elements 5,2, S2 2, 53 2, S4 2 forming stage 2 are set to the X state,while the eight remaining switches are set to the T state. Show that these switch settingssimultaneously connect the output of Ptjk\.o the input of P-.. for all i,j,k. (b) Determinethe switch settings needed to connect P^to /*- for all i,j,k, and also the settings need-ed to connect Pijk to Pv7 for all i,j,k. (c) Explain why N is called a hypercube network.

7.40. Construct a diagram for a four-stage 16 x 16 omega network in the same style as Figure7.74. Show the switch settings required to connect input port 3 to output port 12.

7.41. The through (T) and cross (X) states of the switching element 5 of Figure 7.68 can beaugmented by the two additional states defined in Figure 7.88. These are termed the

CHAPTER 7

Stage 1

Stage 2

Stage 3

''ooo —»•^001 —►-^010 —*■''on —►■''ioo —►■^101 —►•''no —*■Pin —

SECTION 7.5Problems *u   Si*   S..3

lX,

                                                                                    \jS

            su   S2.2   «   yC
                                                                       S2.3

                                                                  y

                                                                  \/\/ *

                                                                            A A

            S3.i   ^3.2 *3.3 /\ /\

            r^i   A

                                                                  V

                                                                            /\

            ■^4.1   •^4.2 *4.3 ^

r000^OOl^010

^011^lOO^lOl

^no

Pill

Figure 7.87

Three-stage indirect hypercube network.

upper (U) and lower (L) broadcast states because they allow an incoming message tobe sent to both output ports simultaneously. Show that if the two-state switch 5 of Figure 7.68 is replaced by the four-state switch 5', then an N x AT omega network has a statethat allows data on any of its input ports to be broadcast directly to any subset of itsoutput ports.

7.42. Show that deleting the final stage of an N x N omega network with n = log2N stagesdestroys its full-access property.

7.43. A MIN linking a set of processors is said to provide dynamic full access if any proces-sor Pj can be connected to any other processor P- by a finite number of passes throughthe MIN, where any intermediate processors visited act as store-and-forward stations.Clearly a full-access network can link any processor-pair in a single pass, (a) Showthat if stage 3 is deleted from the MIN of Figure 7.87 the resulting two-stage MIN hasthe dynamic full-access property but not the full-access property, (b) Is dynamic full-access retained after deleting two stages from this MIN? Justify your answer.

7.44. Determine whether or not the 4 x 4 switching element used in the BBN Butterfly com-puter has the full-access and nonblocking properties.

7.45. A computer series has a mean failure rate of one fault in 5 years; this rate remains fairlyconstant over a normal 10-year life. If a customer purchases a new computer of thistype, what is the probability that at least one fault will occur by the end of the first year?

X,

            *i —

            X-,

      —*z2

            ^

1
      c = 3
c = 2

  (a)   (b)

z,

Figure 7.88

Extended switching elements(a) upper broadcast state U;and (b) lower broadcaststate L.

7.46. A certain computer part is assumed to follow the exponential failure law. The proba-bility that it does not survive more than 50 days is 0.92. How often can one expect tohave to replace this particular part?

7.47. Let F(t) be the unreliability function for a certain class of components. The hazardfunction z(t), which is interpreted as the instantaneous failure rate, is defined by

$z(t) =$

fit)

where$/(r) = d-\hat{}F(t)$ dt

7.48.

Suppose that fit) = 0.25 - 0.03125r, where t is measured in years. Calculate the reli-ability function R(t), the hazard function z(t), and the mean time to failure (MTTF) forthese components.

(a) A system is constructed by connecting n copies of a unit U in parallel. If the reli-ability of U is 0.8, how many copies of U are needed in order for the system reliability7to be (/) at least 0.9, and (ii) at least 0.999? (b) A certain server crashes about once everythree days. It takes an average of 3.5 hours to restore normal operation. What are thesystem's availability and MTTF?

7.49. A variant of TMR called TMR/Simplex has triplicated units and a match circuit to iden-tify the failed unit when the first failure occurs. The system begins operation as a TMRconfiguration. When the first failure is detected, the system structure is changed fromTMR to simplex using one of the two correctly working units. Normal operation thencontinues until the simplex configuration fails. If the reliability of each unit is e~'J andthe voter and match circuit are perfectly reliable, calculate the reliability and MTTF ofthe TMR/Simplex system.

7.50. Consider the 14 x 4 torus that forms the interconnection network linking nodes (sec-tions) in the Tandem Himalaya computer. Determine each of the following parametersfor this network: (a) the diameter of the network; (b) the minimum number of edgesneeded to break the network into two disconnected parts; (c) the minimum number ofedges needed to break it into two disconnected parts, each having the same number ofnodes (this is the bisection width).

587

CHAPTER 7

System

Organization

7.6 REFERENCES

1. Anderson. D. and T. Shanley. Pentium Processor System Architecture. 2nd ed. Reading.MA: Addison-Wesley, 1995.

2. Avizienis, A. "Fault Tolerant Computing—An Overview." IEEE Computer, vol. 4 (Jan-uary /February 1971) pp. 5-8.

3. Chen, P. M. et al. "RAID: High-Performance. Reliable Secondary Storage." ACM Com-puting Sun-eys, vol. 26 (June 1994) pp. 145-85.

4. Crowther. W. et al. "The Butterfly Parallel Processor." IEEE Computer ArchitectureNewsletter (September/December 1985) pp. 18^45.

5. El-Ayat. K. A. "The Intel 8089: An Integrated IO Processor." IEEE Computer, vol. 12(June 1979) pp. 67-78.

6. Feng, T. Y. "A Survey of Interconnection Networks." IEEE Computer, vol. 14 (Decem-ber 1981) pp. 12-27.

7. Flynn, M. J. "Very High-Speed Computing Systems." Proceedings of the IEEE. vol. 54(December 1966) pp. 1901-09.

8. Gustavson. D. B. "Computer Buses—A Tutorial." IEEE Micro, vol. 4 (August 1984)pp. 7-22.

SECTION 7.6 References

588 9. Hayes, J. P. and T. N. Mudge. "Hypercube Computers." Proceedings of the IEEE, vol.

77 (December 1989) pp. 1829—41.

10. Hwang, K. Advanced Computer Architecture. New York: McGraw-Hill, 1993.

11. IBM Corp. IBM System/370 Principles of Operation. White Plains, NY: IBM, 1974.

12. Intel Corp. Peripheral Components. Santa Clara, CA, 1993.

13. Intel Corp. i960 RP I/O Processor. Santa Clara, CA, 1996.

14. Kong, C. "A Hardware Overview of the NonStop Himalaya K10000 Server." TandemSystems Review, vol. 10 (January 1994).

15. Motorola Inc. PowerPC 603 RISC Microprocessor User's Manual. Phoenix, AZ, 1994.(Also published by IBM Microelectronics, Essex Junction, VT, 1994.)

16. Moudgill, M. and S. Vassilliadis. "Precise Interrupts." IEEE Micro, vol. 16 (February1996)pp. 58-87.

17. nCUBE Corp. nCUBE 2 Supercomputers. Beaverton, OR, 1990.

18. Quinn, M. J. Parallel Computing: Theory and Practice. 2nd ed. New York: McGraw-Hill, 1994.

19. Sequent Computer Systems Inc. Symmetry 5000 Series. Beaverton, OR, 1996.

20. Shanley, T. and D. Anderson. PCI System Architecture. 3rded. Reading, MA: Addison-Wesley, 1995.

21. Siegel, H. J. Interconnection Networks for Large-Scale Parallel Processing. 2nd ed.New York: McGraw-Hill, 1990.

22. Siewiorek, D. P. and R. S. Swarz (eds.). Reliable Computer Systems. 2nd ed. Burlington,MA: Digital Press, 1992.

23. Silberschatz, A. and P. B. Galvin. Operating System Concepts. 4th ed. Reading, MA:Addison-Wesley, 1994.

24. Simonds, F. McGraw-Hill LAN Communications Handbook. New York: McGraw-Hill,1994.

25. Southerton, A. Modern UNIX. New York: John Wiley and Sons, 1993.

26. Stenstrom, P. "A Survey of Cache Coherence Schemes for Multiprocessors." IEEEComputer, vol. 23 (June 1990) pp. 12-24.

27. Thurber, K. J. et al. "A Systematic Approach to the Design of Digital Bussing Struc-tures." AFIPS Conference Proceedings, vol. 41 (1972) pp. 719-^10.

28. Triebel, W. A. and A. Singh. The 68000 and 68020 Microprocessors. Englewood Cliffs,NJ: Prentice-Hall, 1991.

591

592

593

INDEX

594

INDEX

601

602

$7.25

Computer Architecture and Orgai

3rd edition, provides a comprehensive and up-to-dateview of the architecture and internal organization ofcomputers from a mainly hardware perspective. Witha balanced treatment of qualitative and quantitativeissues. Hayes focuses on the understanding of thebasic principles while avoiding overemphasis on thearcane aspects of design. This approach best meetsthe needs of undergraduate or beginning graduate-level students.

Some Key Features

■ Emphasis on Basic Principles: The bookprovides a clear and self-contained treatment ofall the fundamental concepts of computer designand their realization in modern computer systems.

■ Broad and Balanced Coverage: The full rangeof computer types, from desktop computers, tohigh-performance multiprocessors, is addressed.Quantitative topics such as performance analysisare given equal treatment with traditionalqualitative issues.

I Systematic, Multilevel Design Methodology:

This edition continues the author's systematicapproach to design, which clearly identifies thedesign levels and the design procedures withineach level.

I Expanded and Extensive Problem Sets: The

much used and often praised problem sets havebeen expanded from 240 to about 3(X) problems.80 percent of which are new in this edition.

9 780071"159975