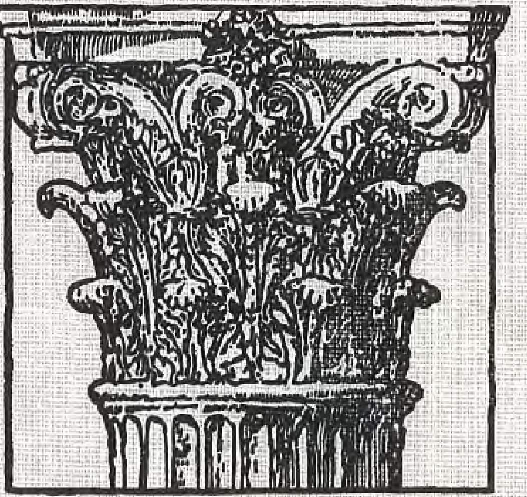
COMPUTER ARCHITECTURE A QUANTITATIVE APPROACH



JOHN L HENNESSY
&
DAVID A PATTERSON

Morgan Kaufmann Publishers, Inc.
P.O. Box 50490
Palo Alto, California 94303

Computer Systems and Design
Electrical Engineering
ISBN 1-55860-069-8

JOHN L HENNESSY
&
DAVID A PATTERSON

COMPUTER ARCHITECTURE
A
QUANTITATIVE APPROACH

MORGAN
KAUFMANN
PUBLISHERS
INC.

Computer Architecture Definitions, Trivia, Formulas, and Rules of Thumb

Definitions

Big Endian: the byte with the binary address "x...x00" is in the most significant position ("big end") of a 32-bit word (page 95).
Clock rate: inverse of clock cycle time, usually measured in MHz (page 36).
CPI: clock cycles per instruction (page 36).
Hit rate: fraction of memory references found in the cache, equal to 1 – Miss rate (page 404).
Hit time: memory-access time for a cache hit, including time to determine if hit or miss (page 405).
Instruction count: number of instructions executed while running a program (page 36).
Little Endian: the byte with the binary address "x...x00" is in the least significant position ("little end") of a 32-bit word (page 95).
MIMD: (multiple instruction stream, multiple data stream) a multiprocessor or multicomputer (page 572).
Miss penalty: time to replace a block in the top level of a cache system with the corresponding block from the lower level (page 405).
Miss rate: fraction of memory references not found in the cache, equal to 1 – Hit rate (page 404).
$N_{1/2}$: the vector length needed to reach one-half of R_∞ (page 384).
N_v : the vector length needed so that vector mode is faster than scalar mode (page 384).
R_∞ : the megaflop rate of an infinite-length vector (page 384).
RAW data hazard: (read after write) instruction tries to read a source before a prior instruction writes it, so it incorrectly gets the old value (page 264).
SIMD: (single instruction stream, multiple data stream) an array processor (page 572).
SISD: (single instruction stream, single data stream) a uniprocessor (page 572).
Spatial locality: (locality in space) if an item is referenced, nearby items will tend to be referenced soon (page 403).
Temporal locality: (locality in time) if an item is referenced, it will tend to be referenced again soon (page 403).
WAR data hazard: (write after read) instruction tries to write a destination before it is read by a prior instruction, so prior instruction incorrectly gets the new value (page 264).
WAW data hazard: (write after write) instruction tries to write an operand before it is written by a prior instruction. The writes are performed in the wrong order, incorrectly leaving the value of the prior instruction in the destination (page 264).

Trivia

Byte order of machines (page 95)

Big Endian: IBM 360, MIPS, Motorola, SPARC, DLX
Little Endian: DEC VAX, DEC RISC, Intel 80x86

Year and User Address Size of Generations of IBM and Intel Computer Families

Year	Model	User address size	Year	Model	User address size
1964	IBM 360	24	1978	Intel 8086	4+16
1971	IBM 370	24	1981	Intel 80186	4+16
1983	IBM 370-XA	31	1982	Intel 80286	16+16
1986	IBM ESA/370	16+31	1985	Intel 80386	16+32 or 32
			1989	Intel 80486	16+32 or 32

Formulas

1. *Amdahl's Law*: $Speedup = \dfrac{1}{(1-\text{Fraction}_{\text{enhanced}}) + \dfrac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$ (page 8)

2. *CPU time* = Instruction count * Clock cycles per instruction * Clock cycle time (page 36)

3 *Average memory-access time* = Hit time + Miss rate * Miss penalty (page 405)

4. *Means*—arithmetic(AM), weighted arithmetic(WAM), harmonic(HM) and weighted harmonic(WHM):

$$\text{AM} = \frac{1}{n}\sum_{i=1}^{n}\text{Time}_i,\ \text{WAM} = \sum_{i=1}^{n}\text{Weight}_i * \text{Time}_i,\ \text{HM} = \frac{n}{\sum_{i=1}^{n}\frac{1}{\text{Rate}_i}},\ \text{WHM} = \frac{1}{\sum_{i=1}^{n}\frac{\text{Weight}_i}{\text{Rate}_i}}$$

where Time_i is the execution time for the ith program of a total of n in the workload, Weight_i is the weighting of the ith program in the workload, and Rate_i is a function of $1/\text{Time}_i$ (page 51).

5. *Cost of integrated circuit* $= \dfrac{\text{Cost of die + Cost of testing die + Cost of packaging}}{\text{Final test yield}}$ (page 55)

6. *Die yield* = Wafer yield * $\left\{1 + \dfrac{\text{Defects per unit area} * \text{Die area}}{\alpha}\right\}^{-\alpha}$

where Wafer yield accounts for wafers that are so bad they need not be tested and α corresponds to the number of masking levels critical to die yield (usually $\alpha \geq 2.0$, page 59).

7. *Pipeline speedup* $= \dfrac{\text{Clock cycle time}_{\text{no pipelining}}}{\text{Clock cycle time}_{\text{pipelined}}} * \dfrac{\text{Ideal CPI} * \text{Pipeline depth}}{\text{Ideal CPI + Pipeline stall cycles per instruction}}$

where Pipeline stall cycles accounts for clock cycles lost due to pipeline hazards (page 258).

8. *System performance:*

$$\text{Time}_{\text{workload}} = \frac{\text{Time}_{\text{CPU}}}{\text{Speedup}_{\text{CPU}}} + \frac{\text{Time}_{\text{I/O}}}{\text{Speedup}_{\text{I/O}}} - \frac{\text{Time}_{\text{overlap}}}{\text{Maximum}(\text{Speedup}_{\text{CPU}}, \text{Speedup}_{\text{I/O}})}$$

where Time_{CPU} means the time the CPU is busy, $\text{Time}_{\text{I/O}}$ means the time the I/O system is busy, and $\text{Time}_{\text{overlap}}$ means the time both are busy. This formula assumes the overlap scales linearly with speedup (page 506).

Rules of Thumb

1. *Amdahl/Case Rule*: A balanced computer system needs about 1 megabyte of main memory capacity and 1 megabit per second of I/O bandwidth per MIPS of CPU performance (page 17).
2. *90/10 Locality Rule*: A program executes about 90% of its instructions in 10% of its code (pages 11–12).
3. *DRAM-Growth Rule*: Density increases by about 60% per year, quadrupling in 3 years (page 17).
4. *Disk-Growth Rule*: Density increases by about 25% per year, doubling in 3 years (page 17).
5. *Address-Consumption Rule*: The memory needed by the average program grows by about a factor of 1.5 to 2 per year; thus, it consumes between 1/2 and 1 address bit per year (page 16).
6. *90/50 Branch-Taken Rule:* About 90% of backward-going branches are taken while about 50% of forward-going branches are taken (page 108).
7. *2:1 Cache Rule*: The miss rate of a direct-mapped cache of size X is about the same as a 2-way–set-associative cache of size X/2 (page 421).

Computer Architecture A Quantitative Approach

Computer Architecture A Quantitative Approach

David A. Patterson
UNIVERSITY OF CALIFORNIA AT BERKELEY

John L. Hennessy
STANFORD UNIVERSITY

With a Contribution by
David Goldberg
Xerox Palo Alto Research Center

MORGAN KAUFMANN PUBLISHERS, INC.
SAN MATEO, CALIFORNIA

Sponsoring Editor Bruce Spatz
Production Manager Shirley Jowell
Technical Writer Walker Cunningham
Text Design Gary Head
Cover Design David Lance Goines
Copy Editor Linda Medoff
Proofreader Paul Medoff
Computer Typesetting and Graphics Fifth Street Computer Services

Library of Congress Cataloging-in-Publication Data
Patterson, David A.
Computer architecture : a quantitative approach / David A. Patterson, John L. Hennessy
p. cm.
Includes bibliographical references
ISBN 1-55860- 069-8
1. Computer architecture. I. Hennessy, John L. II. Title.
QA76.9.A73P377 1990
004.2'2--dc20 89-85227
CIP

Morgan Kaufmann Publishers, Inc.
Editorial Office: 2929 Campus Drive. San Mateo, CA 94403
Order from: P.O. Box 50490, Palo Alto, CA 94303-9953

ADVICE, PRAISE, & ERRORS: Any correspondence related to this publication or intended for the authors should be addressed to the editorial offices of Morgan Kaufmann Publishers, Inc., Dept. P&H APE. Information regarding error sightings is encouraged. Any error sightings that are accepted for correction in subsequent printings will be rewarded by the authors with a payment of $1.00 (U.S.) per correction upon availability of the new printing. Electronic mail can be sent to bugs3@vsop.stanford.edu. (Please include your full name and permanent mailing address.)

INSTRUCTOR SUPPORT: For information on classroom software and other instructor materials available to adopters, please contact the editorial offices of Morgan Kaufmann Publishers, Inc. (415) 578-9911.

Third printing, 1993

To Andrea, Linda, and our four sons

Trademarks

The following trademarks are the property of the following organizations:

Alliant is a trademark of Alliant Computers.

AMD 29000 is a trademark of AMD.

TeX is a trademark of American Mathematical Society.

AMI 6502 is a trademark of AMI.

Apple I, Apple II, and Macintosh are trademarks of Apple Computer, Inc.

ZS-1 is a trademark of Astronautics.

UNIX and UNIX F77 are trademarks of AT&T Bell Laboratories.

Turbo C is a trademark of Borland International.

The Cosmic Cube is a trademark of California Institute of Technology.

Warp, C.mmp, and Cm* are trademarks of Carnegie-Mellon University.

CP3100 is a trademark of Conner Peripherals.

CDC 6600, CDC 7600, CDC STAR-100, CYBER-180, CYBER 180/990, and CYBER-205 are trademarks of Control Data Corporation.

Convex, C-1, C-2, and C series are trademarks of Convex.

CRAY-3 is a trademark of Cray Computer Corporation.

CRAY-1, CRAY-1S, CRAY-2, CRAY X-MP, CRAY X-MP/416, CRAY Y-MP, CFT77 V3.0, CFT, and CFT2 V1.3a are trademarks of Cray Research.

Cydra 5 is a trademark of Cydrome.

CY7C601, 7C601, 7C604, and 7C157 are trademarks of Cypress Semiconductor.

Nova is a trademark of Data General Corporation.

HEP is a trademark of Denelcor.

CVAX, DEC, DECsystem, DECstation, DECstation 3100, DECsystem 10/20, fort, LP11, Massbus, MicroVAX-I, MicroVAX-II, PDP-8, PDP-10, PDP-11, RS-11M/IAS, Unibus, Ultrix, Ultrix 3.0, VAX, VAXstation, VAXstation 2000, VAXstation 3100, VAX-11, VAX-11/780, VAX-11/785, VAX Model 730, Model 750, Model 780, VAX 8600, VAX 8700, VAX 8800, VS FORTRAN V2.4, and VMS are trademarks of Digital Equipment Corporation.

BINAC is a trademark of Eckert-Mauchly Computer Corporation.

Multimax is a trademark of Encore Computers.

ETA 10 is a trademark of the ETA Corporation.

SYMBOL is a trademark of Fairchild Corporation.

Pegasus is a trademark of Ferranti, Ltd.

Ferrari and Testarossa are trademarks of Ferrari Motors.

AP-120B is a trademark of Floating Point Systems.

Ford and Escort are trademarks Ford Motor Co.

Gnu C Compiler is a trademark of Free Software Foundation.

M2361A, Super Eagle, VP100, and VP200 are trademarks of Fujitsu Corporation.

Chevrolet and Corvette are trademarks of General Motors Corporation.

HP Precision Architecture, HP 850, HP 3000, HP 3000/70, Apollo DN 300, Apollo DN 10000, and Precision are trademarks of Hewlett-Packard Company.

S810, S810/200, and S820 are trademarks of Hitachi Corporation.

Hyundai and Excel are trademarks of the Hyundai Corporation.

432, 960 CA, 4004, 8008, 8080, 8086, 8087, 8088, 80186, 80286, 80386, 80486, iAPX 432, i860, Intel, Multibus, Multibus II, and Intel Hypercube are trademarks of Intel Corporation.

Inmos and Transputer are trademarks of Inmos.

Clipper C100 is a trademark of Intergraph.

IBM, 360, 360/30, 360/40, 360/50, 360/65, 360/85, 360/91, 370, 370/135, 370/138, 370/145, 370/155, 370/158, 370/165, 370/168, 370-XA, ESA/370, System/360, System/370, 701, 704, 709, 801, 3033, 3080, 3080 series, 3080 VF, 3081, 3090, 3090/100, 3090/200, 3090/400, 3090/600, 3090/600S, 3090 VF, 3330, 3380, 3380D, 3380 Disk Model AK4, 3380J, 3390, 3880-23, 3990, 7030, 7090, 7094, IBM FORTRAN, ISAM, MVS, IBM PC, IBM PC-AT, PL.8, RT-PC, SAGE, Stretch, IBM SVS, Vector Facility, and VM are trademarks of International Business Machines Corporation.

FutureBus is a trademark of the Institute of Electrical and Electronic Engineers.

Lamborghini and Countach are trademarks of Nuova Automobili Ferrucio Lamborghini, SPA.

Lotus 1-2-3 is a trademark of Lotus Development Corporation.

MB8909 is a trademark of LSI Logic.

NuBus is a trademark of Massachusetts Institute of Technology.

Miata and Mazda are trademarks of Mazda.

MASM, Microsoft Macro Assembler, MS DOS, MS DOS 3.1, and OS/2 are trademarks of Microsoft Corporation.

MIPS, MIPS 120, MIPS/120A, M/500, M/1000, RC6230, RC6280, R2000, R2000A, R2010, R3000, and R3010 are trademarks of MIPS Computer Systems.

Delta Series 8608, System V/88 R32V1, VME bus, 6809, 68000, 68010, 68020, 68030, 68882, 88000, 88000 1.8.4m14, 88100, and 88200 are trademarks of Motorola Corporation.

Multiflow is a trademark of Multiflow Corporation.

National 32032 and 32x32 are trademarks of National Semiconductor Corporation.

Ncube is a trademark of Ncube Corporation.

SX/2, SX/3, and FORTRAN 77/SX V.040 are trademarks of NEC Information Systems.

NYU Ultracomputer is a trademark of New York University.

VAST-2 v.2.21 is a trademark of Pacific Sierra.

Wren IV, Imprimis, Sabre, Sabre 97209, and IPI-2 are trademarks of Seagate Corporation.

Sequent, Balance 800, Balance 21000, and Symmetry are trademarks of Sequent Computers.

Silicon Graphics 4D/60, 4D/240, and Silicon Graphics 4D Series are trademarks of Silicon Graphics.

Stellar GS 1000, Stardent-1500, and Ardent Titan-1 are trademarks of Stardent.

Sun 2, Sun 3, Sun 3/75, Sun 3/260, Sun 3/280, Sun 4, Sun 4/110, Sun 4/260, Sun 4/280, SunOS 4.0.3c, Sun 1.2 FORTRAN compiler, SPARC, and SPARCstation 1 are trademarks of Sun Microsystems.

Synapse N+1 is a trademark of Synapse.

Tandem and Cyclone are trademarks of Tandem Computers.

TI 8847 and TI ASC are trademarks of Texas Instruments Corporation.

Connection Machine and CM-2 are trademarks of Thinking Machines.

Burroughs 6500, B5000, B5500, D-machine, UNIVAC, UNIVAC I, UNIVAC 1103 are trademarks of UNISYS.

Spice and 4.2 BSD UNIX are trademarks of University of California, Berkeley.

Illiac, Illiac IV, and Cedar are trademarks of University of Illinois.

Ada is a trademark of the U.S. Government (Ada Joint Program Office).

Weitek 3364, Weitek 1167, WTL 3110, and WTL 3170 are trademarks of Weitek Computers.

Alto, Ethernet, PARC, Palo Alto Research Center, Smalltalk, and Xerox are trademarks of Xerox Corporation.

Z-80 is a trademark of Zilog.

Foreword

by C. Gordon Bell

I am delighted and honored to write the foreword for this landmark book.

The authors have gone beyond the contributions of Thomas to Calculus and Samuelson to Economics. They have provided the definitive text and reference for computer architecture *and design.* To advance computing, I urge publishers to withdraw the scores of books on this topic so a new breed of architect/engineer can quickly emerge. This book won't eliminate the complex and errorful microprocessors from semicomputer companies, but it will hasten the education of engineers who can design better ones.

The book presents the critical tools to analyze uniprocessor computers. It shows the practicing engineer how technology changes over time and offers the empirical constants one needs for design. It motivates the designer about function, which is a welcome departure from the usual exhaustive shopping list of mechanisms that a naive designer might attempt to include in a single design.

The authors establish a baseline for analysis and comparisons by using the most important machine in each class: mainframe (IBM 360), mini (DEC VAX), and micro/PC (Intel 80x86). With this foundation, they show the coming mainline of simpler pipelined and parallel processors. These new technologies are shown as variants of their pedagogically useful, but highly realizable, processor (DLX). The authors stress technology independence by measuring work done per clock (parallelism), and time to do work (efficiency and latency). These methods should also improve the quality of research on new architectures and parallelism.

Thus, the book is required *understanding* for anyone working with architecture or hardware, including architects, chip and computer system engineers, and compiler and operating system engineers. It is especially useful for software engineers writing programs for pipelined and vector computers. Managers and marketers will benefit by knowing the Fallacies and Pitfalls sections of the book. One can lay the demise of many a computer—and, occasionally, a company—on engineers who fail to understand the subtleties of computer design.

The first two chapters establish the essence of computer design through measurement and the understanding of price/performance. These concepts are applied to the instruction set architecture and how it is measured. They discuss the implementation of processors and include extensive discussions of techniques for designing pipelined and vector processors. Chapters are also devoted to memory hierarchy and the often-neglected input/output. The final chapter presents the opportunities and questions about machines and directions of the future. Now, we need their next book on how to build these machines.

The reason this book sets a standard above all others and is unlikely to be superseded in any foreseeable future is the understanding, experience, taste, and *uniqueness* of the authors. They have stimulated the major change in architecture by their work on RISC (Patterson coined the word). Their university research leading to product development at MIPS and Sun Microsystems established important architectures for the 1990s. Thus, they have done the analysis, evaluated the trade-offs, worked on the compilers and operating systems, and seen their machines achieve significance in use. Furthermore, as teachers, they have seen that the book is pedagogically sound (and have solicited opinions from others through the unprecedented Beta testing program). I know this will be the book of the decade in computer systems. Perhaps its greatest accomplishment would be to stimulate other great architects and designers of higher-level systems (databases, communications systems, languages and operating systems) to write similar books about their domains.

I've already enjoyed and learned from the book, and surely you will too.

—C. Gordon Bell

Contents

Foreword ix
by C. GORDON BELL

Preface xvii

Acknowledgements xxiii

1 **Fundamentals of Computer Design** 2

1.1	Introduction	3
1.2	Definitions of Performance	5
1.3	Quantitative Principles of Computer Design	8
1.4	The Job of a Computer Designer	13
1.5	Putting It All Together: The Concept of Memory Hierarchy	18
1.6	Fallacies and Pitfalls	21
1.7	Concluding Remarks	22
1.8	Historical Perspective and References	23
	Exercises	28

2 **Performance and Cost** 32

2.1	Introduction	33
2.2	Performance	35
2.3	Cost	53
2.4	Putting It All Together: Price/Performance of Three Machines	66
2.5	Fallacies and Pitfalls	70
2.6	Concluding Remarks	76
2.7	Historical Perspective and References	77
	Exercises	81

3 **Instruction Set Design: Alternatives and Principles** 88

3.1	Introduction	89
3.2	Classifying Instruction Set Architectures	90
3.3	Operand Storage in Memory: Classifying General-Purpose Register Machines	92
3.4	Memory Addressing	94
3.5	Operations in the Instruction Set	103
3.6	Type and Size of Operands	109
3.7	The Role of High-Level Languages and Compilers	111
3.8	Putting It All Together: How Programs Use Instruction Sets	122
3.9	Fallacies and Pitfalls	124
3.10	Concluding Remarks	126
3.11	Historical Perspective and References	127
	Exercises	132

4 **Instruction Set Examples and Measurements of Use** **138**

4.1 Instruction Set Measurements: What and Why 139
4.2 The VAX Architecture 142
4.3 The 360/370 Architecture 148
4.4 The 8086 Architecture 153
4.5 The DLX Architecture 160
4.6 Putting It All Together: Measurements of Instruction Set Usage 167
4.7 Fallacies and Pitfalls 183
4.8 Concluding Remarks 185
4.9 Historical Perspective and References 186
Exercises 191

5 **Basic Processor Implementation Techniques** **198**

5.1 Introduction 199
5.2 Processor Datapath 201
5.3 Basic Steps of Execution 202
5.4 Hardwired Control 204
5.5 Microprogrammed Control 208
5.6 Interrupts and Other Entanglements 214
5.7 Putting It All Together: Control for DLX 220
5.8 Fallacies and Pitfalls 238
5.9 Concluding Remarks 240
5.10 Historical Perspective and References 241
Exercises 244

6 **Pipelining** **250**

6.1 What Is Pipelining? 251
6.2 The Basic Pipeline for DLX 252
6.3 Making the Pipeline Work 255
6.4 The Major Hurdle of Pipelining—Pipeline Hazards 257
6.5 What Makes Pipelining Hard to Implement 278
6.6 Extending the DLX Pipeline to Handle Multicycle Operations 284
6.7 Advanced Pipelining—Dynamic Scheduling in Pipelines 290
6.8 Advanced Pipelining—Taking Advantage of More Instruction-Level Parallelism 314
6.9 Putting It All Together: A Pipelined VAX 328
6.10 Fallacies and Pitfalls 334
6.11 Concluding Remarks 337
6.12 Historical Perspective and References 338
Exercises 343

7 **Vector Processors** 350

7.1 Why Vector Machines? 351
7.2 Basic Vector Architecture 353
7.3 Two Real–World Issues: Vector Length and Stride 364
7.4 A Simple Model for Vector Performance 369
7.5 Compiler Technology for Vector Machines 371
7.6 Enhancing Vector Performance 377
7.7 Putting It All Together: Evaluating the Performance of Vector Processors 383
7.8 Fallacies and Pitfalls 390
7.9 Concluding Remarks 392
7.10 Historical Perspective and References 393
Exercises 397

8 **Memory-Hierarchy Design** 402

8.1 Introduction: Principle of Locality 403
8.2 General Principles of Memory Hierarchy 404
8.3 Caches 408
8.4 Main Memory 425
8.5 Virtual Memory 432
8.6 Protection and Examples of Virtual Memory 438
8.7 More Optimizations Based on Program Behavior 449
8.8 Advanced Topics—Improving Cache-Memory Performance 454
8.9 Putting It All Together: The VAX-11/780 Memory Hierarchy 475
8.10 Fallacies and Pitfalls 480
8.11 Concluding Remarks 484
8.12 Historical Perspective and References 485
Exercises 490

9 **Input/Output** 498

9.1 Introduction 499
9.2 Predicting System Performance 501
9.3 I/O Performance Measures 506
9.4 Types of I/O Devices 512
9.5 Buses—Connecting I/O Devices to CPU/Memory 528
9.6 Interfacing to the CPU 533
9.7 Interfacing to an Operating System 535
9.8 Designing an I/O System 539
9.9 Putting It All Together: The IBM 3990 Storage Subsystem 546
9.10 Fallacies and Pitfalls 554
9.11 Concluding Remarks 559
9.12 Historical Perspective and References 560
Exercises 563

10 **Future Directions** **570**

10.1 Introduction 571
10.2 Flynn Classification of Computers 572
10.3 SIMD Computers—Single Instruction Stream, Multiple Data Streams 572
10.4 MIMD Computers—Multiple Instruction Streams, Multiple Data Streams 574
10.5 The Roads to El Dorado 576
10.6 Special-Purpose Processors 580
10.7 Future Directions for Compilers 581
10.8 Putting It All Together: The Sequent Symmetry Multiprocessor 582
10.9 Fallacies and Pitfalls 585
10.10 Concluding Remarks—Evolution Versus Revolution in Computer Architecture 587
10.11 Historical Perspective and References 588
Exercises 592

Appendix A: Computer Arithmetic **A-1**
by DAVID GOLDBERG
Xerox Palo Alto Research Center

A.1 Introduction A-1
A.2 Basic Techniques of Integer Arithmetic A-2
A.3 Floating Point A-12
A.4 Floating-Point Addition A-16
A.5 Floating-Point Multiplication A-20
A.6 Division and Remainder A-23
A.7 Precisions and Exception Handling A-28
A.8 Speeding Up Integer Addition A-31
A.9 Speeding Up Integer Multiplication and Division A-39
A.10 Putting It All Together A-53
A.11 Fallacies and Pitfalls A-57
A.12 Historical Perspective and References A-58
Exercises A-63

Appendix B: Complete Instruction Set Tables **B-1**

B.1 VAX User Instruction Set B-2
B.2 System/360 Instruction Set B-6
B.3 8086 Instruction Set B-9

Appendix C: Detailed Instruction Set Measurements **C-1**

C.1 VAX Detailed Measurements C-2
C.2 360 Detailed Measurements C-3
C.3 Intel 8086 Detailed Measurements C-4
C.4 DLX Detailed Instruction Set Measurements C-5

Appendix D: Time Versus Frequency Measurements D-1

D.1 Time Distribution on the VAX-11/780 D-2
D.2 Time Distribution on the IBM 370/168 D-4
D.3 Time Distribution on an 8086 in an IBM PC D-6
D.4 Time Distribution on a DLX Relative D-8

Appendix E: Survey of RISC Architectures E-1

E.1 Introduction E-1
E.2 Addressing Modes and Instruction Formats E-2
E.3 Instructions: The DLX Subset E-4
E.4 Instructions: Common Extensions to DLX E-9
E.5 Instructions Unique to MIPS E-12
E.6 Instructions Unique to SPARC E-15
E.7 Instructions Unique to M88000 E-17
E.8 Instructions Unique to i860 E-19
E.9 Concluding Remarks E-23
E.10 References E-24

References R-1

Index I-1

Preface

I started in 1962 to write a single book with this sequence of chapters, but soon found that it was more important to treat the subjects in depth rather than to skim over them lightly. The resulting length has meant that each chapter by itself contains enough material for a one semester course, so it has become necessary to publish the series in separate volumes...

Donald Knuth, *The Art of Computer Programming,*
Preface to Volume 1 (of 7) (1968)

Why We Wrote This Book

Welcome to this book! We're glad to have the opportunity to communicate with you! There are so many exciting things happening in computer architecture, but we feel available materials just do not adequately make people aware of this. This is not a dreary science of paper machines that will never work. No! It's a discipline of keen intellectual interest, requiring balance of marketplace forces and cost/performance, leading to glorious failures and some notable successes. And it is hard to match the excitement of seeing thousands of people use the machine that you designed.

Our primary goal in writing this book is to help change the way people learn about computer architecture. We believe that the field has changed from one that can only be taught with definitions and historical information, to one that can be studied with real examples and real measurements. We envision this book as suitable for a course in computer architecture as well as a primer or reference for professional engineers and computer architects. This book embodies a new approach to demystifying computer architecture—it emphasizes a quantitative approach to cost/performance tradeoffs. This does not imply an overly formal approach, but simply one that is grounded in good engineering design. To accomplish this, we've included lots of data about real machines, so that a reader can understand design tradeoffs in a quantitative as well as qualitative fashion. A significant component of this approach can be found in the problem sets at the end of every chapter, as well as the software that accompanies the book. Such exercises have long formed the core of science and engineering education. With the emergence of a quantitative basis for teaching computer architecture, we feel the field has the potential to move toward the rigorous quantitative foundation of other disciplines.

Topic Selection and Organization

We have a conservative approach to topic selection, for there are many interesting ideas in the field. Rather than attempting a comprehensive survey of every architecture a reader might encounter today in practice or in the literature, we've chosen the core concepts of computer architecture that are likely to be included in any new machine. In making these decisions, a key criterion has been to emphasize ideas that have been sufficiently examined to be discussed in quantitative terms. For example, we concentrate on uniprocessors until the final chapter, where a bus-oriented, shared-memory multiprocessor is described. We believe this class of computer architecture will increase in popularity, but despite this perception it only met our criteria by a slim margin. Only recently has this class of architecture been examined in ways that allow us to discuss it quantitatively; a short time ago even this wouldn't have been included. Although large-scale parallel processors are of obvious importance to the future, it is our feeling that a firm basis in the principles of uniprocessor design is necessary before any practicing engineer tries to build a better computer of any organization; especially one incorporating multiple uniprocessors.

Readers familiar with our research might expect this book to be only about reduced instruction set computers (RISCs). This is a mistaken judgment about the content of this book. Our hope is that design principles and quantitative data in this book will restrict discussions of architecture styles to terms like "faster" or "cheaper," unlike previous debates.

The material we have selected has been stretched upon a consistent structure that is followed in every chapter. After explaining the ideas of a chapter, we include a "Putting It All Together" section that ties these ideas together by showing how they are used in a real machine. This is followed by a section, entitled "Fallacies and Pitfalls," that lets readers learn from the mistakes of others.We show examples of common misunderstandings and architectural traps that are difficult to avoid even when you know they are lying in wait for you. Each chapter ends with a "Concluding Remarks" section, followed by a "Historical Perspective and References" section that attempts to give proper credit for the ideas in the chapter and a sense of the history surrounding the inventions, presenting the human drama of computer design. It also supplies references that the student of architecture may want to pursue. If you have time, we recommend reading some of the classic papers in the field that are mentioned in these sections. It is both enjoyable and educational to hear the ideas from the mouths of the creators. Each chapter ends with Exercises, over 200 in total, which vary from one-minute reviews to term projects.

A glance at the Table of Contents shows that neither the amount nor the depth of the material is equal from chapter to chapter. In the early chapters, for example, we have more basic material to ensure a common terminology and background. In talking with our colleagues, we found widely varying opinions of the backgrounds readers have, the pace at which they can pick up new material, and even the order in which ideas should be introduced. Our assumption is that the reader is familiar with logic design, and has had some exposure to at least one instruction set and basic software concepts. The pace varies with the chapters, with the first half gentler than the last half. The organizational decisions were formed in response to reviewer advice. The final organization was selected to conveniently suit the majority of courses (beyond Berkeley and Stanford!) with only minor modifications. Depending on your goals, we see three paths through this material:

Introductory coverage:	Chapters 1, 2, 3, 4, 5, 6.1–6.5, 8.1–8.5, 9.1–9.5, 10, and A.1–A.3.
Intermediary coverage:	Chapters 1, 2, 3, 4, 5, 6.1–6.6, 6.9–6.12, 8.1–8.7, 8.9–8.12, 9, 10, A (except skip division in Section A.9), and E.
Advanced coverage:	Read everything, but Chapters 3 and 5 and Sections A.1–A.2 and 9.3–9.4 may be largely review, so read them quickly.

Alas, there is no single best order for the chapters. It would be nice to know about pipelining (Chapter 6) before discussing instruction sets (Chapters 3 and 4), for example, but it is difficult to understand pipelining without understanding the full set of instructions being pipelined. We ourselves have tried a few different orders in earlier versions of this material, and each has its strengths. Thus, the material was written so that it can be covered in several ways. The organization proved sufficiently flexible for a wide variety of chapter sequences in the Beta test program at 18 schools, where the book was used successfully. Some of these syllabi are reproduced in the accompanying Instructor's Manual. The only restriction is that some chapters should be read in sequence:

Chapters 1 and 2

Chapters 3 and 4

Chapters 5, 6, and 7

Chapters 8 and 9

Readers should start with Chapters 1 and 2 and end with Chapter 10, but the rest can be covered in any order. The only proviso is that if you read Chapters 5, 6, and 7 before Chapters 3 and 4, you should first skim Section 4.5, as the instruction set in this section, DLX, is used to illustrate the ideas found in those three chapters. A compact description of DLX and the hardware description notation we use can be found on the inside back cover. (We selected a modified version of C for our hardware description language because of its compactness, because of the number of people who know the language, and because there is no common description language used in books that could be considered prerequisites.)

We urge everyone to read Chapters 1 and 2. Chapter 1 is intentionally easy to follow so that it can be read quickly, even by a beginner. It gives a few important principles that act as themes guiding the tradeoffs in later chapters. While few would skip the performance section of Chapter 2, some might be tempted to skip the cost section to get to the "technical issues" in the later chapters. Please don't. Computer design is almost always balancing cost and performance, and few understand how price is related to cost, or how to lower cost and price by 10% in a way that minimizes performance loss. The foundations laid in the cost section of Chapter 2 allow cost/performance to be the basis of all tradeoffs in the last half of the book. On the other hand, some subjects are probably best left as reference material. If the book is part of a course, lectures can show how to use the data from these chapters in making decisions in computer design. Chapter 4 is probably the best example of this. Depending on your background, you already may be familiar with some of the material, but we try to include a few new twists for each subject. The section on microprogramming in Chapter 5 will be review for many, for example, but the description of the impact of interrupts on control is rarely found in other books.

We also invested special effort in making this book interesting to practicing engineers and advanced graduate students. Advanced topics sections are found in:

Chapter 6 on pipelining (Sections 6.7 and 6.8, which are about half the chapter)

Chapter 7 on vectors (the whole chapter)

Chapter 8 on memory-hierarchy design (Section 8.8, which is about a third of Chapter 8)

Chapter 10 on future directions (Section 10.7, about a quarter of that chapter)

Those under time pressure might want to skip some of these sections. To make skipping easier, the Putting It All Together sections of Chapters 6 and 8 are independent of the advanced topics.

You might have noticed that floating point is covered in Appendix A rather than in a chapter. Since it is largely independent of the other material, our solution was to include it as an appendix as our surveys indicated that a significant percentage of the readers would be exposed to floating point elsewhere.

The remaining appendices are included both for reference purposes for the computer professional and for the Exercises. Appendix B contains the instruction sets of three classic machines: the IBM 360, Intel 8086, and the DEC VAX. Appendices C and D give the mix of instructions in real programs for these machines plus DLX, either measured by instruction frequency or time frequency. Appendix E offers a more detailed comparative survey of several recent architectures.

Exercises, Projects, and Software

The optional nature of the material is also reflected in the Exercises. Brackets for each question (<chapter.section>) indicate the text sections of primary relevance to answering the question. We hope this helps readers to avoid exercises for which they haven't read the corresponding section, as well as providing the source for review. We have adopted Donald Knuth's technique of rating the Exercises. The ratings give an estimate of how much effort a problem might take:

[10]	1 minute (read and understand)
[20]	15–20 minutes for full answer
[25]	1 hour for full written answer
[30]	Short programming project: less than 1 full day of programming
[40]	Significant programming project: 2 weeks of elapsed time
[50]	Term project (2–4 weeks by two people)
[Discussion]	Topic for discussion with others interested in computer architecture

To facilitate the use of this book in the college curriculum, the book is also accompanied by an Instructor's Manual and software. The software is a UNIX tar tape that includes benchmarks, cache traces, cache and instruction set simulators, and a compiler. Readers interested in obtaining the software will find it available by anonymous FTP via Internet from max.stanford.edu. Copies may also be obtained by contacting Morgan Kaufmann at (415) 578-9911 (duplication and handling charges will apply on these orders).

Concluding Remarks

You might see a masculine adjective or pronoun in a paragraph. Since English does not have gender-neutral pronouns or adjectives, we found ourselves in the unfortunate position of choosing among the standard, consistent use of the masculine, alternating between feminine and masculine, and the grammatically unworkable third person plural. We tried to reduce the occurrence of this problem, but when a pronoun is unavoidable we alternate gender chapter by chapter. Our experience is this practice hurts no one, unlike the standard solution.

If you read the following acknowledgement section you will see that we went to great lengths to correct mistakes. Since a book goes through many printings, we have the opportunity to make even more corrections. If you uncover any remaining resilient bugs, please contact the publisher by electronic mail (bugs2@vsop.stanford.edu) or low-tech mail using the address found on the copyright page. The first reader to report an error that is incorporated in a future printing will be rewarded with a $1.00 bounty.

Finally, this book is unusual in that there is no strict ordering of the authors' names. About half the time you will see Hennessy and Patterson, both in this book and in advertisements, and half the time you will see Patterson and Hennessy. You'll even find it listed both ways in bibliographic publications such as *Books in Print.* (When we reference the book, we will alternate author order.) This reflects the true collaborative nature of this book: Together, we brainstormed about the ideas and method of presentation, then individually wrote one-half of the chapters and acted as reviewer for every draft of the other. (In fact, the final page count suggests each of us wrote exactly the same number of pages!) We could think of no fair way to reflect this genuine cooperation other than to hide in ambiguity—a practice that may help some authors but confuses librarians. Thus, we equally share the blame for what you are about to read.

John Hennessy David Patterson

January 1990

Acknowledgements

This book was written with the help of a great many people—so many, in fact, that some authors would stop here, claiming there are too many to name. We decline to use that excuse, however, because to do so would hide the magnitude of help we needed. Therefore, we name the 137 people and five institutions to whom our thanks go.

When we decided to add a floating-point appendix that featured the IEEE standard, we asked many colleagues to recommend a person who understood that standard and who could write well and explain complex ideas simply. **David Goldberg**, of Xerox Palo Alto Research Center, fulfilled all those tasks admirably, setting a standard to which we hope the rest of the book measures up.

Margo Seltzer of U.C. Berkeley deserves special credit. Not only was she the first teaching assistant of the course at Berkeley using the material, she brought together all the software, benchmarks, and traces that we are distributing with this book. She also ran the cache simulations and instruction set simulations that appear in Chapter 8. We thank her for her promptness and reliability in taking odds and ends of software and putting them together into a coherent package.

Bryan Martin and **Truman Joe** of Stanford also deserve our special thanks for rapidly reading the Exercises for early chapters near the deadline for the fall release. Without their dedication, the Exercises would have been considerably less polished.

Our plan to develop this material was to first try the ideas in the fall of 1988 in courses taught by us at Berkeley and Stanford. We created lecture notes, first trying them on the students at Berkeley (because the Berkeley academic year starts before Stanford), fixing some of the errors, and then exposing Stanford students to these ideas. This may not have been the best experience of their academic lives, so we wish to thank those who "volunteered" to be guinea pigs, as well as the teaching assistants **Todd Narter**, **Margo Seltzer** and **Eric Williams,** who suffered the consequences of this growth experience.

The next step of the plan was to write a draft of the book in the winter of 1989. We expected this to be turning the lecture notes into English, but our feedback from the students and the reevaluation that is part of any writing turned this into a much larger task than we expected. This "Alpha" version was sent out for reviews in the spring of 1989. Special thanks go to **Anoop Gupta** of Stanford University and **Forest Baskett** of Silicon Graphics who used the Alpha version to teach a class at Stanford in the spring of 1989.

Computer architecture is a field that has both an academic side and an industrial side. We relied on both kinds of expertise to review the material in this book. The academic reviewers of the Alpha version include **Thomas Casavant** of Purdue University, **Jim Goodman** of the University of Wisconsin at Madison, **Roger Kieckhafer** of the University of Nebraska, **Hank Levy** of the University of Washington, **Norman Matloff** of the University of California at Davis, **David Meyer** of Purdue University, **Trevor Mudge** of the University of Michigan, **Victor Nelson** of Auburn University, **Richard Reid** of Michigan State University, and **Mark Smotherman** of Clemson University. We also wish to acknowledge those who gave feedback on our outline in the fall of 1989: **Bill Dally** of MIT, and **Jim Goodman**, **Hank Levy**, **David Meyer**, and **Joseph Pfeiffer** of New Mexico State. In April of 1989, a variety of our plans were tested in a discussion group that included **Paul Barr** of Northeastern University, **Susan Eggers** of the University of Washington, **Jim Goodman** and **Mark Hill** of the University of Wisconsin, **James Mooney** of the University of West Virginia, and **Larry Wittie** of SUNY Stony Brook. We appreciate their helpful advice.

Before listing the industrial reviewers, special thanks go to **Douglas Clark** of DEC, who gave us more input on the Alpha version than all other reviewers combined, and whose remarks were always carefully written with an eye toward our sensitive natures. Other people who reviewed several chapters of the Alpha version were **David Douglas** and **David Wells** of Thinking Machines, **Joel Emer** of DEC, **Earl Killian** of MIPS Computer Systems Inc., and **Jim Smith** of Cray Research. **Earl Killian** also explained the mysteries of the Pixie instruction set analyzer and provided an unreleased version for us to collect branch statistics.

Thanks also to **Maurice Wilkes** of Olivetti Research and **C. Gordon Bell** of Stardent for helping us improve our versions of computer history at the end of each chapter.

In addition to those who volunteered to read many chapters, we also wish to thank those who made suggestions of material to include or reviewed the Alpha version of the chapters:

Chapter 1: **Danny Hillis** of Thinking Machines for his suggestion on assigning resources according to their contribution to performance.

Chapter 2: **Andy Bechtolsheim** of Sun Microsystems for advice on price versus cost and workstation cost estimates; **David Hodges** of the University of California at Berkeley, **Ed Hudson** and **Mark Johnson** of MIPS, **Al Marston** and **Jim Slager** of Sun, **Charles Stapper** of IBM, and **David Wells** of Thinking Machines for explaining chip manufacturing and yield; **Ken Lutz** of U.C. Berkeley and the **FAST chip service** of USC/ISI for the price quotes on chips; **Andy Bechtolsheim** and **Nhan Chu** of Sun Microsystems, **Don Lewine** of Data General, and **John Mashey** and **Chris Rowen** of MIPS who also reviewed this chapter.

Chapter 4: **Tom Adams** of Apple and **Richard Zimmermann** of San Francisco State University for their Intel 8086 statistics; **John Crawford** of Intel for reviewing the 80x86 and other material; **Lloyd Dickman** for reviewing IBM 360 material.

Chapter 5: **Paul Carrick** and **Peter Stoll** of Intel for reviews.

Chapter 7: **David Bailey** of NASA Ames and **Norm Jouppi** of DEC for reviews.
Chapter 8: **Ed Kelly** of Sun for help on the explanation of DRAM alternatives and **Bob Cmelik** of Sun for the SPIX statistics; **Anant Agarwal** of MIT, **Susan Eggers** of the University of Washington, **Mark Hill** of the University of Wisconsin at Madison, and **Steven Przybylski** of MIPS for the material from their dissertations; and **Susan Eggers** and **Mark Hill** for reviews.

Chapter 9: **Jim Brady** of IBM for providing references for quantitative data on response time and IBM computers and reviewing the chapter; **Garth Gibson** of the University of California at Berkeley for help with bus references and for reviewing the chapter; **Fred Berkowitz** of Omni Solutions, **David Boggs** of DEC, **Pete Chen** and **Randy Katz** of the University of California at Berkeley, **Mark Hill** of the University of Wisconsin, **Robert Shomler** of IBM, and **Paul Taysom** of AT&T Bell Laboratories for reviews.

Chapter 10: **C. Gordon Bell** of Stardent for his suggestion on including a multiprocessor in the Putting It All Together section; **Susan Eggers, Danny Hillis** of Thinking Machines, and **Shreekant Thakkar** of Sequent Computer for reviews.

Appendix A: The facts about IEEE REM and argument reduction in Section A.6, as well as the $p \leq (q-1)/2$ theorem (page A-29) are taken from unpublished lecture notes of **William Kahan** of U.C. Berkeley (and we don't know of any published sources containing a discussion of these facts). The SDRWAVE data is from **David Hough** of Sun Microsystems. **Mark Birman** of Weitek Corporation, **Merrick Darley** of Texas Instruments, and **Mark Johnson** of MIPS provided information about the 3364, 8847, and R3010, respectively. **William Kahan** also read a draft of this chapter and made many insightful comments. We also thank **Forrest Brewer** of the University of California at Santa Barbara, **Milos Ercegovac** of the University of California at Los Angeles, **Bill Shannon** of Sun Microsystems, and **Behrooz Shirazi** of Southern Methodist University for reviews.

The software that goes with this book was collected and examined by **Margo Seltzer** of the University of California at Berkeley. The following individuals volunteered their software for our distribution:

C compiler for DLX: **Yong-dong Wang** of U.C. Berkeley and the **Free Software Foundation**

Assembler for DLX: **Jeff Sedayo** of U.C. Berkeley

Cache Simulator (Dinero III): **Mark Hill** of the University of Wisconsin

ATUM traces: **Digital Equipment Corporation**, **Anant Agarwal,** and **Richard Sites**

The initial version of the simulator for DLX was developed by **Wie Hong** and **Chu-Tsai Sun** of U.C. Berkeley.

While many advised us to save ourselves some effort and publish the book sooner, we pursued the goal of publishing the cleanest book possible with the help of an additional group of people involved in the final round of review. This book would not be as useful without the help of adventurous instructors, teaching assistants, and willing students, who accepted the role of Beta test sites in the class-testing program; we made hundreds of changes as a result of the Beta testing. (In fact we are so happy with the feedback that we are continuing the error reporting and reward system; see the copyright page.) The Beta test site institutions and instructors were:

Carnegie-Mellon University	**Daniel Siewiorek**
Clemson University	**Mark Smotherman**
Cornell University	**Keshav Pingali**
Pennsylvania State University	**Mary Jane Irwin/Bob Owens**
San Francisco State University	**Vojin Oklobdzija**
Southeast Missouri State University	**Anthony Duben**
Southern Methodist University	**Behrooz Shirazi**
Stanford University	**John Hennessy**
State University of New York at Stony Brook	**Larry Wittie**
University of California at Berkeley	**Vojin Oklobdzija**
University of California at Los Angeles	**David Rennels**
University of California at Santa Cruz	**Daniel Helman**
University of Nebraska	**Roger Kieckhafer**
University of North Carolina at Chapel Hill	**Akhilesh Tyagi**
University of Texas at Austin	**Joseph Rameh**
University of Waterloo	**Bruno Preiss**
University of Wisconsin at Madison	**Mark Hill**
Washington University (St. Louis)	**Mark Franklin**

Special mention should be given to **Daniel Helman**, **Mark Hill**, **Mark Smotherman**, and **Larry Wittie** who were especially generous with their advice. The compilation of exercise solutions for course instructors was aided by contributions from **Evan Tick** of the University of Oregon, **Susan Eggers** of the University of Washington, and **Anoop Gupta** of Stanford University.

The classes at SUNY Stony Brook, Carnegie-Mellon, Stanford, Clemson, and Wisconsin supplied us with the greatest number of bug discoveries in the Beta version. To all of those who qualified for the $1.00 reward program by submitting the first notice of a bug: Your checks are in the mail. We'd also like to note that numerous bugs were hunted and killed by the following people: **Michael Butler**, **Rohit Chandra**, **David Cummings**, **David Filo**, **Carl Feynman**, **John Heinlein**, **Jim Quinlan**, **Andras Radics**, **Peter Schnorf**, and **Malcolm Wing**.

In addition to the class testing, we also asked our friends in industry for help once again. Special thanks go to **Jim Smith** of Cray Research for a thorough review and thoughtful suggestions of the full Beta text. The following individuals also helped us improve the Beta release, and our thanks go to them:

Ben Hao of Sun Microsystems for reviewing the full Beta Release.

Ruby Lee of Hewlett-Packard and **Bob Supnik** of DEC for reviewing several chapters.

Chapter 2: **Steve Goldstein** of Ross Semiconductor and **Sue Stone** of Cypress Semiconductor for photographs and the wafer of the CY7C601 (pages 57–58); **John Crawford** and **Jacque Jarve** of Intel for the photographs and wafer of the Intel 80486 (pages 56 and 58); and **Dileep Bhandarkar** of DEC for help with the VMS version of Spice and TeX used in Chapters 2–4.
Chapter 6: **John DeRosa** of DEC for help with the 8600 pipeline.

Chapter 7: **Corinna Lee** of University of California at Berkeley for measurements of the Cray X-MP and Y-MP and for reviews.

Chapter 8: **Steven Przybylski** of MIPS for reviews.

Chapter 9: **Dave Anderson** of Imprimis for reviews and supplying material on disk access time; **Jim Brady** and **Robert Shomler** of IBM for reviews, and **Pete Chen** of Berkeley for suggestions on the system performance formulas.

Chapter 10: **C. Gordon Bell** for reviews, including several suggestions on classifications of MIMD machines and **David Douglas** and **Danny Hillis** of Thinking Machines for discussions on parallel processors of the future.

Appendix A: **Mark Birman** of Weitek Corporation, **Merrick Darley** of Texas Instruments, and **Mark Johnson** of MIPS for the photographs and floor plans of the chips (pages A-54–A-55); and **David Chenevert** of Sun Microsystems for reviews.

Appendix E: This was added after the Beta version, and we thank the following people for reviews: **Mitch Alsup** of Motorola, **Robert Garner** and **David Weaver** of Sun Microsystems, **Earl Killian** of MIPS Computer Systems, and **Les Kohn** of Intel.

While we have done our best to eliminate errors and to repair those pointed out by the reviewers, we alone are responsible for those that remain!

We also want to thank the **Defense Advanced Research Projects Agency** for supporting our research for many years. That research was the basis of many ideas that came to fruition in this book. In particular, we want to thank these current and former program managers: **Duane Adams**, **Paul Losleben**, **Mark Pullen**, **Steve Squires**, **Bill Bandy**, and **John Toole**.

Thanks go to **Richard Swan** and his colleagues at DEC Western Research Laboratory for providing us a hideout for writing the Alpha and Beta versions, and to **John Ousterhout** of U.C. Berkeley, who was always ready (and even a little too eager) to act as devil's advocate for the merits of ideas in this book during this trek from Berkeley to Palo Alto. Thanks also to **Thinking Machines Corporation** for providing a refuge during the final revision.

This book could not have been published without a publisher. **John Wakerley** gave us valuable advice on how to pick a publisher. We selected Morgan Kaufmann Publishers, Inc., and we have not regretted that decision. (Not all authors feel this way about their publisher!) Starting with lecture notes just after New Year's Day 1989, we completed the Alpha version in four months. In the next three months we received reviews from 55 people and

finished the Beta version. After class testing with 750 students in the fall of 1989 and more reviews from industry, we submitted the final version just before Christmas 1989. Yet the book was available by March, 1990. We are not aware of another publisher who could have kept pace with such a rigorous schedule. We wish to thank **Shirley Jowell** for learning about pipelining and pipeline hazards and seeing how to apply them to publishing. Our warmest thanks to our editor **Bruce Spatz** for his guidance and his humor in our writing adventure. We also want to thank members of the extended Morgan Kaufmann family: **Walker Cunningham** for technical editing, **David Lance Goines** for the cover design, **Gary Head** for the book design, **Linda Medoff** for copy and production editing, **Fifth Street Computer Services** for computer typesetting, and **Paul Medoff** for proofreading and production assistance.

We must also thank our university staff, **Darlene Hadding, Bob Miller, Margaret Rowland**, and **Terry Lessard-Smith**, for countless faxes and express mailings as well as holding down the fort at Stanford and Berkeley while we worked on the book. **Linda Simko** and **Kim Barger** of Thinking Machines also provided numerous express mailings during the fall.

Our final thanks go to our families for their suffering through long nights and early mornings of reading, typing, and neglect.

And now for something completely different.

Monty Python's Flying Circus

1.1 Introduction 3

1.2 Definitions of Performance 5

1.3 Quantitative Principles of Computer Design 8

1.4 The Job of a Computer Designer 13

1.5 Putting It All Together: The Concept of Memory Hierarchy 18

1.6 Fallacies and Pitfalls 21

1.7 Concluding Remarks 22

1.8 Historical Perspective and References 23

Exercises 28

1 Fundamentals of Computer Design

1.1 Introduction

Computer technology has made incredible progress in the past half century. In 1945, there were no stored-program computers. Today, a few thousand dollars will purchase a personal computer that has more performance, more main memory, and more disk storage than a computer bought in 1965 for a million dollars. This rapid rate of improvement has come both from advances in the technology used to build computers and from innovation in computer designs. The increase in performance of machines is plotted in Figure 1.1. While technological improvements have been fairly steady, progress arising from better computer architectures has been much less consistent. During the first 25 years of electronic computers, both forces made a major contribution; but for the last 20 years, computer designers have been largely dependent upon integrated circuit technology. Growth of performance during this period ranges from 18% to 35% per year, depending on the computer class.

More than any other line of computers, mainframes indicate a growth rate due chiefly to technology—most of the organizational and architectural innovations were introduced into these machines many years ago. Supercomputers have grown both via technological enhancements and via architectural enhancements (see Chapter 7). Minicomputer advances have included innovative ways to implement architectures, as well as the adoption of many of the mainframe's techniques. Performance growth of microcomputers has been the fastest, partly because these machines take the most direct advantage of improvements in integrated circuit technology. Also, since 1980, microprocessor technology has been the technology of choice for both new architectures and new implementations of older architectures.

Two significant changes in the computer marketplace have made it easier than ever before to be commercially successful with a new architecture. First, the virtual elimination of assembly language programming has dramatically reduced the need for object-code compatibility. Second, the creation of standardized, vendor-independent operating systems, such as UNIX, has lowered the cost and risk of bringing out a new architecture. Hence, there has been a renaissance in computer design: There are many new companies pursuing new architectural directions, with new computer families emerging—mini-supercomputers, high-performance microprocessors, graphics supercomputers, and a wide range of multiprocessors—at a higher rate than ever before.

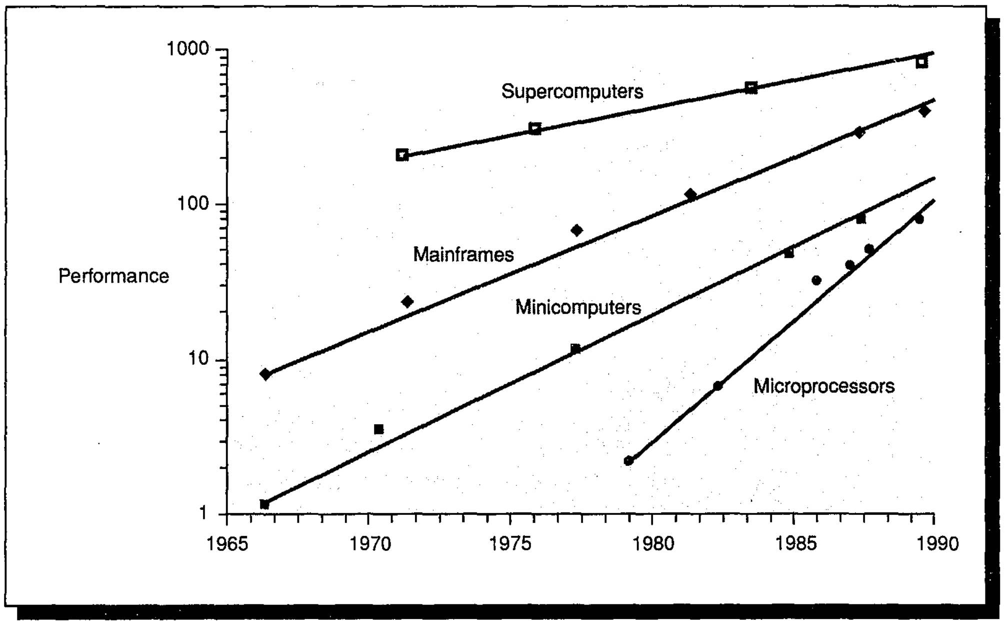


FIGURE 1.1 Different computer classes and their performance growth shown over the past ten or more years. The vertical axis shows relative performance and the horizontal axis is year of introduction. Classes of computers are loosely defined, primarily by their cost. *Supercomputers* are the most expensive—from over one million to tens of millions of dollars. Designed mostly for scientific applications, they are also the highest performance machines. *Mainframes* are high-end, general-purpose machines, typically costing more than one-half million dollars and as much as a few million dollars. *Minicomputers* are midsized machines costing from about 50 thousand dollars up to ten times that much. Finally, *microcomputers* range from small personal computers costing a few thousand dollars to large powerful workstations costing 50 thousand or more. The performance growth rates for supercomputers, minicomputers, and mainframes have been just under 20% per year, while the performance growth rate for microprocessors has been about 35% per year.

Starting in 1985, the computer industry saw a new style of architectures taking advantage of this opportunity and initiating a period in which performance has increased at a much more rapid rate. By bringing together advances in integrated circuit technology, improvements in compiler technology, and new architectural ideas, designers were able to create a series of machines that improved in performance by a factor of almost 2 every year. These ideas are now providing one of the most significant sustained performance improvements in over 20 years. This improvement was only possible because a number of important technological advances were brought together with a much better empirical understanding of how computers were used. From this fusion has emerged a style of computer design based on empirical data, experimentation, and simulation. It is this style and approach to computer design that are reflected in this text.

Sustaining the improvements in cost and performance of the last 25 to 50 years will require continuing innovations in computer design, and the authors believe such innovations will be founded on this quantitative approach to computer architecture. Hence, this book has been written not only to document this design style, but also to stimulate the reader to contribute to this field.

1.2 Definitions of Performance

To familiarize the reader with the terminology and concepts of this book, this chapter introduces some key terms and ideas. Examples of the ideas mentioned here appear throughout the book, and several of them—pipelining, memory hierarchies, CPU performance, and cost measurement—are the focus of entire chapters. Let's begin with definitions of relative performance.

When we say one computer is faster than another, what do we mean? The computer user may say a computer is faster when a program runs in less time, while the computer center manager may say a computer is faster when it completes more jobs in an hour. The computer user is interested in reducing *response time*—the time between the start and the completion of an event—also referred to as *execution time* or *latency*. The computer center manager is interested in increasing *throughput*—the total amount of work done in a given time—sometimes called *bandwidth*. Typically, the terms "response time," "execution time," and "throughput" are used when an entire computing task is discussed. The terms "latency" and "bandwidth" are almost always the terms of choice when discussing a memory system. All of these terms will appear throughout the text.

Example

Do the following system performance enhancements increase throughput, decrease response time, or both?

1. Faster clock cycle time

2. Multiple processors for separate tasks (handling the airlines reservations system for the country, for example)
3. Parallel processing of scientific problems

Answer Decreasing response time usually improves throughput. Hence, both 1 and 3 improve response time and throughput. In 2, no one task gets work done faster, so only throughput increases.

Sometimes these measures are best described with probability distributions rather than constant values. For example, consider the response time to complete an I/O operation to disk. The response time depends on a number of nondeterministic factors, such as what the disk is doing at the time of the I/O request and how many other tasks are waiting to access the disk. Because these values are not fixed, it makes more sense to talk about the average response time of a disk access. Likewise, the effective disk throughput—how much data actually goes to or from the disk per unit time—is not a constant value. For most of this text, we will treat response time and throughput as deterministic values, though this will change in Chapter 9 when we discuss I/O.

In comparing design alternatives, we often want to relate the performance of two different machines, say X and Y. The phrase "X is faster than Y" is used here to mean that the response time or execution time is lower on X than on Y for the given task. In particular, "X is *n*% faster than Y" will mean

$$\frac{\text{Execution time}_{\text{Y}}}{\text{Execution time}_{\text{X}}} = 1 + \frac{n}{100}$$

Since execution time is the reciprocal of performance, the following relationship holds:

$$1 + \frac{n}{100} = \frac{\text{Execution time}_{\text{Y}}}{\text{Execution time}_{\text{X}}} = \frac{\dfrac{1}{\text{Performance}_{\text{Y}}}}{\dfrac{1}{\text{Performance}_{\text{X}}}} = \frac{\text{Performance}_{\text{X}}}{\text{Performance}_{\text{Y}}}$$

Some people think of a performance increase, *n*, as the difference between the performance of the faster and slower machine, divided by the performance of the slower machine. This definition of *n* is exactly equivalent to our first definition, as we can see:

$$n = 100\left(\frac{\text{Performance}_{\text{X}} - \text{Performance}_{\text{Y}}}{\text{Performance}_{\text{Y}}}\right)$$

$$\frac{n}{100} = \frac{\text{Performance}_X}{\text{Performance}_Y} - 1$$

$$1 + \frac{n}{100} = \frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X}$$

The phrase "the throughput of X is 30% higher than Y" signifies here that the number of tasks completed per unit time on machine X is 1.3 times the number completed on Y.

Example If machine A runs a program in 10 seconds and machine B runs the same program in 15 seconds, which of the following statements is true?

- A is 50% faster than B.
- A is 33% faster than B.

Answer Machine A is *n*% faster than machine B can be expressed as

$$\frac{\text{Execution time}_B}{\text{Execution time}_A} = 1 + \frac{n}{100}$$

or

$$n = \frac{\text{Execution time}_B - \text{Execution time}_A}{\text{Execution time}_A} * 100$$

Thus,

$$\frac{15 - 10}{10} * 100 = 50$$

A is therefore 50% faster than B.

To help prevent misunderstandings—and because of the lack of consistent definitions for "faster than" and "slower than"—we will never use the phrase "slower than" in a quantitative comparison of performance.

Because performance and execution time are reciprocals, increasing performance decreases execution time. To help avoid confusion between the terms "increasing" and "decreasing," we usually say "improve performance" or "improve execution time" when we mean *increase* performance and *decrease* execution time.

Throughput and latency interact in a variety of ways in computer designs. One of the most important interactions occurs in pipelining. *Pipelining* is an implementation technique that improves throughput by overlapping the execution of multiple instructions; pipelining is discussed in detail in Chapter 6. Pipelining of instructions is analogous to using an assembly line to manufacture cars. In an assembly line it may take eight hours to build an entire car, but if there are eight steps in the assembly line, a new car is finished every hour. In the assembly line, the latency to build one car is not affected, but the throughput increases proportionally to the number of stages in the line if all the stages are of the same length. The fact that pipelines in computers have some overhead per stage increases the latency by some amount for each stage of the pipeline.

1.3 Quantitative Principles of Computer Design

This section introduces some important rules and observations that arise time and again in designing computers.

Make the Common Case Fast

Perhaps the most important and pervasive principle of computer design is to make the common case fast: In making a design tradeoff, favor the frequent case over the infrequent case. This principle also applies when determining how to spend resources since the impact on making some occurrence faster is higher if the occurrence is frequent. Improving the frequent event, rather than the rare event, will obviously help performance, too. In addition, the frequent case is often simpler and can be done faster than the infrequent case. For example, when adding two numbers in the *central processing unit* (CPU), we can expect overflow to be a rare circumstance and can therefore improve performance by optimizing the more common case of no overflow. This may slow down the case when overflow occurs, but if that is rare, then overall performance will be improved by optimizing for the normal case.

We will see many cases of this principle throughout this text. In applying this simple principle, we have to decide what the frequent case is and how much performance can be improved by making that case faster. A fundamental law, called *Amdahl's Law*, can be used to quantify this principle.

Amdahl's Law

The performance gain that can be obtained by improving some portion of a computer can be calculated using Amdahl's Law. *Amdahl's Law* states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.

Amdahl's Law defines the speedup that can be gained by using a particular feature. What is speedup? Suppose that we can make an enhancement to a machine that will improve performance when it is used. *Speedup* is the ratio

$$\text{Speedup} = \frac{\text{Performance for entire task using the enhancement when possible}}{\text{Performance for entire task without using the enhancement}}$$

Alternatively:

$$\text{Speedup} = \frac{\text{Execution time for entire task without using the enhancement}}{\text{Execution time for entire task using the enhancement when possible}}$$

Speedup tells us how much faster a task will run using the machine with the enhancement as opposed to the original machine.

Example

Consider the problem of going from Nevada to California over the Sierra Nevada mountains and through the desert to Los Angeles. You have several types of vehicles available, but unfortunately your route goes through ecologically sensitive areas in the mountains where you must walk. Your walk over the mountains will take 20 hours. The last 200 miles, however, can be done by high-speed vehicle. There are five ways to complete the second portion of your journey:

1. Walk at an average rate of 4 miles per hour.
2. Ride a bike at an average rate of 10 miles per hour.
3. Drive a Hyundai Excel in which you average 50 miles per hour.
4. Drive a Ferrari Testarossa in which you average 120 miles per hour.
5. Drive a rocket car in which you average 600 miles per hour.

How long will it take for the entire trip using these vehicles, and what is the speedup versus walking the entire distance?

Vehicle for second portion of trip	Hours for second portion of trip	Speedup in the desert	Hours for entire trip	Speedup for entire trip
Feet	50.00	1.0	70.00	1.0
Bike	20.00	2.5	40.00	1.8
Excel	4.00	12.5	24.00	2.9
Testarossa	1.67	30.0	21.67	3.2
Rocket car	0.33	150.0	20.33	3.4

FIGURE 1.2 The speedup ratios obtained for different means of transport depend heavily on the fact that we have to walk across the mountains. The speedup in the desert—once we have crossed the mountains—is equal to the rate using the designated vehicle divided by the walking rate; the final column shows how much faster our entire trip is compared to walking.

Answer

We can find the answer by determining how long the second part of the trip will take and adding that time to the 20 hours needed to cross the mountains. Figure 1.2 shows the effectiveness of using the enhanced mode of transportation.

Amdahl's Law gives us a quick way to find speedup, which depends on two factors:

1. The fraction of the computation time in the original machine that can be converted to take advantage of the enhancement. In the example above, the fraction is $\frac{50}{70}$. This value, which we will call $\text{Fraction}_{\text{enhanced}}$, is always less than or equal to 1.
2. The improvement gained by the enhanced execution mode; that is, how much faster the task would run if *only* the enhanced mode were used. In the above example this value is given in the column labeled "speedup in the desert." This value is the time of the original mode over the time of the enhanced mode and is always greater than 1. We call this value $\text{Speedup}_{\text{enhanced}}$.

The execution time using the original machine with the enhanced mode will be the time spent using the unenhanced portion of the machine plus the time spent using the enhancement:

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} * \left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}\right)$$

The overall speedup is the ratio of the execution times:

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

Example

Suppose that we are considering an enhancement that runs 10 times faster than the original machine but is only usable 40% of the time. What is the overall speedup gained by incorporating the enhancement?

Answer

$$\text{Fraction}_{\text{enhanced}} = 0.4$$

$$\text{Speedup}_{\text{enhanced}} = 10$$

$$\text{Speedup}_{\text{overall}} = \frac{1}{0.6 + \frac{0.4}{10}} = \frac{1}{0.64} \approx 1.56$$

Amdahl's Law expresses the law of diminishing returns: The incremental improvement in speedup gained by an additional improvement in the performance of just a portion of the computation diminishes as improvements are added. An important corollary of Amdahl's Law is that if an enhancement is only usable for a fraction of a task, we can't speed up the task by more than the reciprocal of 1 minus that fraction.

A common mistake in applying Amdahl's Law is to confuse "fraction of time converted to use an enhancement" and "fraction of time after enhancement is in use." If, instead of measuring the time that **could use** the enhancement in a computation, we measure the time **after** the enhancement is in use, the results will be incorrect! (Try Exercise 1.8 to see how wrong.)

Amdahl's Law can serve as a guide to how much an enhancement will improve performance and how to distribute resources to improve cost/performance. The goal, clearly, is to spend resources proportional to where time is spent.

Example

Suppose we could improve the speed of the CPU in our machine by a factor of five (without affecting I/O performance) for five times the cost. Also assume that the CPU is used 50% of the time, and the rest of the time the CPU is waiting for I/O. If the CPU is one-third of the total cost of the computer, is increasing the CPU speed by a factor of five a good investment from a cost/performance viewpoint?

Answer

The speedup obtained is

$$\text{Speedup} = \frac{1}{0.5 + \frac{0.5}{5}} = \frac{1}{0.6} = 1.67$$

The new machine will cost

$$\frac{2}{3} * 1 + \frac{1}{3} * 5 = 2.33 \text{ times the original machine}$$

Since the cost increase is larger than the performance improvement, this change does not improve cost/performance.

Locality of Reference

While Amdahl's Law is a theorem that applies to any system, other important fundamental observations come from properties of programs. The most important program property that we regularly exploit is *locality of reference*: Programs tend to reuse data and instructions they have used recently. A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code. An implication of locality is that based on the program's recent past, one can predict with reasonable accuracy what instructions and data a program will use in the near future.

To examine locality, several programs were measured to determine what percentage of the instructions were responsible for 80% and for 90% of the instructions executed. The data are shown in Figure 1.3, and the programs are described in detail in the next chapter.

Locality of reference also applies to data accesses, though not as strongly as to code accesses. There are two different types of locality that have been observed. *Temporal locality* states that recently accessed items are likely to be accessed in the near future. Figure 1.3 shows one effect of temporal locality. *Spatial locality* says that items whose addresses are near one another tend to be referenced close together in time. We will see these principles applied later in this chapter, and extensively in Chapter 8.

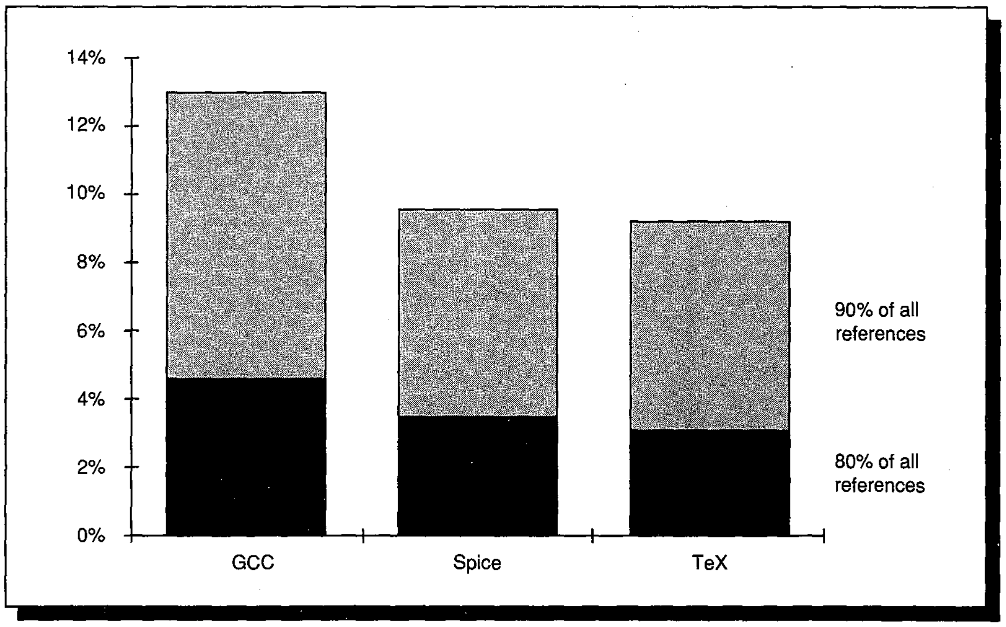


FIGURE 1.3 This plot shows what percentage of the instructions are responsible for 80% and for 90% of the instruction executions. For example, just under 4% of Spice's program instructions (also called the *static* instructions) represent 80% of the dynamically executed instructions, while just under 10% of the static instructions account for 90% of the executed instructions. Less than half the static instructions are executed even once in any one run—in Spice only 30% of the instructions are executed one or more times. Detailed descriptions of the programs and their inputs appear in Figure 2.17 (page 67).

1.4 The Job of a Computer Designer

A computer architect designs machines to run programs. If you were going to design a computer, your task would have many aspects, including instruction set design, functional organization, logic design, and implementation. The implementation may encompass integrated circuit (IC) design, packaging, power, and cooling. You would have to optimize a machine design across these levels. This optimization requires familiarity with a very wide range of technologies, from compilers and operating systems to logic design and packaging.

Some people have used the term *computer architecture* to refer only to instruction set design. They refer to the other aspects of computer design as "implementation," often insinuating that implementation is uninteresting or less challenging. The authors believe this view is not only incorrect, but is even responsible for mistakes in the design of new instruction sets. The architect's or designer's job is much more than instruction set design, and the technical hurdles in the other aspects of the project are certainly as challenging as those encountered in doing instruction set design.

In this book the term *instruction set architecture* refers to the actual programmer-visible instruction set. The instruction set architecture serves as the boundary between the software and hardware, and that topic is the focus of Chapters 3 and 4. The implementation of a machine has two components: organization and hardware. The term *organization* includes the high-level aspects of a computer's design, such as the memory system, the bus structure, and the internal CPU design. For example, two machines with the same instruction set architecture but different organizations are the VAX-11/780 and the VAX 8600. *Hardware* is used to refer to the specifics of a machine. This would include the detailed logic design and the packaging technology of the machine. This book focuses on instruction set architecture and on organization. Two machines with identical instruction set architectures and nearly identical organizations that differ primarily at the hardware level are the VAX-11/780 and the 11/785; the 11/785 used an improved integrated circuit technology to obtain a faster clock rate and made some small changes in the memory system. In this book the word "architecture" is intended to cover all three aspects of computer design.

Functional Requirements

Computer architects must design a computer to meet functional requirements as well as price and performance goals. Often, they also have to determine what the functional requirements are, and this can be a major task. The requirements may be specific features, inspired by the market. Application software often drives the choice of certain functional requirements by determining how the machine will be used. If a large body of software exists for a certain instruction set architecture, the architect may decide that a new machine should implement an existing instruction set. The presence of a large market for a particular class of applications might encourage the designers to incorporate requirements that would make the machine competitive in that market. Figure 1.4 (see page 15) summarizes some requirements that need to be considered in designing a new machine. Many of these requirements and features will be examined in depth in later chapters.

Many of the requirements in Figure 1.4 represent a minimum level of support. For example, modern operating systems use virtual memory and protection. This requirement establishes a minimum level of support, without which the machine would not be viable. Any additional hardware above such thresholds can be evaluated from the viewpoint of cost/performance.

Most of the attributes of a computer—hardware support for different data types, performance of different functions, and so on—can be evaluated on the basis of cost/performance for the intended marketplace. The next section discusses how one might make these tradeoffs.

Balancing Software and Hardware

Once a set of functional requirements has been established, the architect must try to optimize the design. Which design choices are optimal depends, of course, on the choice of metrics. The most common metrics involve cost and performance. Given some application domain, one can try to quantify the performance of the machine by a set of programs that are chosen to represent that application domain. (We will see how to measure performance and what aspects affect cost and price in the next chapter.) Other measurable requirements may be important in some markets; reliability and fault tolerance are often crucial in transaction processing environments.

Throughout this text we will focus on optimizing machine cost/performance. This optimization is largely a question of where is the best place to implement some required functionality? Hardware and software implementations of a feature have different advantages. The major advantages of a software implementation are the lower cost of errors, easier design, and simpler upgrading. Hardware offers performance as its sole advantage, though hardware implementations are not always faster—a superior algorithm in software can beat an inferior algorithm implemented in hardware. Balancing hardware and software will lead to the best machine for the applications of interest.

Sometimes a specific requirement may effectively necessitate the inclusion of hardware support. For example, a machine that is to run scientific applications with intensive floating-point calculations will almost certainly need hardware for floating-point operations. This is not a question of functionality, but rather of performance. Software-based floating point could be used, but it is so much slower that the machine would not be competitive. Hardware-supported floating point is a de facto requirement for the scientific marketplace. By comparison, consider building a machine to support commercial applications written in

Functional requirements	Typical features required or supported
Application area	Target of computer
Special purpose	Higher performance for specific applications (Ch. 10)
General purpose	Balanced performance for a range of tasks
Scientific	High-performance floating point (Appendix A)
Commercial	Support for COBOL (decimal arithmetic), support for data bases and transaction processing
Level of software compatibility	Determines amount of existing software for machine (Ch. 10)
At programming language	Most flexible for designer, need new compiler
Object code or binary compatible	Architecture is completely defined—little flexibility—but no investment needed in software or porting programs
Operating system (OS) requirements	Necessary features to support chosen OS
Size of address space	Very important feature (Ch. 8); may limit applications
Memory management	Required for modern OS; may be flat, paged, segmented (Ch. 8)
Protection	Different OS and application needs: page vs. segment protection (Ch. 8)
Context switch	Required to interrupt and restart program; performance varies (Ch. 5)
Interrupts and traps	Types of support impact hardware design and OS (Ch. 5)
Standards	Certain standards may be required by marketplace
Floating point	Format and arithmetic: IEEE, DEC, IBM (Appendix A)
I/O bus	For I/O devices: VME, SCSI, NuBus, Futurebus (Ch. 9)
Operating systems	UNIX, DOS or vendor proprietary
Networks	Support required for different networks: Ethernet, FDDI (Ch. 9)
Programming languages	Languages (ANSI C, FORTRAN 77, ANSI COBOL) affect instruction set

FIGURE 1.4 Summary of some of the most important functional requirements an architect faces. The left-hand column describes the class of requirement, while the right-hand column gives examples of specific features that might be needed. We will look at these design requirements in more detail in later chapters.

COBOL. Such applications make heavy use of decimal and string operations; thus, many architectures have included instructions for these functions. Other machines have supported these functions using a combination of software and standard integer and logical operations. This is a classic example of a tradeoff between hardware and software implementation, and there is no single correct solution.

In choosing between two designs, one factor that an architect must consider is design complexity. Complex designs take longer to complete, prolonging time to market. This means a design that takes longer will need to have higher performance to be competitive. In general, it is easier to deal with complexity in software than in hardware, chiefly because it is easier to debug and change software. Thus, designers may choose to shift functionality from hardware to software. On the other hand, design choices in the instruction set architecture and in the organization can affect the complexity of the implementation as well as the complexity of compilers and operating systems for the machine. The architect must be constantly aware of the impact of his design choices on the design time for both hardware and software.

Designing to Last Through Trends

If an architecture is to be successful, it must be designed to survive changes in hardware technology, software technology, and application characteristics. The designer must be especially aware of trends in computer usage and in computer technology. After all, a successful new instruction set architecture may last tens of years—the core of the IBM 360 has been in use since 1964. An architect must plan for technology changes that can increase the lifetime of a successful machine.

To plan for the evolution of a machine, the designer must be especially aware of rapidly occurring changes in implementation technology. Figure 1.5 shows some of the most important trends in hardware technology. In writing this book, the emphasis is on design principles that can be applied with new technologies and on accounting for ongoing technology trends.

These technology changes are not continuous but often occur in discrete steps. For example, DRAM (dynamic random-access memory) sizes are always increased by factors of 4 due to the basic design structure. Thus, rather than doubling every year or two, DRAM technology quadruples every three or four years. This stepwise change in technology leads to thresholds that can enable an implementation technique that was previously impossible. For example, when MOS technology reached the point where it could put between 25,000 and 50,000 transistors on a single chip, it became possible to build a 32-bit microprocessor on a single chip. By eliminating chip crossings within the CPU, a dramatic decrease in cost/performance was possible. This design was simply infeasible until the technology reached a certain point. Such technology thresholds are not rare and have a significant impact on a wide variety of design decisions.

The architect will also need to be aware of trends in software and how programs will use the machine. One of the most important software trends is the increasing amount of memory used by programs and their data. The amount of memory needed by the average program has grown by a factor of 1.5 to 2 per year! This translates to a consumption of address bits at a rate of 1/2 bit to 1 bit per year. Underestimating address-space growth is often the major reason why an instruction set architecture must be abandoned. (For a further discussion, see Chapter 8 on memory hierarchy.)

Another important software trend in the past 20 years has been the replacement of assembly language by high-level languages. This trend has resulted in a larger role for compilers and in the redirection of architectures toward the support of the compiler. Compiler technology has been steadily improving. A designer must understand this technology and the direction in which it is evolving since compilers have become the primary interface between user and machine. We will talk about the effects of compiler technology in Chapter 3.

A fundamental change in the way programming is done may demand changes in an architecture to efficiently support the programming model. But the emergence of new programming models occurs at a much slower rate than improvements in compiler technology: As opposed to compilers, which improve yearly, significant changes in programming languages occur about once a decade.

Technology	Density and performance trend
IC logic technology	Transistor count on a chip increases by about 25% per year, doubling in three years. Device speed increases nearly as fast.
Semiconductor DRAM	Density increases by just under 60% per year, quadrupling in three years. Cycle time has improved very slowly, decreasing by about one-third in ten years.
Disk technology	Density increases by about 25% per year, doubling in three years. Access time has improved by one-third in ten years.

FIGURE 1.5 Trends in computer implementation technologies show the rapid changes that designers must deal with. These changes can have a dramatic impact on designers when they affect long-term decisions, such as instruction set architecture. The cost per transistor for logic and the cost per bit for semiconductor or disk memory decrease at very close to the rate at which density increases. Cost trends are considered in more detail in the next chapter. In the past, DRAM (dynamic random-access memory) technology has improved faster than logic technology. This difference has occurred because of reductions in the number of transistors per DRAM cell and the creation of specialized technology for DRAMs. As the improvement from these sources diminishes, the density growth in logic technology and memory technology should become comparable.

When an architect has understood the impact of hardware and software trends on machine design, he can then consider the question of how to balance the machine. How much memory do you need to plan for the targeted CPU speed? How much I/O will be required? To try to give some idea of what would constitute a balanced machine, Case and Amdahl coined two rules of thumb that are now usually combined. The combined rule says that a 1-MIPS *(million instructions per second)* machine is balanced when it has 1 megabyte of memory and 1-megabit-per-second throughput of I/O. This rule of thumb provides a reasonable starting point for designing a balanced system, but should be refined by measuring the system performance of the machine when it is executing the intended applications.

1.5 Putting It All Together: The Concept of Memory Hierarchy

In the "Putting It All Together" sections that appear near the end of every chapter, we show real examples that use the principles in that chapter. In this first chapter, we discuss a key idea in memory systems that will be the sole focus of our attention in Chapter 8.

To begin this section, let's look at a simple axiom of hardware design: *smaller is faster*. Smaller pieces of hardware will generally be faster than larger pieces. This simple principle is particularly applicable to memories for two different reasons. First, in high-speed machines, signal propagation is a major cause of delay; larger memories have more signal delay and require more levels to decode addresses. Second, in most technologies one can obtain smaller memories that are faster than larger memories. This is primarily because the designer can use more power per memory cell in a smaller design. The fastest memories are generally available in smaller numbers of bits per chip at any point in time, but they cost substantially more per byte.

Increasing memory bandwidth and decreasing the latency of memory access are both crucial to system performance, and many of the organizational techniques we discuss will focus on these two metrics. How can we improve these two measures? The answer lies in combining the principles we discussed in this chapter together with the rule that smaller is faster.

The principle of locality of reference says that the data most recently used is likely to be accessed again in the near future. Favoring accesses to such data will improve performance. Thus, we should try to keep recently accessed items in the fastest memory. Because smaller memories will be faster, we want to use smaller memories to try to hold the most recently accessed items close to the CPU and successively larger (and slower) memories as we move further away

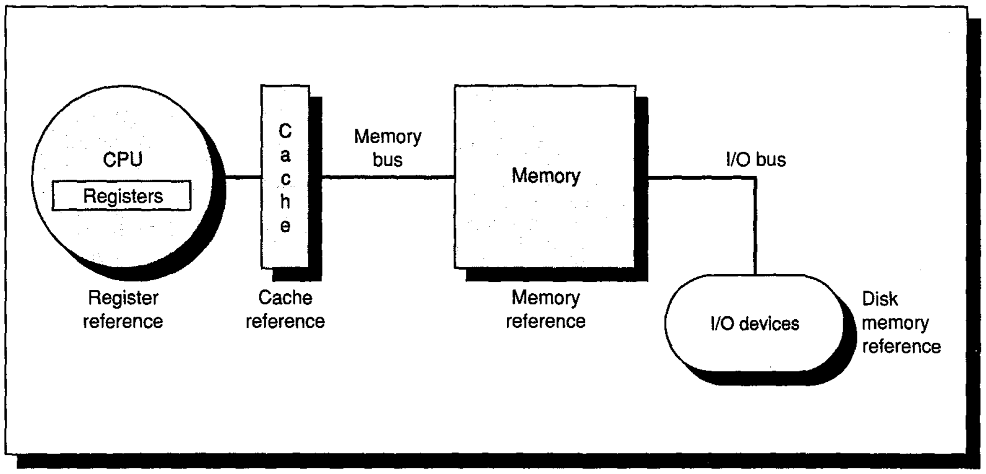


FIGURE 1.6 These are the levels in a typical memory hierarchy. As we move further away from the CPU, the memory in the level becomes larger and slower.

from the CPU. This type of organization is called a *memory hierarchy.* In Figure 1.6, a typical multilevel memory hierarchy is shown. Two important levels of the memory hierarchy are the cache and virtual memory.

A *cache* is a small, fast memory located close to the CPU that holds the most recently accessed code or data. When the CPU does not find a data item it needs in the cache, a *cache miss* occurs, and the data is retrieved from main memory and put into the cache. This usually causes the CPU to pause until the data is available.

Likewise, not all objects referenced by a program need to reside in main memory. If the computer has *virtual memory,* then some objects may reside on disk. The address space is usually broken into fixed-size blocks, called *pages.* At any time, each page resides either in main memory or on disk. When the CPU references an item within a page that is not present in the cache or main memory, a *page fault* occurs, and the entire page is moved from the disk to main memory. The cache and main memory have the same relationship as the main memory and disk.

Level	**1**	**2**	**3**	**4**
Called	Registers	Cache	Main memory	Disk storage
Typical size	< 1 KB	< 512 KB	< 512 MB	> 1 GB
Access time (in ns)	10	20	100	20,000,000
Bandwidth (in MB/sec.)	800	200	133	4
Managed by	Compiler	Hardware	Operating system	Operating system/user
Backed by	Cache	Main memory	Disk	Tape

FIGURE 1.7 The typical levels in the hierarchy slow down and get larger as we move away from the CPU. Sizes are typical for a large workstation or minicomputer. The access time is given in nanoseconds. Bandwidth is given in MB per second, assuming 32-bit paths between levels in the memory hierarchy. As we move to lower levels of the hierarchy, the access times increase, making it feasible to manage the transfer less responsively. The values shown are typical in 1990 and will no doubt change over time.

Machine	**Register size**	**Register access time**	**Cache size**	**Cache access time**
VAX-11/780	16 32-bit	100 ns	8 KB	200 ns
VAXstation 3100	16 32-bit	40 ns	1 KB on chip, 64 KB off chip	125 ns
DECstation 3100	32 32-bit integer; 16 64-bit floating point	30 ns	64 KB instruction; 64 KB data	60 ns

FIGURE 1.8 Sizes and access times for the register and cache levels of the hierarchy vary dramatically among three different machines.

Typical sizes of each level in the memory hierarchy and their access times are shown in Figure 1.7. While the disk and main memory are usually configurable, the register count and cache size are typically fixed for an implementation. Figure 1.8 shows these values for three machines discussed in this text.

Because of locality and the higher speed of smaller memories, a memory hierarchy can substantially improve performance.

Example

Suppose we have a computer with a small, high-speed memory that holds 2000 instructions. Assume that 10% of the instructions are responsible for 90% of the instruction accesses and that the accesses to that 10% are uniform. (That is, each of the instructions in the heavily used 10% is executed an equal number of times.) If we have a program with 50,000 instructions and we know which 10% of the program is most heavily used, what fraction of the instruction accesses can be made to go to high-speed memory?

Answer

Ten percent of 50,000 is 5000. Hence, we can fit 2/5 of the 90%, or 36% of the instructions fetched.

How significant is the impact of memory hierarchy? Let's do a simplified example to illustrate its impact. Though we will evaluate memory hierarchies in a much more precise fashion in Chapter 8, this rudimentary example illustrates the potential impact.

Example

Suppose a cache is five times faster than main memory, and suppose that the cache can be used 90% of the time. How much speedup do we gain by using the cache?

Answer

This is a simple application of Amdahl's Law.

$$\text{Speedup} = \frac{1}{(1 - \%\text{ of time cache can be used}) + \dfrac{\%\text{ of time cache can be used}}{\text{Speedup using cache}}}$$

$$\text{Speedup} = \frac{1}{(1-0.9) + \dfrac{0.9}{5}}$$

$$\text{Speedup} = \frac{1}{0.28} \approx 3.6$$

Hence, we obtain a speedup from the cache of about 3.6 times.

1.6 Fallacies and Pitfalls

The purpose of this section, which will be found in every chapter, is to explain some commonly held misbeliefs or misconceptions that one could acquire. We call such misbeliefs *fallacies*. When discussing a fallacy, we try to give a counterexample. We also discuss *pitfalls*—easily made mistakes. Often pitfalls are generalizations of principles that are true in a limited context. The purpose of these sections is to help you avoid making these errors in machines that you design.

Pitfall: Ignoring the inexorable progress of hardware when planning a new machine.

Suppose you plan to introduce a machine in three years, and you claim the machine will be a terrific seller because it's twice as fast as anything available today. Unfortunately, the machine will probably not sell well, because the performance growth rate for the industry will yield machines of the same performance. For example, assuming a 25% yearly growth rate in performance, a machine with performance x today can be expected to have performance $1.25^3x=1.95x$ in three years. Your machine would have essentially no performance advantage! Many projects within computer companies are canceled, either because they do not pay attention to this rule or because the project slips and the performance of the delayed machine is below the industry average. While this phenomenon can occur in any industry, the rapid improvements in cost/performance make this a major concern in the computer industry.

Fallacy: Hardware is always faster than software.

While a hardware implementation of a well-defined and necessary feature is faster than a software implementation, the functionality provided by the hardware is often more general than the needs of the software. Thus, a compiler may be able to choose a sequence of simpler instructions that accomplishes the required work more efficiently than the more general hardware instruction. A good example is the MVC (move character) instruction in the IBM 360 architecture. This instruction is very general and will move up to 256 bytes of data between two arbitrary addresses. The source and destination may begin at any byte address—and may even overlap. In the worst case, the hardware must move one byte at a time; determining whether the worst case exists requires significant analysis when the instruction is decoded.

Because the MVC instruction is very general, it incurs overhead that is often unnecessary. A software implementation can be faster if it can eliminate this overhead. Measurements have shown that nonoverlapped moves are 50 times more frequent than overlapped moves and that the average nonoverlapped move is only 8 bytes long. In fact, more than half of the nonoverlapped moves move only a single byte! A two-instruction sequence that loads a byte into a register and then stores it in memory is at least twice as fast as `MVC` when moving a single byte. This illustrates the rule of making the frequent case fast.

1.7 Concluding Remarks

The task the computer designer faces is a complex one: Determine what attributes are important for a new machine, then design a machine to maximize performance while staying within cost constraints. Performance can be measured as either throughput or response time; because some environments favor one over the other, this distinction must be borne in mind when evaluating alternatives. Amdahl's Law is a valuable tool to help determine what performance improvement an architectural enhancement can have. In the next chapter we will look at how to measure performance and what properties have the biggest impact on cost.

Knowing what cases are the most frequent is critical to improving performance. In Chapters 3 and 4, we will look at instruction set design and use, watching for common properties of instruction set usage. Based on measurements of instruction sets, tradeoffs can be made by deciding which instructions are the most important and what cases to try to make fast.

In Chapters 5 and 6 we will examine the fundamentals of CPU design, starting with a simple sequential machine and moving to pipelined implementations. Chapter 7 focuses on applying these ideas to high-speed scientific computation in the form of vector machines. Amdahl's Law will be our guiding light throughout Chapter 7.

We have seen how a fundamental property of programs—the principle of locality—can help us build faster computers by allowing us to make effective use of small, fast memories. In Chapter 8, we will return to memory hierarchies, looking in depth at cache design and support for virtual memory. The design of high-performance memory hierarchies has become a key component of modern computer design. Chapter 9 deals with a closely allied topic—I/O systems. As we saw when using Amdahl's Law to evaluate a cost/performance tradeoff, it is not sufficient to merely improve CPU time. To keep a balanced machine, we must also boost I/O performance.

Finally, in Chapter 10, we will look at current research directions focusing on parallel processing. How these ideas will affect the kinds of machines designed and used in the future is not yet clear. What is clear is that an empirical and experimental approach to designing new computers will be the basis for continued and dramatic performance growth.

1.8 Historical Perspective and References

If ... history ... teaches us anything, it is that man in his quest for knowledge and progress, is determined and cannot be deterred.

John F. Kennedy, Address at Rice University, September 12, 1962.

A section of historical perspectives closes each chapter in the text. This section provides some historical background on some of the key ideas presented in the chapter. The authors may trace the development of an idea through a series of machines or describe some important projects. This section will also contain references for the reader interested in examining the initial development of an idea or machine or interested in further reading.

The First Electronic Computers

J. Presper Eckert and John Mauchly at the Moore School of the University of Pennsylvania built the world's first electronic general-purpose computer. This machine, called ENIAC (Electronic Numerical Integrator and Calculator), was funded by the United States Army and became operational during World War II, but was not publicly disclosed until 1946. ENIAC was a general-purpose machine used for computing artillery firing tables. One hundred feet long by eight-and-a-half feet high and several feet wide, the machine was enormous—far beyond the size of any computer built today. Each of the 20, 10-digit registers was two feet long. In total, there were 18,000 vacuum tubes.

While the size was two orders of magnitude bigger than machines built today, it was more than three orders of magnitude slower, with an add taking 200 microseconds. The ENIAC provided conditional jumps and was programmable, which clearly distinguished it from earlier calculators. Programming was done manually by plugging up cables and setting switches. Data was provided on punched cards. Programming for typical calculations required from a half-hour to a whole day. The ENIAC was a general-purpose machine limited primarily by a small amount of storage and tedious programming.

In 1944, John von Neumann was attracted to the ENIAC project. The group wanted to improve the way programs were entered and discussed storing programs as numbers; von Neumann helped crystalize the ideas and wrote a memo proposing a stored-program computer called EDVAC (Electronic Discrete Variable Automatic Computer). Herman Goldstine distributed the memo and put von Neumann's name on it, much to the dismay of Eckert and Mauchly, whose names were omitted. This memo has served as the basis for the commonly used term "von Neumann computer." The authors and several early inventors in the computer field believe that this term gives too much credit to von Neumann, who wrote up the ideas, and too little to the engineers, Eckert and Mauchly, who worked on the machines. For this reason, this term will not appear in this book.

In 1946, Maurice Wilkes of Cambridge University visited the Moore School to attend the latter part of a series of lectures on developments in electronic computers. When he returned to Cambridge, Wilkes decided to embark on a project to build a stored-program computer named EDSAC, for Electronic Delay Storage Automatic Calculator. The EDSAC became operational in 1949 and was the world's first full-scale, operational, stored-program computer [Wilkes, Wheeler, and Gill 1951; Wilkes 1985]. (A small prototype called the Mark I, which was built at the University of Manchester and ran in 1948, might be called the first operational stored-program machine.) The EDSAC was an accumulator-based architecture. This style of machine remained popular until the early 1970s, and the instruction sets looked remarkably similar to the EDSAC. (Chapter 3 starts with a brief summary of the EDSAC instruction set.)

In 1947, Eckert and Mauchly applied for a patent on electronic computers. The dean of the Moore School, by demanding the patent be turned over to the university, may have helped Eckert and Mauchly conclude they should leave. Their departure crippled the EDVAC project, which did not become operational until 1952.

Goldstine left to join von Neumann at the Institute for Advanced Study at Princeton in 1946. Together with Arthur Burks, they issued a report (1946) based on the memo written earlier. The paper led to the IAS machine built by Julian Bigelow at Princeton's Institute for Advanced Study. It had a total of 1024, 40-bit words and was roughly 10 times faster than ENIAC. The group thought about uses for the machine, published a set of reports, and encouraged visitors. These reports and visitors inspired the development of a number of new computers. The paper by Burks, Goldstine, and von Neumann was incredible for the period. Reading it today, one would never guess this landmark paper was written more than 40 years ago, as most of the architectural concepts seen in modern computers are discussed there.

Recently, there has been some controversy about John Atanasoff, who built a small-scale electronic computer in the early 1940s [Atanasoff 1940]. His machine, designed at Iowa State University, was a special-purpose computer that was never completely operational. Mauchly briefly visited Atanasoff before he built ENIAC. The presence of the Atanasoff machine, together with delays in filing the ENIAC patents (the work was classified and patents could not be filed until after the war) and the distribution of von Neumann's EDVAC paper, were used to break the Eckert-Mauchly patent [Larson 1973]. Though controversy still rages over Atanasoff's role, Eckert and Mauchly are usually given credit for building the first working, general-purpose, electronic computer [Stern 1980]. Another early machine that deserves some credit was a special-purpose machine built by Konrad Zuse in Germany in the late 1930s and early 1940s. This machine was electromechanical and, due to the war, was never extensively pursued.

In the same time period as ENIAC, Howard Aiken was building an electromechanical computer called the Mark-I at Harvard. He followed the Mark-I by a relay machine, the Mark-II, and a pair of vacuum tube machines, the Mark-III and Mark-IV. The Mark-III and Mark-IV were being built after the first stored-program machines. Because they had separate memories for instructions and data, the machines were regarded as reactionary by the advocates of stored-program computers. The term *Harvard architecture* was coined to describe this type of machine. Though clearly different from the original sense, this term is used today to apply to machines with a single main memory but with separate instruction and data caches.

The Whirlwind project [Redmond and Smith 1980] was begun at MIT in 1947 and was aimed at applications in real-time radar signal processing. While it led to several inventions, its overwhelming innovation was the creation of magnetic core memory. Whirlwind had 2048, 16-bit words of magnetic core. Magnetic cores served as the main memory technology for nearly 30 years.

Commercial Developments

In December 1947, Eckert and Mauchly formed Eckert-Mauchly Computer Corporation. Their first machine, the BINAC, was built for Northrop and was shown in August 1949. After some financial difficulties, they were acquired by Remington-Rand, where they built the UNIVAC I, designed to be sold as a general-purpose computer. First delivered in June 1951, the UNIVAC I sold for $250,000 and was the first successful commercial computer—48 systems were built! Today, this early machine, along with many other fascinating pieces of computer lore, can be seen at the Computer Museum in Boston, Massachusetts.

IBM, which earlier had been in the punched card and office automation business, didn't start building computers until 1950. The first IBM computer, the IBM 701, shipped in 1952 and eventually sold 19 units. In the early 1950s, many people were pessimistic about the future of computers, believing that the market and opportunities for these "highly specialized" machines were quite limited.

Several books describing the early days of computing have been written by the pioneers [Wilkes 1985; Goldstine 1972]. There are numerous independent histories, often built around the people involved [Slater 1987; Shurkin 1984], as well as a journal, *Annals of the History of Computing,* devoted to the history of computing.

The history of some of the computers invented after 1960 can be found in Chapters 3 and 4 (the IBM 360, the DEC VAX, the Intel 80x86, and the early RISC machines), Chapter 6 (the pipelined processors, including the CDC 6600), and Chapter 7 (vector processors including the TI ASC, CDC Star, and Cray processors).

Computer Generations—
A Capsule Summary of Computer History

Since 1952, there have been thousands of new computers, using a wide range of technologies and having widely varying capabilities. In an attempt to get a perspective on the developments, the industry has tended to group computers into generations. This classification is often based on the implementation technology used in each generation, as shown in Figure 1.9. Typically, each computer generation is eight to ten years in length, though the length and start times—especially of recent generations—is debated. By convention, the first generation is taken to be commercial electronic computers, rather than the mechanical or electromechanical machines that preceded them.

Generation	Dates	Technology	Principal new product	New companies and machines
1	1950-1959	Vacuum tubes	Commercial, electronic computer	IBM 701, UNIVAC I
2	1960-1968	Transistors	Cheaper computers	Burroughs 6500, NCR, CDC 6600, Honeywell
3	1969-1977	Integrated circuit	Minicomputer	50 new companies: DEC PDP-11, Data General Nova
4	1978-199?	LSI and VLSI	Personal computers and workstations	Apple II, Apollo DN 300, Sun 2
5	199?-	Parallel processing?	Multiprocessors?	??

FIGURE 1.9 Computer generations are usually determined by the change in dominant implementation technology. Typically, each generation offers the opportunity to create a new class of computers and for new computer companies to be created. Many researchers believe that parallel processing using high-performance microprocessors will be the basis for the fifth computer generation.

Development of Principles Discussed in This Chapter

What is perhaps the most basic principle was originally stated by Amdahl [1967] and concerned the limitations on speedup in the context of parallel processing:

A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude. [p. 485]

Amdahl stated his law focusing on the implications of speeding up only a portion of the computation. The basic equation can be used as a general technique for measuring the speedup and cost-effectiveness of any enhancement.

Virtual memory first appeared on a machine called Atlas, designed in England in 1962 [Kilburn, et al. 1982]. The IBM 360/85, introduced in the late 1960s, was the first commercial machine to use a cache, but it seems that the idea was discussed for several machines being built in England in the early 1960s (see the discussion in Chapter 8).

Knuth [1971] published the original observations about program locality:

Programs typically have a very jagged profile, with a few sharp peaks. As a very rough approximation, it appears that the nth most important statement of a program from the point of view of execution time accounts for about $(a-1)a^{-n}$ of the running time, for some 'a' and for small 'n'. We also found that less than 4 per cent of a program generally accounts for more than half of its running time. [p. 105]

References

AMDAHL, G. M. [1967]. "Validity of the single processor approach to achieving large scale computing capabilities," *Proc. AFIPS 1967 Spring Joint Computer Conf. 30* (April), Atlantic City, N.J., 483–485.

ATANASOFF, J. V. [1940]. "Computing machine for the solution of large systems of linear equations," Internal Report, Iowa State University.

BELL, C. G. [1984]. "The mini and micro industries," *IEEE Computer* 17:10 (October) 14–30.

BURKS, A. W., H. H. GOLDSTINE, AND J. VON NEUMANN [1946]. "Preliminary discussion of the logical design of an electronic computing instrument," Report to the U.S. Army Ordnance Department, p. 1; also appears in *Papers of John von Neumann,* W. Aspray and A. Burks, eds., The MIT Press, Cambridge, Mass. and Tomash Publishers, Los Angeles, Calif., 1987, 97–146.

GOLDSTINE, H. H. [1972]. *The Computer: From Pascal to von Neumann,* Princeton University Press, Princeton, N.J.

KILBURN, T., D. B. G. EDWARDS, M. J. LANIGAN, AND F. H. SUMNER [1982]. "One-level storage system," reprinted in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), McGraw-Hill, New York.

KNUTH, D. E. [1971]. "An empirical study of FORTRAN programs," *Software Practice and Experience,* Vol. 1, 105–133.

LARSON, JUDGE E. R. [1973]. "Findings of Fact, Conclusions of Law, and Order for Judgment," File No. 4–67, Civ. 138, *Honeywell v. Sperry Rand and Illinois Scientific Development,* U.S. District Court for the District of Minnesota, Fourth Division (October 19).

REDMOND, K. C. AND T. M. SMITH [1980]. *Project Whirlwind—The History of a Pioneer Computer,* Digital Press, Boston, Mass.

SHURKIN, J. [1984]. *Engines of the Mind: A History of the Computer*, W. W. Norton, New York.

SLATER, R. [1987]. *Portraits in Silicon,* The MIT Press, Cambridge, Mass.

STERN, N. [1980]. "Who invented the first electronic digital computer," *Annals of the History of Computing* 2:4 (October) 375–376.

WILKES, M. V. [1985]. *Memoirs of a Computer Pioneer,* The MIT Press, Cambridge, Mass.

WILKES, M. V., D. J. WHEELER, AND S. GILL [1951]. *The Preparation of Programs for an Electronic Digital Computer*, Addison-Wesley Press, Cambridge, Mass.

EXERCISES

1.1 [10/10/10/12/12/12] <1.1,1.2> Here are the execution times in seconds for the Linpack benchmark and 10,000 iterations of the Dhrystone benchmark (see Figure 2.5, page 47) on VAX models:

Model	Year shipped	Linpack execution time (seconds)	Dhrystone execution time (10,000 iterations) (seconds)
VAX-11/780	1978	4.90	5.69
VAX 8600	1985	1.43	1.35
VAX 8550	1987	0.695	0.96

a. [10] How much faster is the 8600 than the 780 using Linpack? How about using Dhrystone?

b. [10] How much faster is the 8550 than the 8600 using Linpack? How about using Dhrystone?

c. [10] How much faster is the 8550 than the 780 using Linpack? How about using Dhrystone?

d. [12] What is the average performance growth per year between the 780 and the 8600 using Linpack? How about using Dhrystone?

e. [12] What is the average performance growth per year between the 8600 and the 8550 using Linpack? How about using Dhrystone?

f. [12] What is the average performance growth per year between the 780 and the 8550 using Linpack? How about using Dhrystone?

1.2–1.5 For the next four questions, assume that we are considering enhancing a machine by adding a vector mode to it. When a computation is run in vector mode it is 20 times faster than the normal mode of execution. We call the percentage of time that could be spent using vector mode the *percentage of vectorization.*

1.2 [20] <1.3> Draw a graph that plots the speedup as a percentage of the computation performed in vector mode. Label the y axis "Net Speedup" and label the x axis "Percent Vectorization."

1.3 [10] <1.3> What percent of vectorization is needed to achieve a speedup of 2?

1.4 [10] <1.3> What percentage of vectorization is needed to achieve one-half the maximum speedup attainable from using vector mode?

1.5 [15] <1.3> Suppose you have measured the percentage of vectorization for programs to be 70%. The hardware design group says they can double the speed of the vector rate with a significant additional engineering investment. You wonder whether the compiler crew could increase the use of vector mode as another approach to increasing performance. How much of an increase in the percentage of vectorization (relative to current usage) would you need to obtain the same performance gain? Which investment would you recommend?

1.6 [12/12] <1.1, 1.4> There are two design teams at two different companies. The smaller and more aggressive company's management demands a two-year design cycle for their products. The larger and less aggressive company's management settles for a four-year design cycle. Assume that today the market they will be selling to demands 25 times the performance of a VAX-11/780.

a. [12] What should the performance goals for each product be, if the growth rates need to be 30% per year?

b. [12] Suppose that the companies have just switched to using 4-megabit DRAMS. Assuming the growth rates in Figure 1.5 (page 17) hold, what DRAM sizes should be planned for use in these projects? Note that DRAM growth is discrete, with each generation being four times larger than the previous generation.

1.7 [12] <1.3> You are considering two alternative designs for an instruction memory: using expensive and fast chips or cheaper and slower chips. If you use the slow chips you can afford to double the width of the memory bus and fetch two instructions, each one word long, every two clock cycles. (With the more expensive fast chips, the memory bus can only fetch one word every clock cycle.) Due to spatial locality, when you fetch two words you often need both. However, in 25% of the clock cycles one of the two words you fetched will be useless. How do the memory bandwidths of these two systems compare?

1.8 [15/10] <1.3> Assume—as in the Amdahl's Law example at the bottom of page 10—that we make an enhancement to a computer that improves some mode of execution by a factor of 10. Enhanced mode is used 50% of the time, measured as a percentage of the execution time when the enhanced mode is in use, rather than as defined in this chapter: the percentage of the running time **without** the enhancement.

a. [15] What is the speedup we have obtained from fast mode?

b. [10] What percentage of the original execution time has been converted to fast mode?

1.9 [15/15] <1.5> Assume we are building a machine with a memory hierarchy for instructions (don't worry about data accesses!). Assume that the program follows the 90-10 rule and that accesses within the top 10% and bottom 90% are uniformly distributed; that is, 90% of the time is spread evenly over 10% of the code and the other 10% of the time is spread evenly over the other 90% of the code. You have three types of memory for use in your memory hierarchy:

Memory type	Access time	Cost per word
Local, fast	1 clock cycle	$0.10
Main	5 clock cycles	$0.01
Disk	5,000 clock cycles	$0.0001

You have exactly 100 programs, each is 1,000,000 words, and all the programs must fit on disk. Assume that only one program runs at a time, and that the whole program must be loaded in main memory. You can spend $30,000 dollars on the memory hierarchy.

a. [15] What is the optimal way to allocate your budget assuming that each word must be statically placed in fast memory or main memory?

b. [15] Ignoring the time for the first loading from disk, what is the average number of cycles for a program to make a memory reference in your hierarchy? (This important measure is called the average memory-access time in Chapter 8.)

1.10 [30] <1.3,1.6> Find a machine that has both a fast and slow implementation of a feature—for example, a system with and without hardware floating point. Measure the speedup obtained when using the faster implementation with a simple loop that uses the feature. Find a real program that makes some use of the feature and measure the speedup. Using this data, compute the percentage of the time the feature is used.

1.11 [Discussion] <1.3,1.4> Often ideas for speeding up processors take advantage of some special properties that certain classes of applications have. Thus, the speedup obtained by an enhancement may be available to only certain applications. How would you decide to make such an enhancement? What factors would be most relevant in the decision? Could these factors be measured or estimated reasonably?

Remember that time is money.

BEN FRANKLIN, *Advice to a Young Tradesman*

2.1 Introduction **33**

2.2 Performance **35**

2.3 Cost **53**

2.4 Putting It All Together: Price/Performance of Three Machines **66**

2.5 Fallacies and Pitfalls **70**

2.6 Concluding Remarks **76**

2.7 Historical Perspective and References **77**

Exercises **81**

2 Performance and Cost

2.1 Introduction

Why do engineers design different computers? Why do people use them? How do customers decide on one computer versus another? Is there a rational basis for their decisions? If so, can engineers use that basis to design better computers? These are some of the questions this chapter addresses.

One way to approach these questions is to see how they have been used in another design field and then apply those solutions by analogy to our own. The automobile, for example, can provide a useful source of analogies for explaining computers: We could say that CPUs are like engines, supercomputers are like exotic race cars, and fast CPUs with slow memories are like hot engines in poor chassis.

Standard measures of performance provide a basis for comparison, leading to improvements of the object measured. Races helped determine which car and driver were faster, but it was hard to separate the skills of the driver from the performance of the car. A few standard performance tests eventually evolved, such as

- Time until the car reaches a given speed, typically 60 miles per hour
- Time to cover a given distance, typically 1/4 mile
- Top speed on a level surface

Standard measures allow designers to select between alternatives quantitatively, which enables orderly progress in a field.

Make and model	Month tested	Price (as tested)	Sec (0-60)	Sec (1/4 mi.)	Top speed	Brake (80-0)	Skidpad g	Fuel MPG
Chevrolet Corvette	2-88	$34,034	6.0	14.6	158	225	0.89	17.5
Ferrari Testarossa	10-89	$145,580	6.2	14.2	181	261	0.87	12.0
Ford Escort	7-87	$5,765	11.2	18.8	95	286	0.69	37.0
Hyundai Excel	10-86	$6,965	14.0	19.4	80	291	0.73	29.9
Lamborghini Countach	3-86	$118,000	5.2	13.7	173	252	0.88	10.0
Mazda Miata	7-89	$15,550	9.2	16.8	116	270	0.83	25.5

FIGURE 2.1 Quantitative automotive cost/performance summary. These data were taken from the October 1989 issue of Road and Track, page 26. "Road Test Summary" is found in every issue of the magazine.

Cars proved so popular that magazines were developed to feed the interest in new cars and to help readers decide which car to purchase. While these magazines have always carried articles describing the impressions of driving a new car—the qualitative experience—over time they have expanded the quantitative basis for comparison, as Figure 2.1 illustrates.

Performance, cost of purchase, and cost of operation dominate these summaries. Performance and cost also form the rational basis for deciding which computer to select. Thus, computer designers must understand both performance and cost if they want to design computers people will consider worth selecting.

Just as there is no single target for car designers, so there is no single target for computer designers. At one extreme, *high-performance design* spares no cost in achieving its goal. Supercomputers from Cray as well as sports cars from Ferrari and Lamborghini fit into this category. At the other extreme is *low-cost design*, where performance is sacrificed to achieve lowest cost. Computers like the IBM PC clones along with their automotive equivalents, such as the Ford Escort and the Hyundai Excel, belong here. In between these extremes is *cost/performance design* where the designer balances cost versus performance. Examples from the minicomputer or workstation industry typify the kinds of tradeoffs with which designers of the Corvette and Miata would feel comfortable.

It is on this middle ground, where neither cost nor performance is neglected, that we will focus our discussion. We begin by looking at performance, the measure of the designer's dream, before going on to describe the accountant's agenda—cost.

2.2 Performance

Time is the measure of computer performance: the computer that performs the same amount of work in the least time is the fastest. Program *execution time* is measured in seconds per program. Performance is frequently measured as a rate of some number of events per second, so that lower time means higher performance. We tend to blur this distinction and talk about performance as either time or a rate, reporting refinements as improved performance rather than using adjectives higher (for rates) or lower (for time).

But time can be defined in different ways depending on what we count. The most straightforward definition of time is called wall-clock time, response time, or *elapsed time*. This is the latency to complete a task, including disk accesses, memory accesses, input/output activities, operating system overhead—everything. However, since with multiprogramming the CPU works on another program while waiting for I/O and may not necessarily minimize the elapsed time of one program, there needs to be a term to take this activity into account. *CPU time* recognizes this distinction and means the time the CPU is computing **not** including the time waiting for I/O or running other programs. (Clearly the response time seen by the user is the elapsed time of the program, not the CPU time.) CPU time can be further divided into the CPU time spent in the program, called *user CPU time*, and the CPU time spent in the operating system performing tasks requested by the program, called *system CPU time*.

These distinctions are reflected in the UNIX time command, which returned the following:

```
90.7u 12.9s 2:39 65%
```

User CPU time is 90.7 seconds, system CPU time is 12.9 seconds, elapsed time is 2 minutes and 39 seconds (159 seconds), and the percentage of elapsed time that is CPU time is (90.7+12.9)/159 or 65%. More than a third of the elapsed time in this example was spent waiting for I/O or running other programs or both. Many measurements ignore system CPU time because of the inaccuracy of operating systems' self-measurement and the inequity of including system CPU time when comparing performance between machines with differing system codes. On the other hand, system code on some machines is user code on others and no program runs without some operating system running on the hardware, so a case can be made for using the sum of user CPU time and system CPU time.

In the present discussion, a distinction is maintained between performance based on elapsed time and that based on CPU time. The term *system performance* is used to refer to elapsed time on an **unloaded** system, while *CPU performance* refers to **user** CPU time. We will concentrate on CPU performance in this chapter.

CPU Performance

Most computers are constructed using a clock running at a constant rate. These discrete time events are called ticks, clock ticks, clock periods, clocks, cycles, or *clock cycles.* Computer designers refer to the time of a clock period by its length (e.g., 10 ns) or by its rate (e.g., 100 MHz).

CPU time for a program can then be expressed two ways:

$$\text{CPU time} = \text{CPU clock cycles for a program} * \text{Clock cycle time}$$

or

$$\text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

Note that it wouldn't make sense to show elapsed time as a function of CPU clock cycle time since the latency for I/O devices is normally independent of the rate of the CPU clock.

In addition to the number of clock cycles to execute a program, we can also count the number of instructions executed—the instruction path length or *instruction count.* If we know the number of clock cycles and the instruction count we can calculate the average number of *clock cycles per instruction* (CPI):

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}}$$

This CPU figure of merit provides insight into different styles of instruction sets and implementations.

By transposing instruction count in the above formula, clock cycles can be defined as instruction count * CPI. This allows us to use CPI in the execution time formula:

$$\text{CPU time} = \text{Instruction count} * \text{CPI} * \text{Clock cycle time}$$

or

$$\text{CPU time} = \frac{\text{Instruction count} * \text{CPI}}{\text{Clock rate}}$$

Expanding the first formula into the units of measure shows how the pieces fit together:

$$\frac{\text{Instructions}}{\text{Program}} * \frac{\text{Clock cycles}}{\text{Instruction}} * \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}$$

As this formula demonstrates, CPU performance is dependent upon three characteristics: clock cycle (or rate), clock cycles per instruction, and instruction count. You can't change one of these in isolation from others because the basic technologies involved in changing each characteristic are also interdependent:

Clock rate—Hardware technology and organization

CPI—Organization and instruction set architecture

Instruction count—Instruction set architecture and compiler technology

Sometimes it is useful in designing the CPU to calculate the number of total CPU clock cycles as

$$\text{CPU clock cycles} = \sum_{i=1}^{n} (\text{CPI}_i * \text{I}_i)$$

where I_i represents number of times instruction *i* is executed in a program and CPI_i represents the average number of clock cycles for instruction *i*. This form can be used to express CPU time as

$$\text{CPU time} = \sum_{i=1}^{n} (\text{CPI}_i * \text{I}_i) * \text{Clock cycle time}$$

and overall CPI as

$$\text{CPI} = \frac{\sum_{i=1}^{n} (\text{CPI}_i * \text{I}_i)}{\text{Instruction count}} = \sum_{i=1}^{n} \left(\text{CPI}_i * \frac{\text{I}_i}{\text{Instruction count}}\right)$$

The latter form of the CPI calculation multiplies each individual CPI_i by the fraction of occurrences in a program.

CPI_i should be measured and not just calculated from a table in the back of a reference manual since it must include cache misses and any other memory system inefficiencies.

Always bear in mind that the real measure of computer performance is time. Changing the instruction set to lower the instruction count, for example, may lead to an organization with a slower clock cycle time that offsets the improvement in instruction count. When comparing two machines, you must look at all three components to understand relative performance.

Example

Suppose we are considering two alternatives for our conditional branch instructions, as follows:

CPU A. A condition code is set by a compare instruction and followed by a branch that tests the condition code.

CPU B. A compare is included in the branch.

On both CPUs, the conditional branch instruction takes 2 cycles, and all other instructions take 1 clock cycle. (Obviously, if the CPI is 1.0 for everything but branches in this simple example we are ignoring losses due to the memory system in this decision; see the fallacy on page 72.) On CPU A, 20% of all instructions executed are conditional branches; since every branch needs a compare, another 20% of the instructions are compares. Because CPU A does not have the compare included in the branch, its clock cycle time is 25% faster than CPU B's. Which CPU is faster?

Answer

Since we are ignoring all systems issues, we can use the CPU performance formula: CPI_A is ((.20*2) + (.80*1)) or 1.2 since 20% are branches taking 2 clock cycles and the rest take 1. Clock cycle time_B is 1.25 * Clock cycle time_A since A is 25% faster. The performance of CPU A is then

$$\begin{aligned}\text{CPU time}_A &= \text{Instruction count}_A * 1.2 * \text{Clock cycle time}_A\\ &= 1.20 * \text{Instruction count}_A * \text{Clock cycle time}_A\end{aligned}$$

Compares are not executed in CPU B, so 20%/80% or 25% of the instructions are now branches, taking 2 clock cycles, and the remaining 75% of the instructions take 1. CPI_B is then ((.25*2) + (.75*1)) or 1.25. Because CPU B doesn't execute compares, Instruction count_B is .80*Instruction count_A. The performance of CPU B is

$$\begin{aligned}\text{CPU time}_B &= (.80*\text{Instruction count}_A) * 1.25 * (1.25*\text{Clock cycle time}_A)\\ &= 1.25 * \text{Instruction count}_A * \text{Clock cycle time}_A\end{aligned}$$

Under these assumptions, CPU A, with the shorter clock cycle time, is faster than CPU B, which executes fewer instructions.

Example

After seeing the analysis, a designer realized that by reworking the organization the difference in clock cycle times can easily be reduced to 10%. Which CPU is faster now?

Answer

The only change from the answer above is that Clock cycle time_B is now 1.10 * Clock cycle time_A since A is just 10% faster. The performance of CPU A is still

$$\text{CPU time}_A = 1.20 * \text{Instruction count}_A * \text{Clock cycle time}_A$$

The performance of CPU B is now

$$\begin{aligned}\text{CPU time}_B &= (.80*\text{Instruction count}_A) * 1.25 * (1.10*\text{Clock cycle time}_A)\\ &= 1.10 * \text{Instruction count}_A * \text{Clock cycle time}_A\end{aligned}$$

With this improvement CPU B, which executes fewer instructions, is now faster.

Example

Suppose we are considering another change to an instruction set. The machine initially has only loads and stores to memory, and then all operations work on the registers. Such machines are called *load/store* machines (see Chapter 3). Measurements of the load/store machine showing the frequency of instructions, called an *instruction mix*, and clock cycle counts per instruction are given in Figure 2.2.

Operation	Frequency	Clock cycle count
ALU ops	43%	1
Loads	21%	2
Stores	12%	2
Branches	24%	2

FIGURE 2.2 An example of instruction frequency. The CPI for each class of instruction is also given. (This frequency comes from the GCC column of Figure C.4 in Appendix C, rounded up to account for 100% of the instructions.)

Let's assume that 25% of the *arithmetic logic unit* (ALU) operations directly use a loaded operand that is not used again.

We propose adding ALU instructions that have one source operand in memory. These new *register–memory instructions* have a clock cycle count of 2. Suppose that the extended instruction set increases the clock cycle count for branches by 1, but it does not affect the clock cycle time. (Chapter 6, on pipelining, explains why adding register–memory instructions might slow down branches.) Would this change improve CPU performance?

Answer

The question is whether the new machine is faster than the old machine. We use the CPU performance formula since we are again ignoring systems issues. The original CPI is calculated by multiplying together the two columns from Figure 2.2:

$$\text{CPI}_{\text{old}} = (.43*1 + .21*2 + .12*2 + .24*2) = 1.57$$

The performance of CPU_{old} is then

$$\begin{aligned}\text{CPU time}_{\text{old}} &= \text{Instruction count}_{\text{old}} * 1.57 * \text{Clock cycle time}_{\text{old}} \\ &= 1.57 * \text{Instruction count}_{\text{old}} * \text{Clock cycle time}_{\text{old}}\end{aligned}$$

Let's give the formula for CPI_{new} first and then explain the components:

$\text{CPI}_{\text{new}} =$

$$\frac{(.43-(.25*.43))*1 + (.21-(.25*.43))*2 + (.25*.43)*2 + .12*2 + .24*3}{1-(.25*.43)}$$

25% of ALU instructions (which are 43% of all instructions executed) become register–memory instructions, changing the first 3 components of the numerator. There are (.25*.43) fewer ALU operations, (.25*.43) fewer loads, and (.25*.43) new register–memory ALU instructions. The rest of the numerator remains the same except the branches take 3 clock cycles instead of 2. We divide by the new instruction count, which is .25*43% smaller than the old one. Simplifying this equation:

$$\text{CPI}_{\text{new}} = \frac{1.703}{.893} = 1.908$$

Since the clock cycle time is unchanged, the performance of the new CPU is

$$\begin{aligned}\text{CPU time}_{\text{new}} &= (.893 * \text{Instruction count}_{\text{old}}) * 1.908 * \text{Clock cycle time}_{\text{old}} \\ &= 1.703 * \text{Instruction count}_{\text{old}} * \text{Clock cycle time}_{\text{old}}\end{aligned}$$

Using these assumptions, the answer to our question is no: It's a bad idea to add register–memory instructions, because they do not offset the increased execution time of slower branches.

MIPS and What Is Wrong with Them

A number of popular measures have been adopted in the quest for a standard measure of computer performance, with the result that a few innocent terms have been shanghaied from their well-defined environment and forced into a service for which they were never intended. The authors' position is that the only consistent and reliable measure of performance is the execution time of real programs, and that all proposed alternatives to time as the metric or to real programs as the items measured have eventually led to misleading claims or even mistakes in computer design. The dangers of a few popular alternatives to our advice are shown first.

One alternative to time as the metric is MIPS, or *million instructions per second.* For a given program, MIPS is simply

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} * 10^6} = \frac{\text{Clock rate}}{\text{CPI} * 10^6}$$

Some find this rightmost form convenient since clock rate is fixed for a machine and CPI is usually a small number, unlike instruction count or execution time. Relating MIPS to time,

$$\text{Execution time} = \frac{\text{Instruction count}}{\text{MIPS} * 10^6}$$

Since MIPS is a rate of operations per unit time, performance can be specified as the inverse of execution time, with faster machines having a higher MIPS rating.

The good news about MIPS is that it is easy to understand, especially by a customer, and faster machines means bigger MIPS, which matches intuition. The problem with using MIPS as a measure for comparison is threefold:

- MIPS is dependent on the instruction set, making it difficult to compare MIPS of computers with different instruction sets;
- MIPS varies between programs on the same computer; and most importantly,
- MIPS can vary inversely to performance!

The classic example of the last case is the MIPS rating of a machine with optional floating-point hardware. Since it generally takes more clock cycles per floating-point instruction than per integer instruction, floating-point programs using the optional hardware instead of software floating-point routines take less time but have a **lower** MIPS rating. Software floating point executes simpler instructions, resulting in a higher MIPS rating, but it executes so many more that overall execution time is longer.

We can even see such anomalies with optimizing compilers.

Example

Assume we build an optimizing compiler for the load/store machine described in the previous example. The compiler discards 50% of the ALU instructions, although it cannot reduce loads, stores, or branches. Ignoring systems issues and assuming a 20-ns clock cycle time (50-MHz clock rate), what is the MIPS rating for optimized code versus unoptimized code? Does the ranking of MIPS agree with the ranking of execution time?

Answer

From the example above $\text{CPI}_{\text{unoptimized}} = 1.57$, so

$$\text{MIPS}_{\text{unoptimized}} = \frac{50\text{ MHz}}{1.57*10^6} = 31.85$$

The performance of unoptimized code is

$$\begin{aligned}\text{CPU time}_{\text{unoptimized}} &= \text{Instruction count}_{\text{unoptimized}} * 1.57 * (20*10^{-9}) \\ &= 31.4*10^{-9} * \text{Instruction count}_{\text{unoptimized}}\end{aligned}$$

For optimized code

$$\text{CPI}_{\text{optimized}} = \frac{(.43/2)*1 + .21*2 + .12*2 + .24*2}{1-(.43/2)} = \frac{.215 + .42 + .24 + .48}{.785} = 1.73$$

since half the ALU instructions are discarded (.43/2) and the instruction count is reduced by the missing ALU instructions. Thus,

$$\text{MIPS}_{\text{optimized}} = \frac{50 \text{ MHz}}{1.73 * 10^6} = 28.90$$

The performance of optimized code is

$$\text{CPU time}_{\text{optimized}} = (.785 * \text{Instruction count}_{\text{unoptimized}}) * 1.73 * (20 * 10^{-9})$$

$$= 27.2 * 10^{-9} * \text{Instruction count}_{\text{unoptimized}}$$

Optimized code is 13% faster, but its MIPS rating is lower!

As examples such as this one show, MIPS can fail to give a true picture of performance in that it does not track execution time. To compensate for this weakness, another alternative to execution time is to use a particular machine, with an agreed-upon MIPS rating, as a reference point. *Relative MIPS*—as distinguished from the original form, called *native MIPS*—is then calculated as follows:

$$\text{Relative MIPS} = \frac{\text{Time}_{\text{reference}}}{\text{Time}_{\text{unrated}}} * \text{MIPS}_{\text{reference}}$$

where

$Time_{reference}$ = execution time of a program on the reference machine

$Time_{unrated}$ = execution time of the same program on machine to be rated

$MIPS_{reference}$ = agreed-upon MIPS rating of the reference machine

Relative MIPS only tracks execution time for the given program and input. Even when they are identified, it becomes harder to find a reference machine on which to run programs as the machine ages. (In the 1980s the dominant reference machine was the VAX-11/780, which was called a 1-MIPS machine; see pages 77–78 in Section 2.7.) The question also arises whether the older machine should be run with the newest release of the compiler and operating system, or whether the software should be fixed so the reference machine does not get faster over time. There is also the temptation to generalize from a relative MIPS rating using one benchmark to relative execution time, even though there can be wide variations in relative performance.

In summary, the advantage of relative MIPS is small since execution time, program, and program input still must be known to have meaningful information.

MFLOPS and What Is Wrong with Them

Another popular alternative to execution time is *million floating-point operations per second*, abbreviated megaFLOPS or MFLOPS but always pronounced "megaflops." The formula for MFLOPS is simply the definition of the acronym:

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

Clearly, a MFLOPS rating is dependent on the machine and on the program. Since MFLOPS were intended to measure floating-point performance, they are not applicable outside that range. Compilers, as an extreme example, have a MFLOPS rating near nil no matter how fast the machine since compilers rarely use floating-point arithmetic.

This term is less innocent than MIPS. Based on operations rather than instructions, MFLOPS is intended to be a fair comparison between different machines. The belief is that the same program running on different computers would execute a different number of instructions but the same number of floating-point operations. Unfortunately, MFLOPS is not dependable because the set of floating-point operations is not consistent across machines. For example, the CRAY-2 has no divide instruction, while the Motorola 68882 has divide, square root, sine, and cosine. Another perceived problem is that the MFLOPS rating changes not only on the mixture of integer and floating-point operations but also on the mixture of fast and slow floating-point operations. For example, a program with 100% floating-point adds will have a higher rating than a program with 100% floating-point divides. The solution for both problems is to give a canonical number of floating-point operations in the source-level program and then divide by execution time. Figure 2.3 shows how the authors of the "Livermore Loops" benchmark calculate the number of normalized floating-point operations per program according to the operations actually found in the source code. Thus, the *native MFLOPS* rating is not the same as the *normalized MFLOPS* rating reported in the supercomputer literature, which has come as a surprise to a few computer designers.

Real FP operations	Normalized FP operations
ADD, SUB, COMPARE, MULT	1
DIVIDE, SQRT	4
EXP, SIN, ...	8

FIGURE 2.3 Real versus normalized floating-point operations. The number of normalized floating-point operations per real operation in a program used by the authors of the Livermore FORTRAN Kernels, or "Livermore Loops," to calculate MFLOPS. A kernel with one ADD, one DIVIDE, and one SIN would be credited with 13 normalized floating-point operations. Native MFLOPS won't give the results reported for other machines on that benchmark.

Example

The Spice program runs on the DECstation 3100 in 94 seconds (see Figures 2.16 to 2.18 for more details on the program, input, compilers, machine, and so on). The number of floating-point operations executed in that program are listed below:

ADDD	25,999,440
SUBD	18,266,439
MULD	33,880,810
DIVD	15,682,333
COMPARED	9,745,930
NEGD	2,617,846
ABSD	2,195,930
CONVERTD	1,581,450
TOTAL	109,970,178

What is the native MFLOPS for that program? Using the conversions in Figure 2.3, what is the normalized MFLOPS?

Answer

Native MFLOPS is easy to calculate:

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

$$\approx \frac{110\text{M}}{94*10^6} \approx 1.2$$

The only operation in Figure 2.3 that is changed for normalized MFLOPS and is in the list above is divide, raising the total of (normalized) floating-point operations, and therefore MFLOPS, almost 50%:

$$\text{Normalized MFLOPS} \approx \frac{157\text{M}}{94*10^6} \approx 1.7$$

Like any other performance measure, the MFLOPS rating for a single program cannot be generalized to establish a single performance metric for a computer. Since normalized MFLOPS is really just a constant divided by execution time for a specific program and specific input (like relative MIPS), MFLOPS is redundant to execution time, our principal measure of performance. And unlike execution time, it is tempting to characterize a machine with a single MIPS or MFLOPS rating without naming the program. Finally, MFLOPS is not a useful measure for all programs.

Choosing Programs to Evaluate Performance

Dhrystone does not use floating point. Typical programs don't ...

RICK RICHARDSON, *Clarification of Dhrystone,* 1988

This program is the result of extensive research to determine the instruction mix of a typical FORTRAN program. The results of this program on different machines should give a good indication of which machine performs better under a typical load of FORTRAN programs. The statements are purposely arranged to defeat optimizations by the compiler.

Anonymous, from comments in the Whetstone benchmark

A computer user who runs the same programs day in and day out would be the perfect candidate to evaluate a new computer. To evaluate a new system he would simply compare the execution time of his *workload*—the mixture of programs and operating system commands that users run on a machine. Few are in this happy situation, however. Most must rely on other methods to evaluate machines and often other evaluators, hoping that these methods will predict performance for their usage of the new machine. There are four levels of programs used in such circumstances, listed below in decreasing order of accuracy of prediction.

1. *(Real) Programs*—While the buyer may not know what fraction of time is spent on these programs, he knows that some users will run them to solve real problems. Examples are compilers for C, text-processing software like TeX, and CAD tools like Spice. Real programs have input, output, and options that a user can select when running the program.

2. *Kernels*—Several attempts have been made to extract small, key pieces from real programs and use them to evaluate performance. Livermore Loops and Linpack are the best known examples. Unlike real programs, no user would run kernel programs, for they exist solely to evaluate performance. Kernels are best used to isolate performance of individual features of a machine to explain the reasons for differences in performance of real programs.

3. *(Toy) Benchmarks*—Toy benchmarks are typically between 10 and 100 lines of code and produce a result the user already knows before he runs the toy program. Programs like Sieve of Erastosthenes, Puzzle, and Quicksort are popular because they are small, easy to type, and run on almost any computer. The best use of such programs is beginning programming assignments.

4. *Synthetic Benchmarks*—Similar in philosophy to kernels, synthetic benchmarks try to match the average frequency of operations and operands of a large set of programs. Whetstone and Dhrystone are popular synthetic benchmarks. (Figures 2.4 and 2.5 on pages 46 and 47 show pieces of the benchmarks.) Like their cousins, the kernels, no user runs synthetic benchmarks

because they don't compute anything a user could use. Synthetic benchmarks are, in fact, even further removed from reality because kernel code is extracted from real programs, while synthetic code is created artificially to match an average execution profile. Synthetic benchmarks are not even **pieces** of real programs, while all the others might be.

If you're not sure how to classify a program, first check to see if there is any input or very much output. A program without input calculates the same result every time it is invoked. (Few buy computers to act as copying machines.) While some programs, notably simulation and numerical analysis applications, use negligible input, every real program has some input.

```
       I = ITER
       ...
       N8 = 899 * I
       ...
       N11 = 93 * I
       ...
       X = 1.0
       Y = 1.0
       Z = 1.0
       IF (N8) 89,89,81
81     DO 88 I = 1, N8, 1
88           CALL P3(X,Y,Z)
89     CONTINUE
       ...
       X = 0.75
       IF (N11) 119,119,111
111    DO 118 I = 1, N11, 1
118          X = SQRT(EXP(ALOG(X)/T1))
119    CONTINUE
       ...
       SUBROUTINE P3 (X,Y,Z)
       COMMON T, T1, T2
       X1 = X
       Y1 = Y
       X1 = T * (X1 + Y1)
       Y1 = T * (X1 + Y1)
       Z = (X1 + Y1) / T2
       RETURN
       END
       ...
```

FIGURE 2.4 Two loops of the Whetstone synthetic benchmark. Based on the frequency of Algol statements in programs submitted to a university batch operating system in the early 1970s, a synthetic program was created to match that profile. (See Curnow and Wichmann [1976].) The statements at the beginning (e.g., N8 = 899*I) control the number of iterations of each of the 12 loops (e.g., the DO loop at line 81). The program was later converted to FORTRAN and became a popular benchmark in marketing literature. (The line labeled 118 is the subject of a fallacy on pages 73–74 in Section 2.5.)

Because computer companies thrive or go bust depending on price/performance of their products relative to others in the marketplace, tremendous resources are available to improve performance of programs widely used in evaluating performance. Such pressures can skew hardware and software engineering efforts to add optimizations that improve performance of synthetic programs, toy programs, or kernels, but not real programs.

An extreme instance of such targeted engineering employed compiler optimizations that were benchmark sensitive. Rather than perform the analysis so that the compiler could properly decide if the optimization could be applied, a person at one startup company used a preprocessor that scanned the text for keywords to try to identify benchmarks by looking for the name of the author and the name of a key subroutine. If the scan confirmed that this program was on a predefined list, the special optimizations were performed. This machine made

```
...
for(Run_Index = 1; Run_Index<=Number_Of_Runs; ++Run_Index)
{
       Proc_5();
       Proc_4();
       Int_1_Loc = 2;
       Int_2_Loc = 3;
       strcpy(Str_2_Loc,"DHRYSTONE PROGRAMS, 2'ND STRING");
       ...
}
...
Proc_4()
{
       Boolean Bool_Loc;

       Bool_Loc = Ch1_1_Glob == 'A';
       Bool_Glob = Bool_Loc | Bool_Glob;
       Ch1_2_Glob = 'B';
} /* Proc_4 */

Proc_5()
{
       Ch1_1_Glob = 'A';
       Bool_Glob = false;
} /* Proc_5 */
...
```

FIGURE 2.5 A section of the Dhrystone synthetic benchmark. Inspired by Whetstone, this program was an attempt to characterize CPU and compiler performance for a typical program. It was based on the frequency of high-level language statements from a variety of publications. The program was originally written in Ada and later converted to C and Pascal (see Weicker [1984]). Note the small size and simple-minded nature of these procedures makes it trivial for an optimizing compiler to avoid procedure-call overhead by expanding them inline. The `strcpy()` on the eighth line is the subject of a fallacy on pages 73–74 in Section 2.5.

a sudden jump in performance—at least according to those benchmarks. Yet these optimizations were not only invalid to programs not on the list, they were useless to the identical code with a few name changes.

The small size of programs in the last three categories makes them vulnerable to such efforts. For example, despite the best intentions, the initial SPEC benchmark suite (page 79) includes a small program. 99% of the execution time of Matrix300 is in a single line (see SPEC [1989]). A minor enhancement of the MIPS FORTRAN compiler (which improved the induction variable elimination optimization—see Section 3.7 in Chapter 3) resulted in a performance increase of 56% on a M/2000 and 117% on an RC 6280. This concentration of execution time led Apollo down the path of temptation: The performance of the DN 10000 is quoted with this line changed to a call to a hand-coded library routine. If the industry adopts real programs to compare performance, then at least resources expended to improve performance will help real users.

So why doesn't everyone run real programs to measure performance? Kernels and toy benchmarks are attractive when beginning a design since they are small enough to easily simulate, even by hand. They are especially tempting when inventing a new machine because compilers may not be available until much later. Small benchmarks are also more easily standardized while large programs are difficult, hence there are numerous published results for small benchmark performance but few for large ones.

While there are rationalizations for use early in the design, there is no current valid rationale for using benchmarks and kernels to evaluate working computer systems. In the past, programming languages were inconsistent among machines, and every machine had its own operating system; so real programs could not be ported without pain and agony. There was also a lack of important software whose source code was freely available. Finally, programs had to be small because the architecture simulator had to run on an old, slow machine.

The popularity of standard operating systems like UNIX and DOS, freely distributed software from universities and others, and faster computers available today remove many of these obstacles. While kernels, toy benchmarks, and synthetic benchmarks were an attempt to make fair comparisons among different machines, use of anything less than real programs after initial design studies is likely to give misleading results and lead the designer astray.

Reporting Performance Results

The guiding principle of reporting performance measurements should be *reproducibility*—list everything another experimenter would need to duplicate the results. Let's compare descriptions of computer performance found in refereed scientific journals to descriptions of car performance found in magazines sold at supermarkets. Car magazines, in addition to supplying 20 performance metrics, list all optional equipment on the test car, the types of tires used in the performance test, and the date the test was made. Computer journals may have only seconds of execution labeled by the name of the program and the name and model of the computer—Spice takes 94 seconds on a DECstation 3100. Left to the reader's imagination are program input, version of the program, version of compiler, optimizing level of compiled code, version of operating system, amount of main memory, number and types of disks, version of the CPU—all of which make a difference in performance.

Car magazines have enough information about the measurement to allow readers to duplicate results or to question the options selected for measurements, but computer journals often do not.

Comparing and Summarizing Performance

Comparing performance of computers is rarely a dull event, especially when the designers are involved. Charges and countercharges fly across an electronic network; one is accused of underhanded tactics and the other of misleading statements. Since careers sometimes depend on the results of such performance comparisons, it is understandable that the truth is occasionally stretched. But more frequently discrepancies can be explained by differing assumptions or lack of information.

We would like to think that if we can just agree on the programs, the experimental environments, and the definition of "faster," then misunderstandings will be avoided, leaving the networks free for scholarly intercourse. Unfortunately, the outcome is not such a happy one, for battles are then fought over what is the fair way to summarize relative performance of a collection of programs. For example, two articles on summarizing performance in the same journal took opposing points of view. Figure 2.6, taken from one of the articles, is an example of the confusion that can arise.

	Computer A	Computer B	Computer C
Program 1 (secs)	1	10	20
Program 2 (secs)	1000	100	20
Total time (secs)	1001	110	40

FIGURE 2.6 Execution times of two programs on three machines. Taken from Figure I of Smith [1988].

Using our definition in Chapter 1 (page 6), the following statements hold:

A is 900% faster than B for program 1.

B is 900% faster than A for program 2.

A is 1900% faster than C for program 1.

C is 4900% faster than A for program 2.

B is 100% faster than C for program 1.

C is 400% faster than B for program 2.

Taken individually, any one of these statements may be of use. Collectively, however, they present a confusing picture—the relative performance of computers A, B, and C is unclear.

Total Execution Time: A Consistent Summary Measure

The simplest approach to summarizing relative performance is to use total execution time of the two programs. Thus

B is 810% faster than A for programs 1 and 2.

C is 2400% faster than A for programs 1 and 2.

C is 175% faster than B for programs 1 and 2.

This summary tracks execution time, our final measure of performance. If the workload consisted of running programs 1 and 2 an equal number of times, the statements above would predict the relative execution times for the workload on each machine.

An average of the execution times that tracks total execution time is the *arithmetic mean*

$$\frac{1}{n}\sum_{i=1}^{n}\text{Time}_i$$

where Time_i is the execution time for the ith program of a total of n in the workload. If performance is expressed as a rate (such as MFLOPS), then the average that tracks total execution time is the *harmonic mean*

$$\frac{n}{\sum_{i=1}^{n}\frac{1}{\text{Rate}_i}}$$

where Rate_i is a function of $1/\text{Time}_i$, the execution time for the ith of n programs in the workload.

Weighted Execution Time

The question arises what is the proper mixture of programs for the workload: Are programs 1 and 2 in fact run equally in the workload as assumed by the arithmetic mean? If not, then there are two approaches that have been tried for summarizing performance. The first approach when given a nonequal mix of programs in the workload is to assign a weighting factor w_i to each program to indicate the relative frequency of the program in that workload. If, for example, 20% of the tasks in the workload were program 1 and 80% of the tasks in the workload were program 2, then the weighting factors would be 0.2 and 0.8. (Weighting factors add up to 1.) By summing the products of weighting factors and execution times, a clear picture of performance of the workload is obtained. This is called the *weighted arithmetic mean*:

$$\sum_{i=1}^{n} \text{Weight}_i * \text{Time}_i$$

where Weight_i is the frequency of the ith program in the workload and Time_i is the execution time of that program. Figure 2.7 shows the data from Figure 2.6 with three different weightings, each proportional to the execution time of a workload with a given mix. The *weighted harmonic mean* of rates will show the same relative performance as the weighted arithmetic means of execution times. The definition is

$$\frac{1}{\displaystyle\sum_{i=1}^{n} \frac{\text{Weight}_i}{\text{Rate}_i}}$$

	A	B	C	W(1)	W(2)	W(3)
Program 1 (secs)	1.00	10.00	20.00	0.50	0.909	0.999
Program 2 (secs)	1000.00	100.00	20.00	0.50	0.091	0.001
Arithmetic mean :W(1)	500.50	55.00	20.00			
Arithmetic mean :W(2)	91.82	18.18	20.00			
Arithmetic mean :W(3)	2.00	10.09	20.00			

FIGURE 2.7 Weighted arithmetic mean execution times using three weightings. W(1) equally weights the programs, resulting in a mean (row 3) that is the same as the nonweighted arithmetic mean. W(2) makes the mix of programs inversely proportional to the execution times on machine B; row 4 shows the arithmetic mean for that weighting. W(3) weights the programs in inverse proportion to the execution times of the two programs on machine A; the arithmetic mean is given in the last row. The net effect of the second and third weightings is to "normalize" the weightings to the execution times of programs running on that machine, so that the running time will be spent evenly between each program for that machine. For a set of n programs each taking T_i time on one machine, the equal-time weightings on that machine are $w_i = \dfrac{1}{T_i * \sum_{j=1}^{n} \left(\frac{1}{T_j}\right)}$.

Normalized Execution Time and the Pros and Cons of Geometric Means

A second approach to nonequal mixture of programs in the workload is to normalize execution times to a reference machine and then take the average of the normalized execution times, similar to the relative MIPS rating discussed above. This measurement gives a warm fuzzy feeling, because it suggests that performance of new programs can be predicted by simply multiplying this number times its performance on the reference machine.

Average normalized execution time can be expressed as either an arithmetic or *geometric* mean. The formula for the geometric mean is

$$\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}$$

where *Execution time ratio$_i$* is the execution time, normalized to the reference machine, for the *i*th program of a total of *n* in the workload. Geometric means also have the nice property that

$$\frac{\text{Geometric mean}(X_i)}{\text{Geometric mean}(Y_i)} = \text{Geometric mean}\left(\frac{X_i}{Y_i}\right)$$

meaning that taking either the ratio of the means or the means of the ratios gets the same results. In contrast to arithmetic means, geometric means of normalized execution times are consistent no matter which machine is the reference. Hence, the arithmetic mean should **not** be used to average normalized execution times. Figure 2.8 shows some variations using both arithmetic and geometric means of normalized times.

	Normalized to A			Normalized to B			Normalized to C		
	A	B	C	A	B	C	A	B	C
Program 1	100%	1000%	2000%	10%	100%	200%	5%	50%	100%
Program 2	100%	10%	2%	1000%	100%	20%	5000%	500%	100%
Arithmetic mean	100%	505%	1001%	505%	100%	110%	2503%	275%	100%
Geometric mean	100%	100%	63%	100%	100%	63%	158%	158%	100%
Total time	100%	11%	4%	910%	100%	36%	2503%	275%	100%

FIGURE 2.8 Execution times from Figure 2.6 normalized to each machine. The arithmetic mean performance varies depending on which is the reference machine—column 2 says B's execution time is 5 times longer than A's while column 4 says just the opposite; column 3 says C is slowest while column 9 says C is fastest. The geometric means are consistent independent of normalization—A and B have the same performance, and the execution time of C is 63% of A or B (100%/158% is 63%). Unfortunately total execution time of A is 9 times longer than B, and B in turn is about 3 times longer than C. As a point of interest, the relationship between the means of the same set of numbers is always harmonic mean ≤ geometric mean ≤ arithmetic mean.

Because weightings of weighted arithmetic means are set proportionate to execution times on a given machine, as in Figure 2.7, they are influenced not only by frequency of use in the workload, but also by the peculiarities of a particular machine and the size of program input. The geometric mean of normalized execution times, on the other hand, is independent of the running times of the individual programs, and it doesn't matter which machine is used to normalize. If a situation arose in comparative performance evaluation where the programs were fixed but the inputs were not, then competitors could rig the results of weighted arithmetic means by making their best performing benchmark have the largest input and therefore dominate execution time. In such a situation the geometric mean would be less misleading than the arithmetic mean.

The strong drawback to geometric means of normalized execution times is that they violate our fundamental principle of performance measurement—they do not predict execution time. The geometric means from Figure 2.8 suggest that for programs 1 and 2 the performance of machines A and B is the same, yet this would only be true for a workload that ran program 1 100 times for every occurrence of program 2 (see Figure 2.6 on page 49). The total execution time for such a workload suggests that machines A and B are about 80% faster than machine C, in contrast to the geometric mean, which says machine C is faster than A and B! In general there is **no workload** for three or more machines that will match the performance predicted by the geometric means of normalized execution times. Our original reason for examining geometric means of normalized performance was to avoid giving equal emphasis to the programs in our workload, but is this solution an improvement?

The ideal solution is to measure a real workload and weight the programs according to their frequency of execution. If this can't be done, then normalizing so that equal time is spent on each program on some machine at least makes the relative weightings explicit and will predict execution time of a workload with that mix (see Figure 2.7 on page 51). The problem above of unspecified inputs is best solved by specifying the inputs when comparing performance. If results must be normalized to a specific machine, first summarize performance with the proper weighted measure and then do the normalizing. Section 2.4 gives an example.

2.3 Cost

While there are computer designs where costs tend to be ignored—specifically supercomputers—cost-sensitive designs are of growing importance. Textbooks have ignored the cost half of cost/performance because costs change, thereby dating books. Yet an understanding of cost is essential for designers to be able to make intelligent decisions about whether or not a new feature should be included in designs where cost is an issue. (Imagine architects designing skyscrapers without any information on costs of steel beams and concrete.) We therefore cover in this section fundamentals of cost that will not change for the life of the book and provide specific examples using costs that, though they may not hold up over time, demonstrate the concepts involved.

The rapid change in cost of electronics is the first of several themes in cost-sensitive designs. This parameter is changing so fast that good designers are basing decisions not on costs of today, but on projected costs at the time the product is shipped. The underlying principle that drives costs down is the *learning curve*—manufacturing costs decrease over time. The learning curve itself is best measured by change in *yield*—the percentage of manufactured devices that survive the testing procedure. Whether it is a chip, a board, or a system, designs that have twice the yield will have basically half the cost. Understanding how the learning curve will improve yield is key to projecting costs over the life of the product.

Lowering cost, however, does not necessarily lower price; it may just increase profits. But when the product is available from multiple sources and demand does not exceed supply, competition does force prices to fall with costs. For the remainder of this discussion we assume that normal competitive forces are at work with a reasonable balance between supply and demand.

As an example of the learning curve in action, the cost per megabyte of DRAM drops over the long term by 40% per year. A more dramatic version of the same information is shown in Figure 2.9, where the cost of a new DRAM chip is depicted over its lifetime. Between the start of a project and the shipping of a product, say two years, the cost of a new DRAM drops by nearly a factor of four. Since not all component costs change at the same rate, designs based on projected costs result in different cost-performance tradeoffs than those using current costs.

A second important theme in cost-sensitive designs is the impact of packaging on design decisions. A few years ago the advantages of fitting a design on a single board meant there was no backplane, no card cage, and a smaller and cheaper box—all resulting in much lower costs and even higher performance. In a few years it will be possible to integrate all the components of a system, except main memory, onto a single chip. The overriding issue will be making the system fit on the chip, thereby avoiding the speed and cost penalties of having multiple chips, which means more interfaces, more pins to interfaces, larger boards, and so forth. The density of integrated circuits and packaging technology determine the resources available at each cost threshold. The designer must know where these thresholds are—or blindly cross them.

Cost of an Integrated Circuit

Why would a computer architecture book have a section on integrated circuit costs? In an increasingly competitive computer marketplace where standard parts—disks, DRAMs, and so on—are becoming a significant portion of any system's cost, integrated circuit costs are becoming a greater portion of the cost that varies between machines, especially in the high volume, cost-sensitive portion of the market. Thus computer designers must understand the costs of chips to understand the costs of current computers. We follow here the American accounting approach to the cost of chips.

While the costs of integrated circuits have dropped exponentially, the basic procedure of silicon manufacture is unchanged: A *wafer* is still tested and chopped into *dies* that are packaged (see Figures 2.10a, b, and c). Thus the cost of a packaged integrated circuit is

$$\text{Cost of integrated circuit} = \frac{\text{Cost of die} + \text{Cost of testing die} + \text{Cost of packaging}}{\text{Final test yield}}$$

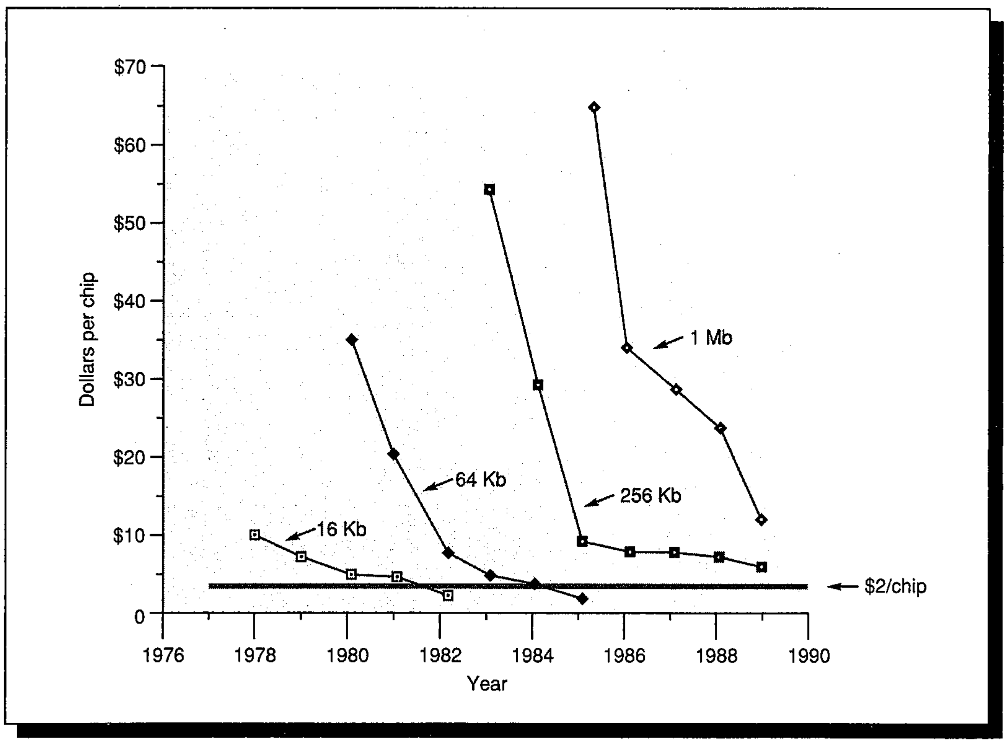


FIGURE 2.9 Prices of four generations of DRAMs over time, showing the learning curve at work. While the longer average is 40% improvement per year, each generation drops in price by nearly a factor of ten over its lifetime. The DRAMs drop to about $1 to $2 per chip over time, independent of capacity. Prices are **not** adjusted for inflation—if they were the graph would show an even greater drop in cost. For a time in 1987–1988, prices of both 256Kb and 1Mb DRAMs were higher than indicated by earlier learning curves due to what seems to have been a temporary excess of demand relative to available supply.

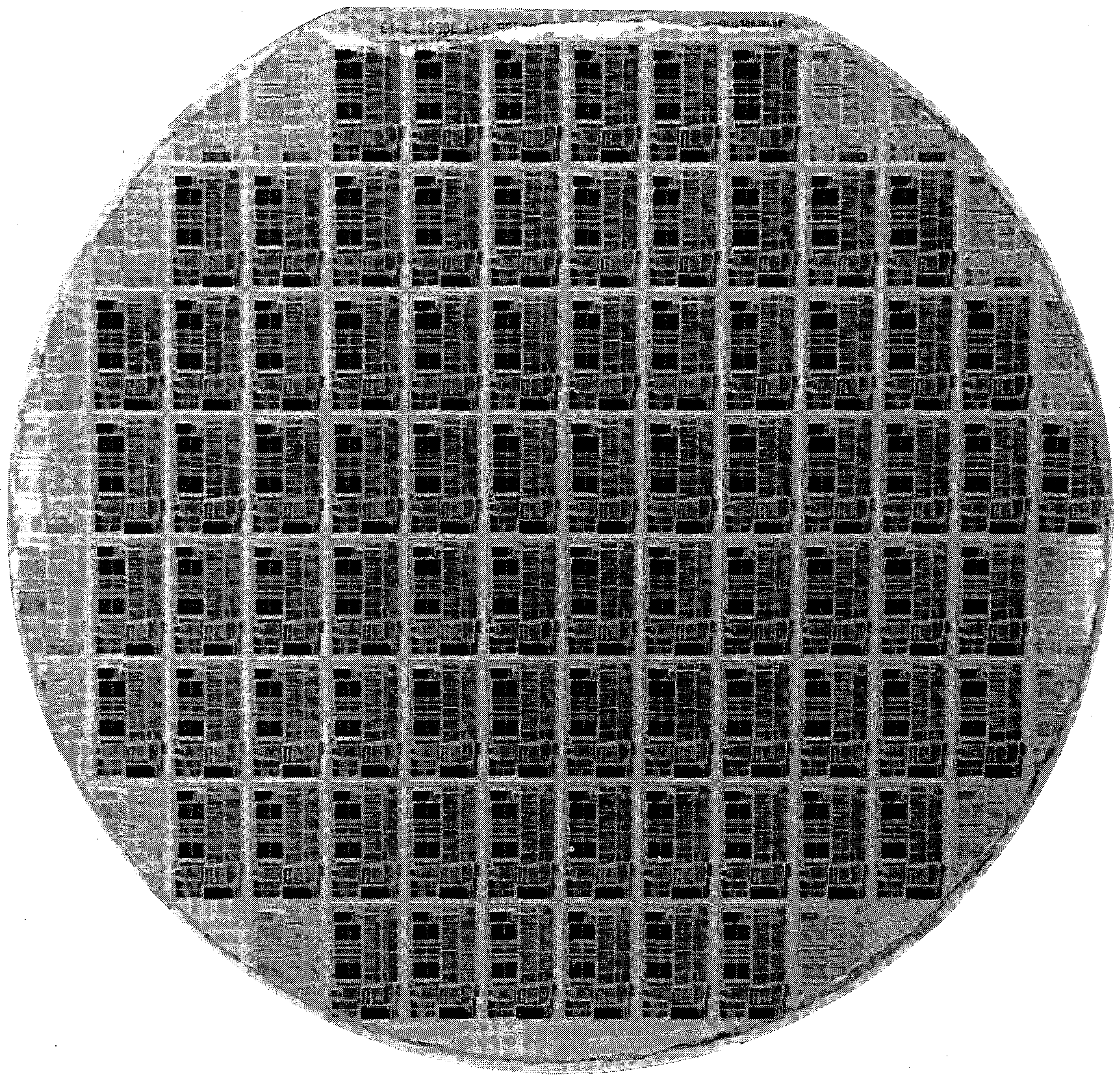


FIGURE 2.10a Photograph of a 6-inch wafer containing Intel 80486 microprocessors. There are 80 1.6 cm x 1.0 cm dies, although four dies are so close to the edge that they may or may not be fully functional. There are no separate test dies; instead, the electrical and parametric test circuits are placed **between** the dies. The 80486 includes a floating point unit, a small cache, and a memory management unit in addition to the integer unit.

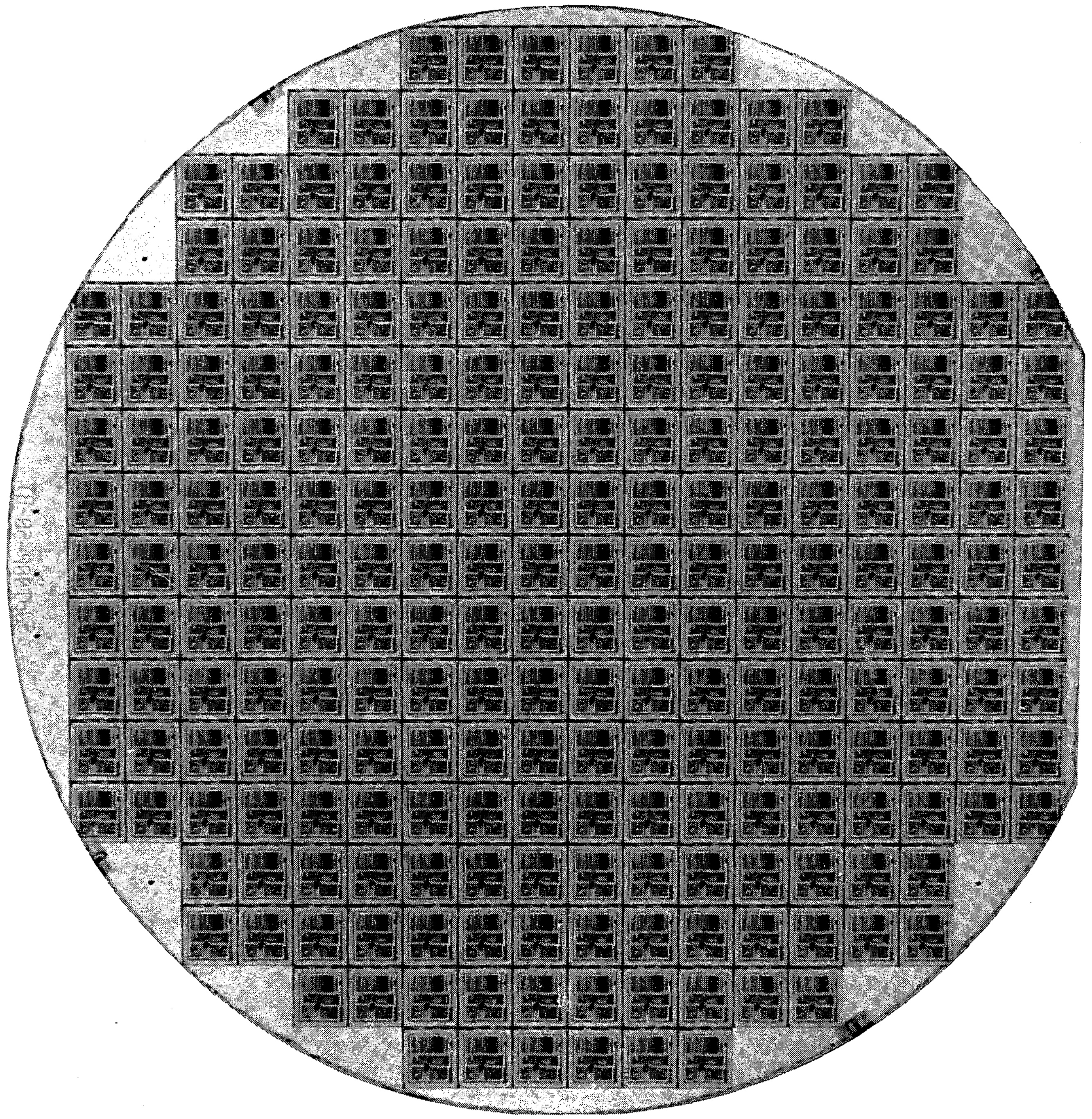


FIGURE 2.10b Photograph of a 6-inch wafer containing Cypress CY7C601 microprocessors. There are 246 full 0.8 cm x 0.7 cm dies, although again four dies are so close to the edge it is hard to tell if they are complete. Like Intel, Cypress places the electrical and parametric test circuits between the dies. These test circuits are removed when the wafer is diced into chips. In contrast to the 80486, the CY7C601 contains the integer unit only.

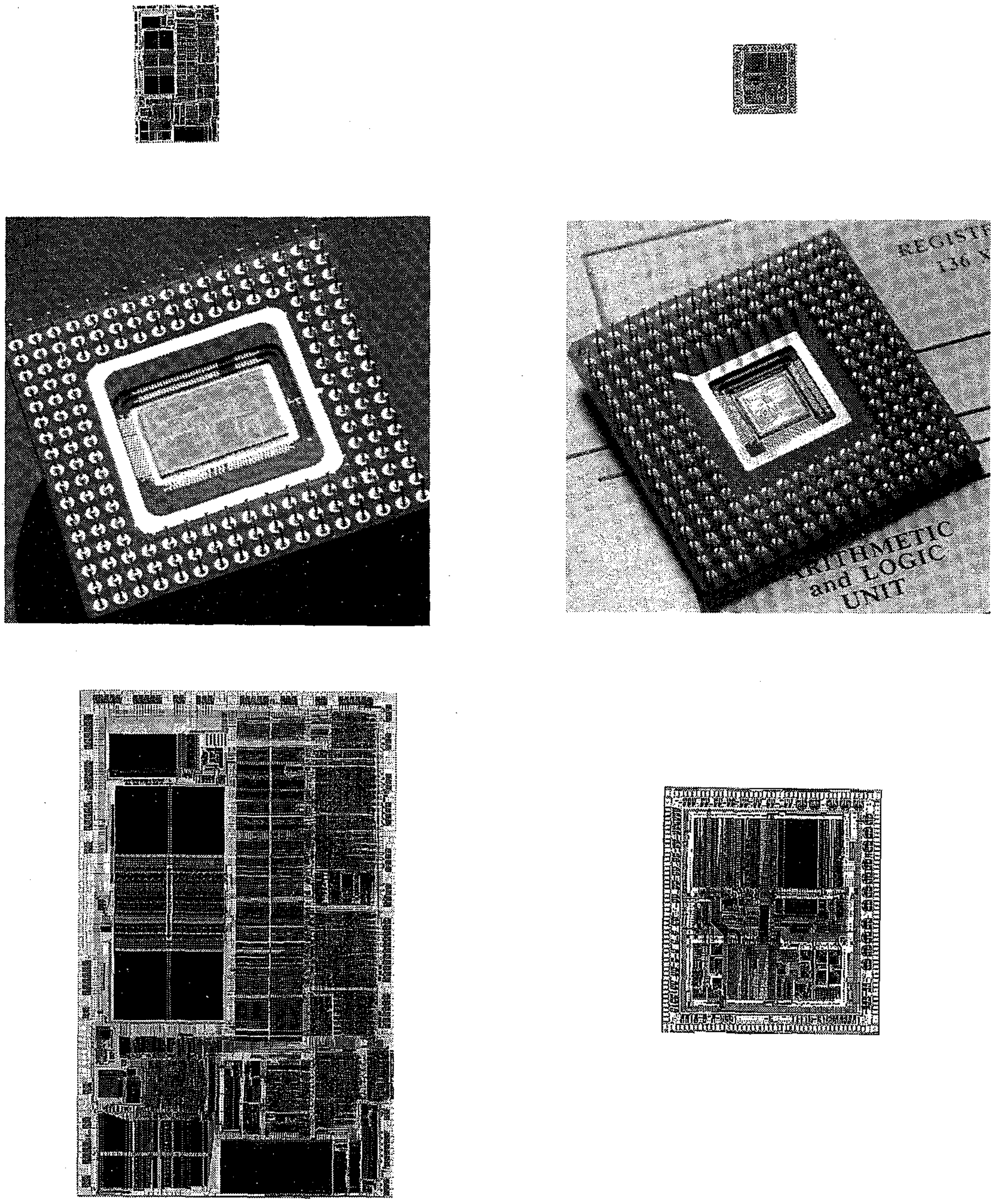


FIGURE 2.10c At the top left is the Intel 80486 die, and the Cypress CY7C601 die is on the right, shown at their actual sizes. Below the dies are the packaged versions of each microprocessor. Note that the 80486 has three rows of pins (168 total) while the 601 has four rows (207 total). The bottom row shows a close-up of the two dies, shown in proper relative proportions.

Cost of Dies

To learn how to predict the number of good chips per wafer requires first learning how many dies fit on a wafer and then how to predict the percentage of those that will work. From there it is simple to predict cost:

$$\text{Cost of die} = \frac{\text{Cost of wafer}}{\text{Dies per wafer} * \text{Die yield}}$$

The most interesting feature of this first term of the chip cost equation is its sensitivity to die size, shown below.

The number of dies per wafer is basically the area of the wafer divided by the area of the die. It can be more accurately estimated by

$$\text{Dies per wafer} = \frac{\pi * (\text{Wafer diameter}/2)^2}{\text{Die area}} - \frac{\pi * \text{Wafer diameter}}{\sqrt{2 * \text{Die area}}} - \text{Test dies per wafer}$$

The first term is the ratio of wafer area (πr^2) to die area. The second compensates for the "square peg in a round hole" problem–rectangular dies near the periphery of round wafers. Dividing the circumference (πd) by the diagonal of a square die is approximately the number of dies along the edge. The last term is for test dies that must be strategically placed to control manufacturing. For example, a 15-cm (≈6-inch) diameter wafer with 5 test dies produces $3.14*225/4 - 3.14*15/\sqrt{2} - 5$ or 138 1-cm-square dies. Doubling die area—the parameter that a computer designer controls—would cut dies per wafer to 59.

But this only gives the maximum number of dies per wafer, and the critical question is what is the fraction or percentage of good dies on a wafer number, or the *die yield.* A simple model of integrated circuit yield assumes defects are randomly distributed over the wafer:

$$\text{Die yield} = \text{Wafer yield} * \left\{ 1 + \frac{\text{Defects per unit area} * \text{Die area}}{\alpha} \right\}^{-\alpha}$$

where *wafer yield* accounts for wafers that are completely bad and so need not be tested and α is a parameter that corresponds roughly to the number of masking levels critical to die yield. α depends upon the manufacturing process. Generally $\alpha = 2.0$ for simple MOS processes and higher values for more complex processes, such as bipolar and BiCMOS. As an example, wafer yield is 90%, *defects per unit area* is 2 per square centimeter, and die area is 1 square centimeter. Then die yield is $90\%*(1 + (2*1)/2.0)^{-2.0}$ or 22.5%.

The bottom line is the number of good dies per wafer, which comes from multiplying dies per wafer by die yield. The examples above predict 138*.225 or 31 good 1-cm-square dies per 15-cm wafer. As mentioned above, both dies per wafer and die yield are sensitive to die size—doubling die area knocks die yield down to 10% and good chips per wafer to just 59*.10, or 6! Die size depends on the technology and gates required by the function on the chip, but it is also limited by the number of pins that can be placed on the border of a square die.

A 15-cm-diameter wafer processed in two-level metal CMOS costs a semiconductor manufacturer about \$550 in 1990. The cost for a 1-cm-square die with two defects per square cm on a 15-cm wafer is \$550/(138*.225) or \$17.74.

What should a computer designer remember about chip costs? The manufacturing process dictates the wafer cost, wafer yield, α, and defects per unit area, so the sole control of the designer is die area. Since α is usually 2 or larger, die costs are proportional to the third (or higher) power of the die area:

$$\text{Cost of die} = f\ (\text{Die area}^3)$$

The computer designer affects die size, and hence cost, both by what functions are included on or excluded from the die and by the number of I/O pins.

Cost of Testing Die and Cost of Packaging

Testing is the second term of the chip-cost equation, and the success rate of testing (die yield) affects the cost of testing:

$$\text{Cost of testing die} = \frac{\text{Cost of testing per hour} * \text{Average die test time}}{\text{Die yield}}$$

Since bad dies are discarded, die yield is in the denominator in the equation—the good must shoulder the costs of testing those that fail. Testing costs about \$150 per hour in 1990 and die tests take about 5 to 90 seconds on average, depending on the simplicity of the die and the provisions to reduce testing time included in the chip. For example, at \$150 per hour and 5 seconds to test, the die test cost is \$0.21. After factoring in die yield for a 1-cm-square die, the costs are \$0.93 per good die. As a second example, let's assume testing takes 90 seconds. The cost is \$3.75 per untested die and \$16.67 per good die. The bill so far for our 1-cm-square die is \$18.67 to \$34.41, depending on how long it takes to test. These two testing-time examples illustrate the importance of reducing testing time in reducing costs.

Cost of Packaging and Final Test Yield

The cost of a package depends on the material used, the number of pins, and the die area. The cost of the material used in the package is in part determined by the ability to dissipate power generated by the die. For example, a *plastic quad flat pack* (PQFP) dissipating less than one watt, with 208 or fewer pins, and containing a die up to one cm on a side costs \$3 in 1990. A ceramic *pin grid array* (PGA) can handle 300 to 400 pins and a larger die with more power, but it costs \$50. In addition to the cost of the package itself is the cost of the labor to place a die in the package and then bond the pads to the pins. We can assume that costs $2. Burn-in exercises the packaged die under power for a short time to catch chips that would fail early. Burn-in costs about $0.25 in 1990 dollars.

We are not finished with costs until we have figured in failure of some chips during assembly and burn-in. Using the estimate of 90% for final test yield, the successful must again pay for the cost of those that fail, so our costs are $26.58 to $96.29 for the 1-cm-square die.

While these specific cost estimates may not hold, the underlying models will. Figure 2.11 shows the dies per wafer, die yield, and their product against the die area for a typical fabrication line, this time using programs that more accurately predict die per wafer and die yield. Figure 2.12 plots the change in area and cost as one dimension of a square die changes. Changes to small dies make little cost difference while 30% increases to large dies can double costs. The wise silicon designer will minimize die area, testing time, and pins per chip and understand the costs of projected packaging options when considering using more power, pins, or area for higher performance.

Cost of a Workstation

To put the costs of silicon in perspective, Figure 2.13 shows the approximate costs of components in a 1990 workstation. Costs of a component can be halved going from low volume to high volume; here we assume high-volume purchasing of 100,000 units. While costs for units like DRAMs will surely drop over time from those in Figure 2.13, units whose prices have already been cut, like displays and cabinets, will change very little.

The processor, floating-point unit, memory-management unit, and cache are only 12% to 21% of the cost of the CPU board in Figure 2.13. Depending on the options included in the system—number of disks, color monitor, and so on—the processor components drop to 9% and 16% of the cost of a system, as Figure 2.14 illustrates. In the future two questions will be interesting to consider: What costs can an engineer control? And what costs can a computer engineer control?

Cost Versus Price—Why They Differ and by How Much

Costs of components may confine a designer's desires, but they are still far from representing what the customer must pay. But why should a computer architecture book contain pricing information? Cost goes through a number of changes before it becomes price, and the computer designer must understand these to determine the impact of design choices. For example, changing cost by $1,000 may change price by $4,000 to $5,000. Without understanding the relationship of cost to price the computer designer may not understand the impact on price of adding, deleting, or replacing components.

Area (sq. cm)	Side (cm)	Die/ wafer	Die yield/ wafer	Cost of die	Cost to test die	Packaging costs	Total cost after final test
0.06	0.25	2778	79.72%	$0.25	$0.63	$5.25	$6.81
0.25	0.50	656	57.60%	$1.46	$0.87	$5.25	$8.42
0.56	0.75	274	36.86%	$5.45	$1.36	$5.25	$13.40
1.00	1.00	143	22.50%	$17.09	$2.22	$5.25	$27.29
1.56	1.25	84	13.71%	$47.76	$3.65	$52.25	$115.18
2.25	1.50	53	8.52%	$121.80	$5.87	$52.25	$199.91
3.06	1.75	35	5.45%	$288.34	$9.17	$52.25	$388.62
4.00	2.00	23	3.60%	$664.25	$13.89	$52.25	$811.54

FIGURE 2.11 Costs for several die sizes. Costs for a working chip are shown in columns 5 through 7. Column 8 is the sum of columns 5 through 7 divided by the final test yield. Figure 2.12 presents this information graphically. This figure assumes a 15.24-cm (6-inch) wafer costing $550, with 5 test die per wafer. The wafer yield is 90%, the defect density is 2.0 per square cm, and α is 2.0. It takes 12 seconds on average to test a die, the tester costs $150 per hour, and the final test yield is 90%. (The numbers differ a little from the text for a 1-cm-square die because the wafer size is calculated at the full 15.24 cm rather than rounded to 15 cm and because of the difference in testing time.)

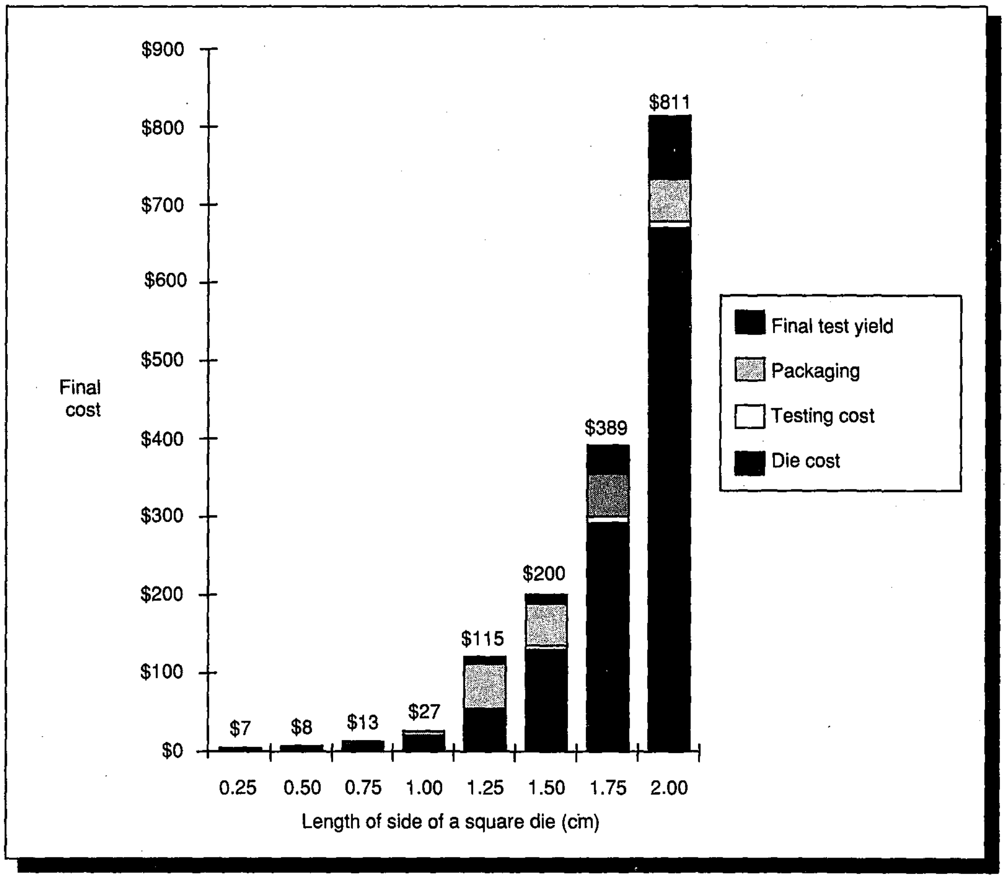


FIGURE 2.12 The costs of a chip from Figure 2.11 presented graphically. Using the parameters given in the text, packaging is a major percentage of the cost of dies of size 1.25-cm square and smaller, with die cost dominating final costs for larger dies.

			Rule of thumb	Lower cost	% Mono WS	Higher cost	% Color WS
CPU cabinet	Sheet metal, plastic			$50	2%	$50	1%
	Power supply and fans		$0.80/watt	$55	3%	$55	1%
	Cables, nuts, bolts			$30	1%	$30	1%
	Shipping box, manuals			$10	0%	$10	0%
	Subtotal			$145	7%	$145	3%
CPU board	IU, FPU, MMU, cache			$200	9%	$800	16%
	DRAM		$150/MB	$1200	56%	$2400	48%
	Video logic (frame buffer, DAC, mono/color)	Mono		$100	5%		
		Color				$500	10%
	I/O interfaces (SCSI, Ethernet, floppy, PROM, time-of-day clock)			$100	5%	$100	2%
	Printed circuit board		8 layers $1.00/sq. in.				
			6 layers $0.50/sq. in.	$50	2%	$50	1%
			4 layers $0.25/sq. in.				
	Subtotal			$1650	77%	$3850	76%
I/O devices	Keyboard, mouse			$50	2%	$50	1%
	Display monitor	Mono		$300	14%		
		Color				$1,000	20%
	Hard disk		100 MB	$400			
	Tape drive		150 MB	$400			
Mono workstation	(8 MB, Mono logic & display, keyboard, mouse, diskless)			$2,145	100%	$2,745	
Color workstation	(16 MB, Color logic & display, keyboard, mouse, diskless)			$4,445		$5,045	100%
File server	(16 MB, 6 disks+tape drive)			$5,595		$6,195	

FIGURE 2.13 Estimated cost of components in a 1990 workstation assuming 100,000 units. IU refers to integer unit of the processor, FPU to floating-point unit, and MMU to memory-management unit. The lower cost column refers to the least expensive options, listed as a Mono workstation in the third row from the bottom. The higher cost column refers to the more expensive options, listed as a Color workstation in the second row from the bottom. Note that about half the cost of the systems is in the DRAMs. Courtesy of Andy Bechtolsheim of Sun Microsystems, Inc.

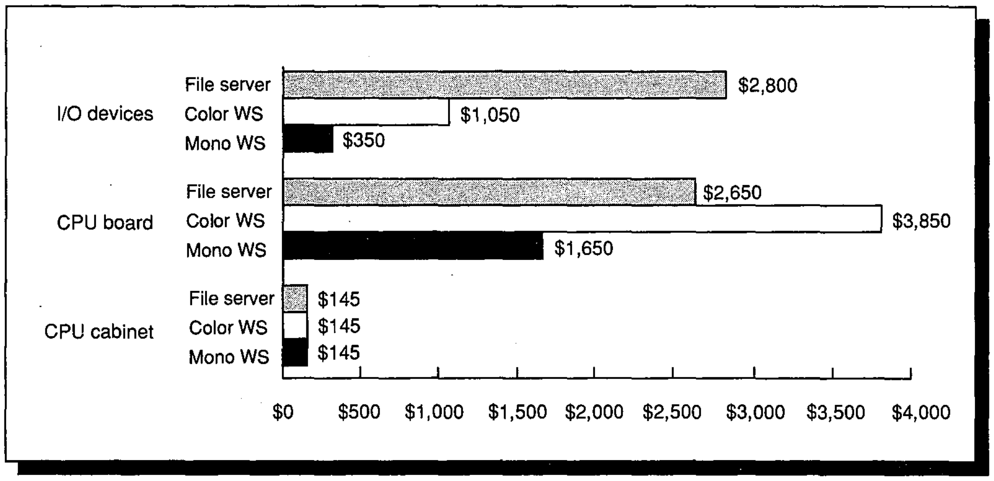


FIGURE 2.14 The costs of each machine in Figure 2.13 divided into the three main categories, assuming the lower cost estimate. Note that I/O devices and amount of memory account for major differences in costs.

The categories that make up price can be shown either as a tax on cost or as a percentage of the price. We will look at the information both ways. Figure 2.15 shows the increasing price of a product from left to right as we add each kind of overhead.

Direct costs refer to the costs directly related to making a product. These include labor costs, purchasing components, scrap (the leftover from yield), and warranty, which covers the costs of systems that fail at the customer's site during the warranty period. Direct cost typically adds 25% to 40% to component cost. Service or maintenance costs are not included because the customer typically pays those costs.

The next addition is called the *gross margin*, the company's overhead that cannot be billed directly to one product. This can be thought of as indirect cost. It includes the company's research and development (R&D), marketing, sales, manufacturing equipment maintenance, building rental, cost of financing, pretax profits, and taxes. When the component costs are multiplied by the direct cost and gross margin we reach the *average selling price*—ASP in the language of MBAs—the money that comes directly to the company for each product sold. The gross margin is typically 45% to 65% of the average selling price.

List price and average selling price are not the same. One reason for this is that companies offer volume discounts, lowering the average selling price. Also, if the product is to be sold in retail stores, as personal computers are, stores want to keep 40% of the list price for themselves. Thus, depending on the distribution system, the average selling price is typically 60% to 75% of the list price. The formula below ties the four terms together:

$$\text{List price} = \frac{\text{Cost} * (1 + \text{Direct costs})}{(1 - \text{Average discount}) * (1 - \text{Gross margin})}$$

Figure 2.16 demonstrates the abstract concepts of Figure 2.15 using dollars and cents by turning the costs of Figure 2.13 into prices. This is done using two business models. Model A assumes 25% (of cost) direct costs, 50% (of ASP) gross margin, and a 33% (of list price) average discount. Model B assumes 40% direct costs, 60% gross margin, and the average discount is dropped to 25%.

Pricing is sensitive to competition. A company striving for market share can therefore adjust to average discount or profits, but must live with its component cost and direct cost, plus the rest of the costs in the gross margin.

Many engineers are surprised to find that most companies spend only 8% to 15% of their income on R&D, which includes all engineering (except for manufacturing and field engineering). This is a well-established percentage that is reported in companies' annual reports and tabulated in national magazines, so this percentage is unlikely to change over time.

The information above suggests that a company uniformly applies fixed-overhead percentages to turn cost into price, and this is true for many companies. But another point of view is R&D should be considered an investment, and so an investment of 8% to 15% of income means every $1 spent on R&D must generate $7 to $13 in sales. This alternative point of view then suggests a different gross margin for each product depending on number sold and the size

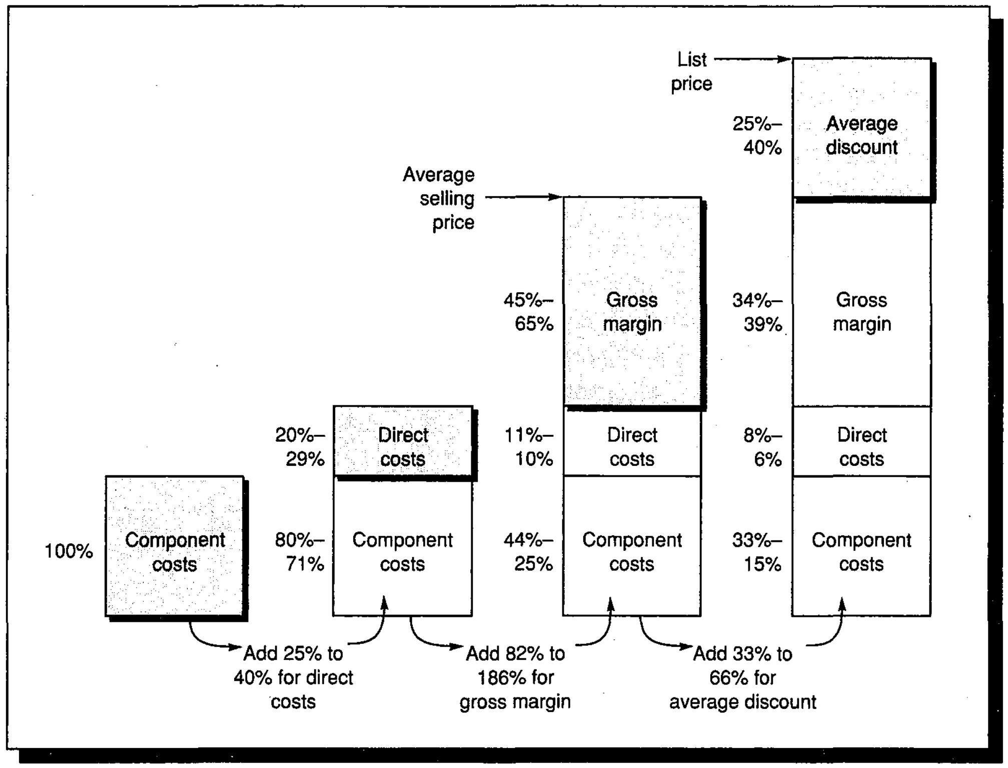


FIGURE 2.15 Starting with component costs, the price increases as we allow for direct costs, gross margin, and average discount, until we arrive at the list price. Each increase is shown along the bottom as a tax on the prior price. On the left of each column are shown the percentages of the new price for all elements.

	Model A	As % of costs	As % of list price	Model B	As % of costs	As % of list price
Component costs	$2,145	100%	27%	$2,145	100%	21%
Component costs + direct costs	$2,681	125%	33%	$3,003	140%	30%
Average selling price (adds gross margin)	$5,363	250%	67%	$7,508	350%	75%
List price	$8,044	375%	100%	$10,010	467%	100%

FIGURE 2.16 The diskless workstation in Figure 2.13 priced using two different business models. For every dollar of increased component cost the average selling price goes up between $2.50 and $3.50, and the list price increases between $3.75 and $4.67.

of the investment. Large expensive machines generally cost more to develop—a machine costing 10 times as much to manufacture may cost many times as much to develop. Since large expensive machines generally do not sell as well as small ones, the gross margin must be greater on the big machines for the company to maintain a profitable return on its investment. This investment model places large machines in double jeopardy—because there are fewer sold **and** they require larger R&D costs—and gives one explanation for a higher ratio of price to cost versus smaller machines.

2.4 Putting It All Together: Price/Performance of Three Machines

Having covered performance and costs, the next step is to measure performance of real programs on real machines and list the costs of those machines. Alas, costs are hard to come by so prices are used instead. We start with the more controversial half of price/performance.

Figure 2.17 lists the programs chosen by the authors for performance measurement in this book. Two of the programs have almost no floating-point operations, and one has a moderate amount of floating-point operations. All three programs have input, output, and options—what you would expect from real programs. Each program has, in fact, a large user community that cares how fast these programs run. (In measuring performance of machines we would like to have a larger sample, but we keep the limit at three throughout the book to make tables and graphs legible.)

Figure 2.18 shows the characteristics of three machines we measure, including the list price as tested and the relative performance as calculated by marketing.

Figure 2.19 (page 69) shows the CPU time and elapsed time measured for these programs. We include total times and several weighted averages, with the weights shown in parentheses. The first weighted arithmetic mean is assuming a workload of just the integer programs (GCC and TeX). The second is the weightings for a floating-point workload (Spice). The next three weighted means give three workloads for equal time spent on each program on one of the machines (see Figure 2.7 on page 51). The only means that are significantly different are the integer and floating-point means for VAXstation 2000. The rest of the means for each machine are within 10% of each other, as can be seen in Figure 2.20 on page 69, which plots the weighted means.

Program name	**Gnu C Compiler for 68000**	**Common TeX**	**Spice**
Version	1.26	2.9	2G6
Lines	79,409	23,037	18,307
Options	-O	'&latex/lplain'	transient analysis, 200 ps steps, for 40 ns
Input	i*.c	bit-set.tex, compiler. tex,...	digsr - digital shift register
Lines/bytes of input	28,009/373,688	10,992/698,914	233/1294
Lines/bytes of output	47,553/664,479	758/524,728	656/4172
% floating-point operations (on the DECstation 3100)	0.01%	0.05%	13.58%
Programming language	C	C	FORTRAN 66
Purpose	Publicly licensed, portable, optimizing C compiler	Document formatting	Computer-aided circuit analysis

FIGURE 2.17 Programs used in this book for performance measurements. The Gnu C compiler is a product of the Free Software Foundation and, for reasons not limited to its price, is preferred by some users over the compilers supplied by the manufacturer. Only 9,540 of the 79,409 lines are specific to the 68000, and versions exist for the VAX, SPARC, 88000, MIPS, and several other instruction sets. The input for GCC are the source files of the compiler that begin with the letter "i." Common TeX is a C version of the document-processing program originally written by Prof. Donald Knuth of Stanford. The input is a set of manual pages for the Stanford SUIF compiler. Spice is a computer-aided circuit-analysis package distributed by the University of California at Berkeley. (These programs and their inputs are available as part of the software package associated with this book. The Preface mentions how to get a copy.)

	VAXstation 2000	VAXstation 3100	DECstation 3100
Year of introduction	1987	1989	1989
Version of CPU/FPU	µVAX II	CVAX	MIPS R2000A/R2010
Clock rate	5 MHz	11.11 MHz	16.67 MHz
Memory size	4 MB	8 MB	8 MB
Cache size	none	1 KB on chip, 64-KB second level	128 KB (split 64-KB instruction and 64-KB data)
TLB size	8 entries fully associative	28 entries fully associative	64 entries fully associative
Base list price	$4,825	$7,950	$11,950
Optional equipment	19" monitor, extra 10 MB	(model 40) extra 8 MB	19" monitor, extra 8 MB
List price as tested	$15,425	$14,480	$17,950
Performance according to marketing	0.9 MIPS	3.0 MIPS	12 MIPS
Operating system	Ultrix 3.0	Ultrix 3.0	Ultrix 3.0
C compiler version	Ultrix and VMS	Ultrix and VMS	1.31
Options for C compiler	-O	-O	-O2 -Olimit 1060
C library	libc	libc	libc
FORTRAN 77 compiler version	fort (VMS)	fort (VMS)	1.31
Options for FORTRAN 77 compiler	-O	-O	-O2 -Olimit 1060
FORTRAN 77 library	lib*77	lib*77	lib*77

FIGURE 2.18 The three machines and software used to measure performance in Figure 2.19. These machines are all sold by Digital Equipment—in fact, the DECstation 3100 and VAXstation 3100 were announced the same day. All three are diskless workstations and run the same version of the UNIX operating system, called Ultrix. The VMS compilers ported to Ultrix were used for TeX and Spice on the VAXstations. We used the native Ultrix C compiler for GCC because GCC would not run using the VMS C compiler. The compilers for the DECstation 3100 are supplied by MIPS Computer Systems. (The "-Olimit 1060" option for the DECstation 3100 tells the compiler not to try to optimize procedures longer than 1060 lines.)

The bottom line for many computer customers is the price they pay for performance. This is graphically depicted in Figure 2.21 (page 70), where arithmetic means of CPU time are plotted against price of each machine.

	VAXstation 2000		VAXstation 3100		DECstation 3100	
	CPU time	Elapsed time	CPU time	Elapsed time	CPU time	Elapsed time
Gnu C Compiler for 68000	985	1108	291	327	90	159
Common TeX	1264	1304	449	479	95	137
Spice	958	973	352	395	94	132
Arithmetic mean	1069	1128	364	400	93	143
Weighted AM—integer only (50% GCC, 50% TeX, 0% Spice)	1125	1206	370	403	93	148
Weighted AM—floating point only (0% GCC, 0% TeX, 100% Spice)	958	973	352	395	94	132
Weighted AM—equal CPU time on V2000 (35.6% GCC, 27.8% TeX, 36.6% Spice)	1053	1113	357	394	93	143
Weighted AM—equal CPU time on V3100 (40.4% GCC, 26.2% TeX, 33.4% Spice)	1049	1114	353	390	93	144
Weighted AM—equal CPU time on D3100 (34.4% GCC, 32.6% TeX, 33.0% Spice)	1067	1127	363	399	93	143

FIGURE 2.19 Performance of the programs in Figure 2.17 on the machines in Figure 2.18. The weightings correspond to integer programs only, and then equal CPU time running on each of the three machines. For example, if the mix of the three programs were proportionate to the weightings in the row "equal CPU time on D3100," the DECstation 3100 would spend a third of its CPU time running Gnu C Compiler, a third running TeX, and a third running Spice. The actual weightings are in parentheses, calculated as shown in Figure 2.7 on page 51.

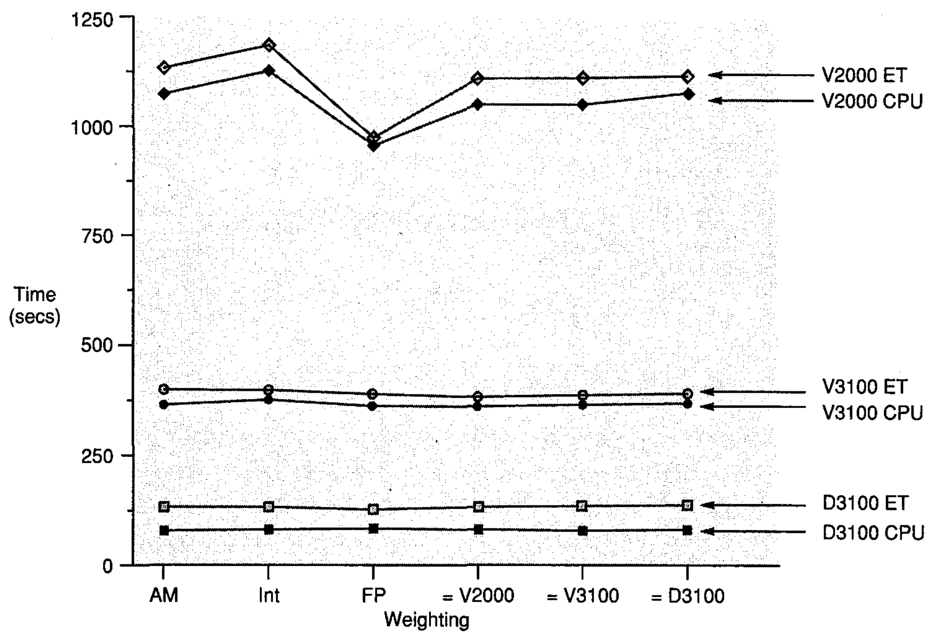


FIGURE 2.20 Plot of means of CPU time and elapsed time from Figure 2.19.

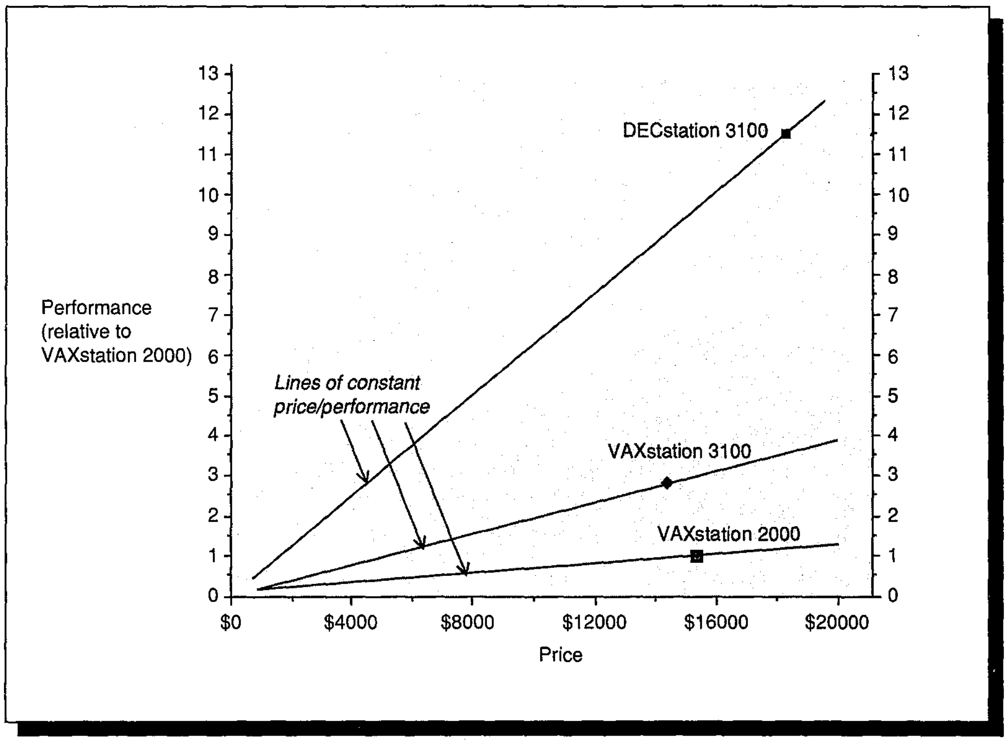


FIGURE 2.21 Price versus performance of VAXstation 2000, VAXstation 3100, and DECstation 3100 for Gnu C Compiler, TeX, and Spice. Based on Figures 2.18–2.19, this figure plots the list price **as tested** of a machine versus performance, where performance is the inverse of the ratio to the arithmetic mean of CPU time on a VAXstation 2000. The lines through the three machines show lines of constant price/performance. For example, a machine at the right end of the VAXstation 3100 line costs $20,000. Since it would cost 30% more, it must have 30% more performance than the VAXstation 3100 to have the same price performance.

2.5 Fallacies and Pitfalls

Cost/performance fallacies and pitfalls have ensnared many computer architects, including ourselves. For this reason, more space is devoted to the warning section in this chapter than in other chapters of this text.

Fallacy: Hardware-independent metrics predict performance.

Because accurately predicting performance is so difficult, the folklore of computer design is filled with suggested shortcuts. These are frequently employed when comparing different instruction sets, especially instruction sets that are paper designs.

One such shortcut is “Code Size = Speed,” or the architecture with the smallest program is fastest. Static code size is important when memory space is at a premium, but it is not the same as performance. As we shall see in Chapter 6, larger programs composed of instructions that are easily fetched, decoded, and executed may run faster than machines with extremely compact instructions that are difficult to decode. "Code Size=Speed" is especially popular with compiler writers, for while it can be difficult to decide if one code sequence is faster than another, it is easy to see which is shorter.

Evidence of the "Code Size=Speed" fallacy can be found on the cover of the book *Assessing the Speed of Algol 60* in Figure 2.22. The CDC 6600's programs are over twice as big, yet the CDC machine runs Algol 60 programs almost six times **faster** than the Burroughs B5500, a machine designed for Algol 60.

Pitfall: Comparing computers using only one or two of three performance metrics: clock rate, CPI, and instruction count.

The CPU performance equation shows why this can mislead. One example is that given in Figure 2.22: The CDC 6600 executes almost 50% more instructions than the Burroughs B5500, yet it is 550% faster. Another example comes from increasing the clock rate so that some instructions execute fast—sometimes called *peak performance*—but making design decisions that also result in a high overall CPI that offsets the clock rate advantage. The Intergraph Clipper C100 has a clock rate of 33 MHz and a peak performance of 33 native MIPS. Yet the Sun 4/280, with half the clock rate and half the peak native MIPS rating, runs programs faster [Hollingsworth, Sachs, and Smith 1989, 215]. Since the Clipper's instruction count is about the same as Sun's, the former machine's CPI must be more than double that of the latter.

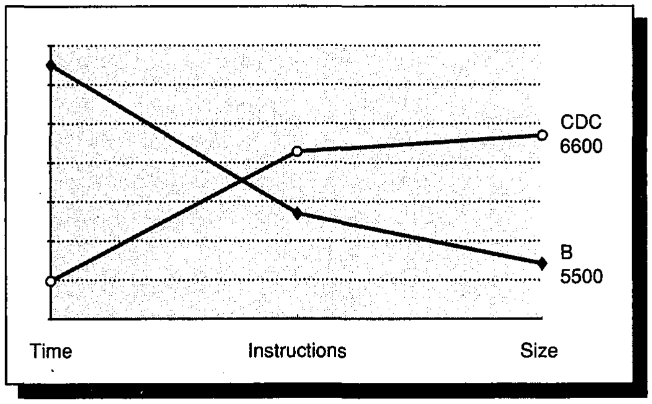


FIGURE 2.22 As found on the cover of *Assessing the Speed of Algol 60* by B. A. Wichmann, the graph shows relative execution time, instruction count, and code size of programs written in Algol 60 for the Burroughs B5500 and the CDC 6600. The results are normalized to a reference machine, with a higher number being worse. This book had a profound effect on one of the authors (DP). Seymour Cray, the designer of the CDC 6600, may not even have known of the existence of this programming language, while Robert Barton, architect of the B5500, designed the instruction set specifically for Algol 60. While the CDC 6600 executes 50% more instructions and has 220% larger code, the CDC 6600 is 550% faster than the B5500.

Fallacy: When calculating relative MIPS, the versions of the compiler and operating system of the reference machine make little difference.

Figure 2.19 shows the VAXstation 2000 taking 958 seconds of CPU time when running Spice with a standard input. Instead of Ultrix 3.0 with the VMS F77 compiler, many systems use Ultrix 3.0 with the standard UNIX F77 compiler. This compiler increases Spice CPU time to 1604 seconds. Using the standard evaluation of 0.9 relative MIPS for the VAXstation 2000, the DECstation 3100 is either 11 or 19 relative MIPS for Spice depending on the compiler of the reference machine.

Fallacy: CPI can be calculated from the instruction mix and the execution times of instructions found in the manual.

Current machines are too complicated to estimate performance from a manual. For example, in Figure 2.19 Spice takes 94 seconds of CPU time on the DECstation 3100. If we calculate the CPI from the DECstation 3100 manual—ignoring memory hierarchy and pipelining inefficiencies for this Spice instruction mix—we get 1.41 for the CPI. When multiplied by the instruction count and clock rate we get only 73 seconds. The missing 25% of CPU time is due to the estimate of CPI based only on the manual. The actual measured value, including all memory-system inefficiencies, is 1.87 CPI.

Pitfall: Summarizing performance by translating throughput into execution time.

The SPEC benchmarks report performance by measuring the elapsed time of each of 10 benchmarks. The sole dual processor workstation in the initial benchmark report ran these benchmarks no faster since the compilers didn't automatically parallelize the code across the two processors. The benchmarker's solution was to run a copy of each benchmark on each processor and record elapsed time for the two copies. This would not have helped if the SPEC release had only summarized performance using elapsed times, since the times were slower due to interference of the processors on memory accesses. The loophole was the initial SPEC release reported geometric means of performance relative to a VAX-11/780 in addition to elapsed times, and these means are used to graph the results. This innovative benchmarker interpreted ratio of performance to a VAX-11/780 as a **throughput** measure, so doubled his measured ratios to the VAX! Figure 2.23 shows the plots as found in the report for the uniprocessor and the multiprocessor. This technique almost doubles the geometric means of ratios, suggesting the mistaken conclusion that a computer that runs two copies of a program simultaneously has the same response time to a user as a computer that runs a single program in half the time.

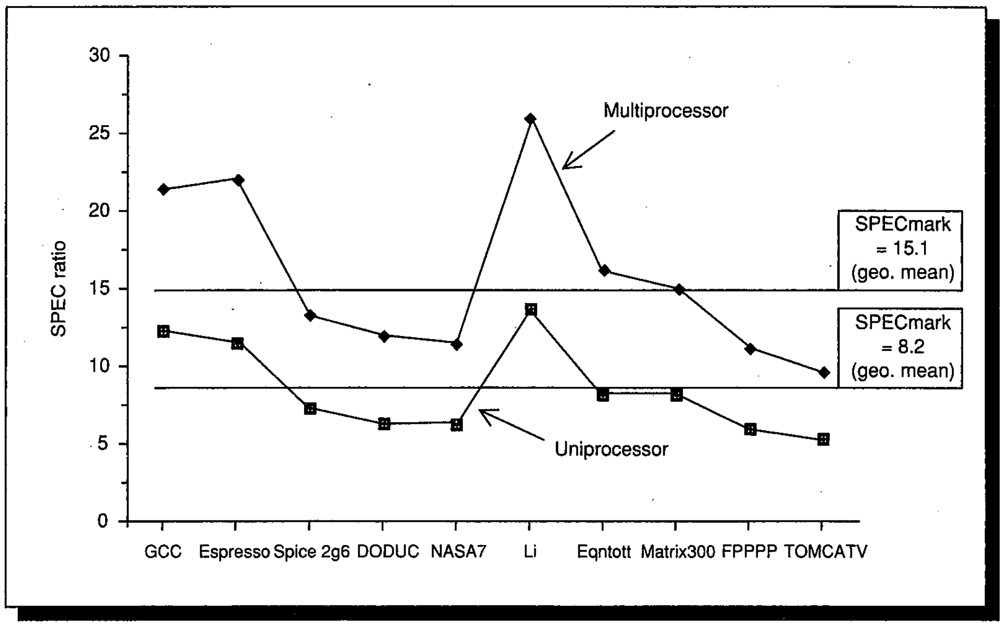


FIGURE 2.23 Performance of uniprocessor and multiprocessor as reported in SPEC Benchmark Press Release. Performance is plotted relative to a VAX-11/780. The ratio for the multiprocessor is really the ratio of elapsed time multiplied by the number of processors.

Fallacy: Synthetic benchmarks predict performance.

The best known examples of such benchmarks are Whetstone and Dhrystone. These are not real programs and, as such, may not reflect program behavior for factors not measured. Compiler and hardware optimizations can artificially inflate performance of these benchmarks but not of real programs. The other side of the coin is that because these benchmarks are not natural programs, they don't reward optimizations of behavior that occur in real programs. Here are some examples:

- Optimizing compilers can discard 25% of the Dhrystone code; examples include loops that are only executed once, making the loop overhead instructions unnecessary. To address these problems the authors of the benchmark "require" both optimized and unoptimized code to be reported. In addition, they "forbid" the practice of inline-procedure expansion optimization. (Dhrystone's simple procedure structure allows elimination of all procedure calls at almost no increase in code size; see Figure 2.5 on page 47.)

- All Whetstone floating-point loops make optimizations via vectorization essentially useless. (The program was written before computers with vector instructions were popular. See Chapter 7.)

- Dhrystone has a long history of optimizations that skew its performance. The most recent comes from a C compiler that appears to include optimizations just for Dhrystone (Figure 2.5). If the proper option flag is set at compile time, the compiler turns the portion of the C version of this benchmark that copies a variable length string of bytes (terminated by an end-of-string symbol) into a loop that transfers a fixed number of words assuming the source and destination of the string is word-aligned in memory. Although it is estimated that between 99.70% to 99.98% of typical string copies could **not** use this optimization, this single change can make a 20% to 30% improvement in overall performance—if Dhrystone is your measure.

- Compilers can optimize a key piece of the Whetstone loop by noting the relationship between square root and exponential, even though this is very unlikely to occur in real programs. For example, one key loop contains the following FORTRAN code (see Figure 2.4 on page 46):

  ```
  X = SQRT( EXP(ALOG(X)/T1) )
  ```

  It could be compiled as if it were

  ```
  X = EXP( ALOG(X)/(2*T1) )
  ```

  since

  $$\text{SQRT(EXP(X))} = \sqrt[2]{e^X} = e^{X/2} = \text{EXP(X/2)}$$

It would be surprising if such optimizations were ever invoked except in this synthetic benchmark. (Yet one reviewer of this book found several compilers that performed this optimization!) This single change converts all calls to the square root function in Whetstone into multiplies by 2, surely improving performance—if Whetstone is your measure.

Fallacy: Peak performance tracks observed performance.

One definition of peak performance is performance a machine is "guaranteed not to exceed." The gap between peak performance and observed performance is typically a factor of 10 or more in supercomputers. (See Chapter 7 on vectors for an explanation.) Since the gap is so large, peak performance is not useful in predicting observed performance unless the workload consists of small programs that normally operate close to the peak.

As an example of this fallacy, a small code segment using long vectors ran on the Hitachi S810/20 at 236 MFLOPS and on the CRAY X-MP at 115 MFLOPS. Although this suggests the S810 is 105% faster than the X-MP, the X-MP runs a

	CRAY X-MP	Hitachi S810/20	Performance
A(i)=B(i)*C(i)+D(i)*E(i) (vector length 1000 done 100,000 times)	2.6 secs	1.3 secs	Hitachi 105% faster
Vectorized FFT (vector lengths 64, 32,...,2)	3.9 secs	7.7 secs	CRAY 97% faster

FIGURE 2.24 Measurements of peak performance and actual performance for the Hitachi S810/20 and the CRAY X-MP. From Lubeck, Moore, and Mendez [1985, 18-20]. Also see the pitfall in the Fallacies and Pitfalls section of Chapter 7.

Machine	Peak MFLOPS rating	Harmonic mean MFLOPS of the Perfect benchmarks	Percent of peak MFLOPS
CRAY X-MP/416	940	14.8	1%
IBM 3090-600S	800	8.3	1%
NEC SX/2	1300	16.6	1%

FIGURE 2.25 Peak performance and harmonic mean of actual performance for the Perfect Benchmarks. These results are for the programs run unmodified. When tuned by hand performance of the three machines moves to 24.4, 11.3, and 18.3 MFLOPS, respectively. This is still 2% or less of peak performance.

program with more typical vector lengths 97% faster than the S810. These data are shown in Figure 2.24.

Another good example comes from a benchmark suite called the Perfect Club (see page 80). Figure 2.25 shows the peak MFLOPS rating, harmonic mean of the MFLOPS achieved for 12 real programs, and the percentage of peak performance for three large computers. They achieve only 1% of peak performance.

While the use of peak performance has been rampant in the supercomputer business, recently this metric spread to microprocessor manufacturers. For example, in 1989 a microprocessor was announced as having the performance of 150 million "operations" per second ("MOPS"). The only way this machine can achieve this performance is for one integer instruction and one floating-point instruction to be executed each clock cycle **and** for the floating-point instruction to perform both a multiply operation and an add. For this peak performance to predict observed performance a real program would have to have 66% of its operations be floating point and no losses for the memory system or pipelining. In contrast to claims, typical measured performance of this microprocessor is under 30 "MOPS."

The authors hope that peak performance can be quarantined to the supercomputer industry and eventually eradicated from that domain; but in any case, approaching supercomputer performance is not an excuse for adopting dubious supercomputer marketing habits.

2.6 Concluding Remarks

Having a standard of performance reporting in computer science journals as high as that in car magazines would be an improvement in current practice. Hopefully, that will be the case as the industry moves toward basing performance evaluation on real programs. Perhaps arguments about performance will even subside.

Computer designs will always be measured by cost and performance, and finding the best balance will always be the art of computer design. As long as technology continues to rapidly improve, the alternatives will look like the curves in Figure 2.26. Once a designer selects a technology, he can't achieve some performance levels no matter how much he pays and, conversely, no matter how much he cuts performance there is a limit to how low the cost can go. It would be better in either case to change technologies.

As a final remark, the number of machines sold is not always the best measure of cost/performance of computers, nor does cost/performance always predict number sold. Marketing is very important to sales. It is easier, however, to market a machine with better cost/performance. Even businesses with high gross margins need to be sensitive to cost/performance, otherwise the company cannot lower prices when faced with stiff competition. Unless you go into marketing, your job is to improve cost/performance!

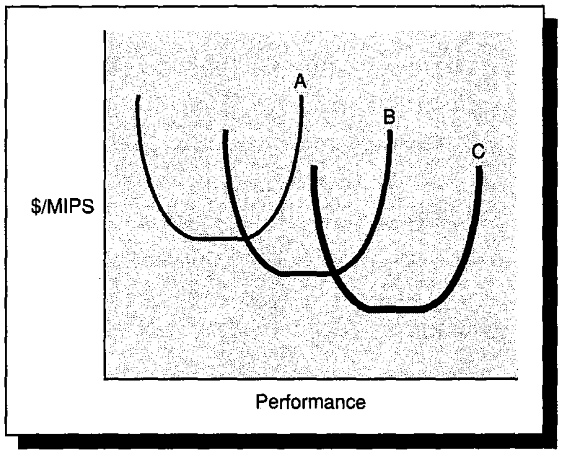


FIGURE 2.26 The cost per MIPS goes up on the y axis, and system performance increases on the x axis. A, B, and C are three technologies, let us say three different semiconductor technologies, to build a processor. Designs in the flat part of the curves can offer varieties of performance at the same cost/performance. If performance goals are too high for a technology it becomes very expensive, and too cheap a design makes the performance too low (cost per MIPS expensive for low MIPS). At either extreme it is better to switch technologies.

2.7 Historical Perspective and References

The anticipated degree of overlapping, buffering, and queuing in the [IBM 360] Model 85 [first computer with a cache] appeared to largely invalidate conventional performance measures based on instruction mixes and program kernels.

Conti, Gibson, and Pitkowsky [1968]

In the earliest days of computing, designers set performance goals—ENIAC was to be 1000 times faster than the Harvard Mark I, and the IBM Stretch (7030) was to be 100 times faster than the fastest machine in existence. What wasn't clear, though, was how this performance was to be measured. In looking back over the years, it is a consistent theme that each generation of computers obsoletes the performance evaluation techniques of the prior generation.

The original measure of performance was time to perform an individual operation, such as addition. Since most instructions took the same execution time, the timing of one gave insight into the others. As the execution times of instructions in a machine became more diverse, however, the time for one operation was no longer useful for comparisons. To take these differences into account, an *instruction mix* was calculated by measuring the relative frequency of instructions in a computer across many programs. The Gibson mix [1970] was an early popular instruction mix. Multiplying the time for each instruction times its weight in the mix gave the user the *average instruction execution time.* (If measured in clock cycles, average instruction execution time is the same as average CPI.) Since instruction sets were similar, this was a more accurate comparison than add times. From average instruction execution time, then, it was only a small step to MIPS (as we have seen, the one is the inverse of the other). MIPS has the virtue of being easy for the layman to understand, hence its popularity.

As CPUs became more sophisticated and relied on memory hierarchies and pipelining, there was no longer a single execution time per instruction; MIPS could not be calculated from the mix and the manual. The next step was benchmarking using kernels and synthetic programs. Curnow and Wichmann [1976] created the Whetstone synthetic program by measuring scientific programs written in Algol 60. This program was converted to FORTRAN and was widely used to characterize scientific program performance. An effort with similar goals to Whetstone, the Livermore FORTRAN Kernels, was made by McMahon [1986] and researchers at Lawrence Livermore Laboratory in an attempt to establish a benchmark for supercomputers. These kernels, however, consisted of loops from real programs.

The notion of relative MIPS came along as a way to resuscitate the easily understandable MIPS rating. When the VAX-11/780 was ready for announcement in 1977, DEC ran small benchmarks that were also run on an IBM 370/158. IBM marketing referred to the 370/158 as a 1-MIPS computer, and since the programs ran at the same speed, DEC marketing called the VAX-11/780 a 1-MIPS computer. (Note that this rating included the effectiveness of the compilers on both machines at the moment the comparison was made.) The popularity of the VAX-11/780 made it a popular reference machine for relative MIPS, especially since relative MIPS for a 1-MIPS computer is easy to calculate: If a machine was five times faster than the VAX-11/780, for that benchmark its rating would be 5 relative MIPS. The 1-MIPS rating was unquestioned for four years until Joel Emer of DEC measured the VAX-11/780 under a time-sharing load. He found that the VAX-11/780 native MIPS rating was 0.5. Subsequent VAXes that run 3 native MIPS for some benchmarks were therefore called 6-MIPS machines because they run 6 times faster than the VAX-11/780.

Although other companies followed this confusing practice, pundits have redefined MIPS as "Meaningless Indication of Processor Speed" or "Meaningless Indoctrination by Pushy Salespersons." At the present time, the most common meaning of MIPS in marketing literature is not native MIPS but "number of times faster than the VAX-11/780" and frequently includes floating-point programs as well. The exception is IBM, which defines MIPS relative to the "processing capacity" of an IBM 370/158, presumably running large system benchmarks (see Henly and McNutt, [1989, 5]). In the late 1980s DEC began using *VAX units of performance* (VUP), meaning ratio to VAX-11/780, so 6 relative MIPS became 6 VUPs.

The 1970s and 1980s marked the growth of the supercomputer industry, which was defined by high performance on floating-point–intensive programs. Average instruction time and MIPS were clearly inappropriate metrics for this industry, and hence the invention of MFLOPS. Unfortunately customers quickly forget the program used for the rating, and marketing groups decided to start quoting peak MFLOPS in the supercomputer performance wars.

A variety of means have been proposed for averaging performance. McMahon [1986] recommends the harmonic mean for averaging MFLOPS. Flemming and Wallace [1986] assert the merits of the geometric mean in general. Smith's reply [1988] to their article gives cogent arguments for arithmetic means of time and harmonic means of rates. (Smith's arguments are the ones followed in "Comparing and Summarizing Performance" under Section 2.2, above.)

As the distinction between architecture and implementation pervaded the computing community (see Chapter 1), the question arose whether the performance of an architecture itself could be evaluated, as opposed to an implementation of the architecture. A study of this question performed at Carnegie-Mellon University is summarized in Fuller and Burr [1977]. Three quantitative measures were invented to scrutinize architectures:

S	Number of bytes for program code
M	Number of bytes transferred between memory and the CPU during program execution for code and data (S measures size of code at compile time, while M is memory traffic during program execution.)

R Number of bytes transferred between registers in a canonical model of a CPU

Once these measures were taken, a weighting factor was applied to them to determine which architecture was "best." Yet there has been no formal effort to see if these measures really matter—do the implementations of an architecture with superior S, M, and R measures outperform implementations of lesser architectures? The VAX architecture was designed in the height of popularity of the Carnegie-Mellon study, and by those measures it does very well. Architectures created since 1985, however, have poorer measures than the VAX, yet their implementations do well against the VAX implementations. For example, Figure 2.27 compares S, M, and CPU time for the VAXstation 3100, which uses the VAX instruction set, and the DECstation 3100, which doesn't. The DECstation 3100 is 200% to almost 400% faster even though its S measure is 35% to 70% worse and its M measure is 5% to 15% worse. The effort to evaluate architecture independent of implementation was a valiant one, it seems, if not a successful one.

	S (code size in bytes)		**M (megabytes code + data transferred)**		**CPU Time (in seconds)**	
	VAX 3100	**DEC 3100**	**VAX 3100**	**DEC 3100**	**VAX 3100**	**DEC 3100**
Gnu C Compiler	409,600	688,128	18	21	291	90
Common TeX	158,720	217,088	67	78	449	95
Spice	223,232	372,736	99	106	352	94

FIGURE 2.27 Code size and CPU time of the VAXstation 3100 and DECstation 3100 for Gnu C Compiler, TeX, and Spice. The programs and machines are described in Figures 2.17 and 2.18. Both machines were announced the same day by the same company and run the same operating system. The difference is in the instruction sets, compilers, clock cycle time, and organization. The *M* measure comes from Figure 3.33 (page 123) for smaller inputs than those in Figure 2.17 (page 67), but the relative performance is unchanged. Code size includes libraries.

A promising development in performance evaluation is the formation of the System Performance Evaluation Cooperative, or SPEC, group in 1988. SPEC contains representatives of many computer companies—the founders being Apollo/Hewlett-Packard, DEC, MIPS, and Sun—who have agreed on a set of real programs and inputs that all will run. It is worth noting that SPEC couldn't have happened before portable operating systems and the popularity of high-level languages. Now compilers, too, are accepted as a proper part of the performance of computer systems and must be measured in any evaluation. (See Exercises 2.8–2.10 on pages 83–84 for more on SPEC benchmarks.)

History teaches us that while the SPEC effort is useful with current computers, it will not be able to meet the needs of the next generation. An effort similar to SPEC, called the Perfect Club, binds together universities and companies interested in parallel computation [Berry et al. 1988]. Rather than being forced to run the existing sequential programs' code, the Perfect Club includes both programs and algorithms, and allows members to write new programs in new languages, which may be needed for the new architectures. Perfect Club members may also suggest new algorithms to solve important problems.

While papers on performance are plentiful, little is available on computer cost. Fuller [1976] wrote the first paper comparing price and performance for the Annual International Symposium on Computer Architecture. This was also the last price/performance paper at this conference. Phister's book [1979] on costs of computers is exhaustive, and Bell, Mudge, and McNamara [1978] describe the computer construction process from DEC's perspective. In contrast, there is a good deal of information on die yield. Strapper [1989] surveys the history of yield modeling, while technical details on the die-yield model used in this chapter are found in Strapper, Armstrong, and Saji [1983].

References

BELL, C. G., J. C. MUDGE, AND J. E. MCNAMARA [1978]. *A DEC View of Computer Engineering*, Digital Press, Bedford, Mass.

BERRY, M., D. CHEN, P. KOSS, D. KUCK [1988]. "The Perfect Club benchmarks: Effective performance evaluation of supercomputers," CSRD Report No. 827 (November), Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign.

CONTI, C. J., D. H. GIBSON, AND S. H. PITKOWSLI [1968]. "Structural aspects of the System/360 Model 85:I general organization," *IBM Systems J.* 7:1, 2–11.

CURNOW, H. J. AND B. A. WICHMANN [1976]. "A synthetic benchmark," *The Computer J.* 19:1.

FLEMMING, P. J. AND J. J. WALLACE [1986]. "How not to lie with statistics: The correct way to summarize benchmarks results," *Comm. ACM* 29:3 (March) 218–221.

FULLER, S. H. [1976]. "Price/performance comparison of C.mmp and the PDP-11," *Proc. Third Annual Symposium on Computer Architecture* (Texas, January 19–21), 197–202.

FULLER, S. H. AND W. E. BURR [1977]. "Measurement and evaluation of alternative computer architectures," *Computer* 10:10 (October) 24–35.

GIBSON, J. C. [1970]. "The Gibson mix," Rep. TR. 00.2043, IBM Systems Development Division, Poughkeepsie, N.Y. (Research done in 1959.)

HENLY, M. AND B. MCNUTT [1989]. "DASD I/O characteristics: A comparison of MVS to VM," Tech. Rep. TR 02.1550 (May), IBM, General Products Division, San Jose, Calif.

HOLLINGSWORTH, W., H. SACHS AND A. J. SMITH [1989]. "The Clipper processor: Instruction set architecture and implementation," *Comm. ACM* 32:2 (February), 200–219.

LUBECK, O., J. MOORE, AND R. MENDEZ [1985]. "A benchmark comparison of three supercomputers: Fujitsu VP-200, Hitachi S810/20, and Cray X-MP/2," *Computer* 18:12 (December) 10–24.

MCMAHON, F. M. [1986]. "The Livermore FORTRAN kernels: A computer test of numerical performance range," Tech. Rep. UCRL-55745, Lawrence Livermore National Laboratory, Univ. of California, Livermore, Calif. (December).

PHISTER, M., JR. [1979]. *Data Processing Technology and Economics*, 2nd ed., Digital Press and Santa Monica Publishing Company.

SMITH, J. E. [1988]. "Characterizing computer performance with a single number," *Comm. ACM* 31:10 (October) 1202–1206.

SPEC [1989]. "SPEC Benchmark Suite Release 1.0," October 2, 1989.

STRAPPER, C. H. [1989]. "Fact and fiction in yield modelling," Special Issue of the *Microelectronics Journal* entitled *Microelectronics into the Nineties*, Oxford, UK; Elsevier (May).

STRAPPER, C. H., F. H. ARMSTRONG, AND K. SAJI, [1983]. "Integrated circuit yield statistics," *Proc. IEEE* 71:4 (April) 453–470.

WEICKER, R. P. [1984]. "Dhrystone: A synthetic systems programming benchmark," *Comm. ACM* 27:10 (October) 1013–1030.

WICHMANN, B. A. [1973]. *Algol 60 Compilation and Assessment*, Academic Press, New York.

EXERCISES

2.1 [20] <2.2> After graduating, you are asked to become the lead computer designer. Your study of usage of high-level–language constructs suggests that procedure calls are one of the most expensive operations. You have invented a scheme that reduces the loads and stores normally associated with procedure calls and returns. The first thing you do is run some experiments with and without this optimization.Your experiments use the same state-of-the-art optimizing compiler that will be used with either version of computer.

Your experiments reveal the following information:

- The clock cycle time of the unoptimized version is 5% faster.
- 30% of the instructions in the nonoptimized version are loads or stores.
- The optimized version executes 1/3 fewer loads and stores than the nonoptimized version. For all other instructions the dynamic execution counts are unchanged.
- All instructions (including load and store) take one clock cycle.

Which is faster? Justify your decision quantitatively.

2.2 [15/15/10] <2.2> Assume the two programs in Figure 2.6 on page 49 each execute 100,000,000 floating-point operations during execution.

a. [15] Calculate the (native) MFLOPS rating of each program.

b. [15] Calculate the arithmetic, geometric, and harmonic mean (native) MFLOPS for each machine.

c. [10] Which of the three means matches the relative performance of total execution time?

Questions 2.3–2.7 require the following information.

The Whetstone benchmark contains 79,550 floating-point operations, not including the floating-point operations performed in each call to the following functions:

- arctangent, invoked 640 times
- sine, invoked 640 times
- cosine, invoked 1920 times
- square root, invoked 930 times
- exponential, invoked 930 times
- and logarithm, invoked 930 times

The basic operations for a single iteration (not including floating-point operations to perform the above functions) are broken down as follows:

Add	37,530
Subtract	3,520
Multiply	22,900
Divide	11,400
Convert integer to fp	4,200
TOTAL	79,550

The total number of floating-point operations for a single iteration can also be calculated by including the floating-point operations needed to perform the functions arctangent, sine, cosine, square root, exponential, and logarithm:

Add	82,014
Subtract	8,229
Multiply	73,220
Divide	21,399
Convert integer to fp	6,006
Compare	4,710
TOTAL	195,578

Whetstone was run on a Sun 3/75 using the F77 compiler with optimization turned on. The Sun 3/75 is based on a Motorola 68020 running at 16.67 MHz, and it includes a floating-point coprocessor. (Assume the coprocessor does not include arctangent, sine, cosine, square root, exponential, and logarithm as instructions.) The Sun compiler allows the floating-point to be calculated with the coprocessor or using software routines, depending on compiler flags. A single iteration of Whetstone took 1.08 seconds using the coprocessor and 13.6 seconds using software. Assume that the CPI using the coprocessor was measured to be 10 while the CPI using software was measured to be 6.

2.3 [15] <2.2> What is the (native) MIPS rating for both runs?

2.4 [15] <2.2> What is the **total** number of instructions executed for both runs?

2.5 [8] <2.2> On the average, how many integer instructions does it take to perform each floating-point operation in software?

2.6 [18] <2.2> What is the native and normalized MFLOPS for the Sun 3/75 with the floating-point coprocessor running Whetstone? (Assume convert counts as a single floating-point operation and use Figure 2.3 for normalized operations.)

2.7 [20] <2.2> Figure 2.3 on page 43 suggests how many floating-point operations it takes to perform the six functions above (arctangent, sine, and so on). From the data above you can calculate the average number of floating-point operations per function. What is the ratio between the estimates in Figure 2.3 and the floating-point operation measurements for the Sun 3? Assume the coprocessor implements only Add, Subtract, Multiply, Divide, and Convert.

Questions 2.8–2.10 require the information in Figure 2.28.

The SPEC Benchmark Release 1.0 Summary [SPEC 89] lists performance as shown in Figure 2.28.

Program	VAX-11/780	DECstation 3100		Delta Series 8608		SPARCstation 1	
Name	Time	Time	Ratio	Time	Ratio	Time	Ratio
GCC	1482	145	10.2	193	7.7	138.9	10.7
Espresso	2266	194	11.7	197	11.5	254.0	8.9
Spice 2g6	23951	2500	9.6	3350	7.1	2875.5	8.3
DODUC	1863	208	9.0	295	6.3	374.1	5.0
NASA7	20093	1646	12.2	3187	6.3	2308.2	8.7
Li	6206	480	12.9	458	13.6	689.5	9.0
Eqntott	1101	99	11.1	129	8.5	113.5	9.7
Matrix300	4525	749	6.0	520	8.7	409.3	11.1
FPPPP	3038	292	10.4	488	6.2	387.2	7.8
TOMCATV	2649	260	10.2	509	5.2	469.8	5.6
Geometric mean	3867.7	381.4	10.1	496.5	7.8	468.5	8.3

FIGURE 2.28 SPEC performance summary 1.0. The four integer programs are GCC, Espresso, Li, and Eqntott, with the rest relying on floating-point hardware.The SPEC report does not describe the version of the compilers or operating system used for the VAX-11/780. The DECstation 3100 is described in Figure 2.18 on page 68. The Motorola Delta Series 8608 uses a 20-MHz MC88100, 16-KB instruction cache, and 16-KB data cache using two M88200s (see Exercise 8.6 in Chapter 8), the Motorola Sys. V/88 R32V1 operating system, the C88000 1.8.4m14 C compiler, and the Absoft SysV88 2.0a4 FORTRAN compiler. The SPARCstation 1 uses a 20-MHz MB8909 integer unit and 20-MHz WTL3170 floating-point unit, a 64-KB unified cache, SunOS 4.0.3c operating system and C compiler, and Sun 1.2 FORTRAN compiler. The size of main memory in these three machines is 16 MB.

2.8 [12/15] <2.2> Compare the relative performance using total execution times for the 10 programs versus using geometric means of ratios of the speed of the VAX-11/780.

a. [12] How do the results differ?

b [15] Compare the geometric mean of the ratios of the four integer programs (GCC, Espresso, Li, and Eqntott) versus the total execution time for these four programs. How do the results differ from each other and from the summaries of all ten programs?

2.9 [15/20/12/10] <2.2> Now let's compare performance using weighted arithmetic means.

a. [15] Calculate the weights for a workload so that running times on the VAX-11/780 will be equal for each of the ten programs (see Figure 2.7 on page 51).

b. [20] Using those weights, calculate the weighted arithmetic means of the execution times of the ten programs.

c. [12] Calculate the ratio of the weighted means of the VAX execution times to the weighted means for the other machines.

d. [10] How do the geometric means of ratios and the ratios of weighted arithmetic means of execution times differ in summarizing relative performance?

2.10 [Discussion] <2.2> What is an interpretation of the geometric means of execution times? What do you think are the advantages and disadvantages of using total execution times versus weighted arithmetic means of execution times using equal running time on the VAX-11/780 versus geometric means of ratios of speed to the VAX-11/780?

Questions 2.11–2.12 require the information in Figure 2.29.

Microprocessor	Size (cm)	Pins	Package	Clock rate	List price	Year available
Cypress CY7C601	0.8 × 0.7	207	Ceramic PGA	33 MHz	$500	1988
Intel 80486	1.6 × 1.0	168	Ceramic PGA	33 MHz	$950	1989
Intel 860	1.2 × 1.2	168	Ceramic PGA	33 MHz	$750	1989
MIPS R3000	0.8 × 0.9	144	Ceramic PGA	25 MHz	$300	1988
Motorola 88100	0.9 × 0.9	169	Ceramic PGA	25 MHz	$695	1989

FIGURE 2.29 Characteristics of microprocessors. List prices were quoted as of 7/15/89 at quantity 1000 purchases.

2.11 [15] <2.3> Pick the largest and smallest microprocessors from Figure 2.29, and use the values found in Figure 2.11 (page 62) for yield parameters. How many good chips do you get per wafer?

2.12 [15/10/10/15/15] <2.3> Let's calculate costs and prices of the largest and smallest microprocessors from Figure 2.29. Use the assumptions on manufacturing found in Figure 2.11 (page 62) unless specifically mentioned otherwise.

a. [15] There are wide differences in defect densities between semiconductor manufacturers. What are the costs of untested dies assuming: (1) 2 defects per square cm; and (2) 1 defect per square cm.

b. [10] Assume that testing costs $150 per hour and the smaller chip takes 10 seconds to test and the larger chip takes 15 seconds, what is the cost of testing each die?

c. [10] Making the assumptions on packaging in Section 2.3, what is the cost of packaging and burn-in?

d. [15] What is the final cost?

e. [15] Given the list price and the calculated cost from the questions above, calculate the gross margin. Assume the direct cost is 40% and average selling discount is 33%. What percentage of the average selling price is the gross margin for both chips?

2.13–2.14 A few companies claim they are doing so well that the defect density is vanishing as the reason for die failures, making wafer yield responsible for the vast majority. For example, Gordon Moore of Intel said in a talk at MIT in 1989 that defect density is improving to the point that some companies have been quoted as producing a 100% yield over the whole run. In fact, he has a 100% yield wafer on his desk.

2.13 [20] <2.3> To understand the impact of such claims, list the costs of the largest and smallest dies in Figure 2.29 for defect densities per square centimeter of 3, 2, 1, and 0. For the other parameters use the values found in Figure 2.11 (page 62). Ignore the costs of testing time, packaging, and final test.

2.14 [Discussion] <2.3> If the statement above becomes true for most semiconductor manufacturers, how would that change the options for the computer designer?

2.15 [10/15] <2.3,2.4> Figure 2.18 (page 68) shows the list price as tested of the DECstation 3100 workstation. Start with the costs of the "higher cost" model in Figure 2.13 on page 63, (assuming a color), workstation but change the cost of DRAM to $100/MB for the full 16 MB of the 3100.

a. [10] Using the average discount and overhead percentages of Model B in Figure 2.16 on page 66, what is the gross margin on the DECstation 3100?

b. [15] Suppose you replace the R2000 CPU of the DECstation 3100 with the R3000, and that this change makes the machine 50% faster. Use the costs in Figure 2.29 for the R3000, and assume the R2000 costs a third as much. Since the R3000 does not require much more power, assume that both the power supply and the cooling of the DECstation 3100 are satisfactory for the upgrade. What is the cost/performance of a diskless black-and-white (mono) workstation with an R2000 versus one with an R3000? Using the business model from the answer to part a, how much must the price of the R3000-based machine be increased?

2.16 [30] <2.2,2.4> Pick two computers and run the Dhrystone benchmark and the Gnu C Compiler. Try running the programs using no optimization and maximum optimization. (Note: GCC is a benchmark, so use the appropriate C compiler to compile both programs. Don't try to compile GCC and use it as your compiler!) Then calculate the following performance ratios:

1. Unoptimized Dhrystone on machine A versus unoptimized Dhrystone on machine B.
2. Unoptimized GCC on A versus unoptimized GCC on B.
3. Optimized Dhrystone on A versus optimized Dhrystone on B.
4. Optimized GCC on A versus optimized GCC on B.
5. Unoptimized Dhrystone versus optimized Dhrystone on machine A.
6. Unoptimized GCC versus optimized GCC on A.
7. Unoptimized Dhrystone versus optimized Dhrystone on B.

8. Unoptimized GCC versus optimized GCC on B.

The benchmarking question is how well the benchmark predicts performance of real programs.

If benchmarks do predict performance, then the following equations should be true about the ratios:

(1) = (2) and (3) = (4)

If compiler optimizations work equally as well on real programs as on benchmarks, then

(5) = (6) and (7) = (8)

Are these equations true? If not, try to find the explanation. Is it the machines, the compiler optimizations, or the programs that explain the answer?

2.17 [30] <2.2,2.4> Perform the same experiment as in question 2.16, except replace Dhrystone by Whetstone and replace GCC by Spice.

2.18 [Discussion] <2.2> What are the pros and cons of synthetic benchmarks? Find quantitative evidence—such as data supplied by answering questions 2.16 and 2.17—as well as listing the qualitative advantages and disadvantages.

2.19 [30] <2.2,2.4> Devise a program in C or Pascal that gets the peak MIPS rating for a computer. Run it on two machines to calculate the peak MIPS. Now run GCC and TeX on both machines. How well do peak MIPS predict performance of GCC and TeX?

2.20 [30] <2.2,2.4> Devise a program in C or FORTRAN that gets the peak MFLOPS rating for a computer. Run it on two machines to calculate the peak MFLOPS. Now run Spice on both machines. How well do peak MFLOPS predict performance of Spice?

2.21 [Discussion] <2.3> Use the cost information in Section 2.3 as a basis for the merits of timesharing a large computer versus a network of workstations. (To determine the potential value of workstations versus timesharing, see Section 9.2 in Chapter 9 on user productivity.)

A n	*Add the number in storage location n into the accumulator*
H n	*Transfer the number in storage location n into the multiplier register.*
E n	*If the number in the accumulator is greater than or equal to zero execute next the order which stands in storage location n; otherwise proceed serially.*
I n	*Read the next row of holes on tape and place the resulting 5 digits in the least significant places of storage location n.*
Z	*Stop the machine and ring the warning bell.*

Selection from the list of 18 machine instructions for the EDSAC from Wilkes and Renwick [1949]

3.1 Introduction **89**
3.2 Classifying Instruction Set Architectures **90**
3.3 Operand Storage in Memory: Classifying General-Purpose Register Machines **92**
3.4 Memory Addressing **94**
3.5 Operations in the Instruction Set **103**
3.6 Type and Size of Operands **109**
3.7 The Role of High-Level Languages and Compilers **111**
3.8 Putting It All Together: How Programs Use Instruction Sets **122**
3.9 Fallacies and Pitfalls **124**
3.10 Concluding Remarks **126**
3.11 Historical Perspective and References **127**
Exercises **132**

3 Instruction Set Design: Alternatives and Principles

3.1 Introduction

In this chapter and the next we will concentrate on instruction set architecture—the portion of the machine visible to the programmer or compiler writer. This chapter introduces the wide variety of design alternatives with which the instruction set architect is presented. In particular, this chapter focuses on three topics. First, we present a taxonomy of instruction set alternatives and give some qualitative assessment of the advantages and disadvantages of various approaches. Second, we present and analyze some instruction set measurements that are largely independent of a specific instruction set. Finally, we address the issue of languages and compilers and their bearing on instruction set architecture. Before we discuss how to classify architectures, we need to say something about the instruction set measurement.

Throughout this chapter and the next, we will be examining a wide variety of architectural measurements. These measurements depend on the programs measured and on the compilers used in making the measurements. The results should not be interpreted as absolute, and you might see different data if you did the measurement with a different compiler or a different set of programs. The authors believe that the measurements shown in these chapters are reasonably indicative of a class of typical applications. The measurements are presented using a small set of benchmarks so that the data can be reasonably displayed, and so that the differences among programs can be seen. An architect for a new machine would want to analyze a **much larger** collection of programs to make his architectural decisions. All the measurements shown are *dynamic*—that is, the frequency of a measured event is determined by the number of times that event occurs during execution of the measured program rather than the number of static occurences in the code.

Now, we will begin exploring how instruction set architectures can be classified and analyzed.

3.2 Classifying Instruction Set Architectures

Instruction sets can be broadly classified along the five dimensions described in Figure 3.1, which are roughly ordered by the role they play in distinguishing instruction sets.

The type of internal storage in the CPU is the most basic differentiation, so we will focus on the alternatives for this portion of the architecture in this section. As shown in Figure 3.2, the major choices are a stack, an accumulator, or a set of registers. Operands may be named explicitly or implicitly: The operands in a *stack architecture* are implicitly on the top of the stack; in an

Operand storage in the CPU	Where are operands kept other than in memory?
Number of explicit operands named per instruction	How many operands are named explicitly in a typical instruction?
Operand location	Can any ALU instruction operand be located in memory or must some or all of the operands be in internal storage in the CPU? If an operand is located in memory, how is the memory location specified?
Operations	What operations are provided in the instruction set?
Type and size of operands	What is the type and size of each operand and how is it specified?

FIGURE 3.1 A set of axes for alternative design choices in instruction sets. The type of storage provided for holding operands in the CPU, as opposed to in memory, is the major distinguishing factor among instruction set architectures. (All architectures known to the authors provide some temporary storage within the CPU.) The type of operand storage in the CPU sometimes dictates the number of operands explicitly named in an instruction. In one class of machines, the number of explicit operands may vary. Among recent instruction sets, the number of memory operands per instruction is another significant differentiating factor. The choice of what operations will be supported in instructions interacts less with other aspects of the architecture. Finally, specifying the data type and the size of an operand is largely independent of other instruction set choices.

accumulator architecture one operand is implicitly the accumulator. *General-purpose register architectures* have only explicit operands—either registers or memory locations. Depending on the architecture, the explicit operands to an operation may be accessed directly from memory or they may need to be first loaded into temporary storage, depending on the class of instruction and choice of specific instruction.

Temporary storage provided	Examples	Explicit operands per ALU instruction	Destination for results	Procedure for accessing explicit operands
Stack	B5500, HP 3000/70	0	Stack	Push and pop onto or from the stack
Accumulator	PDP-8, Motorola 6809	1	Accumulator	Load/store accumulator
Register set	IBM 360, DEC VAX	2 or 3	Registers or memory	Load/store of registers, or memory

FIGURE 3.2 Some alternatives for storing operands within the CPU. Each alternative means that a different number of explicit operands is needed for an instruction with two source operands and a result operand. Instruction sets are usually classified by this internal state as stack machine, accumulator machine, or general-purpose register machine. While most architectures fit cleanly into one or another class, some architectures are hybrids of different approaches. The Intel 8086, for example, is halfway between a general-purpose register machine and an accumulator machine.

Figure 3.3 shows how the code sequence C = A + B would typically appear on these three classes of instruction sets. The primary advantages and disadvantages of each of these approaches are listed in Figure 3.4 (page 92).

While most early machines used stack or accumulator-style architectures, every machine designed in the past ten years and still surviving uses a general-purpose register architecture. The major reasons for the emergence of general-purpose register machines are twofold. First, registers—like other forms of

Stack	Accumulator	Register
`PUSH A`	`LOAD  A`	`LOAD  R1,A`
`PUSH B`	`ADD   B`	`ADD   R1,B`
`ADD`	`STORE C`	`STORE C, R1`
`POP  C`		

FIGURE 3.3 The code sequence for C = A + B for three different instruction sets. It is assumed that A, B, and C all belong in memory and that the values of A and B cannot be destroyed.

Machine type	Advantages	Disadvantages
Stack	Simple model of expression evaluation (reverse polish). Short instructions can yield good code density.	A stack cannot be randomly accessed. This limitation makes it difficult to generate efficient code. It's also difficult to implement efficiently, since the stack becomes a bottleneck.
Accumulator	Minimizes internal state of machine. Short instructions.	Since accumulator is only temporary storage, memory traffic is highest for this approach.
Register	Most general model for code generation.	All operands must be named, leading to longer instructions.

FIGURE 3.4 Primary advantages and disadvantages of each class of machine. These advantages and disadvantages are related to three issues: How well the structure matches the needs of a compiler; how efficient the approach is from an implementation viewpoint; and what the effective code size is relative to other approaches.

storage internal to the CPU—are faster than memory. Second, registers are easier for a compiler to use and can be used more effectively than other forms of internal storage. Because general-purpose register machines so dominate instruction set architectures today—and it seems unlikely that this will change in the future—it is only these architectures that we will consider from this point on. Yet even with this limitation, there is a large number of design alternatives to consider. Some designers have proposed the extension of the register concept to allow additional buffering of multiple sets of registers in a stack-like fashion. This additional level of memory hierarchy is examined in Chapter 8.

3.3 Operand Storage in Memory: Classifying General-Purpose Register Machines

The key advantages of general-purpose register machines arise from effective use of the registers by a compiler, both in computing expression values and, more globally, in using registers to hold variables. Registers permit more flexible ordering in evaluating expressions than do either stacks or accumulators. For example, on a register machine the expression (A*B) – (C*D) – (E*F) may be evaluated by doing the multiplications in any order, which may be more efficient due to the location of the operands or because of pipelining concerns (see Chapter 6). But on a stack machine the expression must be evaluated left to right, unless special operations or swaps of stack positions are done.

More important, registers can be used to hold variables. When variables are allocated to registers, the memory traffic is reduced, the program is sped up (since registers are faster than memory), and the code density improves (since a register can be named with fewer bits than can a memory location). Compiler writers would prefer that all registers be equivalent and unreserved. Many machines compromise this desire—especially older machines with many dedicated registers—effectively decreasing the number of general-purpose registers.

If the number of truly general-purpose registers is too small, trying to allocate variables to registers will not be profitable. Instead, the compiler will reserve all the uncommitted registers for use in expression evaluation.

How many registers are sufficient? The answer of course depends on how they are used by the compiler. Most compilers reserve some registers for expression evaluation, use some for parameter passing, and allow the remainder to be allocated to hold variables. To understand how many registers are sufficient, we really need to examine what variables can be allocated to registers and the allocation algorithm used. We deal with these in our discussion of compilers in Section 3.7 and examine measurements of register usage in that section.

There are two major instruction set characteristics that divide general-purpose register, or *GPR,* architectures. Both characteristics concern the nature of operands for a typical arithmetic or logical instruction, or ALU instruction. The first concerns whether an ALU instruction has two or three operands. In the three-operand format, the instruction contains a result and two source operands. In the two-operand format, one of the operands is both a source and a destination for the operation. The second distinction among GPR architectures concerns how many of the operands may be memory addresses in ALU instructions. The number of memory operands supported by a **typical** ALU instruction may vary from none to three. All possible combinations of these two attributes are shown in Figure 3.5, with examples of machines. While there are seven possible combinations, three serve to classify nearly all existing machines: *register–register* (also called *load/store*), *register–memory*, and *memory–memory.*

The advantages and disadvantages of each of these alternatives are shown in Figure 3.6 (page 94). Of course, these advantages and disadvantages are not absolutes. A GPR machine with memory–memory operations can easily be subsetted by the compiler and used as a register–register machine. The

Number of memory addresses per typical ALU instruction	Maximum number of operands allowed for a typical ALU instruction	Examples
0	2	IBM RT-PC
	3	SPARC, MIPS, HP Precision Architecture
1	2	PDP-10, Motorola 68000, IBM 360
	3	Part of IBM 360 (RS instructions)
2	2	PDP-11, National 32x32, part of IBM 360 (SS instructions)
	3	
3	3	VAX (also has two-operand formats)

FIGURE 3.5 Possible combinations of memory operands and total operands per ALU instruction with examples of machines. Machines with no memory reference per ALU instruction are called load/store or register–register machines. Instructions with multiple memory operands per typical ALU instruction are called register–memory or memory–memory, according to whether they have one or more than one memory operand.

Type	Advantages	Disadvantages
Register–register (0,3)	Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute (see Chapter 6).	Higher instruction count than architectures with memory references in instructions. Some instructions are short and bit encoding may be wasteful.
Register–memory (1,2)	Data can be accessed without loading first. Instruction format tends to be easy to encode and yields good density.	Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction varies by operand location.
Memory–memory (3,3)	Most compact. Doesn't waste registers for temporaries.	Large variation in instruction size, especially for three-operand instructions. Also, large variation in work per instruction. Memory accesses create memory bottleneck.

FIGURE 3.6 Advantages and disadvantages of the three most common types of general-purpose register machines. The notation (*m, n*) means *m* memory operands and *n* total operands. In general, machines with fewer alternatives make the compiler's task simpler since there are fewer decisions for the compiler to make. Machines with a wide variety of flexible instruction formats reduce the number of bits required to encode the program. A machine that uses a small number of bits to encode the program is said to have good *instruction density*—a smaller number of bits do as much work as a larger number on a different architecture. The number of registers also affects the instruction size.

advantages and disadvantages listed in the figure deal primarily with the impact both on the compiler and on the implementation. These advantages and disadvantages are qualitative and their actual impact depends on the compiler and implementation strategy. One of the most pervasive architectural impacts is on instruction encoding and the number of instructions needed to perform a task. In other chapters, we will see the impact of these architectural alternatives on various implementation approaches.

3.4 Memory Addressing

Independent of whether the architecture is register–register (also called *load/store*) or allows any operand to be a memory reference, it must define how memory addresses are interpreted and how they are specified. We will deal with these two topics in this section. The measurements presented here are largely, but not completely, machine independent. In some cases the measurements are significantly affected by the compiler technology. These measurements have been made using an optimizing compiler since compiler technology is playing an increasing role. The measurements will probably reflect what we will be seeing in the future rather than what has been so in the past.

Interpreting Memory Addresses

How is a memory address interpreted? That is, what object is accessed as a function of the address and the length? All the machines discussed in this and the next chapter are byte addressed and provide access for bytes (8 bits), half-words (16 bits), and words (32 bits). Most of the machines also provide access for doublewords (64 bits).

There are two different conventions for ordering the bytes within a word, as shown in Figure 3.7. *Little Endian* byte order puts the byte whose address is "x...x00" at the least significant position in the word (the little end). *Big Endian* byte order puts the byte whose address is "x...x00" at the most significant position in the word (the big end). In Big Endian addressing, the address of a datum is the address of the most significant byte; while in Little Endian, the address of a datum is the least significant byte. When operating within one machine, the byte order is often unnoticeable—only programs that access the same locations as both words and bytes can notice the difference. However, byte order is a problem when exchanging data among machines with different orderings. (The byte orders used by a number of different machines are listed inside the front cover.)

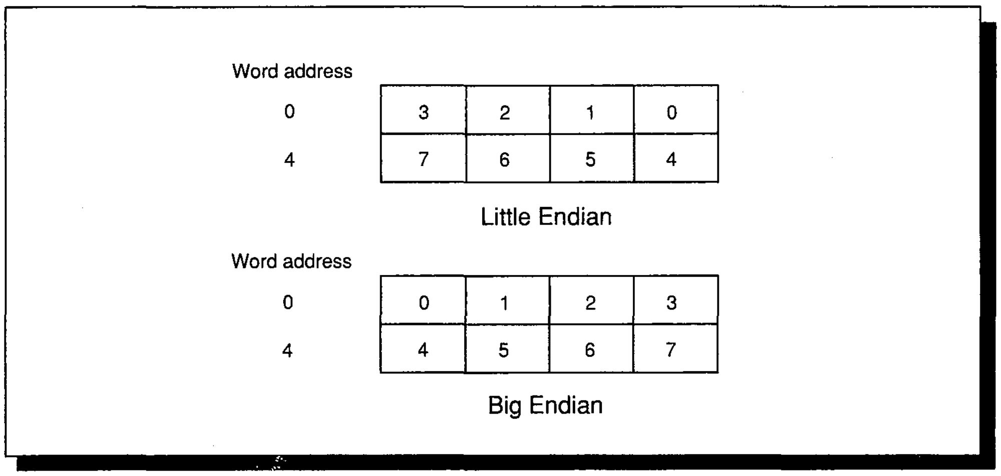


FIGURE 3.7 The two different conventions for ordering bytes within a word. The names "Big Endian" and "Little Endian" come from a famous paper by Cohen [1981]. The paper draws an analogy between the argument over which end to number the bytes from and the argument in Gulliver's Travels over which end of an egg to open. The DEC PDP-11/VAX and Intel 80x86 follow the Little Endian model, while the IBM 360/370 and Motorola 680x0, and others follow the Big Endian model. This numbering applies to bit positions as well, though only a few architectures supply instructions to access bits by their numbered position.

In some machines, accesses to objects larger than a byte must be aligned. An access to an object of size s bytes at byte address A is aligned if A **mod** $s = 0$. Figure 3.8 shows the addresses at which an access is aligned or misaligned.

Object addressed	Aligned at byte offsets	Misaligned at byte offsets
byte	0,1,2,3,4,5,6,7	(never)
halfword	0,2,4,6	1,3,5,7
word	0,4	1,2,3,5,6,7
doubleword	0	1,2,3,4,5,6,7

FIGURE 3.8 Aligned and misaligned accesses of objects. The byte offsets are specified for the low-order three bits of the address.

Why would someone design a machine with alignment restrictions? Misalignment causes hardware complications, since the memory is typically aligned on a word boundary. A misaligned memory access will, therefore, take multiple aligned memory references. Figure 3.9 shows what happens when an access occurs to a misaligned word in a system with a 32-bit-wide bus to memory: Two accesses are required to get the word. Thus, even in machines that allow misaligned access, programs with aligned accesses run faster.

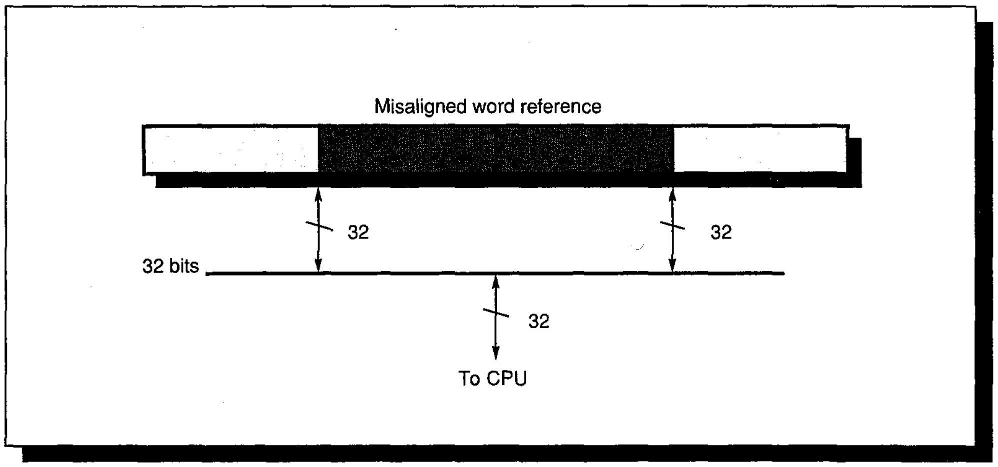


FIGURE 3.9 A word reference is made to a halfword (16-bit) boundary in a memory system that has a 32-bit access path. The CPU or memory system has to perform two separate accesses to get the upper and lower halfword. The two halfwords are then merged to obtain the entire word. With memory organized as independent byte-wide modules it is possible to access only the needed data, but this requires more complex control to supply a different address to each module to select the proper byte.

Even if data is aligned, supporting byte and halfword accesses requires an alignment network to align bytes and halfwords in registers. Depending on the instruction, the machine may also need to sign extend the quantity. On some machines a byte or halfword does not affect the upper portion of a register. For stores only the affected bytes in memory may be altered. Figure 3.10 shows the alignment network for loading or storing a byte from a word in memory into a register. While all the machines discussed in this chapter and the next permit byte and halfword accesses to memory, only the VAX and the Intel 8086 support ALU operations on register operands with a size shorter than a word.

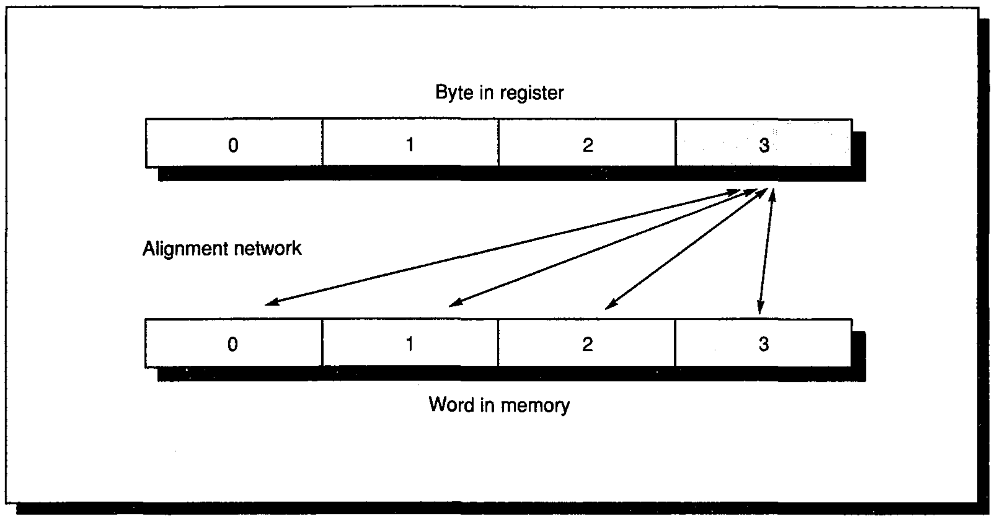


FIGURE 3.10 The alignment network to load or store a byte. The memory system is assumed to be 32 bits wide, and four alignment paths are required for bytes. Accessing aligned halfwords would require two additional paths to move either byte 0 or byte 2 in memory to byte 2 in the register. A 64-bit memory system would require twice as many alignment paths for bytes and halfwords, as well as two 32-bit alignment paths for word accesses. The alignment network only positions the bytes for a store—additional control signals are used to ensure that only the correct byte positions are written in memory. Rather than an alignment network, some machines use a shifter and shift the data only in those cases where alignment is required. This makes the access of a nonword object considerably slower, but eliminating the alignment network speeds up the more common case of accessing a word.

Addressing Modes

We now know what bytes to access in memory given an address. In this section we will look at addressing modes—how architectures specify the address of an object they will access. In GPR machines, an addressing mode can specify a constant, a register, or a location in memory. When a memory location is used, the actual memory address specified by the addressing mode is called the *effective address.*

Figure 3.11 shows all the data addressing modes that arise in the machines discussed in the following chapter. Immediates or literals are usually considered a memory addressing mode (even though the value they access is in the instruction stream), while registers are often separated. We have kept addressing modes that depend on the program counter, called *PC-relative addressing*, separate. PC-relative addressing is used primarily for specifying code addresses in control transfer instructions. The use of PC-relative addressing in control instructions is discussed in Section 3.5.

Figure 3.11 shows the most common names for the addressing modes, though the names differ among architectures. In this figure and throughout the book, we will use an extension of the C programming language as a hardware description notation. In this figure, only two non-C features are used. First, the left arrow (←) is used for assignment. Second, the array `M` is used as the name for memory.

Addressing mode	Example instruction	Meaning	When used
Register	`Add R4,R3`	`R4←R4+R3`	When a value is in a register.
Immediate or literal	`Add R4,#3`	`R4←R4+3`	For constants. In some machines, literal and immediate are two different addressing modes.
Displacement or based	`Add R4,100(R1)`	`R4←R4+M[100+R1]`	Accessing local variables.
Register deferred or indirect	`Add R4,(R1)`	`R4←R4+M[R1]`	Accessing using a pointer or a computed address.
Indexed	`Add R3,(R1 + R2)`	`R3←R3+M[R1+R2]`	Sometimes useful in array addressing—`R1`=base of array; `R2`=index amount.
Direct or absolute	`Add R1,(1001)`	`R1←R1+M[1001]`	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect or memory deferred	`Add R1,@(R3)`	`R1←R1+M[M[R3]]`	If `R3` is the address of a pointer *p*, then mode yields **p*.
Autoincrement	`Add R1,(R2)+`	`R1←R1+M[R2]` `R2←R2+`*d*	Useful for stepping through arrays within a loop. `R2` points to start of array; each reference increments `R2` by size of an element, *d*.
Autodecrement	`Add R1,-(R2)`	`R2←R2-`*d* `R1←R1+M[R2]`	Same use as autoincrement. Autoincrement/decrement can also be used to implement a stack as push and pop.
Scaled or index	`Add R1,100(R2)[R3]`	`R1←R1+M[100+R2+R3*`*d*`]`	Used to index arrays. May be applied to any base addressing mode in some machines.

FIGURE 3.11 Selection of addressing modes with examples, meaning, and usage. The extensions to C used in the hardware descriptions are defined above. In autoincrement/decrement and scaled or index addressing modes, the variable *d* designates the size of the data item being accessed (i.e., whether the instruction is accessing 1, 2, 4, or 8 bytes); this means that these addressing modes are only useful when the elements being accessed are adjacent in memory. In our measurements, we use the first name shown for each mode. A few machines, such as the VAX, encode some of these addressing modes as PC-relative.

Thus, `M[R1]` refers to the contents of the memory location whose address is given by the contents of R1. Later, we will introduce extensions for accessing and transferring data smaller than a word.

Addressing modes have the ability to significantly reduce instruction counts; they also add to the complexity of building a machine. Thus, the usage of various addressing modes is quite important in helping the architect choose what to include. While many measurements of addressing mode usage are machine dependent, others are largely independent of the machine architecture. Some of the more important machine-independent measurements will be examined in this chapter. But, before we look at this type of measurement, let's look at how often these various memory addressing modes are used.

Figure 3.12 shows the results of measuring addressing mode usage patterns in our benchmarks—Gnu C Compiler (GCC), Spice, and TeX—on the VAX, which supports all the modes shown in Figure 3.11. We will look at further measurements of addressing mode usage on the VAX in the next chapter.

As Figure 3.12 shows, immediate and displacement addressing dominate addressing mode usage. Let's look at some properties of these two heavily used modes.

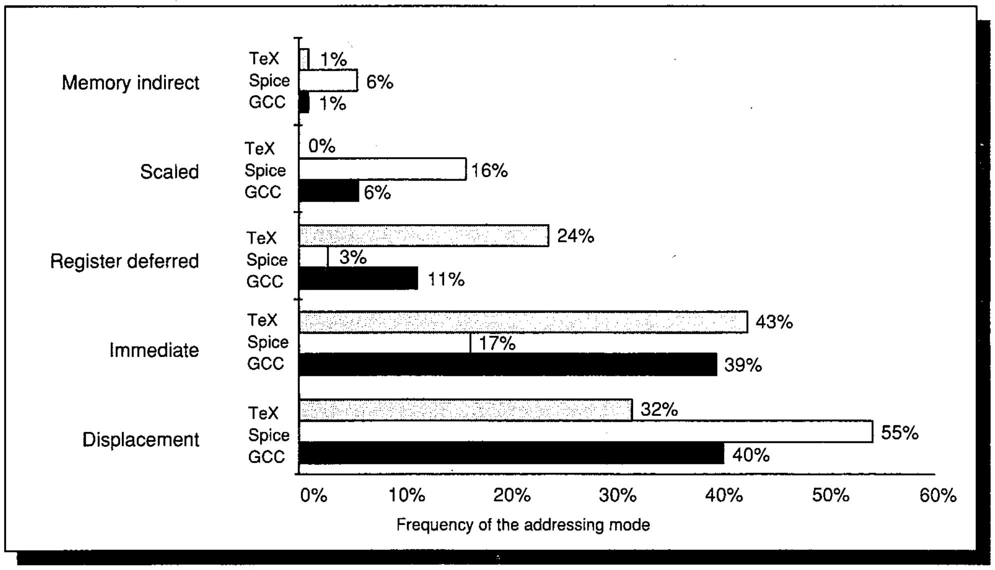


FIGURE 3.12 Summary of use of memory addressing modes (including immediates). The data were taken on a VAX using our three benchmark programs. Only the addressing modes with an average frequency of over 1% are shown. The PC-relative addressing modes, which are used almost exclusively for branches, are not included. Displacement mode includes all displacement lengths (8-, 16-, and 32-bit). Register modes, which are not counted, account for one-half of the operand references, while memory addressing modes (including immediate) account for the other half. The memory indirect mode on the VAX can use displacement, autoincrement, or autodecrement to form the initial memory address; in these programs, almost all the memory indirect references use displacement mode as the base. Of course, the compiler affects what addressing modes are used; we discuss this further in Section 3.7. These major addressing modes account for all but a few percent (0% to 3%) of the memory-accesses.

Displacement or Based Addressing Mode

The major question that arises for a displacement-style addressing mode is that of the range of displacements used. Based on the use of various displacement sizes, a decision of what sizes to support can be made. Choosing the displacement field sizes is important because they directly affect the instruction length. Measurements taken on the data access on a load/store architecture using our three benchmark programs are shown in Figure 3.13. We will look at branch offsets in the next section—data accessing patterns and branches are so different, little is gained by combining them.

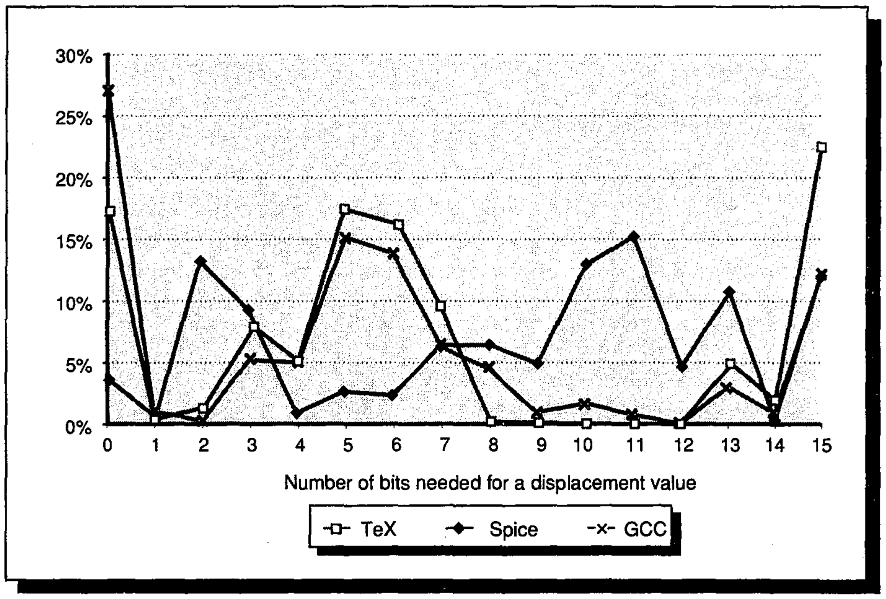


FIGURE 3.13 Displacement values are widely distributed. Though there is a large number of small values, there is also a fair number of large values. The wide distribution of displacement values is due to multiple storage areas for variables and different displacements used to access them. The different storage areas and their access patterns are discussed further in Section 3.7. The chart shows only the magnitude of the displacement and not the sign, which is heavily affected by the storage layout. The entry corresponding to 0 on the x axis shows the percentage of displacements of value 0. The vast majority of the displacements are positive, but a majority of the largest displacements (14+ bits) are negative. Again, this is due to the overall addressing scheme used by the compiler and might change with a different compilation scheme. Since this data was collected on a machine with 16-bit displacements, it cannot tell us anything about accesses that might want to use a longer displacement. Such accesses are broken into two separate instructions—the first of which loads the upper 16 bits of a base register. By counting the frequency of these "load immediate" instructions, which have limited use for other purposes, we can bound the number of accesses with displacements potentially larger than 16 bits. Such an analysis indicates GCC, Spice, and TeX may actually require a displacement longer than 16 bits for up to 5%, 13%, and 27% of the memory references, respectively. Furthermore, if the displacement is larger than 15 bits, it is likely to be quite a bit larger since most constants being loaded are large, as shown in Figure 3.15 (page 102).To evaluate the choice of displacement length, we might also want to examine a cumulative distribution, as shown in Exercise 3.3 (see Figure 3.35 on page 133).

Immediate or Literal Addressing Mode

Immediates can be used in arithmetic operations, in comparisons (primarily for branches), and in moves in which a constant is wanted in a register. The last case occurs for constants written in the code, which tend to be small, and for address constants, which can be large. For the use of immediates it is important to know whether they need to be supported for all operations or for only a subset. The chart in Figure 3.14 shows the frequency of immediates for the general classes of operations in an instruction set.

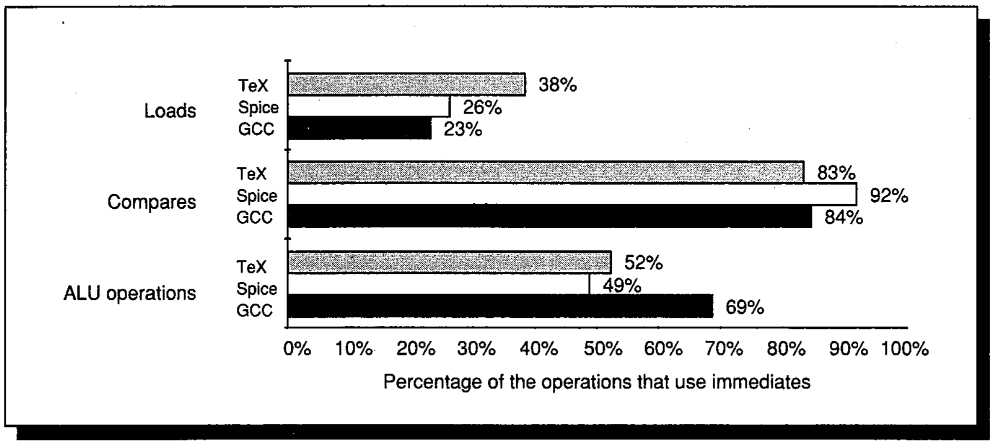


FIGURE 3.14 We see that for ALU operations about half the operations have an immediate operand, while for compares more than 85% of the occurrences use an immediate operand. (For ALU operations, shifts by a constant amount are included as operations with immediate operands.) For loads, the load immediate instructions load 16 bits into either half of a 32-bit register. These load immediates are not loads in a strict sense because they do not reference memory. In some cases, a pair of load immediates may be used to load a 32-bit constant, but this is rare. The compares include comparisons against zero that are done in conditional branches based on this comparison. These measurements were taken on a MIPS R2000 architecture with full compiler optimization. The compiler attempts to use simple compares against zero for branches whenever possible because these branches are efficiently supported in the architecture.

Another important instruction set measurement is the range of values for immediates. Like displacement values, the sizes of immediate values affect instruction lengths. As Figure 3.15 shows, immediate values that are small are most heavily used. However, large immediates are sometimes used, most likely in addressing calculations. The data in Figure 3.15 was taken on a VAX, which provides many instructions that have zero as an implicit operand. These include instructions to compare against zero and to store zero into a word. Because of the use of these instructions, the measurements show relatively infrequent use of zero.

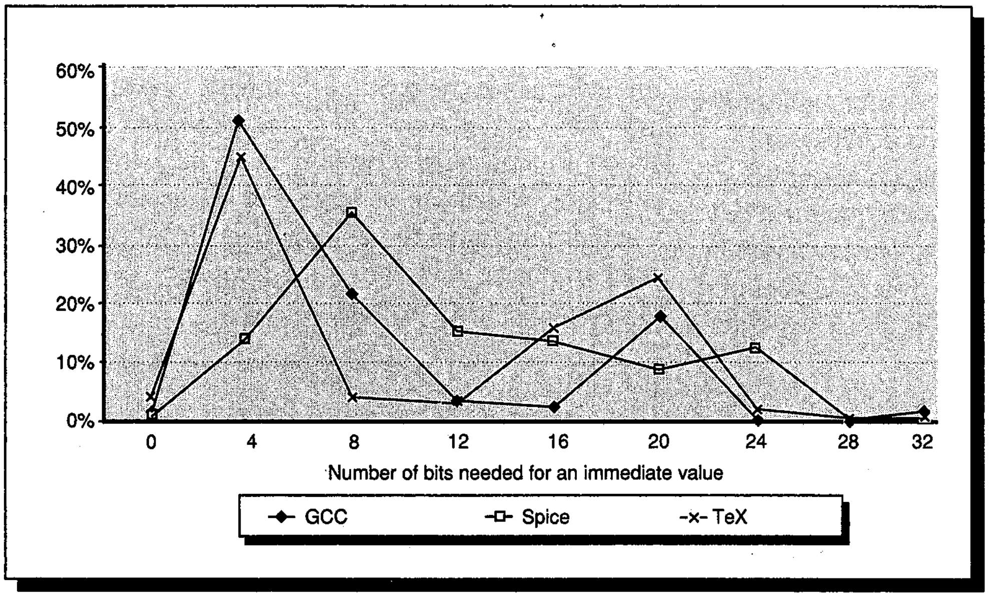


FIGURE 3.15 The distribution of immediate values is shown. The x axis shows the number of bits needed to represent the magnitude of an immediate value—0 means the immediate field value was 0. The vast majority of the immediate values are positive: Overall, less than 6% of the immediates are negative.These measurements were taken on a VAX, which supports a full range of immediates and sizes as operands to any instruction. The measured programs are the standard set—GCC, Spice, and TeX.

Encoding of Addressing Modes

How the addressing modes of operands are encoded depends on the range of addressing modes and the degree of independence between opcodes and modes. For a small number of addressing modes or opcode/addressing mode combinations, the addressing mode can be encoded in the opcode. This works for the IBM 360 with only five addressing modes and most operations offered in only one or two modes. For a large number of combinations, typically a separate *address specifier* is needed for each operand. The address specifier tells what addressing mode the operand is using. In Chapter 4, we will see how these two types of encodings are used in several real instruction formats.

When encoding the instructions, the number of registers and the number of addressing modes both have a significant impact on the size of instructions. This is because the addressing mode field and the register field may appear many times in a single instruction. In fact, for most instructions many more bits are consumed encoding addressing modes and register fields than in specifying the opcode. The architect must balance several competing forces when encoding the instruction set:

1. The desire to have as many registers and addressing modes as possible.

2. The impact of the size of the register and addressing mode fields on the average instruction size and hence on the average program size.
3. A desire to have instructions encode into lengths that will be easy to handle in the implementation. As a minimum, the architect wants instructions to be in multiples of bytes, rather than an arbitrary length. Many architects have chosen to use a fixed-length instruction to gain implementation benefits while sacrificing average code size.

Since the addressing modes and register fields make up such a large percentage of the instruction bits, their encoding will significantly affect how easy it is for an implementation to decode the instructions. The importance of having easily decoded instructions is discussed in Chapters 5 and 6.

3.5 Operations in the Instruction Set

The operators supported by most instruction set architectures can be categorized, as in Figure 3.16. In Section 3.8, we look at the use of operations in a general fashion (e.g. memory references, ALU operations, and branches). In Chapter 4, we will examine the use of various instruction operations in detail for four different architectures. Because the instructions used to implement control flow are largely independent of other instruction set choices and because the measurements of branch and jump behavior are also fairly independent of other measurements, we examine the use of control-flow instructions next.

Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, and, subtract, or
Data transfer	Loads/stores (move instructions on machines with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating point	Floating-point operations: add, multiply
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search

FIGURE 3.16 Categories of instruction operators and examples of each. All machines generally provide a full set of operations for the first three categories. The support for system functions in the instruction set varies widely among architectures, but all machines must have some instruction support for basic system functions. The amount of support in the instruction set for the last three categories may vary from none to an extensive set of special instructions. Floating-point instructions will be provided in any machine that is intended for use in an application that makes much use of floating point. These instructions are sometimes part of an optional instruction set. Decimal and string instructions are sometimes primitives, as in the VAX or the IBM 360, or may be synthesized by the compiler from simpler instructions. Examples of instruction sets appear in Appendix B, while Appendix C contains measurements of typical usage. We will examine four different instruction sets and their usage in detail in Chapter 4.

Instructions for Control Flow

As Figure 3.17 shows, there is no consistent terminology for instructions that change the flow of control. Until the IBM 7030, control-flow instructions were typically called *transfers*. Beginning with the 7030, the name *branch* began to be used. Later, machines introduced additional names. Throughout this book we will use *jump* when the change in control is unconditional and *branch* when the change is conditional.

Machine	Year	"Branch"	"Jump"
IBM 7030	1960	All control transfers—addressing is PC-relative	
IBM 360/370	1965	All control transfers—no PC-relative	
DEC PDP-11	1970	PC-relative only, conditional and unconditional	All addressing modes; unconditional only
Intel 8086	1978		All transfers are jumps; conditional jumps are PC-relative only
DEC VAX	1978	Same as PDP-11	Same as PDP-11
MIPS R2000	1986	Conditional control transfer, always PC-relative	Unconditional jumps and call instructions

FIGURE 3.17 Machines, dates, and the names associated with control transfers in their architectures. These names vary widely based on whether the transfer is conditional or unconditional and on whether it is PC-relative or not. The VAX, PDP-11, and MIPS R2000 architectures allow only PC-relative addressing for branches.

We can distinguish four different types of control-flow change:

1. Conditional branches
2. Jumps
3. Procedure calls
4. Procedure returns

We want to know the relative frequency of these events, as each event is different, may use different instructions, and may have different behavior. The frequencies of these control-flow instructions for a load/store machine running our benchmarks is shown in Figure 3.18.

The destination address of a branch must always be specified. This destination is specified explicitly in the instruction in the vast majority of cases—procedure return being the major exception—since for return the target is not known at compile time. The most common way to specify the destination is to supply a displacement that is added to the *program counter*, or PC. Branches of this sort are called *PC-relative* branches. PC-relative branches are advantageous because the branch target is often near the current instruction, and specifying the position relative to the current PC requires fewer bits. Using PC-relative addressing also permits the code to run independent of where it is loaded. This property, called *position-independence*, can eliminate some work when the program is linked and is also useful in programs linked during execution.

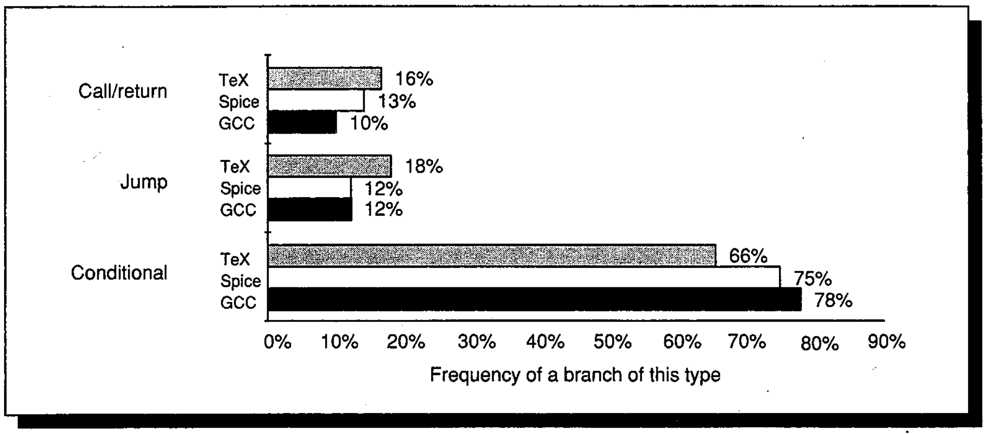


FIGURE 3.18 Breakdown of branches into three classes. Each branch is counted in one of three bars. Conditional branches clearly dominate. On average 90% of the jumps are PC-relative.

To implement returns and indirect branches in which the target is not known at compile time, a method other than PC-relative addressing is required. Here, there must be a way to specify the target dynamically, so that it can change at run-time. This may be as simple as naming a register that contains the target address. Alternatively, the branch may permit any addressing mode to be used to supply the target address.

A key question concerns how far branch targets are from branches. Knowing the distribution of these displacements will help in choosing what branch offsets to support and thus will affect the instruction length and encoding. Figure 3.19 (page 106) shows the distribution of displacements for PC-relative branches in instructions. About 75% of the branches are in the forward direction.

Since most changes in control flow are branches, deciding how to specify the branch condition is important. The three primary techniques in use and their advantages and disadvantages are shown in Figure 3.20 (page 106).

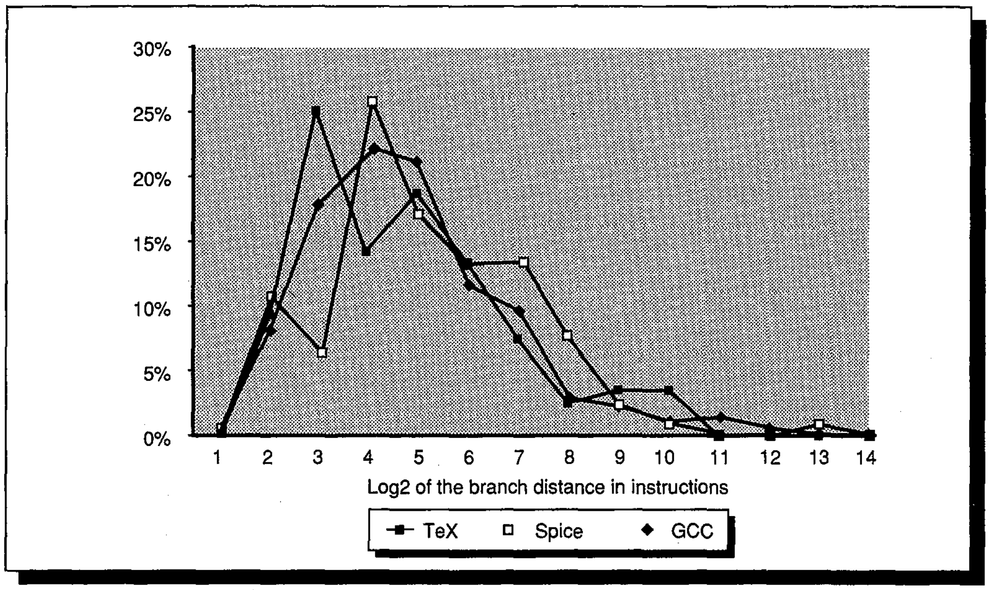


FIGURE 3.19 Branch distances in terms of number of instructions between the target and the branch instruction. The most frequent branches in Spice are to targets that are 8 to 15 instructions away (2^4). The weighted-arithmetic-mean branch target distance is 86 instructions (2^7). This tells us that short displacement fields often suffice for branches and that the designer can gain some encoding density by having a shorter instruction with a smaller branch displacement. These measurements were taken on a load/store machine (MIPS R2000 architecture). An architecture that requires fewer instructions for the same program, such as a VAX, would have shorter branch distances. Similarly, the number of bits needed for the displacement may change if the machine allows instructions to be arbitrarily aligned. A cumulative distribution of this branch displacement data is shown in Exercise 3.3 (see Figure 3.35 on page 133).

Name	How condition is tested	Advantages	Disadvantages
Condition code (CC)	Special bits are set by ALU operations, possibly under program control.	Sometimes condition is set for free.	CC is extra state. Condition codes constrain the ordering of instructions since they pass information from one instruction to a branch.
Condition register	Set arbitrary register with the result of a comparison.	Simple.	Uses up a register.
Compare and branch	Compare is part of the branch. Often compare is limited to subset.	One instruction rather than two for a branch.	May be too much work per instruction.

FIGURE 3.20 The major methods for evaluating branch conditions, their advantages, and disadvantages. Although condition codes can be set by ALU operations that are needed for other purposes, measurements on programs show that this rarely happens. The major implementation problems with condition codes arise when the condition code is set by a large or haphazardly chosen subset of the instructions, rather than being controlled by a bit in the instruction. Machines with compare and branch often limit the set of compares and use a condition register for more complex compares. Often, different techniques are used for branches based on floating-point comparison versus those based on integer comparison. This is reasonable since the number of branches that depend on floating-point comparisons is much smaller than the number depending on integer comparisons.

One of the most noticeable properties of branches is that a large number of the comparisons are simple equality or inequality tests, and a large number are comparisons with zero. Thus, some architectures choose to treat these comparisons as special cases, especially if a *compare and branch* instruction is being used. Figure 3.21 shows the frequency of different comparisons used for conditional branching. The data in Figure 3.14 said that a large percentage of the branches had an immediate operand (86%), and while not shown, 0 was the most heavily used immediate (83% of the immediates in branches). When we combine this with the data in Figure 3.21 we can see that a significant percentage (over 50%) of the compares in branches are simple tests for equality with zero.

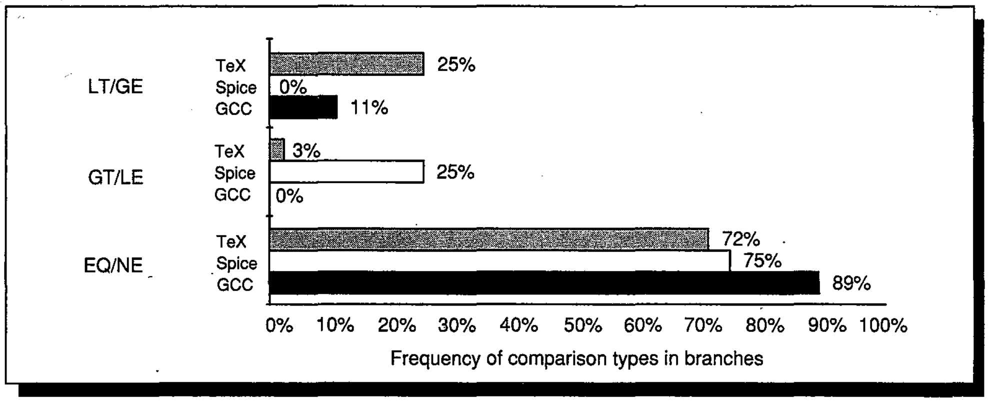


FIGURE 3.21 Frequency of different types of compares in conditional branches. This includes both the integer and floating-point compares in branches. Floating-point comparisons constitute 13% of the branch comparisons in Spice. Remember that earlier data in Figures 3.14 indicate that most comparisons are against an immediate operand. This immediate value is usually 0 (83% of the time).

Program	Percentage of backward branches	Percentage taken branches	Percentage of all control instructions that actually branch
GCC	26%	54%	63%
Spice	31%	51%	63%
TeX	17%	54%	70%
Average	25%	53%	65%

FIGURE 3.22 Branch direction, branch-taken frequency, and frequency that the PC is changed. The first column shows what percentage of all branches (both taken and untaken) are backward-going. The second column shows what percentage of all branches (remember that a branch is always conditional) are taken. The final column shows what percentage of all control-flow instructions actually cause a nonsequential transfer in the flow. This last column is computed by combining data from the second column and the data in Figure 3.18 (page 105).

We will say that a branch is *taken* if the condition tested by the branch is true and the next instruction to be executed is the target of the branch. All jumps, therefore, are taken. Figure 3.22 shows the branch-direction distribution, the frequency of taken (conditional) branches, and the percentage of control-flow instructions that change the PC. Most backward-going branches are loop branches, and typically loop branches are taken with about 90% probability.

Many programs have a higher percentage of loop branches, thus boosting the frequency of taken branches over 60%. Overall, branch behavior is application-dependent and sometimes compiler-dependent. Compiler dependencies arise because of changes to the control flow made by optimizing compilers to improve the execution time of loops.

Example

Assuming that 90% of the backward-going branches are taken, find the probability that a forward-going branch is taken using the averaged data in Figure 3.22.

Answer

The average frequency of taken branches is the sum of the backward-taken and forward-taken times their respective frequencies:

$$\begin{aligned}\%\text{ taken branches} &= (\%\text{ taken backward} * \%\text{ backward}) + \\ &\quad (\%\text{ taken forward} * \%\text{ forward}) \\ 53\% &= (90\% * 25\%) + (\%\text{ taken forward} * 75\%) \\ \%\text{ taken forward} &= \frac{53\% - 22.5\%}{75\%} \\ \%\text{ taken forward} &= 40.7\%\end{aligned}$$

It is not unusual to see the majority of forward branches be untaken. The behavior of forward-going branches often varies among programs.

Procedure calls and returns include control transfer and possibly some state saving; at a minimum the return address must be saved somewhere. Some architectures provide a mechanism to save the registers, while others require the compiler to generate instructions. There are two basic conventions in use to save registers. *Caller-saving* means that the calling procedure must save the registers that it wants preserved for access after the call. *Callee-saving* means that the called procedure must save the registers it wants to use. There are times when caller save must be used due to access patterns to globally visible variables in two different procedures. For example, suppose we have a procedure P1 that calls procedure P2, and both procedures manipulate the global variable *x*. If P1 had allocated *x* to a register it must be sure to save *x* to a location known by P2 before the call to P2. A compiler's ability to discover when a called procedure may access register-allocated quantities is complicated by the possibility of separate compilation, and situations where P2 may not touch *x*, but P2 can call another procedure, P3, that may access *x*. Because of these complications, most compilers will conservatively caller save **any** variable that **may be** accessed during a call.

In the cases where either convention could be used, some will be more optimal with callee-save and some will be more optimal with caller-save. As a result, the most sophisticated compilers use a combination of the two mechanisms, and the register allocator may choose which register to use for a variable based on the convention. Later in this chapter we will examine how well more sophisticated instructions match the needs of the compiler for this function, and in Chapter 8 we will look at hardware buffering schemes for supporting register save and restore.

3.6 Type and Size of Operands

How is the type of an operand designated? There are two primary alternatives: First, the type of an operand may be designated by encoding it in the opcode—this is the method used most often. Alternatively, the data can be annotated with tags that are interpreted by the hardware. These tags specify the type of the operand, and the operation is chosen accordingly. Machines with tagged data, however, are extremely rare. The Burroughs' architectures are the most extensive example of tagged architectures. Symbolics also built a series of machines that used tagged data items for implementing LISP.

Usually the type of an operand—for example, integer, single-precision floating point, character—effectively gives its size. Common operand types include character (one byte), halfword (16 bits), word (32 bits), single-precision floating point (also one word), and double-precision floating point (two words). Characters are represented as either EBCDIC, used by the IBM mainframe architectures, or ASCII, used by everyone else. Integers are almost universally represented as two's complement binary numbers. Until recently, most computer manufacturers chose their own floating-point representation. However, in the past few years, a standard for floating point, the IEEE standard 754, has become the choice of most new computers. The IEEE floating-point standard is discussed in detail in Appendix A.

Some architectures provide operations on character strings, although such operations are usually quite limited and treat each byte in the string as a single character. Typical operations supported on character strings are comparisons and moves.

For business applications, some architectures support a decimal format, usually called *packed decimal*. Packed decimal is *binary-coded decimal*—four bits are used to encode the values 0–9, and two decimal digits are packed into each byte. Numeric character strings are sometimes called unpacked decimal, and operations—called packing and unpacking—are usually provided for converting back and forth between them.

Our benchmarks use byte or character, halfword (short integer), word (integer), and floating-point data types. Figure 3.23 shows the dynamic distribution of the sizes of objects referenced from memory for these programs. The frequency of access to different data types helps in deciding what types are most important to support efficiently. Should the machine have a 64-bit access path, or would taking two cycles to access a doubleword be satisfactory? How important is it to support byte accesses as primitives, which, as we saw earlier, require an alignment network? In Figure 3.23, memory references are used to examine the types of data being accessed. In some architectures, objects in registers may be accessed as bytes or halfwords. However, such access is very infrequent—on the VAX, it accounts for no more than 12% of register references, or roughly 6% of all operand accesses in these programs.

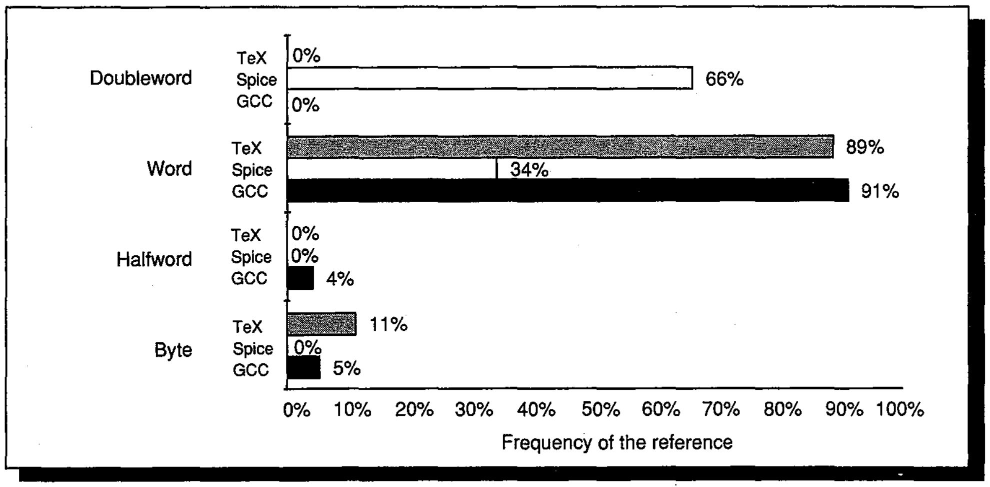


FIGURE 3.23 Distribution of data accesses by size for the benchmark programs. Access to the major data type (word or doubleword) clearly dominates. Reads outnumbered writes of data items by a factor of 1.6 for TeX to a factor of 2.5 for Spice. The doubleword data type is used solely for double-precision floating point in Spice. Spice makes only small use of single-precision floating point; most word references in Spice are to integers. These measurements were taken on the memory traffic generated on a load/store architecture.

In the next chapter we will look extensively at the differences in instruction mix and other architectural measurements on four very different machines. But before we do that, it will be helpful to take a brief look at modern compiler technology and its effect on program properties.

3.7 The Role of High-Level Languages and Compilers

Today most programming is done in high-level languages. This means that since most instructions executed are the output of a compiler, an instruction set architecture is essentially a compiler target. In earlier times, architectural decisions were often made to ease assembly language programming. Because performance of a computer will be significantly affected by the compiler, understanding compiler technology today is critical to designing and efficiently implementing an instruction set. In earlier days it was popular to try to isolate the compiler technology and its effect on hardware performance from the architecture and its performance, just as it was popular to try to separate an architecture from its implementation. This is extremely difficult, if not impossible, with today's compilers and architectures. Architectural choices affect the quality of the code that can be generated for a machine and the complexity of building a good compiler for it. Isolating the compiler from the hardware is likely to be misleading. In this section we will discuss the critical goals in the instruction set primarily from the compiler viewpoint. What features will lead to high-quality code? What makes it easy to write efficient compilers for an architecture?

The Structure of Recent Compilers

To begin, let's look at what optimizing compilers are like today. The structure of recent compilers is shown in Figure 3.24.

A compiler writer's first goal is correctness—all valid programs must be compiled correctly. The second goal is usually speed of the compiled code. Typically, a whole set of other goals follow these first two, including fast compilation, debugging support, and interoperability among languages. Normally, the passes in the compiler transform higher-level, more abstract representations into progressively lower-level representations, eventually reaching the instruction set. This structure helps manage the complexity of the transformations and makes writing a bug-free compiler easier.

The complexity of writing a correct compiler is a major limitation on the amount of optimization that can be done. Although the multiple-pass structure helps reduce compiler complexity, it also means that the compiler must order and perform some transformations before others. In the diagram of the optimizing compiler in Figure 3.24, we can see that certain high-level optimizations are performed long before it is known what the resulting code will look like in detail. Once such a transformation is made, the compiler can't afford to go back and revisit all steps, possibly undoing transformations. This would be prohibitive, both in compilation time and in complexity. Thus, compilers make assumptions about the ability of later steps to deal with certain problems. For example, compilers usually have to choose which procedure calls to expand inline before they know the exact size of the procedure being called. Compiler writers call this problem the *phase-ordering problem.*

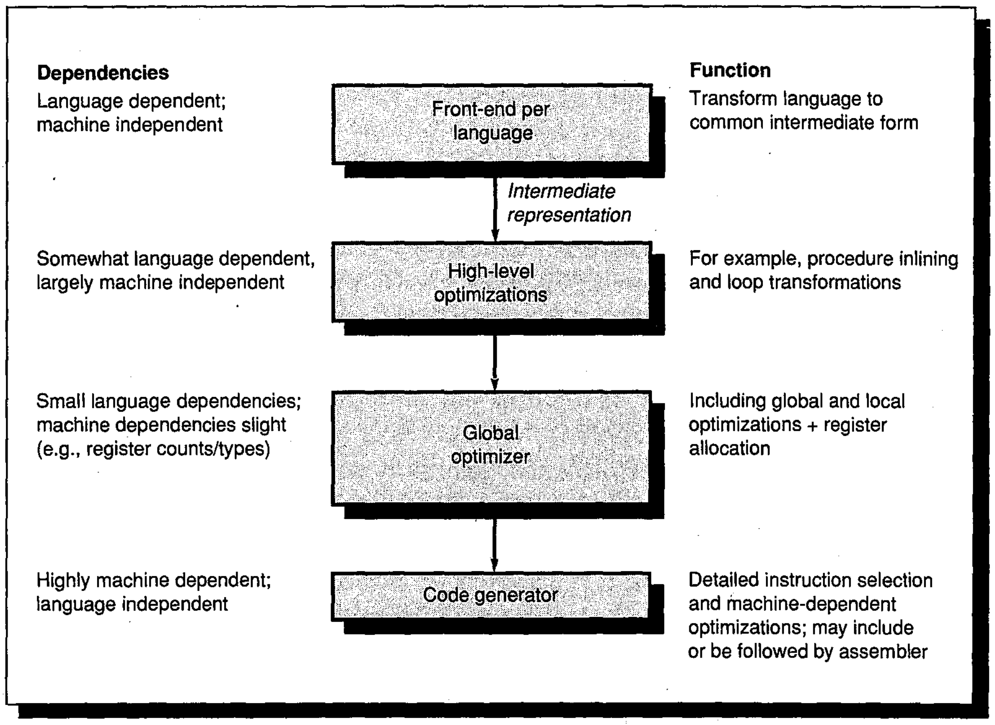
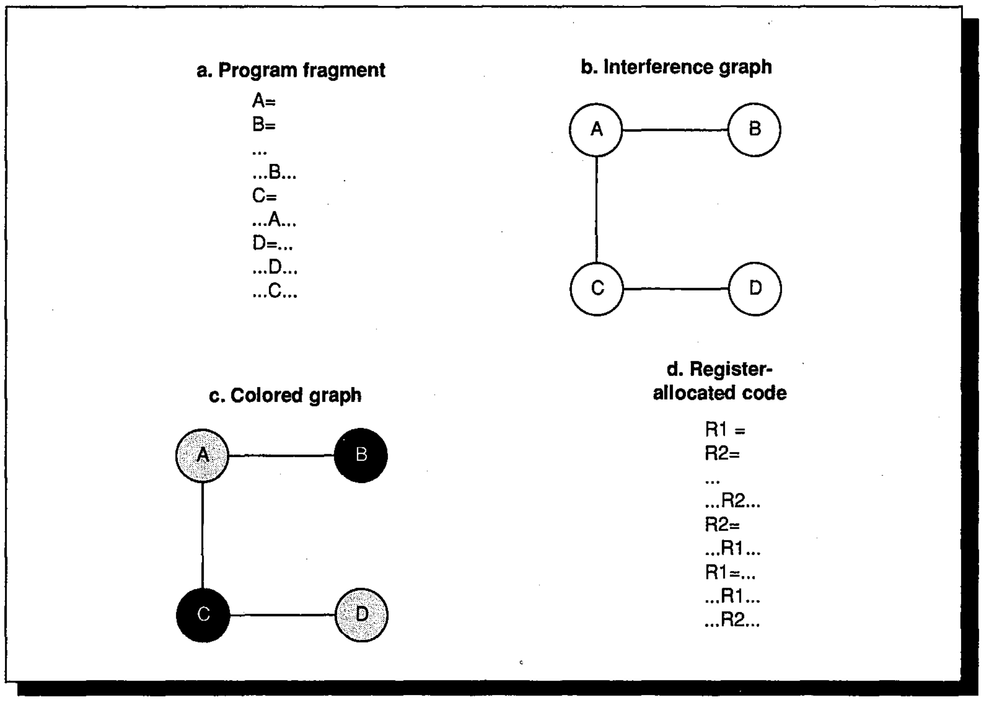

Dependencies
Language dependent; machine independent
Front-end per language
Function
Transform language to common intermediate form
Intermediate representation
Somewhat language dependent, largely machine independent
High-level optimizations
For example, procedure inlining and loop transformations
Small language dependencies; machine dependencies slight (e.g., register counts/types)
Global optimizer
Including global and local optimizations + register allocation
Highly machine dependent; language independent
Code generator
Detailed instruction selection and machine-dependent optimizations; may include or be followed by assembler

FIGURE 3.24 Current compilers typically consist of two to four passes, with more highly optimizing compilers having more passes. A *pass* is simply one phase in which the compiler reads and transforms the entire program. (The term "phase" is often used interchangeably with "pass.") The optimizing passes are designed to be optional and may be skipped when faster compilation is the goal and lower quality code is acceptable. This structure maximizes the probability that a program compiled at various levels of optimization will produce the same output when given the same input. Because the optimizing passes are also separated, multiple languages can use the same optimizing and code-generation passes. Only a new front end is required for a new language. The high-level optimization mentioned here, procedure inlining, is also called *procedure integration.*

How does this ordering of transformations interact with the instruction set architecture? A good example occurs with the optimization called *global common subexpression elimination.* This optimization finds two instances of an expression that compute the same value and saves the value of the first computation in a temporary. It then uses the temporary value, eliminating the second computation of the expression. For this optimization to be significant, the temporary must be allocated to a register. Otherwise, the cost of storing the temporary in memory and later reloading it may negate the savings gained by not recomputing the expression. There are, in fact, cases where this optimization actually slows down code when the temporary is not register allocated. Phase ordering complicates this problem, because register allocation is typically done near the end of the global optimization pass, just before code generation. Thus, an optimizer that performs this optimization **must** assume that the register allocator will allocate the temporary to a register.

Because of the central role that register allocation plays, both in speeding up the code and in making other optimizations useful, it is one of the most important—if not the most important—optimizations. Recent register allocation algorithms are based on a technique called *graph coloring*. The basic idea behind graph coloring is to construct a graph representing the possible candidates for allocation to a register and then to use the graph to allocate registers. As shown in Figure 3.25, each candidate for a register corresponds to a node in the graph, called an *interference graph*. The arcs between the nodes show where the ranges of usage for variables (called *live ranges*) overlap. The compiler then tries to color the graph using a number of colors equal to the number of registers available for allocation. In a graph coloring, no adjacent nodes may have the same color. This restriction is equivalent to saying that no two variables with overlapping uses may be allocated to the same register. However, nodes that are not connected by an arc may have the same color, allowing variables whose uses do not overlap to use the same register. Thus, a coloring of the graph corresponds to an allocation of the active variables to registers. For example, the four nodes in Figure 3.25 can be colored with two colors, meaning the code only needs two registers for allocation. Although the problem of coloring a graph is NP-complete, there are heuristic algorithms that work well in practice.



a. Program fragment

```
A=
B=
...
...B...
C=
...A...
D=...
...D...
...C...
```

d. Register-allocated code

```
R1 =
R2=
...
...R2...
R2=
...R1...
R1=...
...R1...
...R2...
```

FIGURE 3.25 Graph coloring is used to allocate registers by constructing an interference graph that is colored heuristically using a number of colors corresponding to the register count. Part b shows the interference graph corresponding to the code fragment shown in part a. Each variable corresponds to a node, and the arcs show the overlap of the active ranges of the variables. The graph can be colored with two colors, as shown in part c, and this corresponds to the register allocation of part d.

Graph coloring works best when there are at least 16 (and preferably more) general-purpose registers available for global allocation for integer variables and additional registers for floating point. Unfortunately, graph coloring does not work very well when the number of registers is small because the heuristic algorithms for coloring the graph are likely to fail. The emphasis in the approach is to achieve 100% allocation of active variables.

Optimizations performed by modern compilers can be classified by the style of the transformation, as follows:

1. High-level optimizations—often done on the source with output fed to later optimization passes.

Optimization name	**Explanation**	**Percent of the total number of optimizing transforms**
High-level	**At or near the source level; machine-independent**	
Procedure integration	Replace procedure call by procedure body	N.M.
Local	**Within straightline code**	
Common subexpression elimination	Replace two instances of the same computation by single copy	18%
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	22%
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	N.M.
Global	**Across a branch**	
Global common subexpression elimination	Same as local, but this version crosses branches	13%
Copy propagation	Replace all instances of a variable *A* that has been assigned *X* (i.e., *A*=*X*) with *X*	11%
Code motion	Remove code from a loop that computes same value each iteration of the loop	16%
Induction variable elimination	Simplify/eliminate array-addressing calculations within loops	2%
Machine-dependent	**Depends on machine knowledge**	
Strength reduction	Many examples, such as replace multiply by a constant with adds and shifts	N.M.
Pipeline scheduling	Reorder instructions to improve pipeline performance	N.M.
Branch offset optimization	Choose the shortest branch displacement that reaches target	N.M.

FIGURE 3.26 Major types of optimizations and examples in each class. The third column lists the static frequency with which some of the common optimizations are applied in a set of 12 small FORTRAN and Pascal programs. The percentage is the portion of the static optimizations that are of the specified type. These data tell us about the relative frequency of occurrence of various optimizations. There are nine local and global optimizations done by the compiler included in the measurement. Six of these optimizations are covered in the figure, and the remaining three account for 18% of the total static occurrences. The abbreviation "N.M." means that the number of occurrences of that optimization was not measured. Machine-dependent optimizations are usually done in a code generator, and none of those were measured in this experiment. The data are from Chow [1983], and were collected using the Stanford UCODE compiler.

2. Local optimizations—optimize code only within a straightline code fragment (called a *basic block* by compiler people).
3. Global optimizations—extend the local optimizations across branches and introduce a set of transformations aimed at optimizing loops.
4. Register allocation.
5. Machine-dependent optimizations—attempt to take advantage of specific architectural knowledge.

It is sometimes difficult to separate some of the simpler optimizations—local and machine-dependent optimizations—from transformations done in the code generator. Examples of typical optimizations are given in Figure 3.26. The last column of Figure 3.26 indicates the frequency with which the listed optimizing transforms were applied to the source program. Data on the effect of various optimizations on program run-time are shown in Figure 3.27. The data in Figure 3.27 demonstrate the importance of register allocation, which adds the largest single improvement. We will look at the overall effect of optimization on our three benchmarks later in this section.

Optimizations performed	**Percent faster**
Procedure integration only	10%
Local optimizations only	5%
Local optimizations + register allocation	26%
Global and local optimizations	14%
Local and global optimizations + register allocation	63%
Local and global optimizations + procedure integration + register allocation	81%

FIGURE 3.27 Performance effects of various levels of optimization. Performance gains are shown as what percent faster the optimized programs were compared to the unoptimized programs. When register allocation is turned off, data are loaded into, or stored from, the registers on every individual use. These measurements are also from Chow [1983] and are for 12 small FORTRAN and Pascal programs.

The Impact of Compiler Technology on the Architect's Decisions

The interaction of compilers and high-level languages significantly affects how programs use an instruction set. To better understand this interaction, three important questions to ask are:

1. How are variables allocated and addressed? How many registers are needed to allocate variables appropriately?
2. What is the impact of optimization techniques on instruction mixes?

3. What control structures are used and with what frequency?

To address the first questions, we must look at the three separate areas in which current high-level languages allocate their data:

- The *stack*—used to allocate local variables. The stack is grown and shrunk on procedure call or return, respectively. Objects on the stack are addressed relative to the stack pointer and are primarily scalars (single variables) rather than arrays. The stack is used for activation records, **not** as a stack for evaluating expressions. Hence values are almost never pushed or popped on the stack.
- The *global data area*—used to allocate statically declared objects, such as global variables and constants. A large percentage of these objects are arrays or other aggregate data structures.
- The *heap*—used to allocate dynamic objects that do not adhere to a stack discipline. Objects in the heap are accessed with pointers and are typically not scalars.

Register allocation is much more effective for stack-allocated objects than for global variables, and register allocation is essentially impossible for heap-allocated objects because they are accessed with pointers. Global variables and some stack variables are impossible to allocate because they are *aliased*, which means that there are multiple ways to refer to the address of a variable making it illegal to put it into a register. (All heap variables are effectively aliased.) For example, consider the following code sequence (where `&` returns the address of a variable and `*` dereferences a pointer):

```
p = &a       -- gets address of a in p
a = ...      -- assigns to a directly
*p = ...     -- uses p to assign to a
...a...      -- accesses a
```

The variable "a" could not be register allocated across the assignment to `*p` without generating incorrect code. Aliasing causes a substantial problem because it is often difficult or impossible to decide what objects a pointer may refer to. A compiler must be conservative; many compilers will not allocate **any** local variables of a procedure in a register when there is a pointer that may refer to **one** of the local variables.

After register allocation, memory traffic consists of five types of references:

1. Unallocated reference—a potentially allocatable memory reference that was not assigned to a register.
2. Global scalar—a reference to a global scalar variable not allocated to a register. These variables are usually sparsely accessed and thus rarely allocated.
3. Save/restore memory reference—a memory reference made to save or restore a register (during a procedure call) that contains an allocated variable and is not aliased.

4. A required stack reference—a reference to a stack variable that is required due to aliasing possibilities. For example, if the address of the stack variable were taken, then that variable cannot usually be register allocated. Also included in this category are any data items that were caller saved due to aliasing behavior—such as a potential reference by a called procedure.
5. A computed reference—any heap reference or any reference to a stack variable via a pointer or array index.

Figure 3.28 shows how these classes of memory traffic contribute to total memory traffic for GCC and TeX benchmark programs run with an optimizing compiler on a load/store machine and varying the number of registers used. The

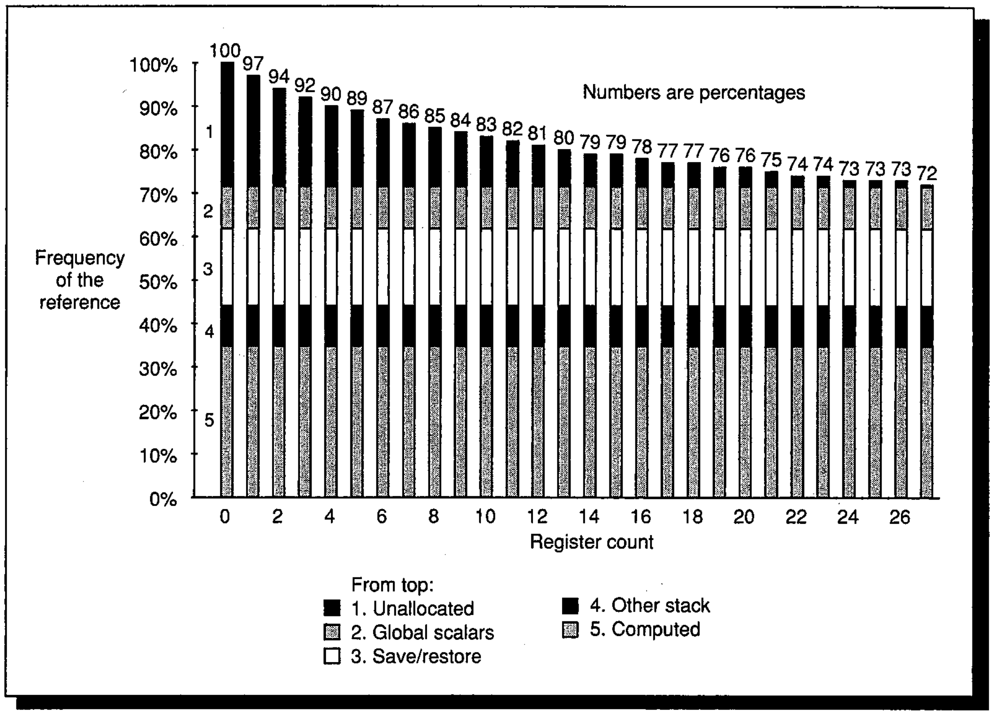


FIGURE 3.28 The percentage of memory references made to different types of variables as the register count increases. This data is averaged between TeX and GCC, which use only integer variables. The decreasing percentage represented by the top bar is the set of references that are candidates for allocation but are not actually allocated with the indicated number of registers. This data was collected for the DLX load/store machine described in the next chapter. The register allocator has 27 integer registers; the first seven integer registers capture about half of the references that can be allocated to registers. While each of the other four components contributes something to the remaining memory traffic, the dominant contribution is computed references to heap-based objects and array elements, which cannot be register allocated. Some small percentage of the required stack references may be contributed when the register allocator runs out of registers; however, from other measurements on the register allocator we know that this contribution is very small [Chow and Hennessy, 1990].

number of memory references to objects in categories 2 through 5 above is constant because they can never be allocated to registers by this compiler. (The save/restore references were measured with the full set of registers.) The number of allocatable references that are unallocated drops as the register count increases. References to objects that could be allocated but that are accessed only once are allocated by the code generator using a set of temporary registers. These references will be counted as required stack references; other allocation strategies might cause them to be treated as save/restore traffic.

The data in Figure 3.28 shows only the integer registers. The percentage of allocatable references with a given register count is computed by examining the frequency of access to registers with a compiler that generally tries to use as small a number of registers as possible. The percentage of the references captured in a given number of registers depends intimately on the compiler and its register-allocation strategy. This compiler cannot use more than the 27 integer registers available for allocating variables; additionally, some registers have a preferred use (such as those used for parameters). We cannot predict from this data how well the compiler might be able to use 100 registers. Given a substantially larger number of registers, the compiler could use the registers to reduce

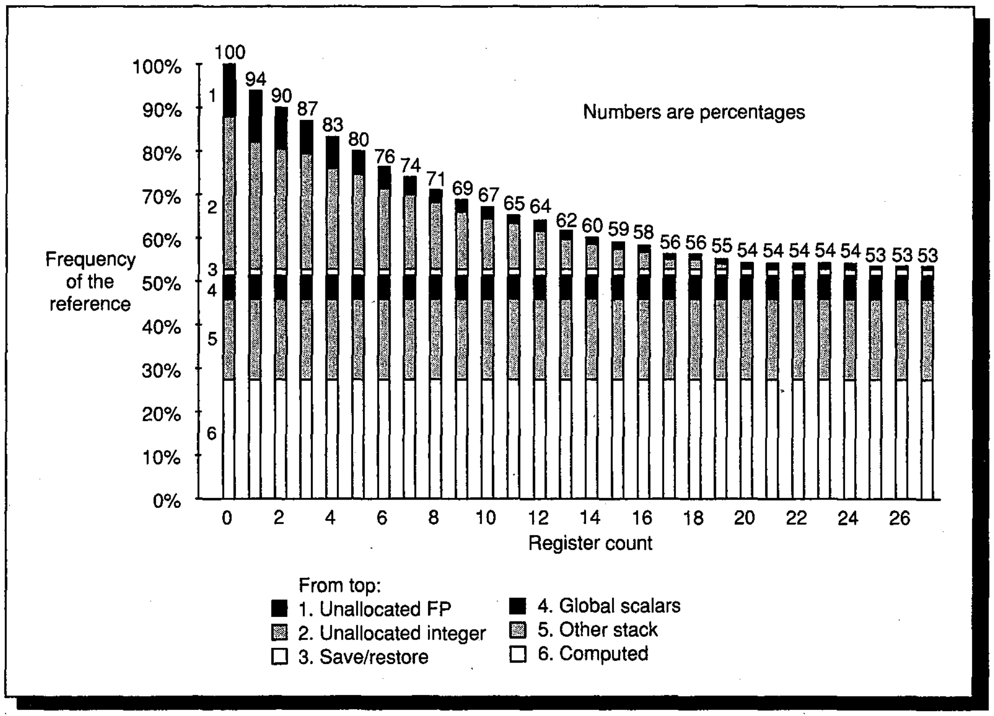


FIGURE 3.29 The percentage of references captured by the integer and floating-point register files for Spice increases to almost 50% with a full register set. Each increment on the x axis adds one integer register and one single-precision (SP), floating-point (FP) register. Thus, the point corresponding to a register count of 12 stands for 12 integer and 12 SP FP registers. Remember that most of the Spice FP data is double precision, which requires two FP registers per datum. As in Figure 3.28, about seven integer registers capture half of the integer references, but only about five registers are needed to capture half the FP references.

the save/restore memory references and the references for global scalars. However, neither class of memory references can be completely eliminated. In the past, compiler technology has made steady progress in its ability to use ever larger register sets, and we can probably expect this to continue, although the percentage of allocatable references may bound the value of larger register sets.

Figure 3.29 shows the same type of data, but this time for Spice, which uses both the integer and floating-point registers. The effect of register allocation is very different for Spice compared to GCC and TeX. First, the percentage of remaining memory traffic is smaller. This probably arises because the absence of pointers in FORTRAN makes register allocation more effective for Spice than for programs in C (i.e., GCC and TeX). Second, the amount of save/restore traffic is much lower. In addition to these differences, we can see that it takes fewer registers to capture the allocatable floating-point references. This is probably because a far smaller percentage of the FP references are allocatable, since the majority are to arrays.

Our second question concerns how an optimizer affects the mix of instructions executed. Figures 3.30 and 3.31 address this issue for the benchmarks used here. The data was taken on a load/store machine using full global optimization that includes all of the global and local optimizations listed in Figure 3.26 (page

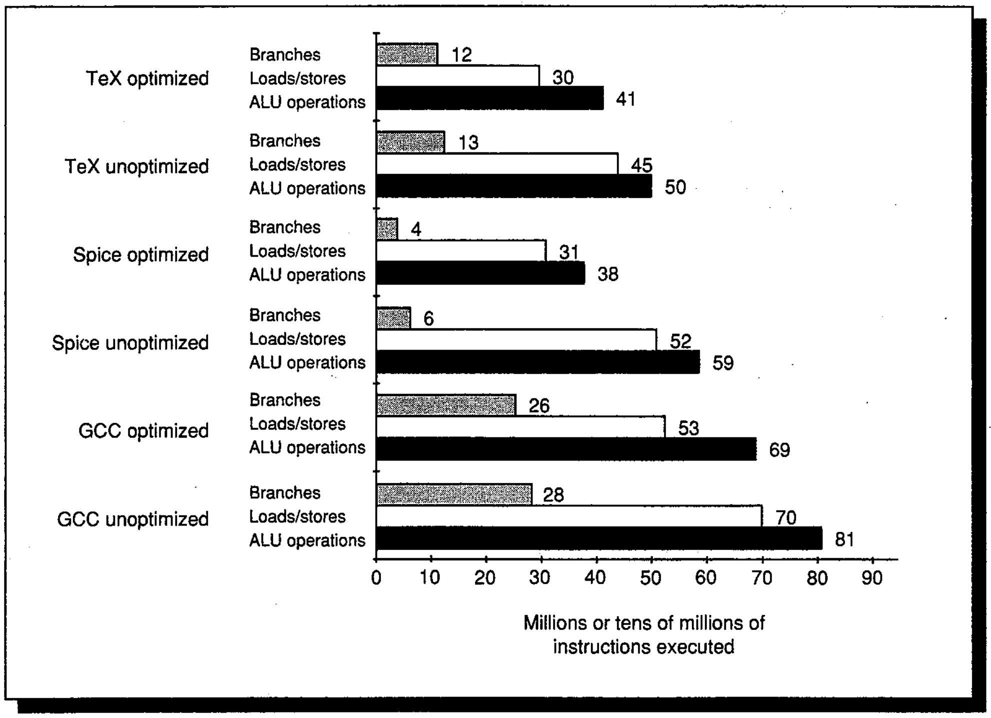


FIGURE 3.30 The effects of optimization in absolute instruction counts. The x axis is the number of instructions executed in millions for GCC and TeX and in tens of millions for Spice. The unoptimized programs execute 21%, 58%, and 30% more instructions for GCC, Spice, and TeX, respectively. This data was taken on a DECstation 3100 using –O2 optimization, as was the data in Figure 3.31. Optimizations that do not affect instruction count, but may affect instruction cycle counts, are not measured here.

114). Differences between optimized and unoptimized code are shown in both absolute and relative terms. The most obvious effect of optimization—besides decreasing the total instruction count—is to increase the relative frequency of branches by decreasing the number of memory references and ALU operations more rapidly than the number of branches (which are decreased only slightly). We show an example of how optimized and unoptimized code differ on a VAX in the Fallacies and Pitfalls section.

Finally, with what frequency are various control structures used? These are important numbers because branches are among the hardest instructions to make go fast and are very difficult to reduce with the compiler. The data in Figures 3.30 and 3.31 give us a good idea of the branch frequency—from 6.5 to 18 instructions are executed between two branches or jumps (including the branch or jump itself). Procedure calls occur about 12 to 13 times less frequently than branches, or in the range of once every 87 to 200 instructions for our programs. Spice has both the lowest percentage of branches and the fewest procedure calls per instruction by nearly a factor of two.

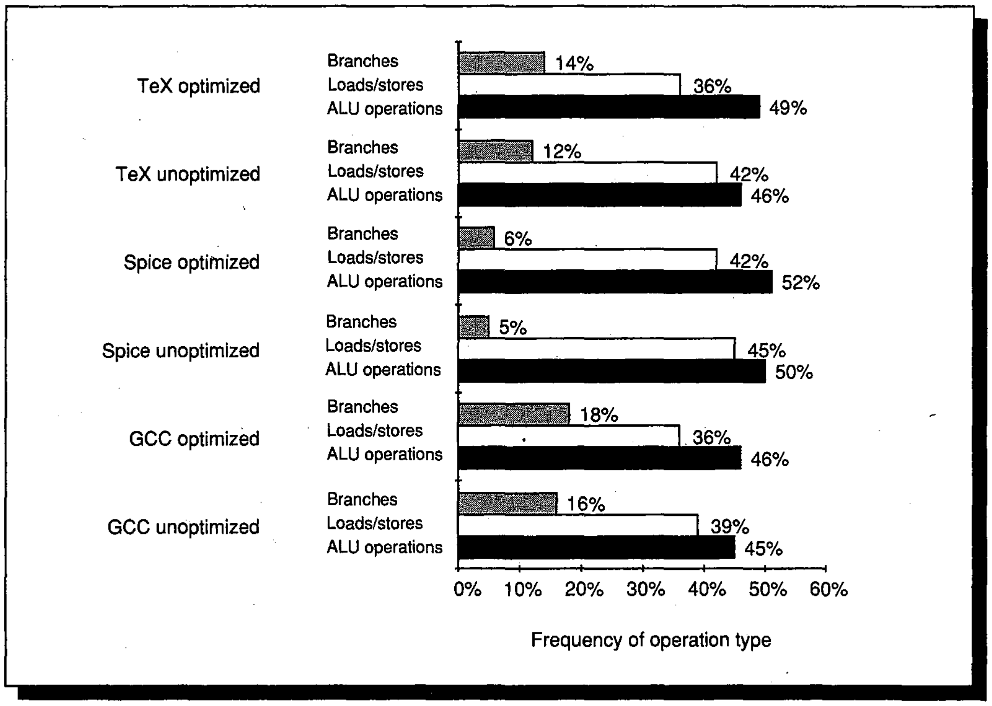


FIGURE 3.31 The effects of optimization on the relative mix of instructions for the data in Figure 3.30.

How the Architect Can Help the Compiler Writer

Today, the complexity of a compiler does not come from translating simple statements like A = B + C. Most programs are "locally simple," and simple translations work fine. Rather, complexity arises because programs are large and globally complex in their interactions, and because the structure of compilers means that decisions must be made about what code sequence is best, one step at a time.

Compiler writers often are working under their own corollary of a basic principle in architecture: "Make the frequent cases fast and the rare case correct." That is, if we know which cases are frequent and which are rare, and if generating code for both is straightforward, then the quality of the code for the rare case may not be very important—but it must be correct!

Some instruction set properties help the compiler writer. These properties should not be thought of as hard and fast rules, but rather as guidelines that will make it easier to write a compiler that will generate efficient and correct code.

1. *Regularity.* Whenever it makes sense, the three primary components of an instruction set—the operations, the data types, and the addressing modes—should be orthogonal. Two aspects of an architecture are said to be *orthogonal* if they are independent. For example, the operations and addressing modes are orthogonal if for every operation to which a certain addressing mode can be applied, all addressing modes are applicable. This helps simplify code generation and is particularly important when the decision about what code to generate is split into two passes in the compiler. A good counterexample of this property is restricting what registers can be used for a certain class of instructions. This can result in the compiler finding itself with lots of available registers, but none of the right kind!

2. *Provide primitives, not solutions.* Special features that "match" a language construct are often unusable. Attempts to support high-level languages may work only with one language, or do more or less than is required for a correct and efficient implementation of the language. Some examples of how these attempts have failed are given in Section 3.9.

3. *Simplify tradeoffs among alternatives.* One of the toughest jobs a compiler writer has is figuring out what instruction sequence will be best for every segment of code that arises. In earlier days, instruction counts or total code size might have been good metrics, but—as we saw in the last chapter—this is no longer true. With caches and pipelining, the tradeoffs have become very complex. Anything the designer can do to help the compiler writer understand the costs of alternative code sequences would help improve the code. One of the most difficult instances of complex tradeoffs occurs in a memory–memory architecture in deciding how many times a variable should be referenced before it is cheaper to load it into a register. This threshold is hard to compute and, in fact, may vary among models of the same architecture.

4. *Provide instructions that bind the quantities known at compile time as constants.* A compiler writer hates the thought of the machine interpreting at run time a value that was known at compile time. Good counterexamples of this

principle include instructions that interpret values that were fixed at compile time. For instance, the VAX procedure call instruction (CALLS) dynamically interprets a mask saying what registers to save on a call, but the mask is fixed at compile time, though in some cases it may not be known by the caller if separate compilation is used.

3.8 Putting It All Together: How Programs Use Instruction Sets

What do typical programs do? This section will investigate and compare the behavior of our benchmark programs running on a load/store architecture and on a memory–memory architecture. The compiler technology for these two different architectures differs and these differences affect the overall measurements.

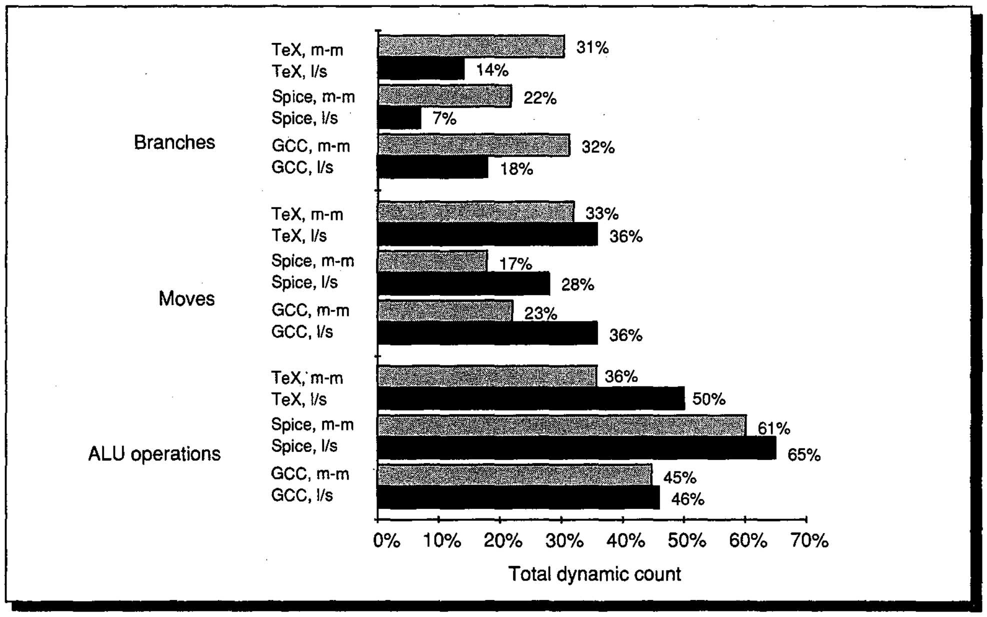


FIGURE 3.32 The instruction distributions for our benchmarks differ in straight forward ways when run on a load/store architecture (l/s) and on a memory–memory architecture (m–m). On the load/store machine, moves are loads or stores. On the memory–memory machine, moves include transfers between two locations; either of the operands may be a register or a memory location. However, the majority of the moves involve one register and a memory location. The load/store machine exhibits a higher percentage of moves because it is a load/store machine—for data to be operated on it must be moved into the registers. The lower relative frequency of branches is primarily a function of the load/store machine's use of more instructions in the other two classes. This data was measured with optimization on a VAXstation 3100 for the memory–memory machine and on DLX, which we discuss in detail in the next chapter, for the load/store machine. The input used is smaller than that in Chapter 2 to make it possible to collect the data on the VAX.

We can examine the behavior of typical programs by looking at the frequency of three basic operations: memory references, ALU operations, and control-flow instructions (branches and jumps). Figure 3.32 does this for a load/store architecture with one addressing mode (a hypothetical machine called DLX that we define in the next chapter) and for a memory–memory architecture with many addressing modes (the VAX). The load/store architecture has more registers, and its compiler places more emphasis on reducing memory traffic. Considering the enormous differences in the instruction sets of these two machines, the results are rather similar.

The same machines and programs are used in Figure 3.33, but the data represent absolute counts of instructions executed, instruction words, and data references. This chart shows a clear difference in instruction count: The load/store machine requires more instructions. Recall that this difference does not imply anything about the relative performance of machines based on these architectures.

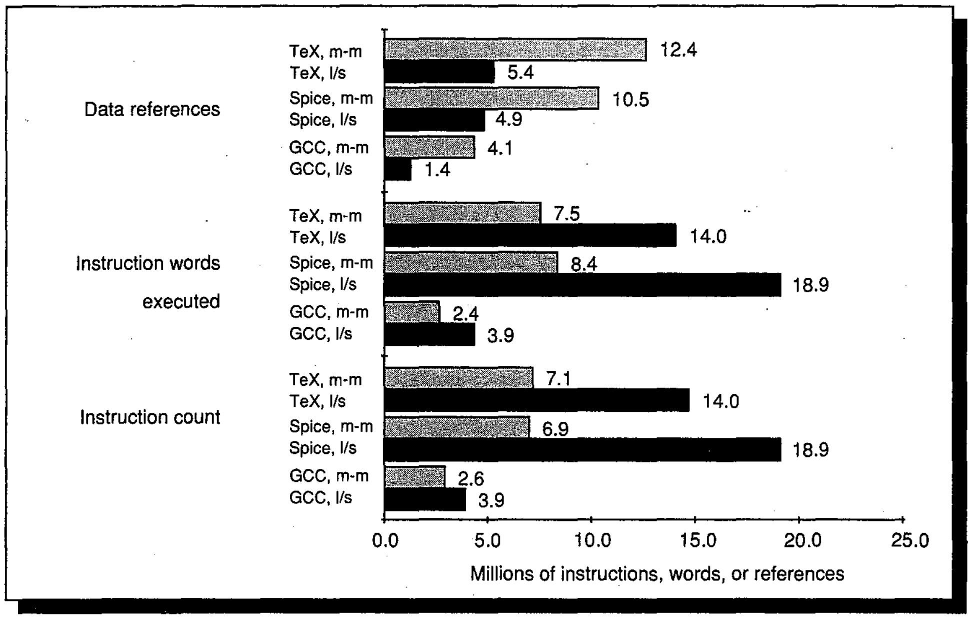


FIGURE 3.33 Absolute counts for dynamic events on a load/store and memory–memory machine. The counts are (from bottom to top) dynamic instructions, instruction words (instruction bytes divided by four), and data references (these may be byte, word, or doubleword). Each reference is counted once. Differences in the size of the register set and the compiler probably explain the large difference in the number of data references. In the case of Spice, the large difference in the total number of registers available for allocation is probably the basic reason for the large difference in total data accesses. This data was collected for the same programs, inputs, and machines as the data in Figure 3.32.

This chart also shows the number of data references made by each machine. From the data in Figure 3.32 and the instruction counts, we might guess that the total number of memory accesses made by the memory–memory machine would be much lower than the number made on the load/store machine. But the data in Figure 3.33 indicate that this hypothesis is false. The large difference in data references balances the difference in instruction references between the architectures, so that the load/store machine uses about the same memory bandwidth at the architectural level. This difference in data references probably arises because the load/store machine has many more registers, and its compiler does a better job of register allocation. For allocating integer quantities, the load/store machine has more than twice as many registers available. In total for integer and floating-point variables, more than four times as many registers are available for the compiler to use on the load/store architecture. This gap in register count combined with compiler differences is the most likely basis for the difference in data bandwidth.

We have seen how architectural measures can run counter to the designer's intuition, and that some of these measures do not directly relate to performance. In the next section we will see that architects' attempts to model machines directly after high-level software features can go awry.

3.9 Fallacies and Pitfalls

Time and again architects have tripped on common, but erroneous, beliefs. In this section we look at a few of them.

Pitfall: Designing a "high-level" instruction set feature specifically oriented to supporting a high-level language structure.

Attempts to incorporate high-level language features in the instruction set have led architects to provide powerful instructions with a wide range of flexibility. But often these instructions do more work than is required in the frequent case or don't match the requirements of the language exactly. Many such efforts have been aimed at eliminating what in the 1970s was called the "semantic gap." While the idea is to supplement the instruction set with additions that bring the hardware up to the level of the language, the additions can generate what Wulf [1981] has called a "semantic clash":

... *by giving too much semantic content to the instruction, the machine designer made it possible to use the instruction only in limited contexts.* [p. 43]

More often the instructions are simply overkill—they are too general for the most frequent case, resulting in unneeded work and a slower instruction. Again, the VAX `CALLS` is a good example. `CALLS` uses a callee-save strategy (the registers to be saved are specified by the callee) **but** the saving is done by the call instruction in the caller. The `CALLS` instruction begins with the arguments pushed on the stack, and then takes the following steps:

1. Align the stack if needed.

2. Push the argument count on the stack.
3. Save the registers indicated by the procedure call mask on the stack (as mentioned in Section 3.7). The mask is kept in the called procedure's code—this permits callee-save to be done by the caller even with separate compilation.
4. Push the return address on the stack, then push the top and base of stack pointers for the activation record.
5. Clear the condition codes, which sets the trap enables to a known state.
6. Push a word for status information and a zero word on the stack.
7. Update the two stack pointers.
8. Branch to the first instruction of the procedure.

The vast majority of calls in real programs do not require this amount of overhead. Most procedures know their argument counts and a much faster linkage convention can be established using registers to pass arguments rather than the stack. Furthermore, the call instruction forces two registers to be used for linkage, while many languages require only one linkage register. Many attempts to support procedure call and activation stack management have failed to be useful either because they do not match the language needs or because they are too general, and hence too expensive to use.

The VAX designers provided a simpler instruction, `JSB`, that is much faster since it only pushes the return PC on the stack and jumps to the procedure (see Exercise 3.11). However, most VAX compilers use the more costly `CALLS` instructions. The call instructions were included in the architecture to standardize the procedure linkage convention. Other machines have standardized their calling convention by agreement among compiler writers and without requiring the overhead of a complex, very general procedure call instruction.

Fallacy: It costs nothing to provide a level of functionality that exceeds what is required in the usual case.

A far more serious architectural pitfall than the previous one was encountered by a few machines, such as the Intel 432, that provided only a high-overhead call instruction that handled the most rare cases. The call instruction on the Intel 432 always creates a new, protected context, and thus is fairly costly (see Chapter 8 for a further discussion on memory protection). However, most calls are within the same module and do not require a protected call. If a simpler call mechanism were available and used when possible, Dhrystone would run 20% faster on the 432 (see Colwell, et al. [1985]). When architects choose to have only a general and expensive instruction, compiler writers have no choice but to use the costly instruction, and suffer the unneeded overhead. A discussion of the experience of designers with providing fine-grain protection domains in hardware appears in the historical section of Chapter 8; the discussion further illustrates this fallacy.

Pitfall: Using a nonoptimizing compiler to measure the instruction set usage made by an optimizing compiler.

The instruction set usage of an optimizing and nonoptimizing compiler may be quite different. We saw some examples in Figure 3.31 (page 120). Figure 3.34 shows the differences in the use of addressing modes on a VAX for Spice, when it is compiled with the nonoptimizing UNIX F77 compiler and when it is compiled with DEC's optimizing FORTRAN compiler.

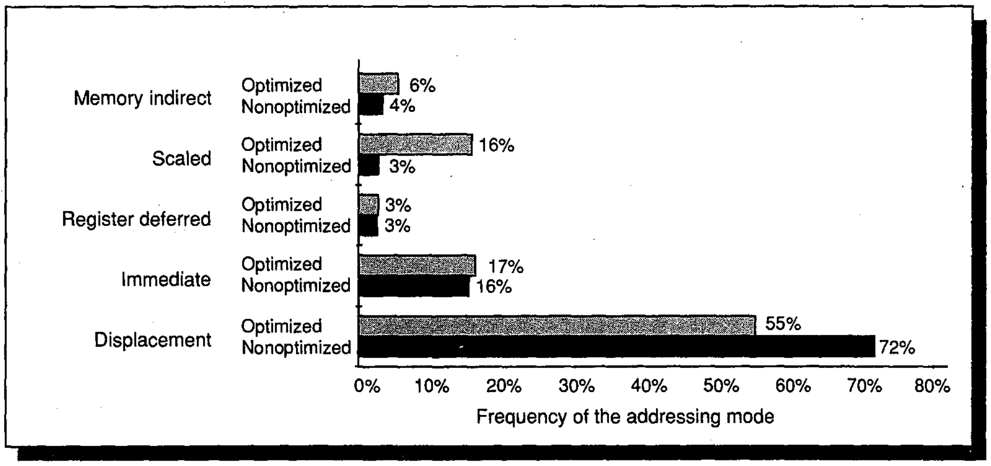


FIGURE 3.34 The address mode usage by an optimizing and nonoptimizing compiler can differ significantly. These measurements show the use of VAX addressing modes by Spice when it is compiled using a nonoptimizing compiler (f77) and an optimizing compiler (fort). In particular, the use of scaled mode is much higher for the optimizing compiler. The other VAX addressing modes account for the remaining 2-3% of the data memory references.

3.10 Concluding Remarks

The earliest architectures were limited in their instruction sets by the hardware technology of that time. As soon as the hardware technology permitted, architects began looking for ways to support high-level languages. This search led to three distinct periods of thought about how to support programs efficiently. In the 1960s, stack architectures became popular. They were viewed as being a good match for high-level languages—and they probably were, given the compiler technology of the day. In the 1970s, the main concern of architects was how to reduce software costs. This concern was met primarily by replacing software with hardware, or by providing high-level architectures that could simplify the task of software designers. The result was both the high-level–language computer architecture movement and powerful architectures like the VAX, which has a large number of addressing modes, multiple data types, and a highly orthogonal architecture. In the 1980s, more sophisticated compiler technology and a renewed emphasis on machine performance has seen a return to simpler architectures, based mainly on the load/store style of machine. Continuing changes in how we program, the compiler technology we use, and the underlying hardware technology will no doubt make another direction more attractive in the future.

3.11 Historical Perspective and References

One's eyebrows should rise whenever a future architecture is developed with a stack- or register-oriented instruction set.

Meyers [1978, 20]

The earliest computers, including the UNIVAC I, the EDSAC, and the IAS machines, were accumulator-based machines. The simplicity of this type of machine made it the natural choice when hardware resources were very constrained. The first general-purpose register machine was the Pegasus, built by Ferranti, Ltd. in 1956. The Pegasus had eight general-purpose registers, with R0 always being zero. Block transfers loaded the eight registers from the drum.

In 1963, Burroughs delivered the B5000. The B5000 was perhaps the first machine to seriously consider software and hardware-software tradeoffs. Barton and the designers at Burroughs made the B5000 a stack architecture (as described in Barton [1961]). Designed to support high-level languages such as ALGOL, this stack architecture used an operating system (MCP) written in a high-level language. The B5000 was also the first machine from a US manufacturer to support virtual memory. The B6500, introduced in 1968 (and discussed in Hauck and Dent [1968]), added hardware-managed activation records. In both the B5000 and B6500, the top two elements of the stack were kept in the CPU and the rest of the stack was kept in memory. The stack architecture yielded good code density, but only provided two high-speed storage locations. The authors of both the original IBM 360 paper [Amdahl et al. 1964] and the original PDP-11 paper [Bell et al. 1970] argue against the stack organization. They cite three major points in their arguments against stacks:

1. Performance is derived from fast registers, not the way they are used.
2. The stack organization is too limiting and requires many swap and copy operations.
3. The stack has a bottom, and when placed in slower memory there is a performance loss.

Stack-based machines fell out of favor in the late 1970s and essentially disappeared in the 1980s.

The term "computer architecture" was coined by IBM in 1964 for use with the IBM 360. Amdahl, Blaauw, and Brooks [1964] used the term to refer to the programmer-visible portion of the instruction set. They believed that a family of machines of the same architecture should be able to run the same software. Although this idea may seem obvious to us today, it was quite novel at that time. IBM, even though it was the leading company in the industry, had **five** different architectures before the 360. Thus, the notion of a company standardizing on a single architecture was a radical one. The 360 designers hoped that six different divisions of IBM could be brought together by defining a common architecture. Their definition of architecture was

... the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine.

The term "machine language programmer" meant that compatibility would hold, even in assembly language, while "timing independent" allowed different implementations.

The IBM 360 was the first machine to sell in large quantities with both byte-addressing using 8-bit bytes and general purpose registers. The 360 also had register–memory and limited memory–memory instructions.

In 1964, Control Data delivered the first supercomputer, the CDC 6600. As discussed in Thornton [1964], he, Cray, and the other 6600 designers were the first to explore pipelining in depth. The 6600 was the first general-purpose, load/store machine. In the 1960s, the designers of the 6600 realized the need to simplify architecture for the sake of efficient pipelining. This interaction between architectural simplicity and implementation was largely neglected during the 1970s by microprocessor and minicomputer designers, but was brought back in the 1980s.

In the late 1960s and early 1970s, people realized that software costs were growing faster than hardware costs. McKeeman [1967] argued that compilers and operating systems were getting too big and too complex and taking too long to develop. Because of inferior compilers and the memory limitations of machines, most systems programs at the time were still written in assembly language. Many researchers proposed alleviating the software crisis by creating more powerful, software-oriented architectures. Tanenbaum [1978] studied the properties of high-level languages. Like other researchers, he found that most programs are simple. He then argued that architectures should be designed with this in mind and should optimize program size and ease of compilation. Tanenbaum proposed a stack machine with frequency-encoded instruction formats to accomplish these goals. However, as we have observed, program size does not translate directly to cost/performance, and stack machines faded out shortly after this work.

Strecker's article [1978] discusses how he and the other architects at DEC responded to this by designing the VAX architecture. The VAX was designed to simplify compilation of high-level languages. Compiler writers had complained about the lack of complete orthogonality in the PDP-11. The VAX architecture was designed to be highly orthogonal and to allow the mapping of a high-level–

language statement into a single VAX instruction. Additionally, the VAX designers tried to optimize code size because compiled programs were often too large for available memories. When it was introduced in 1978, the VAX was the first machine with a true memory–memory architecture.

While the VAX was being designed, a more radical approach, called *High-Level Language Computer Architecture* (HLLCA), was being advocated in the research community. This movement aimed to eliminate the gap between high-level languages and computer hardware—what Gagliardi [1973] called the "semantic gap"—by bringing the hardware "up to" the level of the programming language. Meyers [1982] provides a good summary of the arguments and a history of high-level–language computer architecture projects.

Smith, Rice, and their colleagues [1971] discuss the SYMBOL Project they started at Fairchild. SYMBOL became the largest and most famous of the HLLCA attempts. Its goal was to build a high-level–language, timesharing machine that would dramatically reduce programming time. The SYMBOL machine interpreted programs, written in its own new programming language, directly; the compiler and operating system were built into the hardware. The programming language was very dynamic—there were no variable declarations because the hardware interpreted every statement dynamically.

SYMBOL suffered from many problems, the most important of which were inflexibility, complexity, and performance. The SYMBOL hardware included the programming language, the operating system, and even the text editor. Programmers had no choice in what programming language they used, so subsequent advances in operating systems and programming languages could not be incorporated. The machine was also complicated to design and to debug. Because hardware was used for everything, rare and complex cases needed to be handled completely in hardware, as well as the simpler, more common cases.

Ditzel [1980] observed that SYMBOL had enormous performance problems. While exotic cases ran relatively fast, simple and common cases often ran slowly. Many memory references were needed to interpret a simple statement in a program. While the goal of eliminating the semantic gap seemed like a worthy one, any one of the three problems faced by SYMBOL would have been enough to doom the approach.

HLLCA never had a significant commercial impact. The increase in memory size on machines and the use of virtual memory eliminated the code-size problems arising from high-level languages and operating systems written in high-level languages. The combination of simpler architectures together with software offered greater performance and more flexibility at lower cost and lower complexity.

Studies of instruction set usage began in the late 1950s. The Gibson mix, described in the last chapter, was derived as a study of instruction usage on the IBM 7090. There were several studies in the 1970s of instruction set usage. Among the best known are Foster et al. [1971] and Lunde [1977]. Most of these early studies used small programs because the techniques used to collect data were expensive. Starting in the late 1970s, the area of instruction set measurement and analysis became very active. Because we use data from most of these papers in the next chapter, we will review the contributions there.

Other studies in the 1970s examined the usage of programming-language features. Though many of these studied only static properties, papers by Alexander and Wortman [1975] and Elshoff [1976] studied the dynamic properties of HLL programs. Interest in compiler utilization of instruction sets and interaction between compilers and architecture grew in the 1980s. A conference focusing on the interaction between software systems and hardware systems, called Architectural Support for Programming Languages and Operating Systems (ASPLOS), was created. Many papers on instruction set measurement and interaction between compilers and architectures have been published in this biannual conference.

In the early 1980s, the direction of computer architecture began to swing away from providing high-level hardware support for languages. Ditzel and Patterson [1980] analyzed the difficulties encountered by the high-level–language architectures and argued that the answer lay in simpler architectures. In another paper [Patterson and Ditzel 1980], these authors first discussed the idea of reduced instruction set computers (RISC) and presented the argument for simpler architectures. Their proposal was rebutted by Clark and Strecker [1980]. We will talk more about the effect of these ideas in the next chapter.

About the same time, other researchers published papers that argued for a closer coupling of architectures and compilers, rather than attempts to supplement compilers. These included Wulf [1981], and Hennessy and his colleagues [1982].

The early compiler technology developed for FORTRAN was quite good. Many of the optimization techniques in use in today's compilers were developed and implemented by the late 1960s or early 1970s (see Cocke and Schwartz [1970]). Because FORTRAN had to compete with assembly language, there was tremendous pressure for efficiency in FORTRAN compilers. However, once the benefits of HLL programming were obvious, focus shifted away from optimizing technology. Much of the optimization work in the 1970s was theoretically oriented rather than experimental. In the early 1980s, there was a new focus on developing optimizing compilers. As this technology stabilized, several researchers wrote papers examining the impact of various compiler optimizations on program execution time. Cocke and Markstein [1980] describe the measurements using the IBM PL.8 compiler; Chow [1983] describes the gain obtained with the Stanford UCODE compiler for a variety of machines. As we saw in this chapter, register allocation is the backbone of modern optimizing compilers. The formulation of register allocation as a graph-coloring problem was originally done by Chaitin and his colleagues [1982]. Chow and Hennessy [1984, 1990] extended the algorithm to use priorities in choosing the quantities to allocate. The progress in optimization and register allocation has led to more widespread use of optimizing compilers, and the impact of compiler technology on architectural tradeoffs has increased considerably in the past decade.

References

ALEXANDER, W. G. AND D. B.WORTMAN [1975]. "Static and dynamic characteristics of XPL programs," *Computer* 8:11 (November) 41–46.

AMDAHL, G. M., G. A. BLAAUW, AND F. P. BROOKS, JR. [1964]. "Architecture of the IBM System 360," *IBM J. Research and Development* 8:2 (April) 87–101.

BARTON, R. S. [1961]. "A new approach to the functional design of a computer," *Proc. Western Joint Computer Conf.*, 393–396.

BELL, G., R. CADY, H. MCFARLAND, B. DELAGI, J. O'LAUGHLIN, R. NOONAN, AND W. WULF [1970]. "A new architecture for mini-computers: The DEC PDP-11," *Proc. AFIPS SJCC*, 657–675.

CHAITIN, G. J., M. A. AUSLANDER, A. K. CHANDRA, J. COCKE, M. E. HOPKINS, AND P. W. MARKSTEIN [1982]. "Register allocation via coloring," *Computer Languages* 6, 47–57.

CHOW, F. C. [1983]. *A Portable Machine-Independent Global Optimizer—Design and Measurements,* Ph. D. Thesis, Stanford Univ. (December).

CHOW, F. C. AND J. L. HENNESSY [1984]. "Register allocation by priority-based coloring," *Proc. SIGPLAN '84 Compiler Construction* (*ACM SIGPLAN Notices 19:6* June) 222–232.

CHOW, F. C. AND J. L. HENNESSY [1990]. "The Priority-Based Coloring Approach to Register Allocation," *ACM Trans. on Programming Languages and Systems* 12:4 (October).

CLARK, D. AND W. D. STRECKER [1980]. "Comments on 'the case for the reduced instruction set computer', " *Computer Architecture News* 8:6 (October) 34–38.

COCKE, J., AND J. MARKSTEIN [1980]. "Measurement of code improvement algorithms," *Information Processing* 80, 221–228.

COCKE, J. AND J. T. SCHWARTZ [1970]. *Programming Languages and Their Compilers*, Courant Institute, New York Univ., New York City.

COHEN, D. [1981]. "On holy wars and a plea for peace," *Computer* 14:10 (October) 48–54.

COLWELL, R. P, C. Y. HITCHCOCK, III, E. D. JENSEN, H. M. B. SPRUNT, AND C. P. KOLLAR, [1985]. "Computers, complexity, and controversy," *Computer* 18:9 (September) 8–19.

DITZEL, D. R. [1981]. "Reflections on the high-level language Symbol computer system," *Computer* 14:7 (July) 55–66.

DITZEL, D. R. AND D. A. PATTERSON [1980]. "Retrospective on high-level language computer architecture," in *Proc. Seventh Annual Symposium on Computer Architecture,* La Baule, France (June) 97–104.

ELSHOFF, J. L. [1976]. "An analysis of some commercial PL/I programs," *IEEE Trans. on Software Engineering* SE-2 2 (June) 113–120.

FOSTER, C. C., R. H. GONTER, AND E. M. RISEMAN [1971]. "Measures of opcode utilization," *IEEE Trans. on Computers* 13:5 (May) 582–584.

GAGLIARDI, U. O. [1973]. "Report of workshop 4–software-related advances in computer hardware," *Proc. Symposium on the High Cost of Software*, Menlo Park, Calif., 99–120.

HAUCK, E. A., AND B. A. DENT [1968]. "Burroughs' B6500/B7500 stack mechanism," *Proc. AFIPS SJCC*, 245–251.

HENNESSY, J. L., N. JOUPPI, F. BASKETT, T. R. GROSS, AND J. GILL [1982]. "Hardware/software tradeoffs for increased performance," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems* (March), 2–11.

LUNDE, A. [1977]. "Empirical evaluation of some features of instruction set processor architecture," *Comm. ACM* 20:3 (March) 143–152.

MCKEEMAN, W. M. [1967]. "Language directed computer design," *Proc. 1967 Fall Joint Computer Conf.,* Washington, D.C., 413–417.

MEYERS, G. J. [1978]. "The evaluation of expressions in a storage-to-storage architecture," *Computer Architecture News* 7:3 (October), 20–23.

MEYERS, G. J. [1982]. *Advances in Computer Architecture*, 2nd ed., Wiley, N.Y.

PATTERSON, D. A. AND D. R. DITZEL [1980]. "The case for the reduced instruction set computer," *Computer Architecture News* 8:6 (October) 25–33.

SMITH, W. R., R. R. RICE, G. D. CHESLEY, T. A. LALIOTIS, S. F. LUNDSTROM, M. A. CHALHOUN, L. D. GEROULD, AND T. C. COOK [1971]. "SYMBOL: A large experimental system exploring major hardware replacement of software," *Proc. AFIPS Spring Joint Computer Conf.,* 601–616.

STRECKER, W. D. [1978]. "VAX-11/780: A virtual address extension of the PDP-11 family," *Proc. AFIPS National Computer Conf.* 47, 967–980.

TANENBAUM, A. S. [1978]. "Implications of structured programming for machine architecture," *Comm. ACM* 21:3 (March) 237–246.

THORNTON, J. E. [1964]. "Parallel operation in Control Data 6600," *Proc. AFIPS Fall Joint Computer Conf.* 26, part 2, 33–40.

WILKES, M. V. [1982]. "Hardware support for memory protection: Capability implementations," *Proc. Conf. on Architectural Support for Programming Languages and Operating Systems* (March) 107–116.

WILKES, M. V. AND W. RENWICK [1949]. *Report of a Conf. on High Speed Automatic Calculating Machines,* Cambridge, England.

WULF, W. [1981]. "Compilers and computer architecture," *Computer* 14:7 (July) 41–47.

EXERCISES

3.1 [15/10] <3.7> Use the data in Figures 3.30 and 3.31 (pages 119–120) for GCC for this problem. Assume the following CPIs:

ALU operation	1
Load/store	3
Branch	5

a. [15] Find the CPI for the optimized and unoptimized versions of GCC.

b. [10] How much faster is the optimized program than the unoptimized program?

3.2 [15/15/10] <3.8> Use the data in Figure 3.33 (page 123), in this problem. Assume that each instruction word and each data reference require one memory access.

a. [15] Determine the percentage of memory accesses that are for instructions for each of the three benchmarks on the load/store machine.

b. [15] Determine the percentage of memory accesses that are for instructions for each of the three benchmarks on the memory–memory machine.

c. [10] What is the ratio of total memory accesses on the load/store machine versus the memory–memory machine for each benchmark?

3.3 [20/15/10] <3.3, 3.8> We are designing instruction set formats for a load/store architecture and are trying to decide whether it is worthwhile to have multiple offset lengths for branches and memory references. We will use average measurements for the three benchmarks to make this decision. We have decided that the offsets will be the same for these two classes of instructions. The length of an instruction would be equal to 16 bits + offset length in bits. ALU instructions will be 16 bits. Figure 3.35 contains the data from Figures 3.13 (page 100) and 3.19 (page 106) averaged and put in cumulative form. Assume an additional bit is needed for the sign on the offset.

For instruction set frequencies, use the data from the average of the three benchmarks for the load/store machine in Figure 3.32 (page 122).

Offset bits	Cumulative data references	Cumulative branches
0	16%	0%
1	16%	0%
2	21%	10%
3	29%	27%
4	32%	47%
5	44%	66%
6	55%	79%
7	62%	89%
8	66%	94%
9	68%	97%
10	73%	99%
11	78%	100%
12	80%	100%
13	86%	100%
14	87%	100%
15	100%	100%

FIGURE 3.35 The second and third columns contain the cumulative percentage of the data references and branches, respectively, that can be accommodated with the corresponding number of bits of magnitude in the displacement (i.e., the sign-bit is not included). This data is derived by averaging and accumulating the data in Figures 3.13 and 3.19.

a. [20] Suppose offsets were permitted to be 0, 8, or 16 bits in length including the sign-bit. Based on the dynamic statistics in Figure 3.32, what is the average length of an executed instruction?

b. [15] Suppose we wanted a fixed-length instruction and we chose a 24-bit instruction length (for everything, including ALU instructions). For every offset of longer than 8 bits, an additional instruction is required. Determine the number of instruction bytes fetched in this machine with fixed instruction size versus those fetched with a variable-sized instruction.

c. [10] What if the offset length were 16 and we never required an additional instruction? How would instruction bytes fetched compare to the choice of only an 8-bit offset? Assume ALU instructions will be 16 bits.

3.4 [15/10] <3.2> Several researchers have suggested that adding a register–memory addressing mode to a load/store machine might be useful. The idea is to replace sequences of

```
LOAD   R1,0(Rb)
ADD    R2,R2,R1
```

by

```
ADD    R2,0(Rb)
```

Assume the new instruction will cause the clock cycle to increase by 10%. Use the instruction frequencies for the GCC benchmark on the load/store machine from Figure 3.32 (page 122) and assume that two-thirds of the moves are loads and the rest are stores. The new instruction affects only the clock speed and not the CPI.

a. [15] What percentage of the loads must be eliminated for the machine with the new instruction to have at least the same performance?

b. [12] Show a situation in a multiple instruction sequence where a load of R1 followed immediately by a use of R1 (with some type of opcode) could not be replaced by a single instruction of the form proposed, assuming that the same opcode exists.

3.5 [15/20] <3.1–3.3> For the next two parts of this question, your task is to compare the memory efficiency of four different styles of instruction sets for two code sequences. The architecture styles are:

Accumulator

Memory–Memory—All three operands of each instruction are in memory.

Stack—All operations occur on top of the stack. Only push and pop access memory, and all other instructions remove their operands from stack and replace them with the result. The implementation uses a stack for the top two entries; accesses that use other stack positions are memory references.

Load/store—All operations occur in registers, and register-to-register instructions have three operands per instruction. There are 16 general-purpose registers, and register specifiers are 4 bits long.

To measure memory efficiency, make the following assumptions about all four instruction sets:

- The opcode is always 1 byte (8 bits).
- All memory addresses are 2 bytes (16 bits).
- All data operands are 4 bytes (32 bits).
- All instructions are an integral number of bytes in length.

There are no other optimizations to reduce memory traffic, and the variables A, B, C, and D are initially in memory.

Invent your own assembly language mnemonics and write the best equivalent assembly language code for the high-level–language fragments given.

a. [15] Write the four code sequences for

```
A = B + C;
```

For each code sequence, calculate the instruction bytes fetched and the memory-data bytes transferred. Which architecture is most efficient as measured by code size? Which architecture is most efficient as measured by total memory bandwidth required (code + data)?

b. [20] Write the four code sequences for

```
A = B + C;
B = A + C;
D = A - B;
```

For each code sequence, calculate the instruction bytes fetched and the memory-data bytes transferred (read or written). Which architecture is most efficient as measured by code size? Which architecture is most efficient as measured by total memory bandwidth required (code + data)? If the answers are different from part a, why are they different?

3.6 [20] <3.4> Supporting byte and halfword access requires an alignment network, as in Figure 3.10 (page 97). Some machines have only word accesses, so that a load of a byte or halfword takes two instructions (a load and an extract), and a partial word store takes three instructions (load, insert, store). Use the data for the TeX benchmark from Figure 3.23 (page 110) to determine what percentage of the accesses are to byte or halfwords, and use the data from TeX on the load/store machine from Figure 3.32 (page 122) to find the frequency of data transfers. Assume that loads are twice as frequent as stores independent of the data size. If all instructions on the machine take one cycle, what increase in the clock rate must we obtain to make eliminating partial word accesses a good tradeoff?

3.7 [20] <3.3> We have a proposal for three different machines, M_0, M_8, and M_{16}, that differ in their register count. All three machines have three operand instructions, and any operand can be either a memory reference or a register. The cost of a memory operand on these machines is six cycles and the cost of a register operand is one cycle. Each of the three operands has equal probability of being in a register.

The differences among the machines are described in the following table. The execution cycles per operation are in addition to the cost of operand access. The probability of an operand being in a register applies to each operand individually and is based on Figures 3.28 (page 117) and 3.29 (page 118).

Machine	Register count	Execution cycles per operation ignoring operand accesses	Probability of an operand being in a register as opposed to memory
M_0	0	4 cycles	0.0
M_8	8	5 cycles	0.5
M_{16}	16	6 cycles	0.8

What is the cycle count for an average instruction on each machine?

3.8 [15/10/10] <3.3, 3.7> One place where an architect can drive a compiler writer crazy is in making it difficult to tell if a compiler "optimization" may slow down a program on the machine.

Consider an access to A[*i*], where A is an array of integers and *i* is an integer offset in a register. We wish to generate code to use the value of A[*i*] as a source operand throughout this problem. Assume that all instructions take one clock cycle plus the cost of the memory addressing mode:

- Indexed addressing costs four clock cycles for the memory reference (for a total of five clock cycles for the instruction).
- Register indirect addressing costs three clock cycles for the memory reference (for a total of four clock cycles).
- Register-register instructions have no memory access cost, requiring only one cycle.

Assume that the value A[*i*] must be stored in memory at the end of the code sequence and that the base address of A is already in R1 and the value of *i* is in R2.

a. [15] Assume that the array element A[*i*] cannot be kept in a register, but the address of A[*i*] may be kept in a register once computed. Then, there are two different methods to access A[*i*]:

   (1) compute the address of A[*i*] into a register and use register indirect, and

   (2) use the indexed addressing mode.

   Write the code sequence for both methods. How many references to A[*i*] must occur for method 1 to be better?

b. [10] Suppose you choose method 1, but you ran out of registers and had to save the address of A[*i*] on the stack and restore it. How many references must occur now for method 1 to be better?

c. [10] Suppose that the value A[*i*] can be kept in a register (versus just the address of A[*i*]). How many references must occur to make this the best approach versus using method 2?

3.9 [Discussion] <3.2–3.8> What are the **economic** arguments (i.e., more machines sold) **for and against** changing instruction set architecture?

3.10 [25] <3.1–3.3> Find an instruction set manual for some older machine (libraries and private bookshelves are good places to look). Summarize the instruction set with the discriminating characteristics used in Figures 3.1 and 3.5 (pages 90 and 93). Does the machine fit nicely into one of the categories shown in Figures 3.4 and 3.6 (pages 92 and 94)? Write the code sequence for this machine for the statements in both parts of Exercise 3.5.

3.11 [30] <3.7, 3.9> Find a machine that has a powerful instruction set feature, such as the `CALLS` instruction on the VAX. Replace the powerful instruction with a simpler sequence that accomplishes what is needed. Measure the resultant running time. How do the two compare? Why might they be different? In the early 1980s, engineers at DEC did a quick experiment to evaluate the impact of replacing `CALLS`. They found a 30% improvement in run time on a very call-intensive program when the `CALLS` was simply replaced (parameters remained on the stack). How do your results compare?

The emphasis on performance rather than aesthetics is deliberate. Without an interest in performance the study of architecture is a sterile exercise, since all computable problems can be solved using trivial architectures, given enough time. The challenge is to design computers that make the best use of available technology; in doing so we may be assured that every increase in processing speed can be used to advantage in current problems or will make previously impractical problems tractable.

Leonard J. Shustek, *Analysis and Performance of Computer Instruction Sets* (1978)

4.1 Instruction Set Measurements: What and Why **139**

4.2 The VAX Architecture **142**

4.3 The 360/370 Architecture **148**

4.4 The 8086 Architecture **153**

4.5 The DLX Architecture **160**

4.6 Putting It All Together: Measurements of Instruction Set Usage **167**

4.7 Fallacies and Pitfalls **183**

4.8 Concluding Remarks **185**

4.9 Historical Perspective and References **186**

Exercises **191**

4 Instruction Set Examples and Measurements of Use

4.1 Instruction Set Measurements: What and Why

In this chapter we will be examining some specific architectures and then detailed measurements of the architectures. Before doing so, however, let's discuss what we might want to measure and why, as well as how to measure it.

To understand performance, we are usually most interested in *dynamic measurements*—measurements that are made by counting the number of occurrences of an event during execution. Some measurements, such as code size, are inherently *static measurements,* which are made on a program independent of execution. Static and dynamic measurements may differ dramatically, as shown in Figure 4.1—using only the static data for this program would be significantly misleading. Throughout this text the data given is dynamic, unless otherwise specified. Exceptions are when only static measurements make sense (as with code size—the most important use of static measurement) and when it is interesting to compare static and dynamic measurements. As we will see in Fallacies and Pitfalls and the Exercises, the dynamic frequency of occurrence of two instructions and the time spent on those two instructions can sometimes be very different.

Our primary focus in this chapter will be on introducing the architectures and measuring instruction usage for each architecture. Although this suggests a concentration on opcodes, we will also examine addressing mode and instruction format usage.

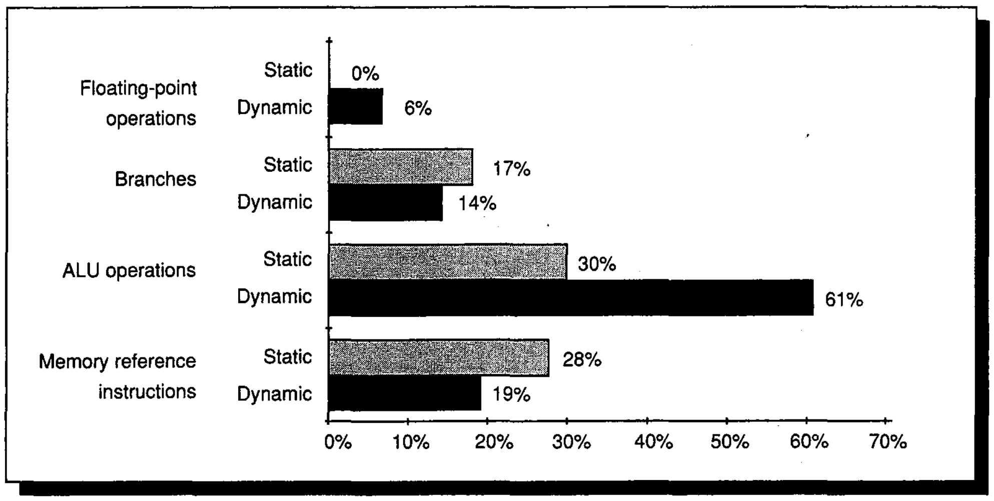


FIGURE 4.1 Data from a measurement of the IBM 360 FORTRAN benchmark, which we describe in detail in Section 4.6. The top 20 dynamically executed instructions have been broken into four wide classes, showing how different the static and dynamic occurrences can be. In the case of the dynamic measurements, these 20 instructions account for nearly 100% of the instruction executions, but only 75% of the static instruction occurrences.

The instruction set measurements in Section 4.6 can be used in two ways. At a high level, the measurements allow one to form broad approximations of the instruction usage patterns within each architectural approach. For example, we will see that "PC-changing" instructions for a powerful instruction set like the VAX average nearly 25% of all instruction executions. This tells us that techniques that try to optimize the fetching of the next sequential instruction (instruction prefetch buffers—discussed in Section 8.7 of Chapter 8) will be significantly limited because every fourth instruction is a branch. The data on the IBM 360 will show that the use of decimal and string instructions is almost nonexistent in programs written in languages other than COBOL. This leads us to conclude that support for such operations need not be included in a machine targeted at the scientific market. Measurements of the frequency of memory operands—about 40% of the operands on the 8086—can be used in the design of both the pipeline and the cache. This type of high-level, general measurement is background data that a computer architect will use on an almost daily basis.

The other purpose of such measurements is to serve as the knowledge data base that an architect would use in making detailed design tradeoffs. Such tradeoffs would be required in choosing what to include in an instruction set and what to omit, or in implementing a defined instruction set and choosing what cases to try to make fast. For example, the low frequency of use for the memory-indirect addressing modes on the VAX might encourage the architect to omit this addressing mode from a new architecture. If he was implementing a VAX, he would know that the performance penalty for disfavoring this complex addressing mode would be small. Another example that would use detailed information might be the evaluation of branching based on condition codes. By looking at the frequency of conditional branches and instructions whose only function is to set the condition code, we can evaluate the frequency with which the condition code is set implicitly (about 35% of the occurrences on the VAX). We could use this value to decide what kind of conditional branches to design in a new architecture, or we could use the data to optimize the implementation of conditional branches in a VAX. In this chapter and subsequent ones we will see many examples of how this data is applied to specific design problems.

We have chosen four machines to examine: the DEC VAX, the IBM 360, the Intel 8086, and a generic load/store machine called DLX. These architectures play a dominant role in the computer marketplace, and each has a set of unique and interesting characteristics. The Intel 8086 is the most popular general-purpose computer in the world; tens of millions of machines containing this microprocessor have been sold. The IBM 360 and DEC VAX represent architectures that have existed for long periods of time (25+ and 10+ years, respectively) and have each sold hundreds of thousands of units. DLX is representative of a new breed of machines that has become very popular since the late 1980s. These machines are also very different in architectural style, as we will see.

To try to simplify the reader's task, a common format is used for the syntax of instructions. This format puts the destination of a multiple-operand instruction first, followed by the first and second source operands. So, an instruction that subtracts R3 from R2 and puts the result in R1 is written as:

```
SUB   R1,R2,R3
```

This format follows the convention used on the Intel 8086, and is close to the convention on the 360. The only significant difference on the 360 is for store instructions, which place the source register first. While the VAX syntax always puts the source operands first and the destination last, we will show VAX code in our common format. Of course, this ordering is purely a syntactic convention and the architecture defines the encoding of operands in the binary instruction format.

The next four sections are summaries of the four architectures. Although these summaries are concise, the important attributes and most heavily used features are all discussed. Tables containing all the operations in the instruction sets are contained in Appendix B. To describe these architectures accurately, we need to introduce a few additional extensions to our C description language to explain the functions of the instructions. The additions are as follows:

- A subscript is appended to the symbol $\leftarrow$ whenever the length of the datum being assigned might not be clear. Thus, $\leftarrow_n$ means transfer an n-bit quantity.
- A subscript is used to indicate selection of a bit from a field. Bits are labeled from the most significant bit starting at 0. The subscript may be a single digit

(e.g., $R4_0$ yields the sign bit of R4) or a subrange (e.g., $R3_{24..31}$ yields the least significant byte of R3).

- A superscript is used to replicate a field (e.g., 0^{24} yields a field of zeros of length 24 bits).
- The variable `M` is used as an array that stands for main memory. The array is indexed by a byte address and may transfer any number of bytes.
- The symbol ## is used to concatenate two fields and may appear on either side of a data transfer.

A summary of the entire description language appears on the page preceding the back inside cover. As an example, assuming that `R8` and `R10` are 32-bit registers:

$$\mathtt{R10}_{16..31} \leftarrow_{16} (\mathtt{M[R8]}_0)^8 \;\#\#\; \mathtt{M[R8]}$$

means that the byte at the memory location addressed by the contents of `R8` is sign-extended to form a 16-bit quantity that is stored into the lower half of `R10`. (The upper half of `R10` is unchanged.)

Following the instruction set architecture summaries in the next four sections, we examine and contrast dynamic use measurements of the four architectures.

4.2 The VAX Architecture

The DEC VAX was introduced with its first model, the VAX-11/780, in 1977. The VAX was designed to be a 32-bit extension of the PDP-11 architecture. Among the goals of the VAX, two stand out as both important and having had a substantial impact on the VAX architecture.

First, the designers wanted to make the existing PDP-11 customer base feel comfortable with the VAX architecture and view it as an extension of the PDP-11. This motivated the name VAX-11/780, the use of a very similar assembly language syntax, inclusion of the PDP-11 data types, and emulation support for the PDP-11. Second, the designers wanted to ease the task of writing compilers and operating systems. This translated to a set of goals that included defining interfaces between languages, the hardware, and OS; and supporting a highly orthogonal architecture.

In terms of addressing modes and operations supported in instructions, the other architectures discussed in this chapter are largely subsets of the VAX. For this reason our discussion begins with the VAX, which will serve as a basis for comparison. The reader should be aware that there are entire books devoted to the VAX architecture as well as a number of papers reporting instruction set measurements. Our summary of the VAX instruction set—like the other