

S E C O N D E D I T I O N

Computer Organization and Design

T H E H A R D W A R E / S O F T W A R E I N T E R F A C E

John L. Hennessy

Stanford University

David A. Patterson

University of California, Berkeley

With a contribution by
James R. Larus
University of Wisconsin



Morgan Kaufmann Publishers, Inc.
San Francisco, California

Dartmouth

Sponsoring Editor Denise Penrose
Production Manager Yonie Overton
Production Editor Julie Pabst
Editorial Coordinator Jane Elliott
Text and Cover Design Ross Carron Design
Illustration Alexander Teshin Associates, with second edition modifications by Dartmouth Publishing, Inc.
Chapter Opener Illustrations Canary Studios
Copyeditor Ken DellaPenta
Composition Nancy Logan
Proofreader Jennifer McClain
Indexer Steve Rath
Printer Courier Corporation

Morgan Kaufmann Publishers, Inc.
Editorial and Sales Office:
340 Pine Street, Sixth Floor
San Francisco, CA 94104-3205
USA

Telephone 415/392-2665
Facsimile 415/982-2665
Email mkp@mkp.com
WWW <http://www.mkp.com>
Order toll free 800/745-7323

© 1998 by Morgan Kaufmann Publishers, Inc.
All rights reserved
Printed in the United States of America

02 01 00 99 5 4 3

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, recording, or otherwise—without the prior written permission of the publisher.

Advice, Praise, and Errors: Any correspondence related to this publication or intended for the authors should be sent electronically to cod2bugs@mkp.com. 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 at the time of their implementation in a reprint.

Library of Congress Cataloging-in-Publication Data

Patterson, David A.

Computer organization and design : the hardware/software interface
/ David A. Patterson, John L. Hennessy.—2nd ed.
p. cm.

Includes bibliographical references and index.

ISBN 1-55860-428-6 (cloth).—ISBN 1-55860-491-X (paper)

1. Computer organization. 2. Computers—Design and construction.

3. Computer interfaces. I. Hennessy, John L. II. Title
QA76.9.C643H46 1997

004.2'2—dc21

97-16050

Contents

Foreword vi
by John H. Crawford

Worked Examples xiii

Computer Organization and Design Online xvi

Preface xix

C H A P T E R S

1

Computer Abstractions and Technology 2

- 1.1 Introduction** 3
- 1.2 Below Your Program** 5
- 1.3 Under the Covers** 10
- 1.4 Integrated Circuits: Fueling Innovation** 21
- 1.5 Real Stuff: Manufacturing Pentium Chips** 24
- 1.6 Fallacies and Pitfalls** 29
- 1.7 Concluding Remarks** 30
- 1.8 Historical Perspective and Further Reading** 32
- 1.9 Key Terms** 44
- 1.10 Exercises** 45

2

The Role of Performance 52

- 2.1 Introduction** 54
- 2.2 Measuring Performance** 58
- 2.3 Relating the Metrics** 60
- 2.4 Choosing Programs to Evaluate Performance** 66
- 2.5 Comparing and Summarizing Performance** 69
- 2.6 Real Stuff: The SPEC95 Benchmarks and Performance of Recent Processors** 71
- 2.7 Fallacies and Pitfalls** 75
- 2.8 Concluding Remarks** 82

2.9 Historical Perspective and Further Reading	83
2.10 Key Terms	89
2.11 Exercises	90

3**Instructions: Language of the Machine 104**

3.1 Introduction	106
3.2 Operations of the Computer Hardware	107
3.3 Operands of the Computer Hardware	109
3.4 Representing Instructions in the Computer	116
3.5 Instructions for Making Decisions	122
3.6 Supporting Procedures in Computer Hardware	132
3.7 Beyond Numbers	142
3.8 Other Styles of MIPS Addressing	145
3.9 Starting a Program	156
3.10 An Example to Put It All Together	163
3.11 Arrays versus Pointers	171
3.12 Real Stuff: PowerPC and 80x86 Instructions	175
3.13 Fallacies and Pitfalls	185
3.14 Concluding Remarks	187
3.15 Historical Perspective and Further Reading	189
3.16 Key Terms	196
3.17 Exercises	196

4**Arithmetic for Computers 208**

4.1 Introduction	210
4.2 Signed and Unsigned Numbers	210
4.3 Addition and Subtraction	220
4.4 Logical Operations	225
4.5 Constructing an Arithmetic Logic Unit	230
4.6 Multiplication	250
4.7 Division	265
4.8 Floating Point	275
4.9 Real Stuff: Floating Point in the PowerPC and 80x86	301
4.10 Fallacies and Pitfalls	304
4.11 Concluding Remarks	308
4.12 Historical Perspective and Further Reading	312
4.13 Key Terms	322
4.14 Exercises	322

5**The Processor: Datapath and Control 336**

5.1 Introduction	338
5.2 Building a Datapath	343
5.3 A Simple Implementation Scheme	351

5.4	A Multicycle Implementation	377
5.5	Microprogramming: Simplifying Control Design	399
5.6	Exceptions	410
5.7	Real Stuff: The Pentium Pro Implementation	416
5.8	Fallacies and Pitfalls	419
5.9	Concluding Remarks	421
5.10	Historical Perspective and Further Reading	423
5.11	Key Terms	426
5.12	Exercises	427

6**Enhancing Performance with Pipelining** 434

6.1	An Overview of Pipelining	436
6.2	A Pipelined Datapath	449
6.3	Pipelined Control	466
6.4	Data Hazards and Forwarding	476
6.5	Data Hazards and Stalls	489
6.6	Branch Hazards	496
6.7	Exceptions	505
6.8	Superscalar and Dynamic Pipelining	510
6.9	Real Stuff: PowerPC 604 and Pentium Pro Pipelines	517
6.10	Fallacies and Pitfalls	520
6.11	Concluding Remarks	521
6.12	Historical Perspective and Further Reading	525
6.13	Key Terms	529
6.14	Exercises	529

7**Large and Fast: Exploiting Memory Hierarchy** 538

7.1	Introduction	540
7.2	The Basics of Caches	545
7.3	Measuring and Improving Cache Performance	564
7.4	Virtual Memory	579
7.5	A Common Framework for Memory Hierarchies	603
7.6	Real Stuff: The Pentium Pro and PowerPC 604 Memory Hierarchies	611
7.7	Fallacies and Pitfalls	615
7.8	Concluding Remarks	618
7.9	Historical Perspective and Further Reading	621
7.10	Key Terms	627
7.11	Exercises	628

8**Interfacing Processors and Peripherals** 636

8.1	Introduction	638
8.2	I/O Performance Measures: Some Examples from Disk and File Systems	641
8.3	Types and Characteristics of I/O Devices	644

- 8.4 Buses: Connecting I/O Devices to Processor and Memory** 655
- 8.5 Interfacing I/O Devices to the Memory, Processor, and Operating System** 673
- 8.6 Designing an I/O System** 684
- 8.7 Real Stuff: A Typical Desktop I/O System** 687
- 8.8 Fallacies and Pitfalls** 688
- 8.9 Concluding Remarks** 690
- 8.10 Historical Perspective and Further Reading** 694
- 8.11 Key Terms** 700
- 8.12 Exercises** 700

9**Multiprocessors** 710

- 9.1 Introduction** 712
- 9.2 Programming Multiprocessors** 714
- 9.3 Multiprocessors Connected by a Single Bus** 717
- 9.4 Multiprocessors Connected by a Network** 727
- 9.5 Clusters** 734
- 9.6 Network Topologies** 736
- 9.7 Real Stuff: Future Directions for Multiprocessors** 740
- 9.8 Fallacies and Pitfalls** 743
- 9.9 Concluding Remarks—Evolution versus Revolution in Computer Architecture** 746
- 9.10 Historical Perspective and Further Reading** 748
- 9.11 Key Terms** 756
- 9.12 Exercises** 756

A P P E N D I C E S**A****Assemblers, Linkers, and the SPIM Simulator** A-2

by James R. Larus, University of Wisconsin

- A.1 Introduction** A-3
- A.2 Assemblers** A-10
- A.3 Linkers** A-17
- A.4 Loading** A-19
- A.5 Memory Usage** A-20
- A.6 Procedure Call Convention** A-22
- A.7 Exceptions and Interrupts** A-32
- A.8 Input and Output** A-36
- A.9 SPIM** A-38
- A.10 MIPS R2000 Assembly Language** A-49
- A.11 Concluding Remarks** A-75
- A.12 Key Terms** A-76
- A.13 Exercises** A-76

B**The Basics of Logic Design** B-2

- B.1 Introduction** B-3
- B.2 Gates, Truth Tables, and Logic Equations** B-4
- B.3 Combinational Logic** B-8
- B.4 Clocks** B-18
- B.5 Memory Elements** B-21
- B.6 Finite State Machines** B-35
- B.7 Timing Methodologies** B-39
- B.8 Concluding Remarks** B-44
- B.9 Key Terms** B-45
- B.10 Exercises** B-45

C**Mapping Control to Hardware** C-2

- C.1 Introduction** C-3
- C.2 Implementing Combinational Control Units** C-4
- C.3 Implementing Finite State Machine Control** C-8
- C.4 Implementing the Next-State Function with a Sequencer** C-21
- C.5 Translating a Microprogram to Hardware** C-28
- C.6 Concluding Remarks** C-31
- C.7 Key Terms** C-32
- C.8 Exercises** C-32

Glossary G-1**Index** I-1

Worked Examples

Chapter 2: The Role of Performance

Throughput and Response Time 56
 Relative Performance 57
 Improving Performance 60
 Using the Performance Equation 62
 Comparing Code Segments 64
 MIPS as a Performance Measure 78

Chapter 3: Instructions: Language of the Machine

Compiling Two C Assignment Statements into MIPS 108
 Compiling a Complex C Assignment into MIPS 109
 Compiling a C Assignment Using Registers 110
 Compiling an Assignment When an Operand Is in Memory 112
 Compiling Using Load and Store 113
 Compiling Using a Variable Array Index 114
 Translating a MIPS Assembly Instruction into a Machine Instruction 117
 Translating MIPS Assembly Language into Machine Language 119
 Compiling an *If* Statement into a Conditional Branch 123
 Compiling *if-then-else* into Conditional Branches 124
 Compiling a Loop with Variable Array Index 126
 Compiling a *while* Loop 127
 Compiling a Less Than Test 128
 Compiling a *switch* Statement by Using a Jump Address Table 129
 Compiling a Procedure that Doesn't Call Another Procedure 134
 Compiling a Recursive Procedure, Showing Nested Procedure Linking 136
 Compiling a String Copy Procedure, Showing How to Use C Strings 143
 Translating Assembly Constants into Machine Language 145
 Loading a 32-Bit Constant 147
 Showing Branch Offset in Machine Language 149
 Branching Far Away 150
 Decoding Machine Code 154
 Linking Object Files 160
 Compiling an Assignment Statement into Accumulator Instructions 190
 Compiling an Assignment Statement into Memory-Memory Instructions 192
 Compiling an Assignment Statement into Stack Instructions 193

Chapter 4: Arithmetic for Computers

ASCII versus Binary Numbers 212
 Binary to Decimal Conversion 214
 Signed versus Unsigned Comparison 215
 Negation Shortcut 216
 Sign Extension Shortcut 217
 Binary-to-Hexadecimal Shortcut 218

- Binary Addition and Subtraction 220
- C Bit Fields 229
- Both Levels of the Propagate and Generate 247
- Speed of Ripple Carry versus Carry Lookahead 248
- First Multiply Algorithm 253
- Second Multiply Algorithm 256
- Third Multiply Algorithm 257
- Booth's Algorithm 261
- Multiply by 2^i via Shift 262
- First Divide Algorithm 268
- Third Divide Algorithm 271
- Floating-Point Representation 279
- Converting Binary to Decimal Floating Point 280
- Decimal Floating-Point Addition 282
- Decimal Floating-Point Multiplication 287
- Compiling a Floating-Point C Program into MIPS Assembly Code 293
- Compiling Floating-Point C Procedure with Two-Dimensional Matrices into MIPS 294
- Rounding with Guard Digits 297

Chapter 5: The Processor: Datapath and Control

- Composing Datapaths 351
- Implementing Jumps 370
- Performance of Single-Cycle Machines 373
- Performance of a Single-Cycle CPU with Floating-Point Instructions 375
- CPI in a Multicycle CPU 397

Chapter 6: Enhancing Performance with Pipelining

- Single-Cycle versus Pipelined Performance 438
- Stall on Branch Performance 442
- Forwarding with Two Instructions 446
- Reordering Code to Avoid Pipeline Stalls 447
- Labeled Pipeline Execution, Including Control 471
- Dependency Detection 479
- Forwarding 485
- Pipelined Branch 498
- Loops and Prediction 501
- Comparing Performance of Several Control Schemes 504
- Exception in a Pipelined Computer 507
- Simple Superscalar Code Scheduling 513
- Loop Unrolling for Superscalar Pipelines 513

Chapter 7: Large and Fast: Exploiting Memory Hierarchy

- Bits in a Cache 550
- Mapping an Address to a Multiword Cache Block 556
- Calculating Cache Performance 565
- Cache Performance with Increased Clock Rate 567
- Associativity in Caches 571
- Size of Tags versus Set Associativity 575
- Performance of Multilevel Caches 576
- Overall Operation of a Memory Hierarchy 595

Chapter 8: Interfacing Processors and Peripherals

- Impact of I/O on System Performance 639
- Disk Read Time 648

Performance of Two Networks 654
 FSM Control for I/O 662
 Performance Analysis of Synchronous versus Asynchronous Buses 662
 Performance Analysis of Two Bus Schemes 665
 Overhead of Polling in an I/O System 676
 Overhead of Interrupt-Driven I/O 679
 Overhead of I/O Using DMA 681
 I/O System Design 685

Chapter 9: Multiprocessors

Speedup Challenge 715
 Speedup Challenge, Bigger Problem 716
 Parallel Program (Single Bus) 718
 Parallel Program (Message Passing) 729

Appendix A: Assemblers, Linkers, and the SPIM Simulator

Local and Global Labels A-11
 String Directive A-15
 Macros A-15
 Stack in Recursive Procedure A-28
 Interrupt Handler A-34

Appendix B: The Basics of Logic Design

Truth Tables B-5
 Logic Equations B-6
 Sum of Products B-11
 PLAs B-13
 Don't Cares B-16

Appendix C: Mapping Control to Hardware

Logic Equations for Next-State Outputs C-12
 Control ROM Entries C-17

7.2 The Basics of Caches

Cache: a safe place for hiding or storing things.

*Webster's New World Dictionary of the American Language,
Third College Edition (1988)*

In our library example, the desk acted as a cache—a safe place to store things (books) that we needed to examine. *Cache* was the name chosen to represent the level of the memory hierarchy between the CPU and main memory in the first commercial machine to have this extra level. Today, although this remains the dominant use of the word *cache*, the term is also used to refer to any storage managed to take advantage of locality of access. Caches first appeared in research machines in the early 1960s and in production machines later in that same decade; virtually every general-purpose machine built today, from the fastest to the slowest, includes a cache.

In this section, we begin by looking at a very simple cache in which the processor requests are each one word and the blocks also consist of a single word. Figure 7.4 shows such a simple cache, before and after requesting a data item that is not initially in the cache. Before the request, the cache contains a collection of recent references $X_1, X_2, \dots, X_{n-1}$, and the processor requests a word X_n that is not in the cache. This request results in a miss, and the word X_n is brought from memory into cache.

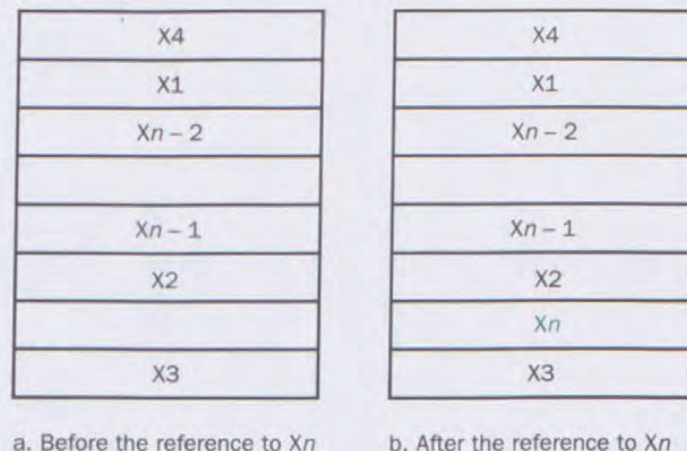


FIGURE 7.4 The cache just before and just after a reference to a word X_n that is not initially in the cache. This reference causes a miss that forces the cache to fetch X_n from memory and insert it into the cache.

Looking at the scenario in Figure 7.4, we can see that there are two questions we must answer: How do we know if a data item is in the cache? And, if it is, how do we find it? The answers to these two questions are related. If each word can go in exactly one place in the cache, then we will know how to find the word if it is in the cache. The simplest way to assign a location in the cache for each word in memory is to assign the cache location based on the address of the word in memory. This cache structure is called *direct mapped*, since each memory location is mapped to exactly one location in the cache. The typical mapping between addresses and cache locations for a direct-mapped cache is usually simple. For example, almost all direct-mapped caches use the mapping:

$$(\text{Block address}) \bmod (\text{Number of cache blocks in the cache})$$

This mapping is attractive because if the number of entries in the cache is a power of two, then modulo can be computed simply by using only the low-order $\log_2$ (cache size in blocks) bits of the address; hence the cache may be accessed directly with the low-order bits. For example, Figure 7.5 shows a direct-mapped cache of eight words and the memory addresses between 1_{ten} (00001_{two}) and 29_{ten} (11101_{two}) that map to locations 1_{ten} (001_{two}) and 5_{ten} (101_{two}) in the cache.

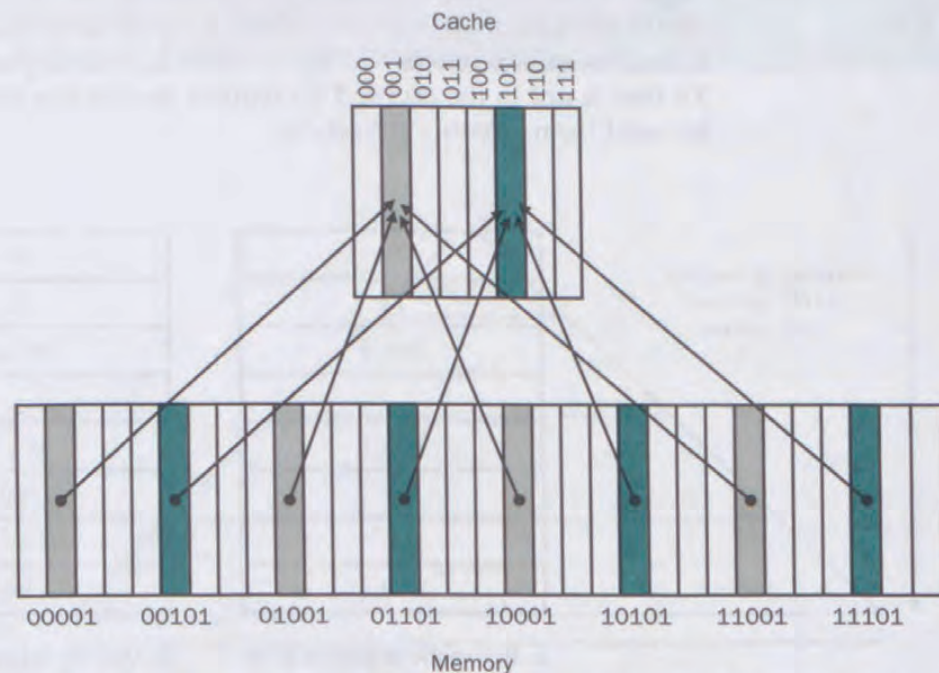


FIGURE 7.5 A direct-mapped cache with eight entries showing the addresses of memory words between 0 and 31 that map to the same cache locations. Because there are eight words in the cache, an address X maps to the cache word $X \bmod 8$. That is, the low-order $\log_2(8) = 3$ bits are used as the cache index. Thus, addresses 00001_{two} , 01001_{two} , 10001_{two} , and 11001_{two} all map to entry 001_{two} of the cache, while addresses 00101_{two} , 01101_{two} , 10101_{two} , and 11101_{two} all map to entry 101_{two} of the cache.

Because each cache location can contain the contents of a number of different memory locations, how do we know whether the data in the cache corresponds to a requested word? That is, how do we know whether a requested word is in the cache or not? We can plan for this by adding a set of *tags* to the cache. The tags contain the address information required to identify whether a word in the cache corresponds to the requested word. The tag needs only to contain the upper portion of the address, corresponding to the bits that are not used as an index into the cache. The reason for this is that the bits corresponding to the index are used to select the unique entry in the cache corresponding to the supplied address, and thus we only need to ensure that the upper portion of the supplied address matches the tag. For example, in Figure 7.5 we need only have 2 of the 5 address bits in the tag, since the lowest 3 bits of the address select the block.

We also need a way to recognize that a cache block does not have valid information. For instance, when a processor starts up, the cache will be empty, and the tag fields will be meaningless. Even after executing many instructions, some of the cache entries may still be empty, as in Figure 7.4. Thus we need to know that the tag should be ignored for such entries. The most common method is to add a *valid bit* to indicate whether an entry contains a valid address. If the bit is not set, there cannot be a match for this block.

For the rest of this section, we will focus on explaining how reads work in a cache and how the cache control works for reads. In general, handling reads is a little simpler than handling writes, since reads do not have to change the contents of the cache. After seeing the basics of how reads work and how cache misses can be handled, we'll examine the cache designs for two real machines and detail how these caches handle writes.

Accessing a Cache

Figure 7.6 shows the contents of an eight-word direct-mapped cache as it responds to a series of requests from the processor. Since there are eight blocks in the cache, the low-order 3 bits of an address give the block number. Here is the action for each reference:

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110 _{two}	miss (7.5b)	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010 _{two}	miss (7.5c)	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
22	10110 _{two}	hit	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010 _{two}	hit	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000 _{two}	miss (7.5d)	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
3	00011 _{two}	miss (7.5e)	$(00011_{\text{two}} \bmod 8) = 011_{\text{two}}$
16	10000 _{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
18	10010 _{two}	miss (7.5f)	$(10010_{\text{two}} \bmod 8) = 010_{\text{two}}$

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a. The initial state of the cache after power-on

Index	V	Tag	Data
000	N		
001	N		
010	Y	11 _{two}	Memory (11010 _{two})
011	N		
100	N		
101	N		
110	Y	10 _{two}	Memory (10110 _{two})
111	N		

c. After handling a miss of address (11010_{two})

Index	V	Tag	Data
000	Y	10 _{two}	Memory (10000 _{two})
001	N		
010	Y	11 _{two}	Memory (11010 _{two})
011	Y	00 _{two}	Memory (00011 _{two})
100	N		
101	N		
110	Y	10 _{two}	Memory (10110 _{two})
111	N		

e. After handling a miss of address (00011_{two})

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10 _{two}	Memory(10110 _{two})
111	N		

b. After handling a miss of address (10110_{two})

Index	V	Tag	Data
000	Y	10 _{two}	Memory (10000 _{two})
001	N		
010	Y	11 _{two}	Memory (11010 _{two})
011	N		
100	N		
101	N		
110	Y	10 _{two}	Memory (10110 _{two})
111	N		

d. After handling a miss of address (10000_{two})

Index	V	Tag	Data
000	Y	10 _{two}	Memory (10000 _{two})
001	N		
010	Y	10 _{two}	Memory (10010 _{two})
011	Y	00 _{two}	Memory (00011 _{two})
100	N		
101	N		
110	Y	10 _{two}	Memory (10110 _{two})
111	N		

f. After handling a miss of address (10010_{two})

FIGURE 7.6 The cache contents are shown after each reference request that misses, with the index and tag fields shown in binary. The cache is initially empty, with all valid bits (V entry in cache) turned off (N). The processor requests the following addresses: 10110_{two} (miss), 11010_{two} (miss), 10110_{two} (hit), 11010_{two} (hit), 10000_{two} (miss), 00011_{two} (miss), 10000_{two} (hit), and 10010_{two} (miss). The figures show the cache contents after each miss in the sequence has been handled. When address 10010_{two} (18) is referenced, the entry for address 11010_{two} (26) must be replaced, and a reference to 11010_{two} will cause a subsequent miss. The tag field will contain only the upper portion of the address. The full address of a word contained in cache block i with tag field j for this cache is $j \times 8 + i$, or equivalently the concatenation of the tag field j and the index i . For example, in cache f above, index 010 has tag 10 and corresponds to address 10010.

When the word at address 18 (10010_{two}) is brought into cache block 2 (010_{two}), the word at address 26 (11010_{two}), which was in cache block 2 (010_{two}), must be replaced by the newly requested data. This behavior allows a cache to take advantage of temporal locality: recently accessed words replace less recently

referenced words. This situation is directly analogous to needing a book from the shelves and having no more space on your desk—some book already on your desk must be returned to the shelves. In a direct-mapped cache, there is only one place to put the newly requested item and hence only one choice of what to replace.

We know where to look in the cache for each possible address: the low-order bits of an address can be used to find the unique cache entry to which the address could map. Figure 7.7 shows how a referenced address is divided into

- a cache index, which is used to select the block
- a tag field, which is used to compare with the value of the tag field of the cache

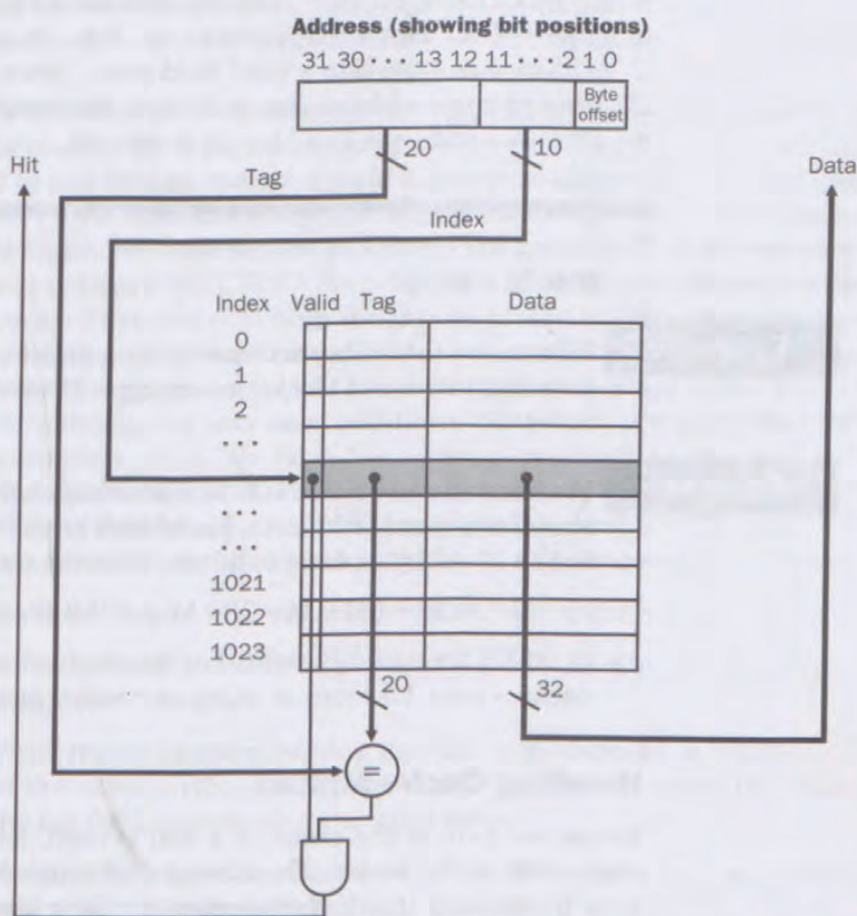


FIGURE 7.7 For this cache, the lower portion of the address is used to select a cache entry consisting of a data word and a tag. The tag from the cache is compared against the upper portion of the address to determine whether the entry in the cache corresponds to the requested address. Because the cache has 2^{10} (or 1024) words, and a block size of 1 word, 10 bits are used to index the cache, leaving $32 - 10 - 2 = 20$ bits to be compared against the tag. If the tag and upper 20 bits of the address are equal and the valid bit is on, then the request hits in the cache, and the word is supplied to the processor. Otherwise, a miss occurs.

Because a given address can appear in exactly one location, the tag need only correspond to the upper portion of the address, which is not used to index the cache. Thus, the index of a cache block, together with the tag contents of that block, uniquely specify the memory address of the word contained in the cache block. Because the index field is used as an address to access the cache and because an n -bit field has 2^n values, the total number of entries in the cache must be a power of two. In the MIPS architecture, the least significant 2 bits of every address specify a byte within a word and are not used to select the word in the cache.

The total number of bits needed for a cache is a function of the cache size and the address size because the cache includes both the storage for the data and for the tags. Assuming the 32-bit byte address, a direct-mapped cache of size 2^n words with one-word (4-byte) blocks will require a tag field whose size is $32 - (n + 2)$ bits, because 2 bits are used for the byte offset and n bits are used for the index. The total number of bits in a direct-mapped cache is $2^n \times (\text{block size} + \text{tag size} + \text{valid field size})$. Since the block size is one word (32 bits) and the address size is 32 bits, the number of bits in such a cache is $2^n \times (32 + (32 - n - 2) + 1) = 2^n \times (63 - n)$.

Bits in a Cache

Example

How many total bits are required for a direct-mapped cache with 64 KB of data and one-word blocks, assuming a 32-bit address?

Answer

We know that 64 KB is 16K words, which is 2^{14} words, and, with a block size of one word, 2^{14} blocks. Each block has 32 bits of data plus a tag, which is $32 - 14 - 2$ bits, plus a valid bit. Thus the total cache size is

$$2^{14} \times (32 + (32 - 14 - 2) + 1) = 2^{14} \times 49 = 784 \times 2^{10} = 784 \text{ Kbits}$$

or 98 KB for a 64-KB cache. For this cache, the total number of bits in the cache is over 1.5 times as many as needed just for the storage of the data.

Handling Cache Misses

Before we look at the cache of a real system, let's see how the control unit deals with cache misses. The control unit must detect a miss and process the miss by fetching the data from memory (or a lower-level cache). If the cache reports a hit, the machine continues using the data as if nothing had happened. Consequently, we can use the same basic control that we developed in

Chapter 5 and enhanced to accommodate pipelining in Chapter 6. The memories in the datapath used in Chapters 5 and 6 are simply replaced by caches.

Modifying the control of a processor to take a hit into account is trivial; misses, however, require some extra work. The basic approach is to stall the CPU, freezing the contents of all the registers. A separate controller handles the cache miss, fetching the data into the cache from memory. Once the data is present, execution is restarted at the cycle that caused the cache miss. The cache miss handling is done with the processor control unit and with a separate controller that initiates the memory access and refills the cache. The processing of a cache miss creates a stall, similar to the pipeline stalls discussed in Chapter 6, as opposed to an interrupt, which would require saving the state of all registers. For a cache miss, we can stall the entire machine, essentially freezing the contents of the temporary and programmer-visible registers, while we wait for memory. In contrast, pipeline stalls, discussed in Chapter 6, are more complex because we must continue executing some instructions while we stall others.

Let's look a little more closely at how instruction misses are handled for either the multicycle or pipelined datapath; the same approach can be easily extended to handle data misses. If an instruction access results in a miss, then the contents of the Instruction register are invalid. To get the proper instruction into the cache, we must be able to instruct the lower level in the memory hierarchy to perform a read. Since the program counter is incremented in the first clock cycle of execution in both the pipelined and multicycle processors, the address of the instruction that generates an instruction cache miss is equal to the value of the program counter minus four. We can compute this value using the ALU, although we may need additional temporary storage in the pipelined implementation. Once we have the address, we need to instruct the main memory to perform a read. We wait for the memory to respond (since the access will take multiple cycles), and then write the word into the cache.

We can now define the steps to be taken on an instruction cache miss:

1. Send the original PC value (current PC - 4) to the memory.
2. Instruct main memory to perform a read and wait for the memory to complete its access.
3. Write the cache entry, putting the data from memory in the data portion of the entry, writing the upper bits of the address (from the ALU) into the tag field, and turning the valid bit on.
4. Restart the instruction execution at the first step, which will refetch the instruction, this time finding it in the cache.

The control of the cache on a data access is essentially identical: on a miss, we simply stall the processor until the memory responds with the data. In the

rest of this section we describe two different caches from real machines, and we examine how they handle both reads and writes. In section 7.5, we will describe the handling of writes in more detail.

Elaboration: To reduce the penalty of cache misses, designers employ two techniques, one of which we discuss here and another that we will discuss later. To reduce the number of cycles that a processor is stalled for a cache miss, we can allow a processor to continue executing instructions while the cache miss is handled. This strategy does not help for instruction misses because we cannot fetch new instructions to execute. In the case of data misses, however, we can allow the machine to continue fetching and executing instructions until the loaded word is required. Hence the name for this technique: *stall on use*. While this additional effort may save cycles, it will probably not save very many cycles because the loaded data will likely be needed very shortly and because other instructions will need access to the cache. A more sophisticated technique is used in some modern processors; we describe it in section 7.6.

An Example Cache: The DECStation 3100

The DECStation 3100 was a workstation that used a MIPS R2000 as the processor and a very simple cache implementation. Near the end of the chapter, we will examine the cache design of a more recent computer, but we start with this simple, yet real, example for pedagogical reasons.

This processor has a pipeline similar to that discussed in Chapter 6. When operating at peak speed, the processor requests both an instruction word and a data word on every clock. To satisfy the demands of the pipeline without stalling, separate instruction and data caches are used. Each cache is 64 KB, or 16K words, with a one-word block. Figure 7.8 shows the organization of the DECStation 3100 data cache.

Read requests for the cache are straightforward. Because there are separate data and instruction caches, separate control signals will be needed to read and write each cache. (Remember that we need to write into the instruction cache when a miss occurs.) Thus the steps for a read request to either cache are as follows:

1. Send the address to the appropriate cache. The address comes either from the PC (for an instruction read) or from the ALU (for a data access).
2. If the cache signals hit, the requested word is available on the data lines. If the cache signals miss, we send the address to the main memory. When the memory returns with the data, we write it into the cache.

Writes work somewhat differently. Suppose on a store instruction, we wrote the data into only the data cache (without changing main memory); then, after

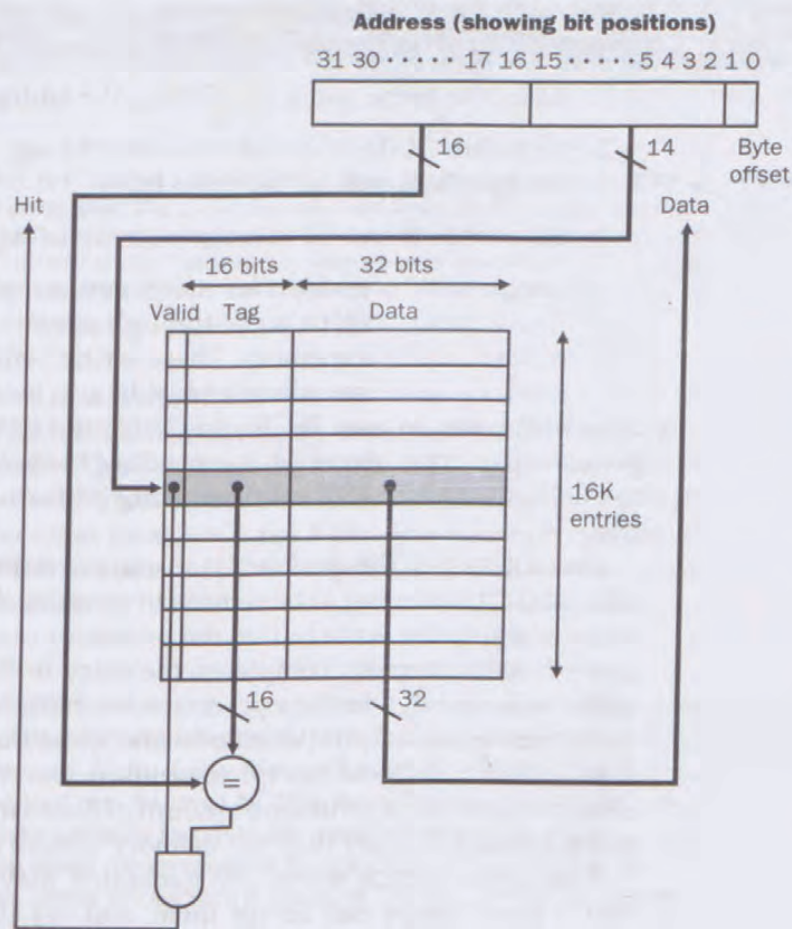


FIGURE 7.8 The caches in the DECStation 3100 each contain 16K blocks with one word per block. This means that the index is 14 bits and that the tag contains 16 bits.

the write into the cache, memory would have a different value from that in the cache. In such a case, the cache and memory are said to be *inconsistent*. The simplest way to keep the main memory and the cache consistent is to always write the data into both the memory and the cache. This scheme, which the DECStation 3100 uses, is called *write-through*.

The other key aspect of writes is what occurs on a write miss. Because the data word in the cache is being written by the processor, there is no reason to read a word from memory; it would just be overwritten by the processor. In fact, for this simple cache we can always just write the word into the cache, updating both the tag and data. We do not need to consider whether a write hits

or misses in the cache. This observation leads to the following simple scheme for processing writes, used on the DECStation 3100:

1. Index the cache using bits 15–2 of the address.
2. Write bits 31–16 of the address into the tag, write the data word into the data portion, and set the valid bit.
3. Also write the word to main memory using the entire address.

Although this design handles writes very simply, it would not provide very good performance. With a write-through scheme, every write causes the data to be written to main memory. These writes will take a long time and could slow down the machine considerably. In gcc, for example, 13% of the instructions are stores. In the DECStation 3100, the CPI without cache misses for a program like gcc is about 1.2, so spending 10 cycles on every write would lead to a CPI of $1.2 + 10 \times 13\% = 2.5$, reducing performance by more than a factor of two.

One solution to this problem is to use a *write buffer*. A write buffer stores the data while it is waiting to be written to memory. After writing the data into the cache and into the write buffer, the processor can continue execution. When a write to main memory completes, the entry in the write buffer is freed. If the write buffer is full when the processor reaches a write, the processor must stall until there is an empty position in the write buffer. Of course, if the rate at which the memory can complete writes is less than the rate at which the processor is generating writes, no amount of buffering can help because writes are being generated faster than the memory system can accept them.

The rate at which writes are generated may also be *less* than the rate at which the memory can accept them, and yet stalls may still occur. This can happen when the writes occur in bursts. To reduce the occurrence of such stalls, machines usually increase the depth of the write buffer beyond a single entry. For example, the DECStation 3100 write buffer is four words deep. As the difference between the rate at which programs can generate writes and the rate at which the memory system can accept them increases, even deeper write buffers (e.g., 10 entries) are appropriate.

The alternative to a write-through scheme is a scheme called *write-back*. In a write-back scheme, when a write occurs, the new value is written only to the block in the cache. The modified block is written to the lower level of the hierarchy when it is replaced. Write-back schemes can improve performance, especially when processors can generate writes as fast or faster than the writes can be handled by main memory; a write-back scheme is, however, more complex to implement than write-through.

What sort of cache miss rates are attained with a cache structure like that used by the DECStation 3100? Figure 7.9 shows the miss rates for the instruction and data caches for two programs, which we have seen before. The com-

Program	Instruction miss rate	Data miss rate	Effective combined miss rate
gcc	6.1%	2.1%	5.4%
spice	1.2%	1.3%	1.2%

FIGURE 7.9 Instruction and data miss rates for the DECStation 3100 when executing two different programs. The combined miss rate is the effective miss rate seen for the combination of the 64-KB instruction cache and 64-KB data cache. It is obtained by weighting the instruction and data individual miss rates by the frequency of instruction and data references. Remember that data misses include only data reads because writes cannot miss in the DECStation 3100 cache.

bined miss rate is the effective miss rate per reference for each program after accounting for the differing frequency of instruction and data accesses.

Remember that although miss rate is an important characteristic of cache designs, the ultimate measure will be the effect of the memory system on program execution time; we'll see how miss rate and execution time are related shortly. First we must explore how the memory system can take advantage of spatial locality.

Elaboration: A combined cache of the total size equal to the sum of the two split caches will usually have a better hit rate. This higher rate occurs because the combined cache does not rigidly divide the number of entries that may be used by instructions from those that may be used by data. Nonetheless, many machines use a split instruction and data cache to increase the *bandwidth* from the cache.

Here are some measurements for the DECStation 3100 for the program gcc, and for a combined cache whose size is equal to the total of the two caches on the 3100:

- Total cache size: 128 KB
- Split cache effective miss rate: 5.4%
- Combined cache miss rate: 4.8%

The miss rate of the split cache is only slightly worse.

For many systems, the advantage of doubling the cache bandwidth, by supporting both an instruction and data access simultaneously, easily overcomes the disadvantage of a slightly increased miss rate. This observation is another reminder that we cannot use miss rate as the sole measure of cache performance.

Taking Advantage of Spatial Locality

The cache we have described so far, while simple, does nothing to take advantage of spatial locality in requests, since each word is in its own block. As we noted in section 7.1, spatial locality exists naturally in programs. To take advantage of spatial locality, we want to have a cache block that is larger than one word in length. When a miss occurs, we will then fetch multiple words that are adjacent and carry a high probability of being needed shortly.

Figure 7.10 shows a cache that holds 64 KB of data, but with blocks of four words (16 bytes) each. Compared with Figure 7.8 on page 553, which shows the same total size cache with a one-word block, an extra block index field occurs in the address of the cache in Figure 7.10. This block index field is used to control the multiplexor (shown at the bottom of the figure), which selects the requested word from the four words in the indexed block. The total number of tags and valid bits in the cache with a multiword block is smaller because each tag and valid bit is used for four words. This sharing of tags improves the efficiency of memory use in the cache.

How do we find the cache block for a particular address? We can use the same mapping that we used for a cache with a one-word block:

$$(\text{Block address}) \bmod (\text{Number of cache blocks})$$

The block address is simply the word address divided by the number of words in the block (or equivalently, the byte address divided by the number of bytes in the block).

Mapping an Address to a Multiword Cache Block

Example

Consider a cache with 64 blocks and a block size of 16 bytes. What block number does byte address 1200 map to?

Answer

The block is given by

$$(\text{Block address}) \bmod (\text{Number of cache blocks})$$

Where the address of the block is

$$\frac{\text{Byte address}}{\text{Bytes per block}}$$

Notice that this block address is the block containing all addresses between

$$\left\lfloor \frac{\text{Byte address}}{\text{Bytes per block}} \right\rfloor \times \text{Bytes per block}$$

and

$$\left\lfloor \frac{\text{Byte address}}{\text{Bytes per block}} \right\rfloor \times \text{Bytes per block} + (\text{Bytes per block} - 1)$$

Thus, with 16 bytes per block, byte address 1200 is block address

$$\left\lfloor \frac{1200}{16} \right\rfloor = 75$$

which maps to cache block number $(75 \bmod 64) = 11$.

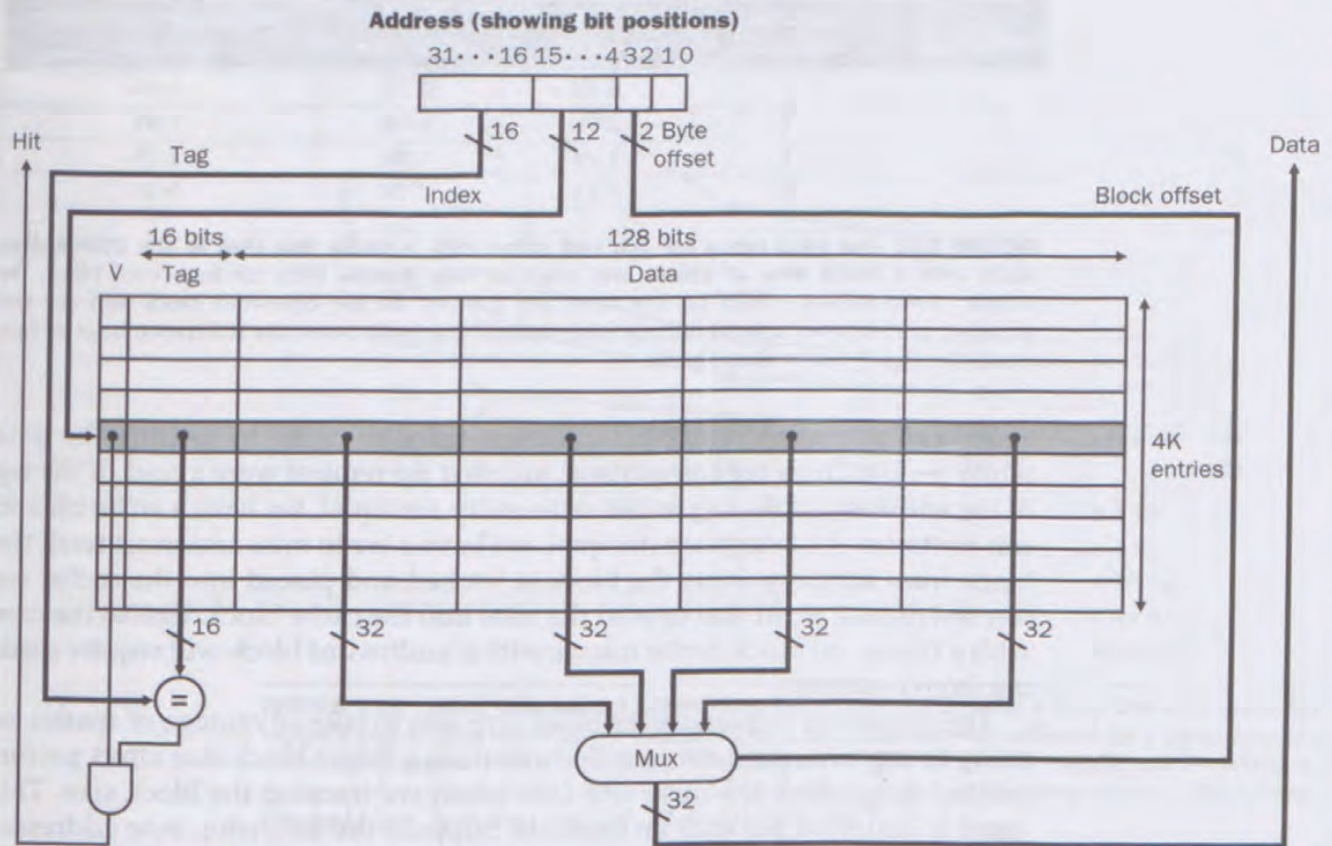


FIGURE 7.10 A 64-KB cache using four-word (16-byte) blocks. The tag field is 16 bits wide and the index field is 12 bits wide, while a 2-bit field (bits 3-2) is used to index the block and select the word from the block using a 4-to-1 multiplexor. In practice, the low-order bits of the address (bits 2 and 3 in this case) are used to enable only those RAMs that contain the desired word, eliminating the need for the multiplexor. Another way to eliminate the multiplexor is to have a large RAM for the data (with the tags stored separately) and use the block offset to supply 2 address bits for the RAM. The RAM must be 32 bits wide and have four times as many words as blocks in the cache.

Read misses are processed the same way for a multiword block as for a single-word block; a miss always brings back the entire block. Write hits and misses, however, must be handled differently than they were in the DECStation 3100 cache. Because the block contains more than a single word, we cannot just write the tag and data. To see why this is true, assume that there are two memory addresses, X and Y, that both map to cache block C, which is a four-word block that currently contains Y. Now consider writing to address X by simply overwriting the data and tag in cache block C. After the write, block C will have the tag for X, but the data portion of block C will contain one word of X and three words of Y!

Program	Block size in words	Instruction miss rate	Data miss rate	Effective combined miss rate
gcc	1	6.1%	2.1%	5.4%
	4	2.0%	1.7%	1.9%
spice	1	1.2%	1.3%	1.2%
	4	0.3%	0.6%	0.4%

FIGURE 7.11 The miss rates for gcc and spice with a cache like that in the DECStation 3100 with a block size of either one word or four words. With the four-word block, we include write misses, which do not incur any penalty for the one-word block and are not included in that case. If write misses were included in both cases, the difference in miss rate would be slightly larger than shown.

We can solve this problem for a write-through cache by writing the data while performing a tag comparison, just as if the request were a read. If the tag of the address and the tag in the cache entry are equal, we have a write hit and can continue. If the tags are unequal, we have a write miss and must fetch the block from memory. After the block is fetched and placed into the cache, we can rewrite the word that caused the miss into the cache block. Unlike the case with a one-word block, write misses with a multiword block will require reading from memory.

The reason for increasing the block size was to take advantage of spatial locality to improve performance. So how does a larger block size affect performance? In general, the miss rate falls when we increase the block size. This trend is easiest to see with an example. Suppose the following byte addresses are requested by a program: 16, ..., 24, ..., 20 and none of these addresses is in the cache. Spatial locality tells us that some pattern of this form is highly probable, although the order of the references may vary. If the cache has a four-word block, then the miss to address 16 will cause the block containing addresses 16, 20, 24, and 28 to be loaded into the cache. Only one miss is encountered for the three references, provided that an intervening reference doesn't bump the block out of the cache. With a one-word block, two additional misses are required because each miss brings in only a single word.

Figure 7.11 shows the miss rates for the programs gcc and spice with one- and four-word blocks. The instruction cache miss rates drop at a rate that is nearly equal to the increase in block size; this larger decrease in the instruction versus data miss rate occurs because the instruction references have better spatial locality.

The miss rate may actually go up if the block size becomes a significant fraction of the cache size because the number of blocks that can be held in the cache will become small, and there will be a great deal of competition for those blocks. As a result, a block will be bumped out of the cache before many of its words are accessed. As Figure 7.12 shows, increasing the block size usually decreases the miss rate. However, the spatial locality among the words in a block decreases with a very large block; consequently, the improvements in the miss rate become smaller—and the miss rate can eventually even increase.

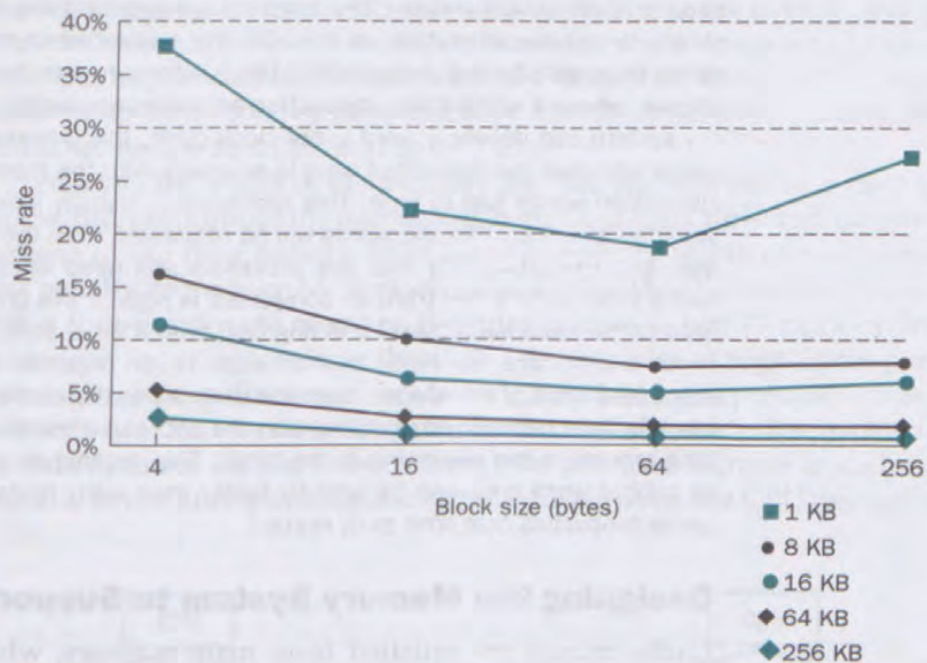


FIGURE 7.12 Miss rate versus block size. For a small 1-KB cache, a large 256-byte block size causes a higher miss rate than the smaller block sizes. This data was collected for a direct-mapped cache using traces (SAVE0) collected by Agarwal for the VAX. More details can be found in A. Agarwal, *Analysis of Cache Performance for Operating Systems and Multiprogramming*, Ph.D. thesis, Stanford Univ., Tech. Rep. No. CSL-TR-87-332 (May 1987).

A more serious problem associated with just increasing the block size is that the cost of a miss increases. The miss penalty is determined by the time required to fetch the block from the next lower level of the hierarchy and load it into the cache. The time to fetch the block has two parts: the latency to the first word and the transfer time for the rest of the block. Clearly, unless we change the memory system, the transfer time—and hence the miss penalty—will increase as the block size grows. Furthermore, the improvement in the miss rate starts to decrease as the blocks become larger. The result is that the increase in the miss penalty overwhelms the decrease in the miss rate for large blocks, and cache performance thus decreases. Of course, if we design the memory to transfer larger blocks more efficiently, we can increase the block size and obtain further improvements in cache performance. We discuss this topic in the next section.

Elaboration: The major disadvantage of increasing the block size is that the cache miss penalty increases. Although it is hard to do anything about the latency component of the miss penalty, we may be able to hide some of the transfer time so that the miss

penalty is effectively smaller. The simplest method for doing this, called *early restart*, is simply to resume execution as soon as the requested word of the block is returned, rather than wait for the entire block. Many machines use this technique for instruction access, where it works best. Instruction accesses are largely sequential, so if the memory system can deliver a word every clock cycle, the processor may be able to restart operation when the requested word is returned, with the memory system delivering new instruction words just in time. This technique is usually less effective for data caches because it is likely that the words will be requested from the block in a less predictable way, and the probability that the processor will need another word from a different cache block before the transfer completes is high. If the processor cannot access the data cache because a transfer is ongoing, then it must stall.

An even more sophisticated scheme is to organize the memory so that the requested word is transferred from the memory to the cache first. The remainder of the block is then transferred, starting with the address after the requested word and wrapping around to the beginning of the block. This technique, called *requested word first*, or *critical word first*, can be slightly faster than early restart, but it is limited by the same properties that limit early restart.

Designing the Memory System to Support Caches

Cache misses are satisfied from main memory, which is constructed from DRAMs. In section 7.1, we saw that DRAMs are designed with the primary emphasis on density rather than access time. Although it is difficult to reduce the latency to fetch the first word from memory, we can reduce the miss penalty if we increase the bandwidth from the memory to the cache. This reduction allows larger block sizes to be used while still maintaining a low miss penalty, similar to that for a smaller block.

To understand the impact of different organizations for memory, let's define a set of hypothetical memory access times:

- 1 clock cycle to send the address
- 15 clock cycles for each DRAM access initiated
- 1 clock cycle to send a word of data

If we have a cache block of four words and a one-word-wide bank of DRAMs, the miss penalty would be $1 + 4 \times 15 + 4 \times 1 = 65$ clock cycles. Thus the number of bytes transferred per clock cycle for a single miss would be

$$\frac{4 \times 4}{65} = 0.25$$

Figure 7.13 shows three options for designing the memory system. The first option follows what we have been assuming so far: memory is one word wide, and all accesses are made sequentially. The second option increases the bandwidth to memory by widening the memory and the buses between the processor and memory; this allows parallel access to all the words of the block.

The third option increases the bandwidth by widening the memory but not the interconnection bus. Thus we still pay a cost to transmit each word, but we can avoid paying the cost of the access latency more than once. Let's look at how much these other two options improve the 65-cycle miss penalty that we would see for the first option (Figure 7.13a).

Increasing the width of the memory and the bus will increase the memory bandwidth proportionally, decreasing both the access time and transfer time portions of the miss penalty. With a main memory width of two words, the miss penalty drops from 65 clock cycles to $1 + 2 \times 15 + 2 \times 1 = 33$ clock cycles. With a four-word-wide memory, the miss penalty is just 17 clock cycles. The bandwidth for a single miss is then 0.48 (almost twice as high) bytes per clock cycle for a memory that is two words wide, and 0.94 bytes per clock cycle when the memory is four words wide (almost four times higher). The major costs of this enhancement are the wider bus and the potential increase in cache access time due to the multiplexor and control logic between the CPU and cache.

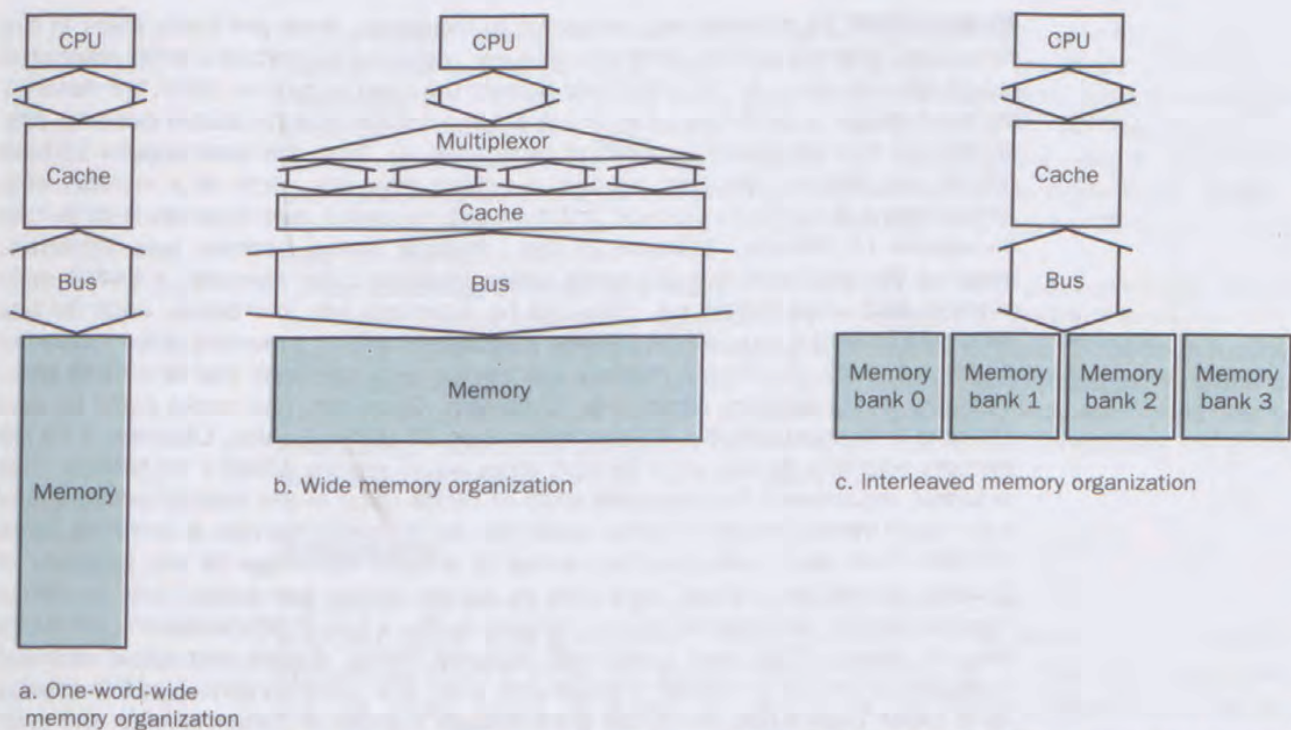


FIGURE 7.13 The primary method of achieving higher memory bandwidth is to increase the physical or logical width of the memory system. In this figure, there are two ways in which the memory bandwidth is improved. The simplest design, (a), uses a memory where all components are one word wide; (b) shows a wider memory, bus, and cache; while (c) shows a narrow bus and cache with an interleaved memory. In (b), the logic between the cache and CPU consists of a multiplexor used on reads and control logic to update the appropriate words of the cache on writes.

Instead of making the entire path between the memory and cache wider, the memory chips can be organized in banks to read or write multiple words in one access time rather than reading or writing a single word each time. Each bank could be one word wide so that the width of the bus and the cache need not change, but sending an address to several banks permits them all to read simultaneously. This scheme, which is called *interleaving*, retains the advantage of incurring the full memory latency only once. For example, with four banks, the time to get a four-word block would consist of 1 cycle to transmit the address and read request to the banks, 15 cycles for all four banks to access memory, and 4 cycles to send the four words back to the cache. This yields a miss penalty of $1 + 1 \times 15 + 4 \times 1 = 20$ clock cycles. This is an effective bandwidth per miss of 0.80 bytes per clock, or about three times the bandwidth for the one-word-wide memory and bus. Banks are also valuable on writes. Each bank can write independently, quadrupling the write bandwidth and leading to fewer stalls in a write-through cache. As we will see, there is an alternative strategy for writes that makes interleaving even more attractive.

Elaboration: As capacity per memory chip increases, there are fewer chips in the same-sized memory system. Memory chips are organized to produce a small number of output bits, usually 1 to 16, with 4 and 8 being the most popular in 1997. We describe the organization of a RAM as $d \times w$, where d is the number of addressable locations (the depth) and w is the output (or width of each location). Thus, the most popular 16-Mbit DRAMs are $4M \times 4$. As memory chip densities grow, the width of a memory chip remains constant (or grows slowly), but the depth increases (see Appendix B for further discussion of DRAMs). Because of this, multiple banks become less attractive, because the minimum memory configuration increases. For example, a 64-MB main memory built using 4-Mbit $\times$ 1 chips can be organized into four banks, each 32 bits wide, and holding a total of 128 DRAMs. If we use 16-Mbit $\times$ 1 memory chips instead of 4-Mbit $\times$ 1 chips, our 64-MB memory can contain only one bank that is 32 bits wide, since the entire memory needs only 32 DRAMs. Of course, four banks could be constructed if we used 4-Mbit $\times$ 4 chips rather than 16-Mbit $\times$ 1 chips. Likewise, a 64-MB memory with four banks using 64-Mbit chips would require 4-Mbit $\times$ 16 DRAMs. This potential requirement for increased width in DRAM chips is the main disadvantage of interleaved memory banks. Another possibility for improving the rate at which we transfer data from the memory to the caches is to take advantage of the structure of DRAMs. DRAMs are logically organized as square arrays, and access time is divided into row access and column access. DRAMs buffer a row of bits inside the DRAM for column access. They also come with optional timing signals that allow repeated accesses to the buffer without a row-access time. One common version of this capability is called *page mode*, which has gone through a series of enhancements, the most recent called EDO (Extended Data Out) RAMs. In page mode, the buffer acts like an SRAM; by changing column address, random bits can be accessed in the buffer until the next row access. This capability changes the access time significantly, since the access time to bits in the row is much lower. For example, the total row and column access times without page mode is about 120 ns; basic page mode provides 60-ns access to data in the page, while EDO mode provides 25-ns access to data in the page. Figure 7.14 shows how the density, cost, and access time of DRAMS have changed over the years.

Year introduced	Chip size	\$ per MB	Total access time to a new row/column	Column access time to existing row
1980	64 Kbit	1500	250 ns	150 ns
1983	256 Kbit	500	185 ns	100 ns
1985	1 Mbit	200	135 ns	40 ns
1989	4 Mbit	50	110 ns	40 ns
1992	16 Mbit	15	90 ns	30 ns
1996	64 Mbit	10	60 ns	20 ns

FIGURE 7.14 DRAM size increases by multiples of four approximately once every three years. The improvements in access time have been slower but continuous, and cost almost tracks density improvements, although cost is often affected by other issues, such as availability and demand. Column access time usually determines the time to perform a page mode access. DRAMs are almost always available in narrower configurations initially (e.g., 4 Mbit $\times$ 4). Wider configurations (e.g., 1 Mbit $\times$ 16) usually track availability of the narrower configuration, and they cost more. Reasons for this are that both the testing cost and package cost for the narrower configuration are slightly cheaper, the narrower configuration is usually the commodity product, and the die size of the narrower configuration is slightly smaller.

The newest development are SDRAMs (synchronous DRAMs). SDRAMs provide for a burst access to data from a series of sequential locations in the DRAM. An SDRAM is supplied with a starting address and a burst length. The data in the burst is transferred under control of a clock signal, which in 1997 can run at up to 100 MHz. The two key advantages of SDRAMs are the use of a clock that eliminates the need to synchronize and the elimination of the need to supply successive addresses in the burst. Together these advantages help lower the time between successive bits from 25 ns for an EDO RAM to 8–10 ns for an SDRAM.

The advantage of these optimizations is that they use the circuitry already largely on the DRAMs, adding little cost to the system while achieving a significant improvement in bandwidth. (The same is true of interleaving.) Furthermore, these DRAM options allow us to increase the bandwidth without incurring system disadvantages in terms of expandability and minimum memory size that are associated with wider memories or interleaving. The internal architecture of DRAMs and how these optimizations are implemented are described in section B.5 of Appendix B.

Summary

We began the previous section by examining the simplest of caches: a direct-mapped cache with a one-word block. In such a cache, both hits and misses are simple, since a word can go in exactly one location and there is a separate tag for every word. To keep the cache and memory consistent, a write-through scheme can be used, so that every write into the cache also causes memory to be updated. The alternative to write-through is a write-back scheme that copies a block back to memory when it is replaced; we'll discuss this scheme further in upcoming sections.

To take advantage of spatial locality, a cache must have a block size larger than one word. The use of a larger block decreases the miss rate and improves

the efficiency of the cache by reducing the amount of tag storage relative to the amount of data storage in the cache. Although a larger block size decreases the miss rate, it can also increase the miss penalty. If the miss penalty increased linearly with the block size, larger blocks could easily lead to lower performance. To avoid this, the bandwidth of main memory is increased to transfer cache blocks more efficiently. The two common methods for doing this are making the memory wider and interleaving. In both cases, we reduce the time to fetch the block by minimizing the number of times we must start a new memory access to fetch a block, and, with a wider bus, we can also decrease the time needed to send the block from the memory to the cache.

7.3

Measuring and Improving Cache Performance

In this section, we begin by looking at how to measure and analyze cache performance; we then explore two different techniques for improving cache performance. One focuses on reducing the miss rate by reducing the probability that two different memory blocks will contend for the same cache location. The second technique reduces the miss penalty by adding an additional level to the hierarchy. This technique, called *multilevel caching*, first appeared in high-end machines selling for over \$100,000 in 1990; since then it has become common on desktop computers selling for less than \$3,000!

CPU time can be divided into the clock cycles that the CPU spends executing the program and the clock cycles that the CPU spends waiting for the memory system. Normally, we assume that the cost of cache accesses that are hits are part of the normal CPU execution cycles. Thus,

$$\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory-stall clock cycles}) \times \text{Clock cycle time}$$

The memory-stall clock cycles come primarily from cache misses, and we make that assumption here. We also restrict the discussion to a simplified model of the memory system. In real processors, the stalls generated by reads and writes can be quite complex, and accurate performance prediction usually requires very detailed simulations of the processor and memory system.

Memory-stall clock cycles can be defined as the sum of the stall cycles coming from reads plus those coming from writes:

$$\text{Memory-stall clock cycles} = \text{Read-stall cycles} + \text{Write-stall cycles}$$

The read-stall cycles can each be defined in terms of the number of read accesses per program, the miss penalty in clock cycles for a read, and the read miss rate:

$$\text{Read-stall cycles} = \frac{\text{Reads}}{\text{Program}} \times \text{Read miss rate} \times \text{Read miss penalty}$$

Writes are more complicated. For a write-through scheme, we have two sources of stalls: write misses, which usually require that we fetch the block before continuing the write (see the elaboration on page 607 for more details