LIBRARY
LIBRARY OF CONGRESS

Office of Business Enterprises
Duplication Services Section

THIS IS TO CERTIFY that the collections of the Library of Congress contain a book entitled ***Memory Systems and Pipelined Processors*** by Harvey G. Cragon, and that the attached photocopies—the spine, front cover, back cover, front end page, title page, publication information page, table of contents, page 18, pages 82 through 95, pages 109 through 113, pages 191 through 193, and pages 231 through 234—are a true representation from that work.

THIS IS TO CERTIFY FURTHER, that the spine bears a Library of Congress call number that reads "QA 76 .5 .C68 1998 COPY 2" and a label that reads "FT MEADE GenColl"; that the front end page bears a Library of Congress stamp; and the back cover bears a label that reads "Library of Congress 0 003 464 349.0."

IN WITNESS WHEREOF, the seal of the Library of Congress is affixed hereto on September 27, 2024.

Rosalina Delgado-Jones
Assistant Business Enterprises Officer
Office of Business Enterprises
Library of Congress



101 Independence Avenue, SE Washington, DC 20540–4917 Tel 202.707.5650 www.loc.gov;
duplicationservices@loc.gov

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000001
IPR2025-00486 (Advanced Micro Devices, Inc., Pensando Systems, Inc. v. Concurrent Ventures, LLC, XtreamEdge, Inc.)

Memory Systems and Pipelined Processors

Crago

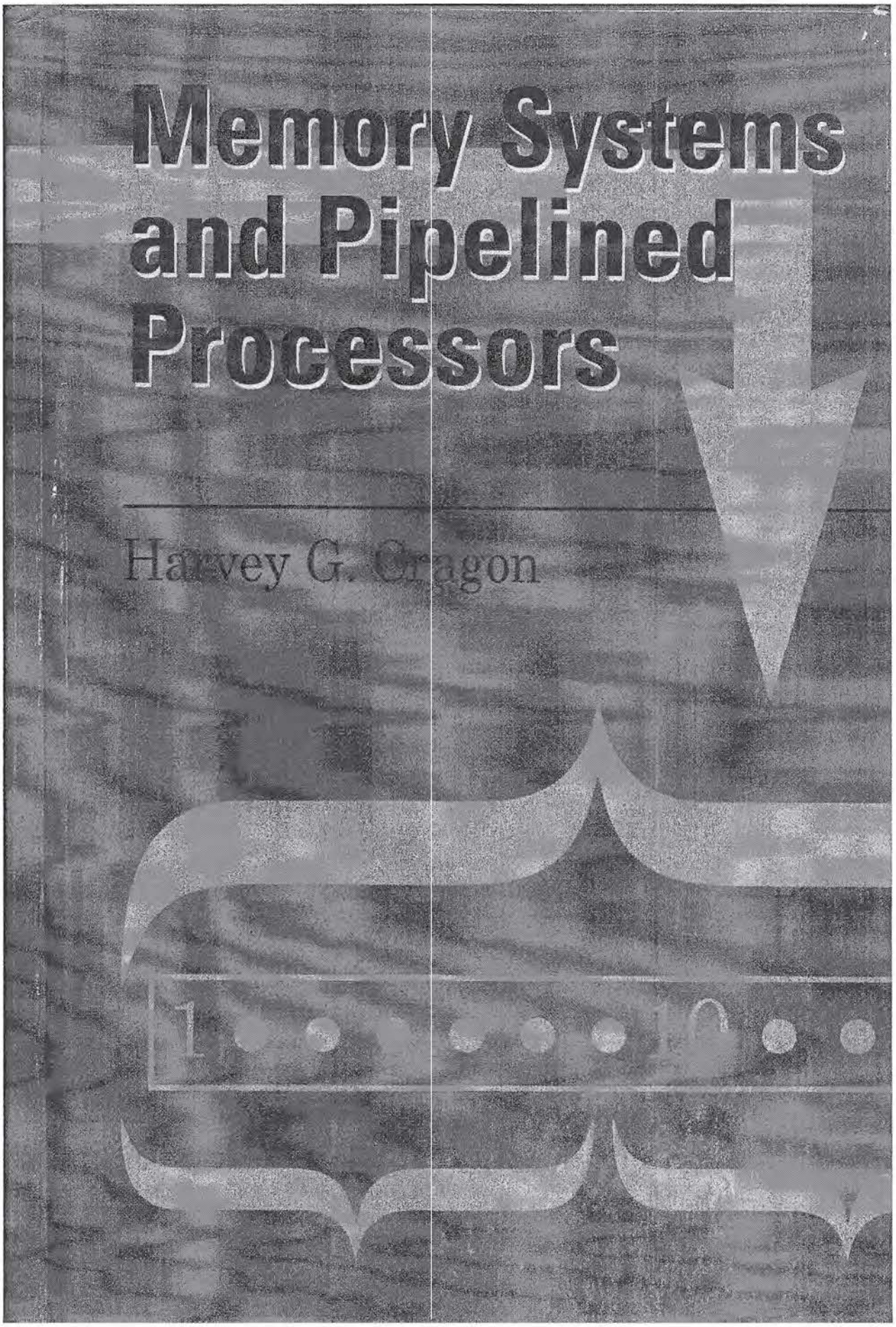
QA 76
.5
.C68
1996
COPY 2

FT MEADE
GenColl

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000002

Memory Systems and Pipelined Processors

Harvey G. Cragon



Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000003

AXIOMS FOR IMPLEMENTATION

Computer designers strive to provide the best performance consistent with cost objectives. I have collected a few rules for implementation that are called *Axioms of Implementation* and are listed below; the order does not signify relative importance.

1 Functions that consume the most time should be the object of effort to speedup the total process while recognizing that the ultimate speedup is limited by the function(s) that cannot be speeded up.

"This axiom is a restatement of Amdahl's observation that "effort expended on achieving high parallel processing rate is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude."



2 Spatial and/or temporal concurrency in hardware can improve performance.

Spatial concurrency is found in split caches, superscalar implementations that issue more than one instruction in a clock, multiple I/O channels, and the like. Temporal concurrency is found, for example, in pipelined processors, pipelined caches, and concurrent input/compute/output scheduling.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000004

Memory Systems and Pipelined Processors

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000005

Memory Systems and Pipelined Processors

Harvey G. Cragon

The University of Texas at Austin



Jones and Bartlett Publishers

Sudbury, Massachusetts

Boston London Singapore

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000006

LIBRARY OF CONGRESS
OCT 3 0 1995
COPYRIGHT OFFICE

QA76
.5
.C68
1996
Copy2

Editorial, Sales, and Customer Service Offices
Jones and Bartlett Publishers
One Exeter Plaza
Boston, MA 02116
1-800-832-0034
617-859-3900

Jones and Bartlett Publishers International
7 Melrose Terrace
London W6 7RL
England

Copyright © 1996 by Jones and Bartlett Publishers, Inc.

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission from the copyright owner.

Trademarked products mentioned in this book are listed on page 576.

Library of Congress Cataloging-in-Publication Data

Cragon, Harvey G.
Memory systems and pipelined processors / Harvey G. Cragon.
p. cm.
Includes bibliographical references and index.
ISBN 0-86720-474-5
1. Computers, Pipeline. 2. Computer storage devices. I. Title.
QA76.5.C68 1995
004.25'6--dc20
95-22155
CIP

Acquistions Editor: Carl Hesler
Production Editor: Anne S. Noonan
Manufacturing Buyer: Dana L. Cerrito
Editorial Production Service: Superscript Editorial Production Services
Typesetting: Scanway Graphics International
Printing and Binding: Braun-Brumfield
Cover Design: Beth Santos
Cover Printing: John P. Pow Company

Printed in the United States of America
99 98 97 96 95 10 9 8 7 6 5 4 3 2 1

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000007

Contents

Preface xi

1 Memory Systems 1

1.0 Overview 1
1.1 Memory Latency 4
1.1.1 Hierarchical Memory Systems 5
1.1.2 Memory Efficieny 7
1.2 Memory Bandwidth 10
1.3 Queues 13
1.4 Summary 14

2 Caches 15

2.0 Overview 15
2.1 Cache Organization and Placement Policies 19
2.1.1 Set Index Allocation 27
2.1.2 Late/Early Select Caches 29
2.1.3 Control Bits: Tags, Valid, Dirty, Shared and LRU 31
2.1.4 Nonblocking Caches 32
2.2 Read Policies and Performance Models 34
2.2.1 Transport Time 37
2.2.2 Cache Miss Characteristics 39
2.2.3 Sources of Miss and Hit Data 41
2.2.4 Fetch Policies 42

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000008

2.2.5 Write Policies 45
2.2.6 Replacement Policy 54
2.2.7 Block Size Considerations 56
2.2.8 Bus Design Considerations 57
2.3 Instruction Buffering Schemes 60
2.3.1 Instruction Buffers 61
2.3.2 Instruction Queues 62
2.3.3 Instruction Caches 68
2.3.4 Published Instruction Caches Hit Data 69
2.3.5 Contemporary Instruction Caches 72
2.4 Data Caches 72
2.4.1 Published Data Cache Miss Data 73
2.4.2 Contemporary Data Caches 73
2.4.3 Data Queues 78
2.5 Unified Caches 78
2.5.1 Published Performance Data 81
2.5.2 Contemporary Unified Caches 82
2.6 Split and Multilevel Caches 82
2.7 Multilevel Caches 91
2.7.1 Published Performance Data 94
2.8 Other Cache Considerations 95
2.8.1 Processor Control 95
2.8.2 Transport Time Hiding or Reduction Methods 97
2.8.3 Compiler Optimization 107
2.8.4 Pipelined Caches 107

3 Virtual Memory 109

3.0 Overview 109
3.1 Virtual Memory Concepts 113
3.2 Memory Mapping 117
3.2.1 Creating Large Virtual Addresses 118
3.2.2 One-Level Map Address Extension 118
3.2.3 Register Extensions 120
3.2.4 Summary 120
3.3 Virtual Memory Organizations 121
3.3.1 Page Allocation Systems 123
3.3.2 Segmented Systems 153
3.3.3 Paged Segmentation 156
3.4 Translation Lookaside Buffers 162
3.4.1 TLB Organization 165
3.4.2 TLB Miss Ratio Data 174
3.4.3 TLB Impact on Virtual Memory Performance 177
3.4.4 Instruction Support for TLBs 177

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000009

3.5 Virtual Memory Accessing Rules 178
3.5.1 Read Accesses 178
3.5.2 Allocation 180
3.5.3 Write Accesses 184
3.5.4 Location of Tables 184
3.5.5 Table Walking Methods 185
3.5.6 Instruction Set Support for Virtual Memory 187
3.5.7 Memory Access Control 189
3.5.8 Choice of Page Size 193
3.5.9 Addressing the Disk 195
3.6 Virtual Memory Performance Issues and Models 197
3.6.1 Published Page Fault Ratio Data 199

4 Memory Addressing and I/O Coherency 201

4.0 Overview 201
4.1 Cache Addressing, Virtual or Real 203
4.1.1 Real Address Caches 204
4.1.2 Pipelined Real Caches 206
4.1.3 Virtual Address Cache 209
4.1.4 Restricted Virtual Caches 212
4.1.5 Summary 213
4.1.6 TLB Addressing 215
4.2 I/O System Addressing Design Issues 215
4.2.1 Cache Coherency 217
4.2.2 Snoopy Controllers 220
4.2.3 DMA I/O Configurations 222
4.2.4 Processor Control Over Snoopy Controllers 225
4.3 Other Considerations 227
4.3.1 Memory-Mapped I/O 227
4.3.2 Cache Coherency Protocols 227
4.3.3 Snooping on Write Queues 229
4.3.4 Summary 230

5 Interleaved Memory and Disk Systems 231

5.0 Overview 231
5.1 Interleaved Memory 231
5.1.1 Performance Models 234
5.1.2 Reducing the Effect of Strides≠1 237
5.1.3 Superinterleaving 241
5.1.4 Interleaving for Multiport Access 248
5.1.5 Cache-Memory Transfers 249
5.1.6 Nibble, Page, and Static Column Mode Drams 250

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000010

5.1.7 Wide-Word Interleaved Modules 251
5.2 Disk Systems 252
5.2.1 Disk Requirements 254
5.2.2 Disks in File I/O and Virtual Memory 265
5.2.3 Disk Arrays 272
5.2.4 Disk Caches 283

6 Pipelined Processors 287

6.0 Overview 287
6.1 Pipeline Performance Models 290
6.2 Pipeline Design Considerations 294
6.3 Other Pipeline Performance Models 298
6.3.1 Pipeline Efficiency 300
6.4 Pipeline Partitioning 301
6.5 Contemporary Pipelined Microprocessors 305
6.6 Exclusion or Synchronization Instructions 306
6.6.1 Processors That Address Memory 309
6.6.2 Load/Store Processors 311
6.6.3 Synchronization Summary 315

7 Branching 317

7.0 Overview 317
7.1 Modeling Technique 318
7.2 Pipeline Freeze Strategies 319
7.3 Prediction Strategies 321
7.3.1 Static Prediction Strategies 324
7.3.2 Semistatic Prediction Strategies 326
7.3.3 Dynamic Prediction Strategies 327
7.4 Instruction Sequence Alteration Strategies 339
7.4.1 Delayed Branch Strategy 339
7.4.2 Delayed Branch with Abandon 341
7.4.3 Delayed Branch with Fast Compare 343
7.4.4 Branch Spreading with Condition Codes 343
7.5 Fetch Multiple-Paths Strategies 344
7.5.1 Fetch Two Paths 345
7.5.2 Fetch All Paths 347
7.5.3 Hardware Requirements 349

8 Dependencies 351

8.0 Overview 351
8.1 Instruction Issue and Release Techniques 355

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000011

8.2 Data Register or Data Path Dependencies 358
8.2.1 True Dependencies 358
8.2.2 Resource Dependencies 375
8.2.3 Other Methods 380
8.3 Memory Dependencies 385
8.3.1 Anti- and Output Dependencies 385
8.3.2 True Dependencies 385
8.3.3 Load Bypass 387
8.3.4 Self-Modifying Code: True Dependencies 387
8.3.5 MU5 Name Store 388
8.4 Special Register Dependencies 390
8.4.1 Address Registers 390
8.4.2 Status Registers 391
8.4.3 Program Counter 393
8.4.4 Global Impact Registers 394
8.5 Performance Models 397
8.6 Summary Comments 401
8.7 Structural Dependencies 402
8.7.1 Single-Function Pipelines 403
8.7.2 Multi-Function Dynamic Pipelines 408
8.7.3 Arithmetic Pipelines 412

9 Exceptions and Interrupts 425

9.0 Overview 425
9.1 Precise Interrupts 429
9.1.1 Interrupted Instruction and Saved Program Counter 433
9.1.2 Handling Preceding Instructions 434
9.1.3 Handling Following Instructions 455
9.2 Performance and Cost of Precise Interrupts 461
9.3 Precise Interrupt Comments 462

10 Enhanced Implementations 467

10.0 Overview 467
10.1 Superpiplined Processors 468
10.1.1 Elementary Performance Model 469
10.1.2 A Refined Model 471
10.1.3 Superpipeline Design Issues 473
10.1.4 Current Example of Superpipelining 474
10.2 Superscalar Processors 475
10.2.1 Elementary Performance Model 476

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000012

10.2.2 Superscalar Design Issues 481
10.2.3 Superscalar Branching Strategies 491
10.2.4 Dependency Detection and Resolution 492
10.3 Performance Comparison 494
10.4 Superscalar Superpipelined Processor 497
10.4.1 Processor Parallelism 497
10.4.2 Instruction-Level Parallelism 499
10.4.3 Processor Performance 500
10.5 Very Long Instruction Word (VLIW) Processors 502

11 Vector Processors 507

11.0 Overview 507
11.1 Vector Operations 509
11.2 Memory-Processor Interface 510
11.2.1 Memory-to-Memory Architecture 511
11.2.2 Load/Store Architecture 512
11.3 Vector Performance 513
11.3.1 Assessment by Vector Instruction Latency 514
11.3.2 Streaming Rate 515
11.3.3 Assessment by MFLOPS 518
11.3.4 Assessment by Half Performance 520
11.3.5 Vector-Scalar Code Mix Problem 521
11.4 Vector Instructions 523
11.5 Mask Operations 532
11.6 Dependencies 534
11.6.1 Intrastatement Dependencies 534
11.6.2 Interstatement Dependencies 535
11.6.2 Chaining 536
11.7 Benchmark Performance 538

References 541

Subject Index 567

Processors Index 574

Trademarks 576

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000013

Read, a read access is made by the processor to a cache for instructions or data.
Write, a write access is a processor write into the cache and/or main memory.
Fetch, a read access to memory to load a block into the cache.
Access, a generic reference to the cache or memory.

Caches can be classified by the interconnection topology; the selection and interconnection of the various cache components. A multilevel cache is one that has a first-level cache (either instruction, unified or split) and a second-level unified cache. The second-level cache is designed to accept the missed references from the first-level cache(s). Each of the two levels can be

N Nil, not implemented;
I Instruction cache, buffer or queue;
D Data cache;
S Split cache;
U Unified cache.

Some reasonable, or known, two-level cache topologies are shown in Figure 2.1. A more complete description of various topologies is found in Section 2.6.

Historically, caches have been viewed as a pure implementation technique for improving performance that is transparent to the program. However, following this view, the program can misuse the cache with a performance penalty when there is no recognition of the presence or absence of a cache. This view of the cache has, of necessity, changed; software support for caches is now becoming the norm. There are things the program can or should explicitly do with the cache. For example, two of these actions are: (1) prefetch a cache block and (2) clear the cache to maintain coherency. These program actions are specific to individual architectures and are discussed in Section 2.8.1.

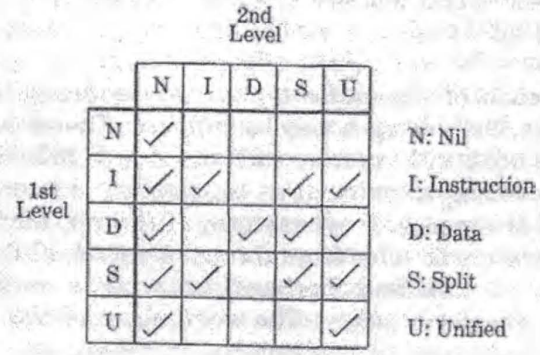


FIGURE 2.1 Common cache topologies.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000014

shows excerpted P_{miss} for the benchmarks, a combination of floating point and integer, and a block size of 16 bytes.

A unified cache, as with instruction and data caches, shows that the size of the cache is paramount and the organization is of secondary importance. For example, an 8-K cache, organized as (64, 8, 16, 1), has a P_{miss} of 0.041 while the organization (512, 1, 16, 1) has a P_{miss} of 0.056. On the other hand, for cache design where the number of sets is constrained for some reason (discussed in Chapter 4), doubling the size of the cache by increasing the number of sectors can cause a significant decrease in P_{miss}. For example, an organization (256, 2, 16, 1) has P_{miss} = 0.047 while (256, 4, 16, 1) has P_{miss} = 0.029.

2.5.2 *Contemporary Unified Caches*

Many contemporary first-level caches are designed for VLSI implementation of new processors and new operating systems. For a number of reasons these caches are split as discussed in Section 2.5. All known new processors are implemented as pipelines that place heavy demands on the memory system. Because of this factor, some form of first-level split caches or multilevel caches is the design of choice today. The exception to the universal use of split caches is the IBM RS/6000 Model 220, which has a unified cache purely because of a marketing consideration; this processor model is targeted for a low-cost, low-performance market.

However, with split on-chip caches, a second-level unified cache is the norm. For example, the VAX 9000 has a first-level instruction cache followed by a second-level unified cache. The i486 has a second-level unified cache and an instruction queue. These caches are discussed in Section 2.6.

2.6 Split and Multilevel Caches

I can find no historical evidence of separate data and instruction caches before the mid-1980s. During the generation typified by IBM S/360 and VAX architectures, the caches were generally unified. It is only since microprocessor pipelining that the possibility of making concurrent references to instructions and data and with VLSI technology that designers have considered split and multilevel caches.

The major reason for using split caches is to provide a second path to memory from the processor. With pipelined processors, an instruction fetch and a data read/write can occur on the same clock, creating a structural hazard if there is only a unified cache. Two caches eliminate this hazard and permit concurrent references. Early use of two paths to memory were called "*Harvard Architectures*" [CRAG80]. Structural

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000015

hazards are discussed further in Chapter 8. However, a split cache has problems with self-modifying code because a specific instruction could have two values—one in the instruction cache and another value in the data cache.

Another reason for the use of split caches is that P_{miss} is different for instructions and data. A look at the data from Tables 2.12, 2.13, 2.14, and 2.15 shows that significant differences between instruction P_{miss}, data P_{miss}, and unified P_{miss} can exist for caches of equal size. For example, a 16-Kbyte cache organized as (512, 2, 1, 16) will have instruction $P_{miss} = 0.01$, data $P_{miss} = 0.104$, and unified $P_{miss} = 0.047$ for the overall SPEC benchmarks. With these differences, the possibility exists that the design of the three caches of a two-level cache system can be customized. There is a significant problem with split caches if self-modifying code is used, resulting in coherency problems between the two caches. A modified instruction in the data cache may be referenced in the instruction cache—a problem that must be managed either by software or hardware means.

Most of the early research on hierarchical memory systems recognized that there would be a continuum of levels [MATT70, MEAD70] and not just the processor/cache/memory levels. With practical VLSI and pipeline implementations, interest in multilevel caches has revived. Przybylski [PRZY90] discussed two reasons for renewed interest. First, all of the information presented previously shows that the P_{miss} of a cache is not significantly improved if the cache size is increased beyond 64 Kbytes. Thus, ever larger caches are ineffective. Second, transport time can be reduced with a second-level cache, which reduces the number of cycles of a first-level cache miss. The reason for this is basically that the clock period for determining transfer time between the first- and second-level caches is much shorter than that between a single-level cache and the main memory.

Baer [BAER87] reports that the only system, as of 1987, to have a multilevel cache is the Facom M-3825. The first-level cache is organized as (64, 16, 1, 64) 64-Kbytes write through, while the second level is organized as (1024, 4, 1, 64) 256 Kbytes write back. Note that contemporary multilevel cache systems usually employ a very large second-level cache. The reason for this is that with VLSI memory devices, the second-level cache can be made large enough to hold a complete working set, thereby significantly reducing bus traffic to the main memory.

As shown in Figure 2.1, there are a number of topological configurations of cache systems that cannot be simply classified. This section will describe some configurations that have been employed. A partial listing of processors and their topologies is given in Table 2.17. Examples of the cache designs are described in the paragraphs below.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000016

First Level	Second Level	Examples
I-Queue	Unified	i386, AM29000
I-Queue	Unified, off-chip	i486, PowerPC 601
I + D Unified I-Cache D-Cache	Unified, off-chip	MC68030, MC68040 MC88200, RS/6000
I-Queue I-Cache	Unified	VAX 9000
I-Queue D-Cache	Unified, off-chip	Intel Pentium

TABLE 2.17 Processor cache topologies.

First level, I-Queue;
Second level, Unified Cache;
Example, i386

The i386 processor is implemented with limited chip area with space for only an instruction queue. The supply of instructions to the queue and the supply of data are provided by an off-chip unified cache [ELAY85]; the organization is shown in Figure 2.23.

As previously discussed in Section 2.3.2, the on-chip instruction queue is used to smooth out the flow of variable-length instructions to the processor and hide cache misses. The off-chip unified cache organization and size are the responsibilities of the system designer for this processor. However, a suggested cache design is given by Intel in

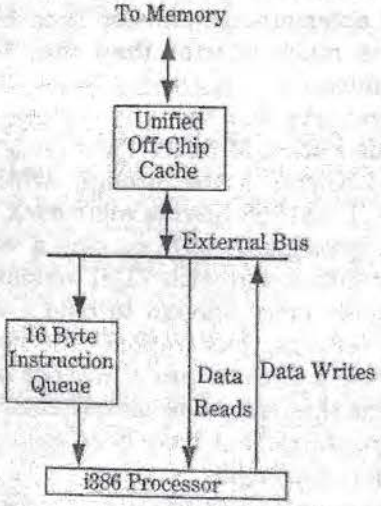


FIGURE 2.23 i386 cache and instruction queue.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000017

[INTE86] along with an application note on the design of caches in general [SHIR86].

One of the major considerations that led to the selection of a unified cache is that the i386 had to execute old versions of MS-DOS that have self-modifying code. This consideration dictated a unified cache, external to the processor, for the i386. Intel also provides a cache component (part number 82396SX) for use with the i386. This cache organization is (256, 4, 1, 16) using a pseudo-LRU replacement algorithm and is the precursor to the i486 on-chip data cache, which is discussed below. Shires [SHIR86] provides hit ratio and performance ratios of various i386 unified cache organizations, executing an anticipated workload. This data indicates the dominant effect of cache size rather than organization as noted in previous sections.

First level, I-Queue;
Second level, Unified Cache;
Example, AMD 29000

AMD moved away from split caches for the AM29000 family. Software portability problems, in addition to complex bus issues, led to a return to a single bus access to the main memory. There is a four-word instruction queue (called a prefetch buffer by AMD) and a very large (192-word) register file that provide cache-like characteristics. A single bus services requests to memory that may be partitioned into instruction and data regions [AMD90].

First level, I-Queue + Unified;
Second level, Unified Cache;
Example, i486

The designers of the Intel i486, [FU89, CRAW90], needed to support two memory accesses (instruction and data) per clock period. The instruction length is 1 byte to 15 bytes and the data word is 1 byte to 8 bytes. Thus the cache bandwidth needs to be in the range of 2–23 bytes per clock. In addition, the designers had a chip budget of 8 Kbytes in which to provide an on-chip cache.

As with the i386 cache, a unified cache is needed to support old MS-DOS and OS/2 systems that did not have "cache flush" instructions that evict the complete cache. These instructions are generally needed to support cache consistency. See Section 2.8.1 for more details.

The organization of the i486 cache is (128, 4, 1, 16) [INTE92]. To provide the ability to read instructions and read/write data simultaneously, the i486 has an instruction queue, as shown in Figure 2.24 and described in Section 2.3.2. The size of the cache and the number of sets are directly related to the size of the page (4 Kbytes) used in the virtual

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000018

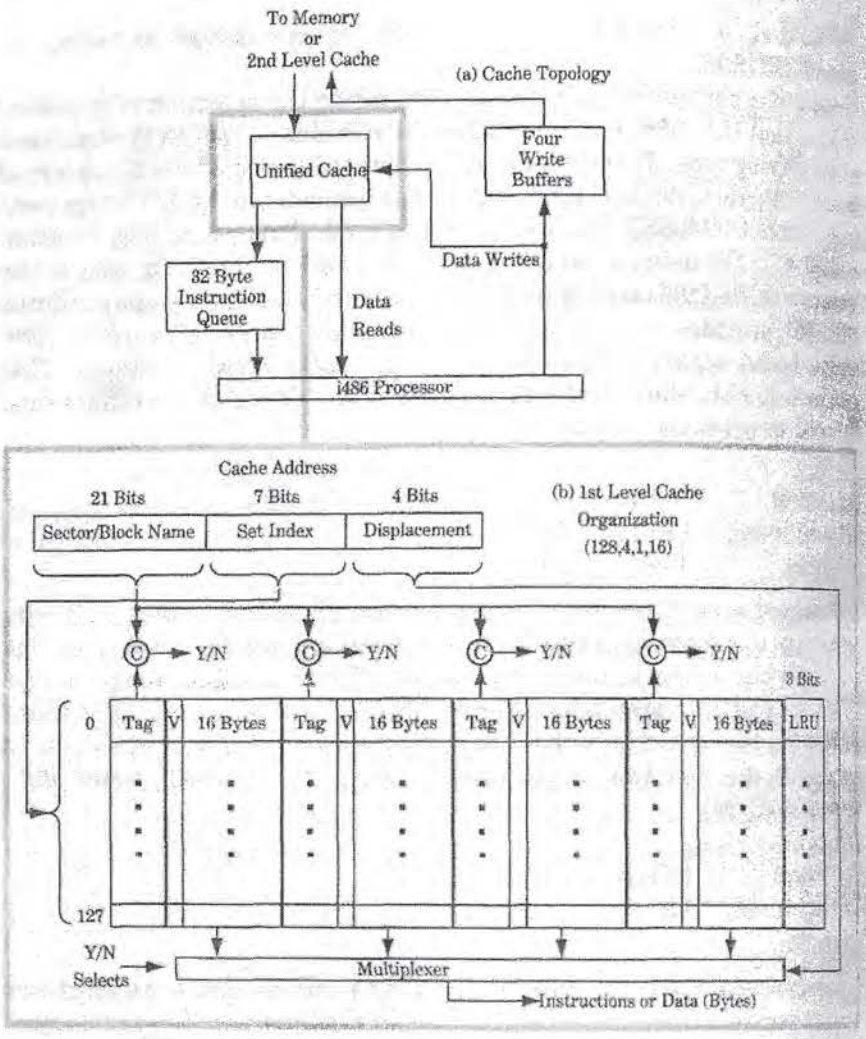


FIGURE 2.24 Intel 486 cache.

memory. By matching these dimensions, one-half of the cache (two sector-blocks from all of the sets) is equal to one page. This organization permits a cache access and an access to the TLB to be made in parallel (an issue that is discussed in Chapter 4).

Each of the 128 sets has four sector-blocks with tags and valid bits. Because the write policy is write through, dirty bits are not required. However, each set has three bits (B2, B1, B0) for maintaining a pseudo-random LRU status of the sector-blocks within each set. These bits are set/reset as the cache is filled and sector-blocks are modified as shown in Table 2.18.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000019

Sector-Block Used	B2	B1	B0
S0	X	1	1
S1	X	0	1
S2	1	X	0
S3	0	X	0

TABLE 2.18 i486 cache LRU bits.

If all of the valid bits are "set," a sector is selected for replacement using the LRU bits that have recorded the use patterns of the sectors. Selection is simply the decoding of the B bits. For example, [0X0] identifies S3 as the LRU sector-block, and it should be overwritten; write back is not required since the write policy is write through.

A complete discussion of the design tradeoffs of the i486 cache is found in [GROC89]. The i486 cache design was tested by trace-driven simulation and gave a $P_{miss} = 0.08$ versus the $P_{miss} = 0.044$ shown in Table 2.16 for the same organization executing the SPEC benchmarks. These simulations showed that with a fast bus, the write through policy is only slightly faster than a write back policy. As the bus slows down, the write through cache slows down more than the write back and the bus utilization increases more for the write through cache. However, a write through cache is simpler to implement and was selected for the i486. With a write through policy, the bus can be expected to be utilized approximately 68% of the time, making shared memory multiprocessor configurations difficult to implement.

The instruction queue prefetches instructions via a 16-byte-wide bus from the unified cache in one clock into the prefetch queues. Thus, one block, which is four average-length instructions, is fetched into the queue in one clock. Instructions are then read from the queue into the processor.

It is interesting to note that, because the cache is on the same chip as the processor, the 16-byte bus between the cache and the prefetch queues poses minimum wiring problems as would be the case if such a wide bus is required to traverse chip boundaries.

First level, I-Queue + D-Cache;
Second level, Unified Cache;
Example, MC68030

The Motorola MC68030 has on-chip split instruction and data caches. Due to the limited chip area, these caches are only 256 bytes each. Because these caches are small, an effective application of this processor demands an off-chip second-level unified cache. Figure 2.25 illustrates the cache topology and the identical instruction and data caches.

The instruction and data caches are organized identically as (16, 1,

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000020

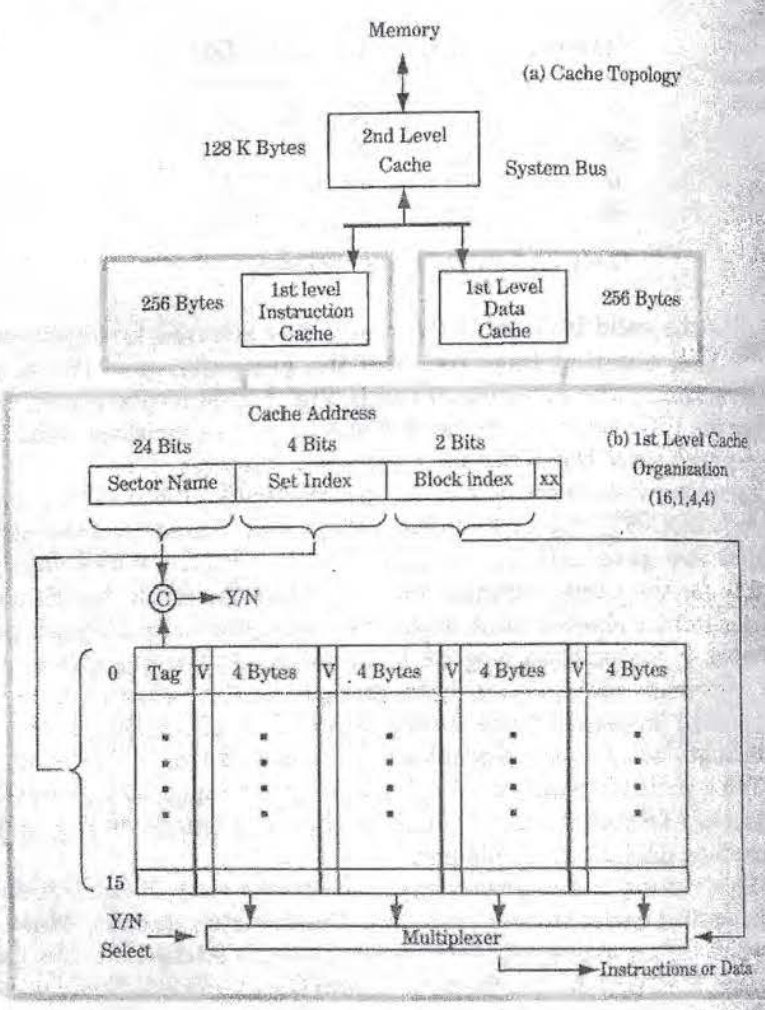


FIGURE 2.25 Motorola 68030 instruction and data caches.

4, 4); the set index of four bits selects one of the 16 sets. If the set is present, the two block index bits select one of the four 4-byte words in the block; each block has a valid bit. The data cache uses a write through policy on a write hit. The data cache policy on a write miss is program-selectable for either write-allocate write back or no-write-allocate write through. As with all instruction caches, there is no write policy for the instruction cache. Furthermore, as self-modifying code is not supported, the instruction read process does not have to check the data cache for an instruction that has been modified.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000021

First level, I-Queue + D-Cache;
Second level, Unified Cache;
Example, DEC Alpha

The DEC Alpha processor, introduced in 1992, has a split cache on-chip and a second-level cache off-chip. The data cache has a capacity of 8 Kbytes, write through, direct mapped, and write-allocate. The data cache is byte addressable and the organization is (256, 1, 1, 32). The instruction cache has a capacity of 8 Kbytes, is also direct mapped with a 32-byte block, and is organized (256, 1, 1, 32). Both caches are addressed with the real addresses. The second-level cache, named the B-Cache, is unified, and the design is the system designer's responsibility.

First level, I-Queue + I-Cache;
Second level, Unified Cache;
Example, VAX 9000

The VAX 9000 [FITE90] has a multilevel cache consisting of a first-level instruction cache of 8 Kbytes and a second-level unified cache of 128 Kbytes as shown in Figure 2.26. There is no first-level data cache; data transfers directly to and from the register file and the second-level cache. The variable-length instructions are issued from a 25-byte instruction queue.

The instruction cache organization is (256, 1, 2, 16) and is addressed with virtual addresses to reduce latency and simplify the design. The cache address has a one-bit (B) block address and a one-bit (D) displacement address. As the cache AU is eight bytes, the three LSBs of the address are not used. A miss can load either one or two blocks as there is a valid bit per block. There are two sets of 256 valid bits (one for each sector-block); these bits are not integrated into the cache array (similar to the IBM S/360/85 cache).

The second-level unified cache is organized as (1024, 2, 1, 64) and is also addressed with virtual addresses. This cache is 2-way set associative and uses a FIFO block replacement scheme. It is believed that a write back policy is used in conjunction with a write queue.

First level, I-Queue + I-Cache + D-Cache;
Second level, Unified Cache;
Example, Intel Pentium

The recently announced Intel Pentium employs a split cache, unlike previous members of the ix86 family. The very large number of transistors available to the designers allowed them to solve the self-modifying code problem. The instruction read path is, starting at the processor, via a decoded instruction queue, an instruction queue, an instruction cache,

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000022

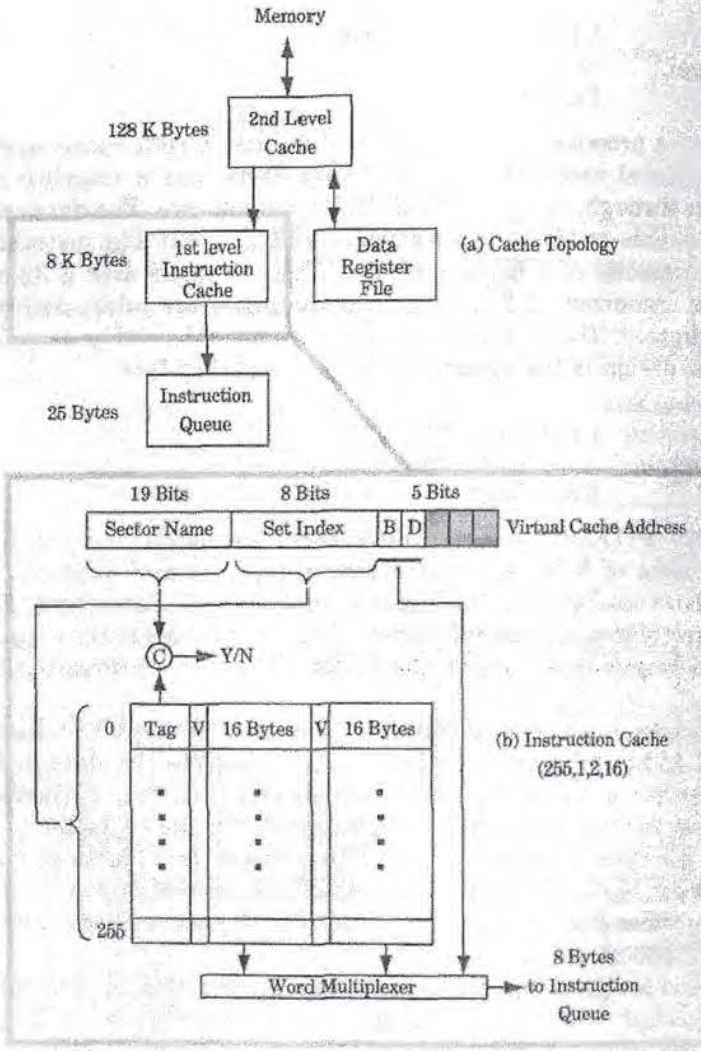


FIGURE 2.26 VAX 9000 cache.

a unified cache, and the main memory. The data read path is via a data cache, unified cache, and the main memory.

As discussed previously, the Intel ix86 family of microprocessors have employed unified caches because of the problem of self-modifying code. This design philosophy has changed with the Pentium processor [INTE93, CRAW92, GWEN92, SLAT92]. There are two on-chip caches, an 8-Kbyte instruction cache and an 8-Kbyte data cache, and both are organized as (128, 2, 8, 4). The transfer path between the caches and the main memory is via a 64-bit bus. Logic is included to test for true

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000023

dependencies in the data cache to determine if instructions have been modified. Dependency management is discussed in Chapter 8.

The instruction queue can supply up to three instructions in one clock, while the data cache can supply two 32-bit words to the register file. The data cache has a unique interleaved organization that supplies operands to the superscalar pipelines (this design is described in Section 10.2.2).

2.7 Multilevel Caches

The idea at work with a multilevel cache is that a small and fast first-level cache is used and that the second-level cache can (1) reduce the transport time to the first-level cache and (2) reduce the bus loading. With the proper interplay of these ideas, the cache system performance will be improved. Reduction in transport time is advantageous because on-chip caches were initially small with a resulting high P_{miss}. Because the weighted time to reference an AU not in the cache is the product of P_{miss} and transport time, these two parameters are kept in balance with a second-level cache. (Another form of multilevel cache, *disk cache*, is discussed in Section 4.2.4.) Bus-loading reduction is important for multiprocessor systems where a number of processors must communicate over one bus to the main memory.

A simple two-level unified cache system is shown in Figure 2.27. This system has caches L1 and L2 between the processor and the main memory. The access time of L1 is t_1, the transport times for the second-level cache and the main memory are t_2 and t_3, respectively; and P_{miss1} and P_{miss2} are the *local* P_{miss} for the two caches. *Local* P_{miss} is the number of misses in a specific cache divided by the number of references to that cache.

The effective access time, as seen by the processor, for the simple two-level cache system using the sequential-forward model and local P_{miss} values is

$$t_{es}(\text{two levels}) = t_1 + P_{miss1} \times t_2 + P_{miss1} \times P_{miss2} \times t_3 .$$

The performance of a two-level cache can be compared to a one-level

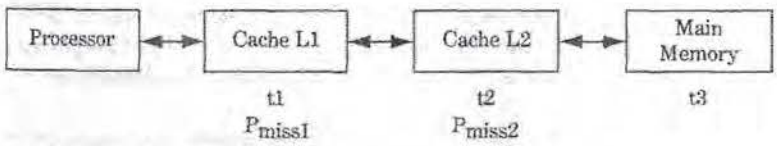


FIGURE 2.27 Two-level cache.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000024

cache under the assumption that the P_{miss} of a single-level cache is the same as the P_{miss} of the first-level cache C1. The speedup is

$$S = \frac{t_{ea}(\text{one level})}{t_{ea}(\text{two levels})} = \frac{t_1 + t_{ea} P_{miss} t_3}{t_1 + P_{miss1} t_2 + P_{miss1} P_{miss2} t_3}.$$

For example, assume the parameters

$$\begin{aligned} t_1 &\quad 1, \\ t_2 &\quad 4, \\ t_3 &\quad 10, \\ P_{miss1} &\quad 0.05, \\ P_{miss2} &\quad 0.02. \end{aligned}$$

Using these parameters, the speedup of the two-level cache over the single-level cache is $S = 1.5/1.21 = 1.24$. In other words, adding this second-level cache provides a reduction of approximately 20% in effective access time. Note that P_{miss} for different levels are not independent. Changing the design of one level will, in many caches, influence the P_{miss} of another level. In the context of multilevel caches, two other definitions are of interest:

Global P_{miss} is the number of misses in a cache or memory in the system divided by the total number of references to the memory system from the processor.

Global P_{hit} is the number of references satisfied in a cache or memory in the system divided by the total number of references to the memory system from the processor. The sum of all global P_{hit} values of a system is 1.

For an example of global P_{miss} values and global P_{hit} values, consider the following. If local $P_{miss1} = 0.1$ and local $P_{miss2} = 0.05$, then the *global miss ratio* for level two is $0.1 \times 0.05 = 0.005$. A more useful global metric is the global hit ratio. In this example, 90% of all references are satisfied in the first-level cache, 9.5% in the second-level cache, and 0.5% in the main memory.

The complexity of selecting an organization and policy for a one-level cache is compounded in the case of a two-level cache. The number of points in the design space is approximately the square of the number of design points of a one-level cache. For example, what is the policy when there is a miss on one of the caches? And, what is the policy when there is a miss on both of the caches. J.-L. Baer [BAER87] provides a discussion on these design problems.

To illustrate the complexity of a policy for a two-level cache, consider that the four write policy designs discussed in Section 2.2.6 will expand

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000025

	L1	L2	ACTION
READ	Hit	X	read L1 → processor
*	Miss	Hit	Reload L2 → L1, read L1 → processor
	Miss	Miss	Reload Mem → L2, L2 → 1, read L1 → Processor
WRITE	Hit	Hit	Write processor → L1 → L2
*	Hit	Miss	Allocate to L2, write processor → L1 → L2
*	Miss	Hit	Write processor → L2
	Miss	Miss	Allocate to L2, write processor → L2

*Does not provide multilevel inclusion.

TABLE 2.19 Multilevel cache policies.

to sixteen possible designs for a two-level cache. Not all of these designs are reasonable. One example that illustrates a particular selection of write strategies is found in the policy described in [BAER 87] and in Table 2.19. The write policies of the two caches are

Level 1: (hit) Write Through, (miss) No Write-allocate (Design A);
Level 2: (hit) Write Back, (miss) Write-allocate (Design C).

With this multilevel cache if, for example, there is a miss on L1 and a hit on L2, the block in L1 is replaced by the block in L2 and the new ALU is read into the processor. The actions in the table are viewed as logically sequential; however, a parallel implementation is possible. In other words, the new block of this example can be placed in L1 and forwarded to the processor.

For a single-level cache, the assumption is made that resident blocks in the cache must be found in the main memory. In other words, all blocks in the cache are *included* in the main memory. The cache system cannot have a valid block that is not represented in the main memory; the values may not be the same due to the delay waiting for a write back. With the addition of another cache level, *multilevel inclusion* (MLI), which means that the contents of a cache at level $l + 1$ is a superset of the contents of its cache at level l (MATT70, LAM79, BAER87]), may not exist unless explicitly enforced.

The lack of MLI is illustrated by an example. Consider a block that has been moved to the L2 and L1 caches due to misses at both levels. A subsequent reference to the L2 cache (from I/O, say) may, due to a set conflict miss, evict that block. Now the block exists only in the L1 cache and in the main memory, not in the L2 cache. In this case,

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000026

multilevel inclusion is not provided. When there is a multiprocessor with a cache for each processor, MLI is more difficult to achieve.

Note in the example of Table 2.19 that this cache design does not provide MLI. For the read cases, as there can be an L1 hit and L2 miss, L2 is not a superset of L1. Likewise, as there can be a write with a miss on L1 and a hit on L2 and a hit on L1 with a miss on L2, L2 is not a superset of L1. Baer states that the first-level cache must be write through if multilevel inclusion properties are to be maintained.

There is also the issue of addresses presented to the various cache levels. The question of whether a cache should be addressed with real or virtual addresses will be discussed in Chapter 4. With a multilevel cache the issue is even more difficult. In fact, [WANG89] argues that the first-level cache should be addressed with virtual and the second-level with real addresses, an organization named *V-R caching*.

2.7.1 *Published Performance Data*

Przybylski [PRZY88] provides a wealth of information on the design and optimization of multilevel cache. Data on the effect of a second-level cache on bus utilization and increases in system performance are found in [GOLB92] for i486 processors. Bus utilization reduction is compared to an i486DX processor without a second-level cache. Two bus configurations are evaluated, namely, a slow bus that has a one clock hit and a four clock miss time and a fast bus with a one clock hit and a four clock miss time. The second-level cache is set at 128 Kbytes, 256 Kbytes, and 512 Kbytes. Excerpted bus utilization data are shown in Table 2.20. Note that a large second-level cache reduces the bus utilization to a negligible amount.

The performance improvement with a second-level cache was also measured and normalized to a 25-MHz i486 without a second-level cache. A 50 MHz without a second-level cache was evaluated to show the performance improvement obtainable with raw processor performance, and then this system was measured with a 256-Kbyte second-level cache.

Second-level Cache	Slow Bus	Fast Bus
None	88%	68%
128 Kbytes	19%	11%
256 Kbytes	15%	6%
512 Kbytes	5%	4%

TABLE 2.20 i486 bus utilization with a two-level cache.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000027

System	Benchmarks		
	ispec	fspec	spec
25-MHz i486 (none)	1	1	1
50-MHz i486 (none)	1.61	1.62	1.61
50-MHz i486 (256 Kbytes)	2.09	2.1	2.1

TABLE 2.21 i486 performance improvement with a second-level cache.

The results of measurement while executing three SPEC benchmarks are excerpted in Table 2.21.

The interesting results here are that while doubling the clock rate of the processor gives a speedup of 1.6, adding a second-level cache gives an additional speedup of 1.3. This result compares favorably with the results of the speedup model developed earlier.

2.8 Other Cache Considerations

In addition to the design issues of organization, policies, and performance, there are other issues that influence the design and efficacy of caches. Some of these issues are control of the cache by the processor, transport time reduction techniques, compiler optimization, and pipelined caches.

2.8.1 *Processor Control*

Caches have historically been viewed as transparent to the user or processor. That is, the cache provides lower latency reads and writes than can be provided by the main memory, and the executing program should not be aware of the presence of a cache. This view of a transparent cache is no longer valid, and it has been found that there are situations that require direct processor control of the cache. A cache that has processor control is known as a *software visible cache*.

Bakoglu discusses the requirements for software visible caches [BAKO90].

> Software visible caches have two implications: The first is that any program that uses data references to create instructions (for example, loaders, debuggers, and simulators) must explicitly force these instructions from the data cache into the instruction cache. Additionally, to perform input or output operations on the processor main

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000028

3

Virtual Memory

3.0 Overview

Virtual memory has evolved in response to a number of factors. First, the address space of early processors was closely matched to the size of the *real memory* (main memory) and, for many reasons, was found to be inadequate. Second, software schemes that enabled the operating system to manage the transfer of information from bulk storage were developed. Third, virtual memory helped to overcome the performance penalty generated by these software memory management schemes.

The organization, or structure, of the virtual memory system results from matching the data structures employed by programmers. Program (code and data) modularity has been an accepted practice for many decades to facilitate independent development and checkout of software. Modules can be viewed as a hierarchy, and the allocation system should facilitate the modular view. A common data structure for representing modules is the tree, as shown in Figure 3.1, that reflects the hierarchical nature of many commonly occurring data structures. Files are referenced from the Root Directory via interior nodes or subdirectories. The files for early versions of UNIX were variable-length segments.

It is informative to survey the evolution of software memory management systems that preceded virtual memory. Figure 3.2(a) shows that the early *simple batch monitors* provided a single contiguous memory partition for a single user. When the user program is changed—due perhaps to the termination of a job—the active user partition is written out to disk and a new job is read in. The major drawback with this system is the overhead incurred in swapping and the inefficient use of processor and memory resources.

These early batch monitor systems were also called *roll-in/roll-out* because of the way jobs are moved from disk to the main memory. Upon the first instance of the job being moved to the main memory, real addresses are assigned that are used in subsequent references. This

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000029

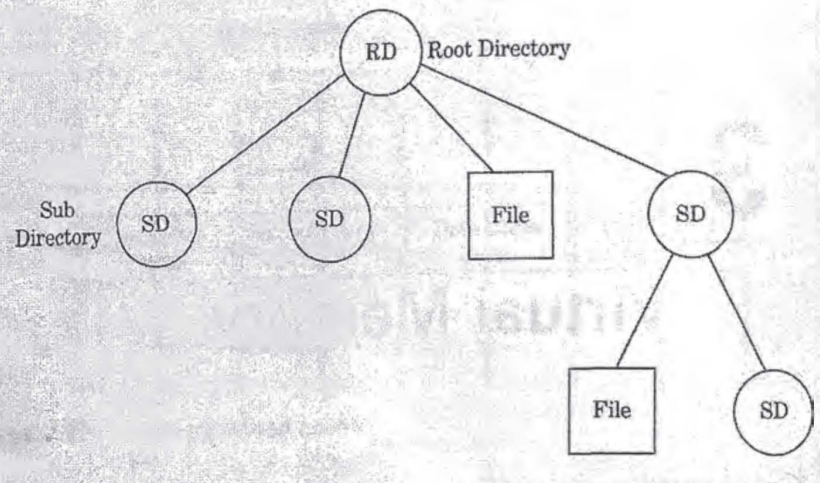


FIGURE 3.1 Data structure.

presents the problem that a new job may not be loadable from disk, even though there is available memory space, because the particular required address space is occupied.

The efficiency problems of the batch monitor are addressed in software-managed memory systems by *partitioned allocations*, as shown in Figure 3.2(b). Given that real memory is large enough to hold a number of partitions, each partition is allocated its own address space in real memory. The monitor then manages these partitions, assigning space, adjusting addresses, and swapping partitions in and out. As jobs are completed, space in memory is vacated and the monitor attempts to load another partition. The problem with this system is primarily that memory becomes fragmented and is used inefficiently.

Figure 3.2(b) shows a 30-K job (Job 5) waiting to be allocated. Even after Jobs 1 and 4 are complete (the assumed completion sequence) Job 5 cannot be allocated, even though 35K of memory space are available. Only after Job 2 is completed does space in memory become available for allocation of Job 5. Fragmentation clearly results in the inefficient use of memory.

Early research into solutions for the problem of fragmentation led to the identification of two types of fragmentation in a virtual memory system [RAND68, RAND69].

External fragmentation is caused in real memory when the allocation system does not relocate used space to free up unused space for allocation to a new task. Thus, the real address space becomes fragmented into groups of unusable spaces. (This fragmentation is the situation described above.)

Internal fragmentation occurs when the average space required for

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000030

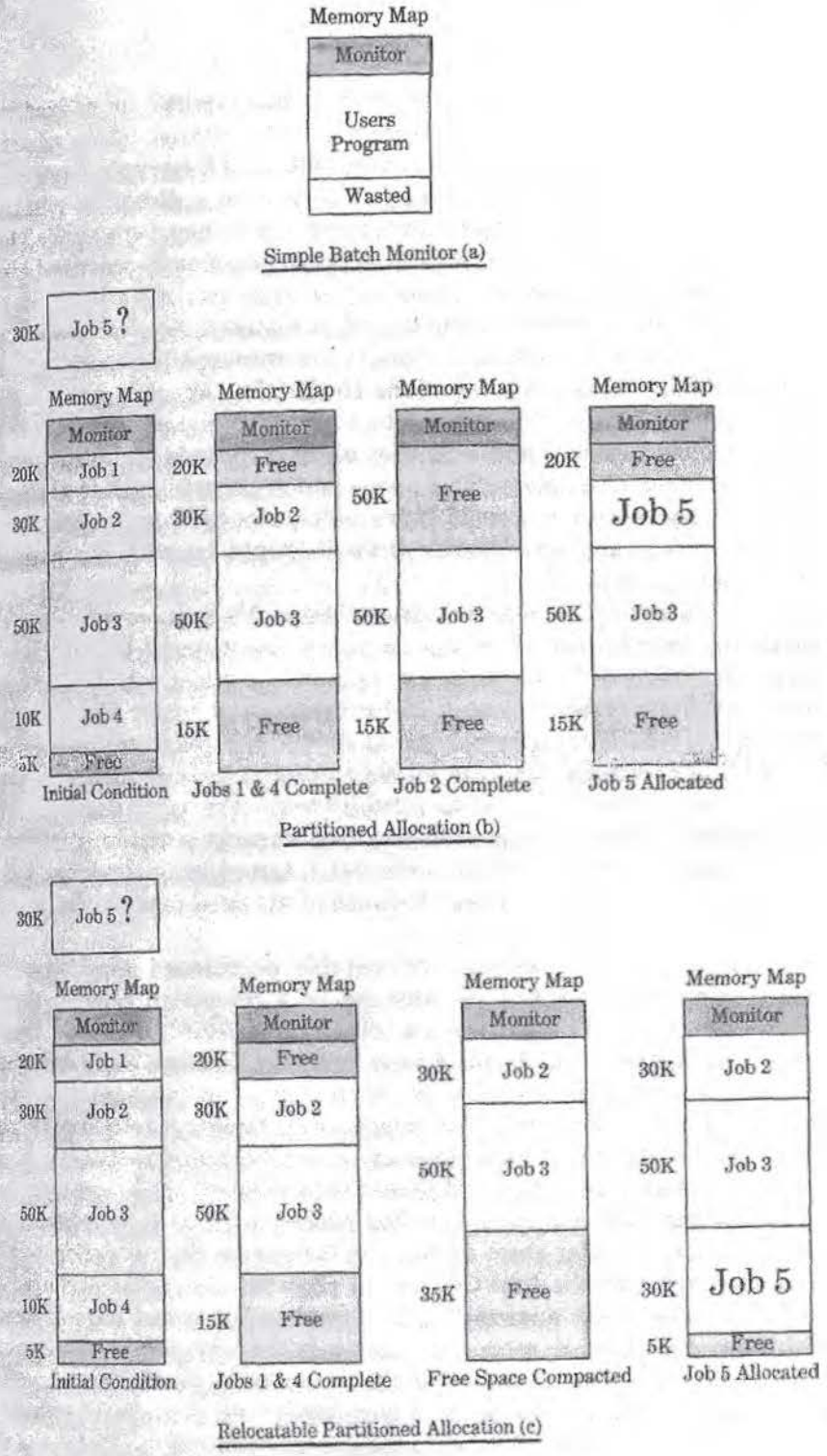


FIGURE 3.2 Early software managed memory systems.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000031

an allocation is slightly larger than a page, which is a fixed-length allocation unit. Thus an extra page must be allocated, which wastes real memory space. Note that external and internal fragmentation were first applied in relation to the allocation of files on a disk. The meaning of these terms has been somewhat altered since they were applied to virtual memory systems. (This form of fragmentation is described later.)

External Fragmentation was solved by means of a *relocatable partitioned allocation*, which is illustrated in Figure 3.3(c). Relocatable partitioned allocation means that as jobs are terminated, the operating system compacts the remaining partitions so that free space is made available to waiting jobs. Note that after Jobs 1 and 4 are completed, the operating system moves Jobs 2 and 3 so that all free space is compacted into one address space in memory. This space is now large enough to allocate Job 5. Not only is memory used more efficiently by this process, Job 5 is available for execution earlier than it would be with the partitioned allocation system.

Relocation introduces an addressing problem, however. When partitions are moved, all addresses in the program must be changed. This problem is solved by the use of a *relocation register*, which points to the start of the partition that had been allocated under the assumption that the partition starts at address 0. All accesses to memory have the relocation register's contents added to the address. Relocation registers do not eliminate the need to actually move the partition, but they do minimize the addressing problems. Relocatable partitioned allocations go a long way to solve the external fragmentation problem but can introduce significant overhead because of the need to move the partitions around in memory.

Paged allocation extends relocatable partitioned allocations into a more generalized technique. Instead of a relocation register for each partition, paged allocation uses a relocation register for pages of approximately 4 Kbytes (size varies across systems). In some early systems the operating system managed these page relocation registers. Because the partitions are subdivided into pages, each having a relocation register, it is unnecessary to move pages as with relocatable partitions. A partition is broken into pages and placed into available page locations in the address space of real memory. The program can still be written under the assumption of a start address of 0 because the relocation registers change each address into the correct page location address. Paged allocation is the basic approach found in *demand paged virtual memory*, which is described in more detail later in this chapter.

Segmented allocation involves relocatable partitions of variable length and may be viewed as a variable-length page that supports the program's view of the data structure. Segmentation facilitates the sharing of programs and data.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000032

The problems of software support of the above-mentioned memory management schemes have led researchers to see if the processor itself could make these tasks transparent to the executing program, in other words, the development of *virtual memory*. The first description of a virtual memory applied to the Atlas computer [KILB62], a design motivated by the desire for a large linear address space that hid the memory management tasks from the programmer. Another early experimental paged system was implemented at Berkeley [LAMP66]. The IBM S/370 family, first delivered in 1970, extended the IBM S/360 family to paged virtual memory capability. Cocke [COCK59] described the "virtual memory" of Stretch; however, this scheme is in reality an early form of cache.

A "pure" segmented system (pure means that a complete segment is fetched at one time from disk) was first implemented in the late 1950s on the experimental Rice University computer. The Burroughs B5000, first delivered in 1965, was the first large-scale commercial implementation of virtual memory and used segmentation [LEVY84]. In England, ICL implemented a segmentation machine in the early 1960s [ILIF68].

For almost three decades, research activity has produced hundreds of research papers and reports. An authoritative early work on virtual memories is by Denning [DENN70], a paper that explains most of the tradeoffs in the design of a virtual memory system. [PARM72] is another review paper, describing work at IBM, with an annotated bibliography. This paper looks not only at virtual memory but virtual machine concepts as well. [DONO72] reflects the MIT Project MAC view of memory management and virtual memory.

3.1 Virtual Memory Concepts

The term *virtual memory* has been used to denote a computer with a large linear address space that hides memory management tasks, discussed above, from the programmer. Virtual memory can be implemented with pure software or combinations of hardware and software. The following sections describe the use of hardware support to remove from the operating system some of the requirements that are necessary to manage the details of allocation. For the ensuing discussion, I define the following terms.

Symbolic name is the set of all names used by the programmer.

Effective address is the address generated by the processor and may, in some cases, be extended into a virtual address.

Virtual address is the machine equivalent of a symbolic name.

Real address (physical address) is the set of all addresses in the real memory.

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000033

level, or at the block level if block translation is employed. Examples of the use of control bits that provide protection can be found in a number of processors. The i386, i486, and Pentium devote two bits in the descriptor tables, which provide four levels of protection. These levels are: kernel, system services, custom extensions, and application. Two protection bits are also used in the page frame table entry of the RS/6000. In the most general implementation of rings of protection, if control is being transferred to a ring other than the current one, an interrupt occurs and the operating system is invoked to perform the proper housekeeping tasks.

Proper Access

Protecting memory from an improper access is the second issue of access control. One task should not be able to interfere with another task without mutual agreement. Four possible ways [SITE80] of providing memory protection are

1. bounds registers,
2. memory keys,
3. translation table keys,
4. distinct addresses spaces.

These four protection methods are discussed individually below. It should be noted, however, that combinations of the four methods are implemented in some specific computers.

Bounds registers were first used in nonvirtual memory machines. Bounds registers provide two pointers: one to an upper address limit, the other to a lower address limit. Bounds can also be specified with a starting address and length. If a reference is found to be outside these bounds, an error interrupt is signaled. The bounds registers are loaded and stored by the operating systems using privileged instructions.

For a system that will have a number of tasks resident in memory at one time, there may be a number of bounds register pairs, one for each task. The register pairs are addressed by a pointer that is set when a task is switched in. The use of bounds registers assumes that a task, program, and/or data, is allocated within a contiguous address space. Bounds registers permit sharing between tasks of an address space by identifying overlapping spaces by the bounds registers. Bounds registers, however, present restrictions with, for example, a single word containing a semaphore that is accessed by a number of tasks. The semaphore would have to be in the highest address of one task and in the lowest address in a second task. Access by three tasks is difficult.

For paged systems, protection is at the page or block level and for segmented systems at the segment level. For example, the segment

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000034

length field of a segment table operates in the fashion of a bounds register. An example of this is the block translation system of the PowerPC 601 that has a block partition of memory that is similar to a segment. A block name is presented to the block TLB and is checked for being within the starting address and the length of the block. Block length is encoded with six bits, rather than having an upper address, in the BAT registers discussed previously. The length of the block is a binary progression: 128 Kbytes, 256 Kbytes, 512 Kbytes, . . . , 8 Mbytes, an encoding designed to facilitate length checking [MOTO93].

Memory keys specify the type of access that is permitted to a memory location, page, or segment. Typical access types are: Unlimited, Read Only, Execute Only, and/or a task I.D. The executing program key is usually found in the processor status register that, in combination with the type of access, is compared to the "lock" associated with the memory and found in the name translation system. If the key matches the lock, the requested access is performed. There is usually a "skeleton key" that can be used by the operating system to access all of memory regardless of the key associated with each of the keyed memory spaces.

The memory keys are usually contained in the name translation maps and/or the TLB. Thus, an improper access is terminated before the read or write is actually performed. Memory keys permit access to noncontiguous page frames in real memory, unlike bounds registers. Sharing is more difficult, however, as sharing is at the block level. Shared variables may need to be relocated with a context switch increasing context switch overhead. If different accesses are legal for different tasks, the protection fields in the TLB must be changed when tasks are switched.

The PowerPC 601 illustrates the use of memory keys. The supervisor mode has the keyed accesses of read/write and read only, while the user mode has the keyed accesses of no access, read only, and read/write. The operating system can block user programs from accesses by evoking the no access key. This description is not complete; additional information can be found in [MOTO93].

Translation table keys provide protection by having an access table for each task. These tables, one for each task, contain the legal access methods for each page of real memory and permit different legal accesses for different tasks with a minimum of overhead. As there are multiple tables, only a switch of a pointer is needed to select the table for the new task. Images of the keys are found in the page tables and are checked along with the check of the virtual address tag.

Distinct address spaces can be used to provide protection by assuring that different tasks do not share the same virtual address space. This protection is accomplished by making the task name a part of the virtual address. With the task name in the most significant bits, there can be no

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000035

overlap between addresses in different tasks. One example of a distinct address space system is found in the ix86 processors that have segment registers associated with each type of access, for example, code and stack.

3.5.8 *Choice of Page Size*

This section addresses four tradeoffs in determining page size.

1. Reducing internal fragmentation.
2. Reducing disk/memory transfer time.
3. Matching the page size to the levels of the name translation tables.
4. Reducing the miss ratio.

Internal fragmentation was introduced in Section 3.1. When a page size is selected, there is a waste of usable memory space. First, because the allocation block is not an exact multiple of the page size, internal fragmentation occurs. Internal fragmentation is reduced with small pages because the wasted space is limited to a fraction of a smaller page. Second, when a program and its data is allocated to real memory, space must be provided for the page table entry of the allocated page. Page table space is not available for the program and data; it is pure overhead space. As the page size decreases, more pages must be allocated and the space used for page table entries increases. Thus, what page size will balance these two counterforces and minimize the loss of memory? Early work on this design problem, and the model derived below, is found in [WOLM65], who attributes the first version of the model to J. B. Kruskal. This model determines an optimum page size to minimize wasted real memory.

Assume that the number of AUs that must be allocated for a program (instructions and data) is n; the page size is p; and a AUs are required for each page table entry. A direct one-level page table, in memory, is assumed. Thus, the use of main memory by page tables and internal fragmentation represents a loss of real memory to the user.

Memory loss = loss due to internal fragmentation + loss for allocating a page table entry,

$$\text{Loss due to internal fragmentation} = \frac{p}{2},$$

$$\text{Loss for allocating a page table entry} \approx \frac{an}{p},$$

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000036

5

Interleaved Memory and Disk Systems

5.0 Overview

This chapter discusses two important subjects, interleaved memory and disk systems, that are usually given only minor attention. Interleaved memory, which is used to provide high bandwidth, was of great importance prior to the advent of practical high-speed memories for caches. Currently, innovative architectures of modern DRAMS and various forms of interleaving are receiving design and research attention. Disk technology has had a remarkable life since the first commercial disks in 1956, the IBM 350 RAMAC. Disks have survived all predictions that the technology could go no further; indeed, cost and performance of disks have continued to improve.

5.1 Interleaved Memory

Interleaved memories provide high-memory bandwidth by distributing the read or write accesses over a number of memory modules, which is a form of parallelism. The array disk to be described in Section 5.2.3 is another instance of memory interleaving. Early research into interleaving was concurrent with and a "back up" to hierarchical memory research. In the late 1950s and 1960s no one was sure if cache technology development would be successful.

The first known reference to interleaving is found in a description of the IBM Stretch [BLOC59]: "The memories themselves are interleaved so that the first two memories have their address distributed modulo 2 and the other four are interleaved modulo 4."

A later reference to the IBM S/360 [FAGG64] points out that: "When

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000037

accesses are made to sequential storage addresses, the storage units operate in an interleaved fashion."

Pirtle [PIRT67] observes: ". . . the addresses are distributed over an n module memory so that address 0 is in $m(0)$, address 1 is in $m(1)$ and, in general, address X is in $m(i)$ where $i \equiv X$ modulo n."

We can see from these statements that there are two uses of the word *interleaved*. One use concerns the *units* or *modules*. Interleaved modules subdivide the module cycle time into clock periods that establish the streaming rate of the memory system. The degree of module interleaving is m and is formally defined as the number of requests that can be issued by the processor in one memory module cycle [BURN70]. Note that the clock period of the memory and the bus system are related as $m = t_m/t_c$ where t_m denotes the memory cycle time and t_c the bus or processor clock time. The other use of the term interleaving concerns *addresses*. Interleaving addresses across modules establishes the sequence of module access for a stream of requests. The degree of address interleaving is n, which is defined as the number of memory modules over which the addresses are interleaved [BURN70].

In this chapter, memory systems are described for $1 < m \leq n$. With many systems, the two degrees of interleaving are equal; that is, $m = n$. However, with some systems, such as those found with supercomputers, the degree of address interleaving is greater than the degree of module interleaving; that is, $m < n$, which is called *superinterleaving* and is discussed in Section 5.1.3.

Interleaving presents two questions of interest to researchers. First, can interleaved memories provide enhanced bandwidth to supply instruction and data streams to a processor? And second, what are the design parameters of an interleaved memory that supports vector reads and writes of supercomputers? With the success of caches, the first research objective has almost disappeared as an interesting topic. The second research topic is still an area of active research and is discussed in more detail in Section 5.1.3. Note that the IBM RT uses a noncache interleaved memory system [ROWL86] to provided instruction and data bandwidth.

The requirement for accessing vectors comes from two sources. The first source is found in the main memory references of supercomputers. Scientific programs frequently access arrays of data stored in memory as vectors. The second source is found in the interface between a cache and the main memory. A block of AUs are transferred between these two memories; this block is a vector of length 4 to 16 and is discussed in Chapter 2.

Interleaved memories are ideal for providing high bandwidth access to a long vector of data (either read or write). An interleaved memory requires two addresses: the address of the module and the address of

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000038

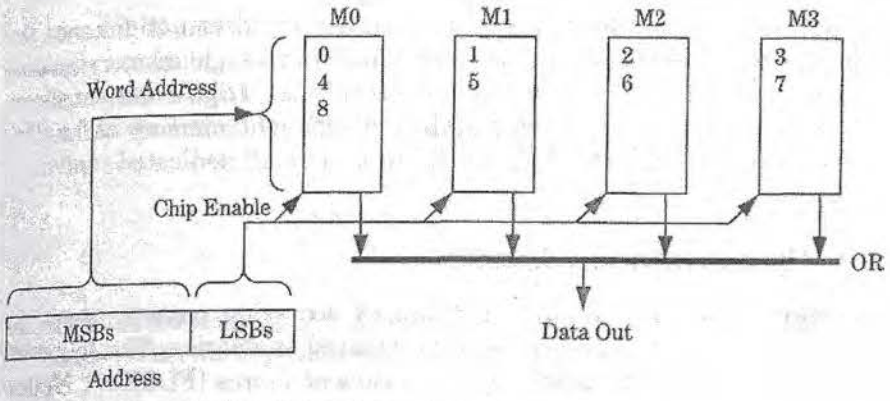


FIGURE 5.1 Four-way interleaved memory.

the word within the module. Low-order interleaving generates these two addresses by dividing the memory address by the address interleaving degree n: the quotient is the word address presented to each module and the reminder selects the module. This division by an integer power of two is a trivial operation for designs having a degree of interleaving that is an integer power of two.

Consider a memory of total size 2^s, with $m = n = 4$, as shown in Figure 5.1. The four modules are addressed by the two LSBs of the address, and the word within the addressed module is selected by the high-order bits. This process is called *low-order interleaving*.

Addressable units are stored in the modules according to the sequence

Module	Addresses
0	$0, 4, \ldots, 2^s - m + 0$
1	$1, 5, \ldots, 2^s - m + 1$
2	$2, 6, \ldots, 2^s - m + 2$
3	$3, 7, \ldots, 2^s - m + 3$

A major disadvantage of low-order interleaving is the difficulty of expanding the installed memory. If the installed memory is to be expanded by a factor of 2, for example, how is this to be done? Should the degree of interleaving be increased by 2, or should the module size be increased by a factor of 2? Increasing the interleaving factor requires a major hardware modification. Increasing the size of each of the modules is a rather straightforward task. However, both expansion methods become completely intractable if the expansion is something other than an integer power of 2.

Another form of interleaving is called *high-order interleaving*, in which the quotient selects the module and the remainder selects the

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000039

word within the module. High-order interleaving does not enhance the bandwidth of the system over the bandwidth of a single memory module but instead provides for easy memory expansion. High-order interleaving also permits the easy intermixing of different memory technology such as a combination of RAM and ROM in a small dedicated controller.

5.1.1 *Performance Models*

Performance models for various types of accessing patterns were developed during early research on interleaved memories. The following paragraphs will briefly describe the models of Flores [FLOR64], Hellerman [HELL67], and Burnett and Coffman [BURN70]. These models were created to assist designers evaluate the performance impact of various interleaved memories with different workloads.

There are three useful metrics for measuring and evaluating the performance of an interleaved memory. These are

1. Speedup, S = ratio of the bandwidth of an interleaved memory to the bandwidth of a single module.
2. Acceptance ratio, AR = steady-state ratio of accepted memory requests to total memory requests [CHEU86], $0 \leq AR \leq 1$. AR is an indicator of how effectively the memory design sustainable bandwidth is being used.
3. Mean acceptance ratio, MAR = acceptance ratio that prorates the latency of accessing the first module over all of the AUs accessed [CHEU86]; $0 \leq MAR \leq 1$.

Instruction and Data Streams

The models for estimating the performance of an interleaved memory are of three forms: (1) instruction references, (2) data references, and (3) combined references. The addressing patterns of these three streams are quite different and require different performance models. Note that the problems or conflicts associated with interleaved memory systems are special cases of *structural dependencies* (discussed in Chapter 8).

Flores [FLOR64] modeled the situation where a processor and I/O channels are contending for access to an interleaved memory. He modeled the time that a request must wait, waiting time, as a function of the degree of interleaving, the ratio of I/O time to processor time, and the fraction of time that the memory is busy. For an equal ratio of I/O time and as the memory busy time approaches one, the model gives the expected result that the speedup (ratio of waiting time of an interleaved memory to a single module memory) is equal to m, the number of interleaved modules.

The next published model is by Hellerman [HELL67]. This model

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000040

Memory Systems an
Pipelined Processo

LIBRARY OF CONGRESS
0 003 464 349 0

Harvey G. Cragon

The current widespread demand for high performance personal computers and workstations has resulted in a renaissance of computer design. To meet the challenge that this presents to students and professional computer architects, this graduate level text offers an in-depth treatment of the implementation details of memory systems and pipelined processors, the "microarchitecture" of modern computers and microprocessors.

The text explores techniques for solving the design problems inherent in computers with high levels of concurrency, such as the demand for a memory system with low latency and high bandwidth, branching, providing precise interrupts, managing dependencies and insuring coherency. Additionally, it examines the difficulties presented by virtual memory in high performance computers. As a thorough compendium of both historical and contemporary implementation techniques, this is an essential sourcebook for computer architecture students and practicing professionals.

Features

- Text is ideal for a one-semester course
- Provides references to the origins of ideas that guide readers to relevant literature
- Offers a complete treatment of the ways in which features can be implemented
- Includes a chapter on diversity of addresses

About the Author

Harvey G. Cragon has held the Ernest Cockrell, Jr. Centennial Chair in Engineering at the University of Texas at Austin since 1984. He has over 30 years of industrial experience in architectural design and implementation of digital computers, including 25 years with Texas Instruments. Mr. Cragon's current research interests are in computer architecture design methodology and high-speed computers.

Other Titles of Interest

Computer Architecture: Pipelined and Parallel Processor Design, Michael J. Flynn
Algorithms and Data Structures in Computer Engineering, E. Stewart Lee

ISBN 0-86720-474-5

Jones and Bartlett Publishers
One Exeter Plaza
Boston, MA 02116
800-832-0034 • 617-859-3900

9 780867 204742 >

Advanced Micro Devices, Inc., Pensando Systems, Inc. - Ex. 1037, Page 000041