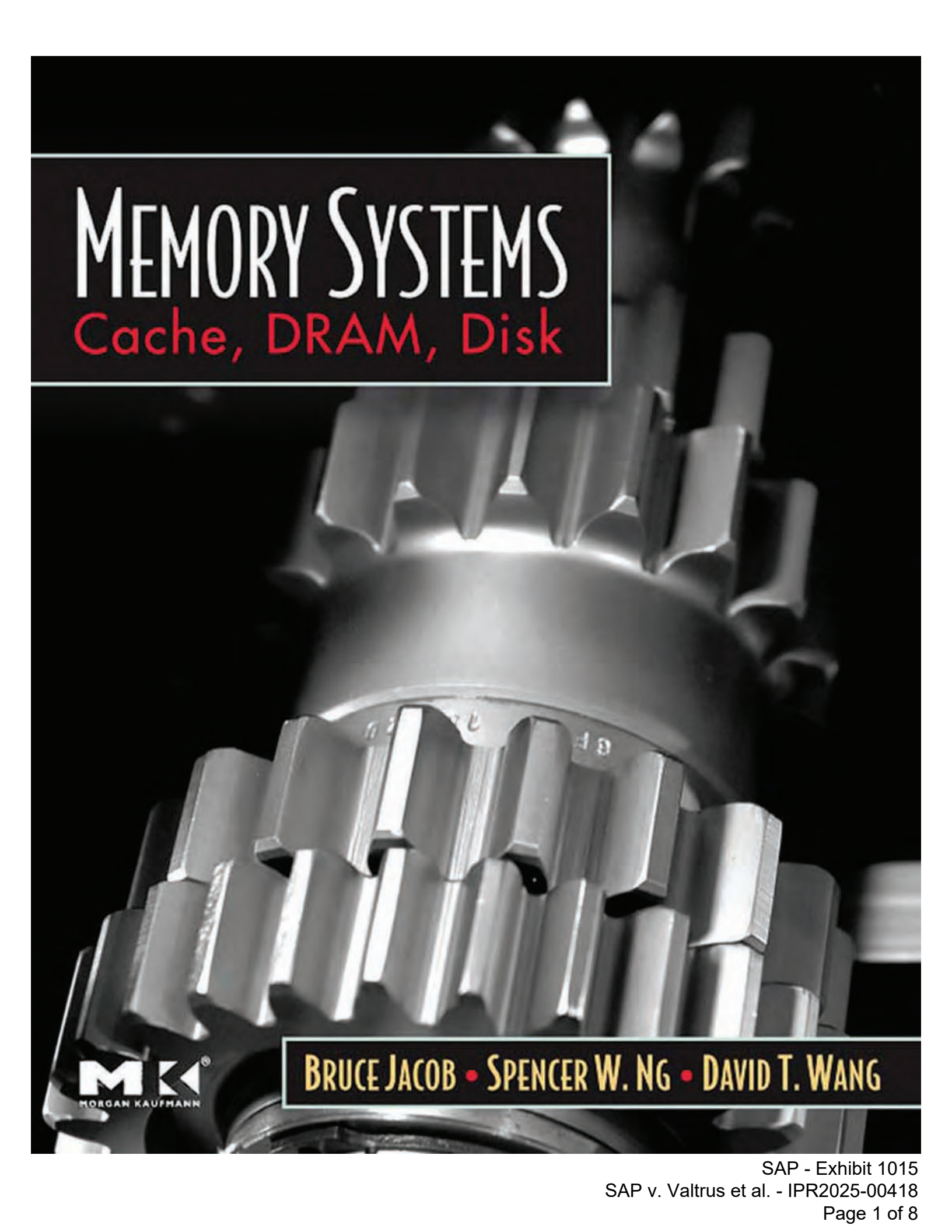




MEMORY SYSTEMS

Cache, DRAM, Disk



BRUCE JACOB • SPENCER W. NG • DAVID T. WANG

Memory Systems

Cache, DRAM, Disk

Memory Systems

Cache, DRAM, Disk

Bruce Jacob

University of Maryland at College Park

Spencer W. Ng

Hitachi Global Storage Technologies

David T. Wang

MetaRAM

With Contributions By

Samuel Rodriguez

Advanced Micro Devices



AMSTERDAM • BOSTON • HEIDELBERG LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann is an imprint of Elsevier



MORGAN KAUFMANN PUBLISHERS

<i>Publisher</i>	Denise E.M. Penrose
<i>Acquisitions Editor</i>	Chuck Glaser
<i>Publishing Services Manager</i>	George Morrison
<i>Senior Production Editor</i>	Paul Gottehrer
<i>Developmental Editor</i>	Nate McFadden
<i>Assistant Editor</i>	Kimberlee Honjo
<i>Cover Design</i>	Joanne Blank
<i>Text Design</i>	Dennis Schaefer
<i>Composition</i>	diacriTech
<i>Interior printer</i>	Maple-Vail Book Manufacturing Group
<i>Cover printer</i>	Phoenix Color

Morgan Kaufmann Publishers is an imprint of Elsevier.
30 Corporate Drive, Suite 400, Burlington, MA 01803, USA

This book is printed on acid-free paper.

© 2008 by Elsevier Inc. All rights reserved.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which Morgan Kaufmann Publishers is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

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, scanning, or otherwise—without prior written permission of the publisher.

Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: phone: (+44) 1865 843830, fax: (+44) 1865 853333, E-mail: permissions@elsevier.com. You may also complete your request online via the Elsevier homepage (<http://elsevier.com>), by selecting "Support & Contact" then "Copyright and Permission" and then "Obtaining Permissions."

Library of Congress Cataloging-in-Publication Data

Application submitted

ISBN: 978-0-12-379751-3

For information on all Morgan Kaufmann publications,
visit our Web site at www.mkp.com or www.books.elsevier.com

Printed in the United States of America

08 09 10 11 12 5 4 3 2 1

Working together to grow
libraries in developing countries

www.elsevier.com | www.bookaid.org | www.sabre.org

ELSEVIER

BOOK AID
International

Sabre Foundation

2.2 Transparently Addressed Caches

A transparently addressed cache uses the same namespace as the backing store; that is, a cache lookup for the desired data object uses the same object name one would use when accessing the backing store. For example, if the backing store is the disk drive, then the disk block number would be used to search (or “probe”) the cache. If the backing store is main memory, then the physical memory address would be used to search the cache. If the backing store is a magneto-optical or tape jukebox, then its associated identifier would be used. The mechanism extends to software caches as well. For example, if the backing store is the operating system’s file system, then the file name would be used to search the cache. If the backing store is the world-wide web, then the URL would be used, and so on. The point is that the client (e.g., application software or even an operating system) need not know anything about the cache’s organization or operation to benefit from it. The client simply makes a request to the backing store, and the cache is searched for the requested object implicitly.

Because the cache must be *searched* (a consequence of the fact that it is addressed transparently and not explicitly), there must be a way to determine the cache’s contents, and thus, an implicitly addressed cache must have keys or tags, most likely taken from the backing store’s namespace as object identifiers. For instance, a software cache could be implemented as a binary search tree, a solid-state cache could be implemented with a tag store that matches the organization of the data store, or a DRAM-based cache fronting a disk drive could have a CAM lookup preceding data access to find the correct physical address [Braunstein et al. 1989]. Note that this is precisely what the TLB does.

Because the cost of searching can be high (consider, for example, a large cache in which every single key or tag is checked for every single lookup), different logical organizations have been developed that try to accomplish simultaneously two goals:

1. Reduce the number of tags checked
2. Keep the likelihood of a cache hit high

Note that these goals tend to pull a cache design in opposite directions. For instance, an easy way to

satisfy Goal 1 is to assign a particular data object a single space in the cache, kind of like assigning the parking spaces outside a company’s offices. The problem with this is that, unless the cache is extremely large, one can quickly run out of spaces to assign, in which case the cache might have to start throwing out useful data (as an analogy, a company might have to reassign to a new employee a parking spot belonging to the company’s CEO).

Basic organization for implicitly addressed caches has already been discussed briefly in Chapter 1. The main point is that a transparent cache typically divides its storage into equivalence classes, or *sets*, and assigns blocks to sets based on the block ID. The single, unique set to which a particular chunk of data belongs is typically determined by a subset of bits taken from its block ID, as shown in Figure 2.2. Once the set is determined, the block may be placed anywhere within that set. Because a block’s ID, taken from its location in the memory space, does not change over the course of its lifetime (exceptions including virtually indexed caches), a block will not typically migrate between cache sets. Therefore, one can view these sets as having an exclusive relationship with each other, and so *any* heuristic that chooses a partitioning of data between exclusive storage units would be just as appropriate as the simple bit-vector method shown in Figure 2.2.

A cache can comprise one or more sets; each set can comprise one or more storage slots. This gives rise to the following organizations for solid-state designs.

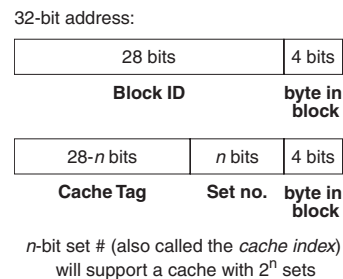


FIGURE 2.2: Different views of an address.

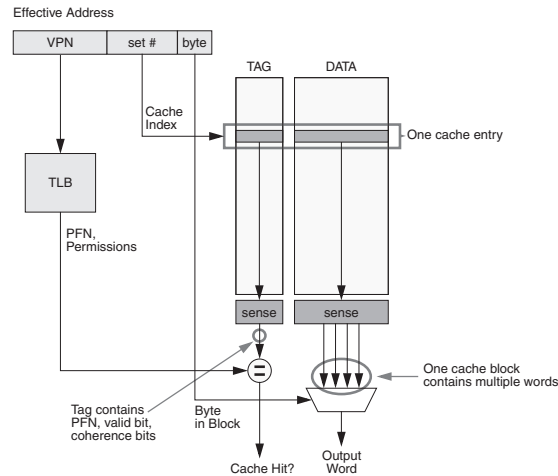


FIGURE 2.3: Block diagram for a direct-mapped cache. Note that data read-out is controlled by tag comparison with the TLB output as well as the block-valid bit (part of the tag or metadata entry) and the page permissions (part of the TLB entry). Data is not read from the cache if the valid bit is not set or if the permissions indicate an invalid access (e.g., writing a read-only block). Note also that the cache size is equal to the virtual memory page size (the cache index does not use bits from the VPN).

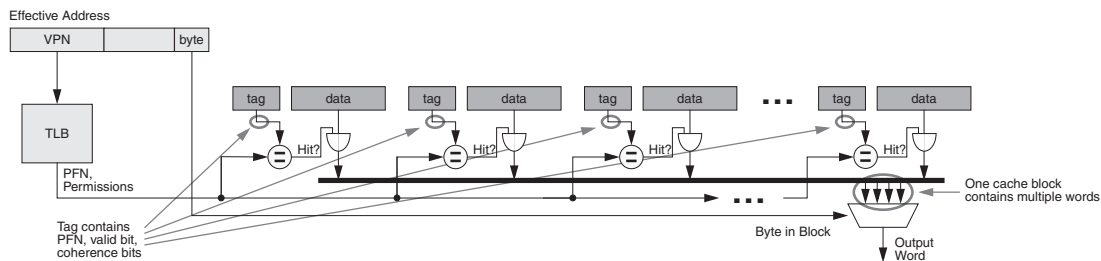


FIGURE 2.4: Fully associative lookup mechanism. This organization is also called a CAM, for content-addressable memory. It is similar to a database in that any entry that has the same tag as the lookup address matches, no matter where it is in the cache. This organization reduces cache contention, but the lookup can be expensive, since the tag of every entry is matched against the lookup address.

- Direct-Mapped Organization:** Figure 2.3 illustrates a read operation in a *direct-mapped* cache organization. In this organization, a given datum can only reside in one entry of the cache, usually determined by a subset of the datum's address bits. Though the most common index is the low-order bits of the tag field, other indexing schemes exist that use a different set of bits or hash the address to compute an index value bearing
 - no obvious correspondence to the original address. Whereas in the associative scheme there are n tag matches, where n is the number of cache lines, in this scheme there is only one tag match because the requested datum can only be found at one location: it is either there or nowhere in the cache.
- Fully Associative Organization:** Figure 2.4 demonstrates a *fully associative* lookup. In a fully associative organization, the cache

is truly a small hardware database. A datum can be placed anywhere in the cache; the tag field identifies the data contents. A search checks the tag of every datum stored in the cache. If any one of the tags matches the tag of the requested address, it is a *cache hit*: the cache contains the requested data.

The benefit of a direct-mapped cache is that it is extremely quick to search and dissipates relatively little power, since there can only be one place that any particular datum can be found. However, this introduces the possibility that several different data might need to reside in the cache at the same place, causing what is known as *contention* for the desired data entry. This results in poor performance, with entries in the cache being replaced frequently; these are *conflict misses*. The problem can be solved by using a fully associative cache, a design choice traditionally avoided due to its high dynamic power dissipation (in a typical lookup, every single tag is compared to the search tag). However, as dynamic power decreases in significance relative to leakage power, this design choice will become more attractive.

- **Set-Associative Organization:** As a compromise between the two extremes, a set-associative cache lies in between direct-mapped and fully associative designs and often reaps

the benefits of both designs—fast lookup and lower contention. Figures 2.5 and 2.6 illustrate two set-associative organizations. Figure 2.5 shows an implementation built of a small number of direct-mapped caches, typically used for low levels of set associativity, and Figure 2.6 shows an implementation built of a small number of content-addressable memories, typically used for higher degrees of set associativity.

These are the basic choices of organization for transparently addressed caches. They are applicable to solid-state implementations (evident from the diagrams) as well as software implementations. For instance, an operating system's buffer cache or distributed memory system's object cache could be implemented as a large array of blocks, indexed by a hash of the block ID or object ID, which would correspond to a direct-mapped organization as in Figure 2.3; it could be implemented as a binary search tree, which would correspond to a fully associative organization, because the search tree is simply software's way to “search all tags in parallel” as in Figure 2.4; or it could be implemented as a set of equivalence classes, each of which maintains its own binary search tree, an arrangement that would correspond to a set-associative organization as in Figure 2.6 (each row in the figure is a separate equivalence class). The last organization of software

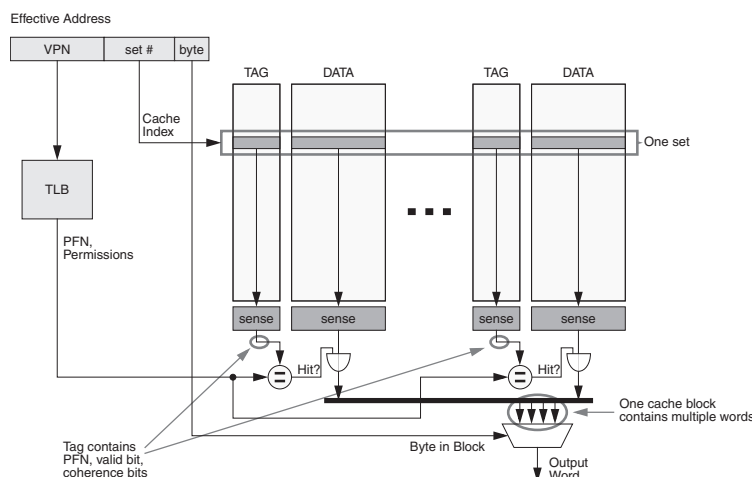


FIGURE 2.5: Block diagram for an n -way set-associative cache. A set-associative cache can be made of several direct-mapped caches operated in parallel. Note that data read-out is controlled by tag comparison with the TLB output as well as the block-valid bit (part of the tag or metadata entry) and the page permissions (part of the TLB entry). Note also that the cache column size is equal to the virtual memory page size (the cache index does not use bits from the VPN).

cache could significantly reduce the search time for any given object, assuming that the assignment of objects to equivalence classes is evenly distributed.

An important point to note is that, because the transparently addressed cache uses the same namespace as the backing store, it is possible (and this is indeed the typical case) for the cache to maintain only a *copy* of the datum that is held in the backing store. In other words, caching an object does not necessarily supersede or invalidate the original. The original stays where it is, and it is up to the consistency-management solution to ensure that the original is kept consistent with the cached copy.

What does this have to do with logical organization? Because the transparently addressed cache maintains a copy of the object's identifier (e.g., its physical address, if the backing store is main memory), it is possible for the cache to keep the copy in the backing store consistent transparently. Without this information, the application software must be responsible for maintaining consistency between the cache and the backing store. This is a significant issue for processor caches: if the cache does not maintain a physical tag for each cached item, then the cache hardware is not capable of ensuring the consistency of the copy in main memory.

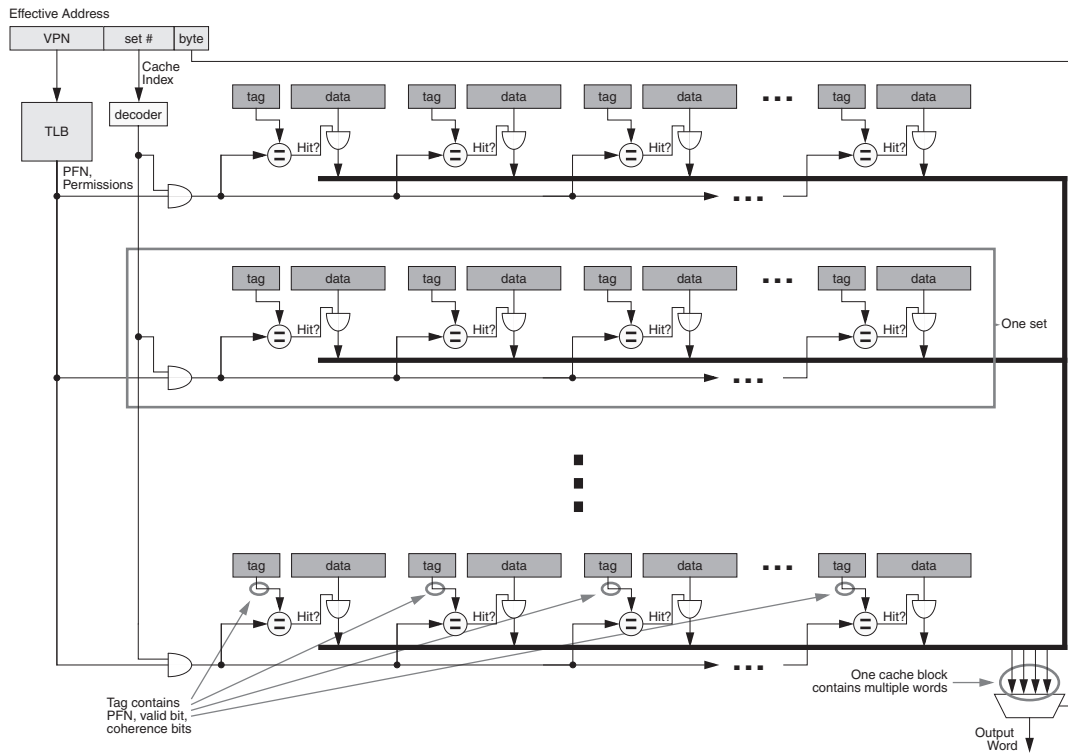


FIGURE 2.6: Set-associative cache built from CAMs. The previous figure showed a set-associative cache built of several direct-mapped caches, which is appropriate when the degree of associativity is low. When the degree of associativity is high, it is more appropriate to build the cache out of CAMs (content-addressable memories, i.e., fully associative caches), where each CAM represents one data set or equivalence class. For example, the Strong ARM's 32-way set-associative cache is built this way.