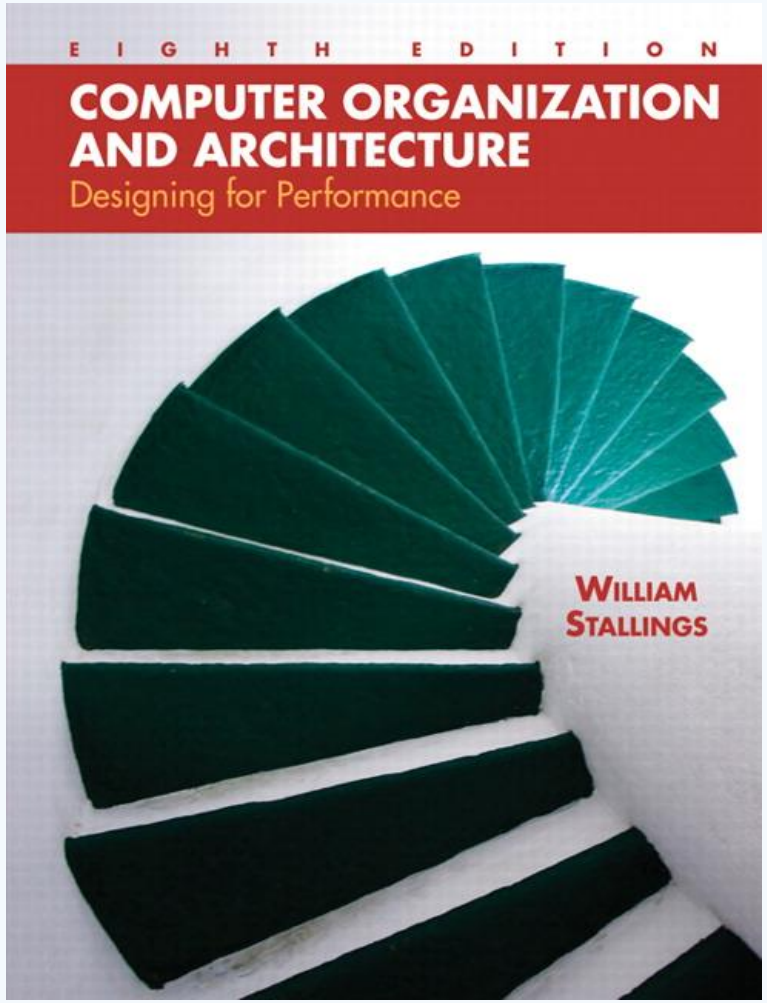
Computer Org & Arch (COA)

Chapter 4 Cache Memory

Go Green! Please think before printing these lecture slides.

Text Book

- **Title:** Computer Organization and Architecture: Designing for Performance
- **Author:** William Stallings
- **Edition:** 8th
- **Publisher:** Prentice-Hall India



Text Book

The book is organized into five parts:

- Part One: Provides an overview of computer organization and architecture and looks at how computer design has evolved. (Chapter 1 to 2)
- Part Two: Examines the major components of a computer and their interconnections, both with each other and the outside world. This part also includes a detailed discussion of internal and external memory and of input–output (I/O). Finally, the relationship between a computer's architecture and the operating system running on that architecture is examined. (Chapter 3 to 8)
- Part Three: Examines the internal architecture and organization of the processor. This part begins with an extended discussion of computer arithmetic. Then it looks at the instruction set architecture. The remainder of the part deals with the structure and function of the processor, including a discussion of reduced instruction set computer (RISC) and superscalar approaches. (Chapter 8 to 14)
- Part Four: Discusses the internal structure of the processor's control unit and the use of microprogramming. (Chapter 15 to 16)

- Part Five: Deals with parallel organization, including symmetric multiprocessing, clusters, and multicore architecture. (Chapter 17 to 18)
- This text is intended to acquaint you with the design principles and implementation issues of contemporary computer organization and architecture. Accordingly, a purely conceptual or theoretical treatment would be inadequate. This book uses examples from a number of different machines to clarify and reinforce the concepts being presented. Many, but by no means all, of the examples are drawn from two computer families: the Intel x86 family and the ARM (Advanced RISC Machine) family. These two systems together encompass most of the current computer design trends. The Intel x86 architecture is essentially a complex instruction set computer (CISC) with some RISC features, while the ARM is essentially a RISC. Both systems make use of superscalar design principles and both support multiple processor and multicore configurations.

Chapter 4
Cache Memory

Cache Memory

Computer memory exhibits a wide range of type, technology, organization, performance, and cost. The typical computer system is equipped with a hierarchy of memory subsystems, some internal (directly accessible by the processor) and some external (accessible by the processor via an I/O module). Chapter 4 begins with an overview of this hierarchy. Next, the chapter deals in detail with the design of cache memory, including separate code and data caches and two-level caches.

Top Level View of Computer Function & Interconnection

- Characteristics of Memory Systems
  - Location
  - Capacity
  - Unit of transfer
- Memory Hierarchy
  - How much?
  - How fast?
  - How expensive?
- Cache memory principles
- Elements of cache design
  - Cache addresses
  - Cache size
  - Mapping function
  - Replacement algorithms
  - Write policy
  - Line size
  - Number of caches
- Pentium 4 cache organization
- ARM cache organization

KEY POINTS

- Computer memory is organized into a hierarchy. At the highest level (closest to the processor) are the processor registers. Next comes one or more levels of cache, When multiple levels are used, they are denoted L1, L2, and so on. Next comes main memory, which is usually made out of dynamic random-access memory (DRAM). All of these are considered internal to the computer system. The hierarchy continues with external memory, with the next level typically being a fixed hard disk, and one or more levels below that consisting of removable media such as optical disks and tape.

- As one goes down the memory hierarchy, one finds decreasing cost/bit, increasing capacity, and slower access time. It would be nice to use only the fastest memory, but because that is the most expensive memory, we trade off access time for cost by using more of the slower memory. The design challenge is to organize the data and programs in memory so that the accessed memory words are usually in the faster memory.

Introduction

- In general, it is likely that most future accesses to main memory by the processor will be to locations recently accessed. So the cache automatically retains a copy of some of the recently used words from the DRAM. If the cache is designed properly, then most of the time the processor will request memory words that are already in the cache.

- Although seemingly simple in concept, computer memory exhibits perhaps the widest range of type, technology, organization, performance, and cost of any feature of a computer system. No one technology is optimal in satisfying the memory requirements for a computer system. As a consequence, the typical computer system is equipped with a hierarchy of memory subsystems, some internal to the system (directly accessible by the processor) and some external (accessible by the processor via an I/O module).

- This chapter and the next focus on internal memory elements, while Chapter 6 is devoted to external memory. To begin, the first section examines key characteristics of computer memories. The remainder of the chapter examines an essential element of all modern computer systems: cache memory.

Key Characteristics of Computer Memory Systems

Location	**Performance**
Internal (e.g. processor registers, cache, main memory)	Access time
External (e.g. optical disks, magnetic disks, tapes)	Cycle time
Capacity	Transfer rate
Number of words	**Physical Type**
Number of bytes	Semiconductor
Unit of Transfer	Magnetic
Word	Optical
Block	Magneto-optical
Access Method	**Physical Characteristics**
Sequential	Volatile/nonvolatile
Direct	Erasable/nonerasable
Random	**Organization**
Associative	Memory modules

Table 4.1 Key Characteristics of Computer Memory Systems

The complex subject of computer memory is made more manageable if we classify memory systems according to their key characteristics. The most important of these are listed in Table 4.1.

Key Characteristics of Computer Memory Systems

- The term location in Table 4.1 refers to whether memory is internal and external to the computer. Internal memory is often equated with main memory. But there are other forms of internal memory. The processor requires its own local memory, in the form of registers (e.g., see Figure 2.3).
- Further, as we shall see, the control unit portion of the processor may also require its own internal memory. We will defer discussion of these latter two types of internal memory to later chapters. Cache is another form of internal memory. External memory consists of peripheral storage devices, such as disk and tape, that are accessible to the processor via I/O controllers.
- An obvious characteristic of memory is its capacity. For internal memory, this is typically expressed in terms of bytes (1 byte = 8 bits) or words. Common word lengths are 8, 16, and 32 bits. External memory capacity is typically expressed in terms of bytes.

Key Characteristics of Computer Memory Systems

- A related concept is the unit of transfer. For internal memory, the unit of transfer is equal to the number of electrical lines into and out of the memory module. This may be equal to the word length, but is often larger, such as 64, 128, or 256 bits. To clarify this point, consider three related concepts for internal memory:
  - Word: The "natural" unit of organization of memory. The size of a word is typically equal to the number of bits used to represent an integer and to the instruction length. Unfortunately, there are many exceptions. For example, the CRAY C90 (an older model CRAY supercomputer) has a 64-bit word length but uses a 46-bit integer representation. The Intel x86 architecture has a wide variety of instruction lengths, expressed as multiples of bytes, and a word size of 32 bits.

Key Characteristics of Computer Memory Systems

- Addressable units: In some systems, the addressable unit is the word. However, many systems allow addressing at the byte level. In any case, the relationship between the length in bits A of an address and the number N of addressable units is $2^A = N$.
- Unit of transfer: For main memory, this is the number of bits read out of or written into memory at a time. The unit of transfer need not equal a word or an addressable unit. For external memory, data are often transferred in much larger units than a word, and these are referred to as blocks

Method of Accessing Units of Data

Sequential access

- Memory is organized into units of data called records
- Access must be made in a specific linear sequence
- Access time is variable

Direct access

- Involves a shared read-write mechanism
- Individual blocks or records have a unique address based on physical location
- Access time is variable

Random access

- Each addressable location in memory has a unique, physically wired-in addressing mechanism
- The time to access a given location is independent of the sequence of prior accesses and is constant
- Any location can be selected at random and directly addressed and accessed
- Main memory and some cache systems are random access

Associative

- A word is retrieved based on a portion of its contents rather than its address
- Each location has its own addressing mechanism and retrieval time is constant independent of location or prior access patterns
- Cache memories may employ associative access

Method of Accessing Units of Data

Another distinction among memory types is the method of accessing units of data. These include the following:

- Sequential access: Memory is organized into units of data, called records. Access must be made in a specific linear sequence. Stored addressing information is used to separate records and assist in the retrieval process. A shared read–write mechanism is used, and this must be moved from its current location to the desired location, passing and rejecting each intermediate record. Thus, the time to access an arbitrary record is highly variable. Tape units, discussed in Chapter 6, are sequential access.

- Direct access: As with sequential access, direct access involves a shared read–write mechanism. However, individual blocks or records have a unique address based on physical location. Access is accomplished by direct access to reach a general vicinity plus sequential searching, counting, or waiting to reach the final location. Again, access time is variable. Disk units, discussed in Chapter 6, are direct access.

Method of Accessing Units of Data

- Random access: Each addressable location in memory has a unique, physically wired-in addressing mechanism. The time to access a given location is independent of the sequence of prior accesses and is constant. Thus, any location can be selected at random and directly addressed and accessed. Main memory and some cache systems are random access.
- Associative: This is a random access type of memory that enables one to make a comparison of desired bit locations within a word for a specified match, and to do this for all words simultaneously. Thus, a word is retrieved based on a portion of its contents rather than its address. As with ordinary random-access memory, each location has its own addressing mechanism, and retrieval time is constant independent of location or prior access patterns. Cache memories may employ associative access.

Capacity and Performance:

The two most important characteristics of memory

Three performance parameters are used:

Access time (latency)

- For random-access memory it is the time it takes to perform a read or write operation
- For non-random-access memory it is the time it takes to position the read-write mechanism at the desired location

Memory cycle time

- Access time plus any additional time required before second access can commence
- Additional time may be required for transients to die out on signal lines or to regenerate data if they are read destructively
- Concerned with the system bus, not the processor

Transfer rate

- The rate at which data can be transferred into or out of a memory unit
- For random-access memory it is equal to 1/(cycle time)

Capacity and Performance:

From a user's point of view, the two most important characteristics of memory are capacity and performance. Three performance parameters are used:

- Access time (latency): For random-access memory, this is the time it takes to perform a read or write operation, that is, the time from the instant that an address is presented to the memory to the instant that data have been stored or made available for use. For non-random-access memory, access time is the time it takes to position the read–write mechanism at the desired location.
- Memory cycle time: This concept is primarily applied to random-access memory and consists of the access time plus any additional time required before a second access can commence. This additional time may be required for transients to die out on signal lines or to regenerate data if they are read destructively. Note that memory cycle time is concerned with the system bus, not the processor.
- Transfer rate: This is the rate at which data can be transferred into or out of a memory unit. For random-access memory, it is equal to 1/(cycle time).

Memory

- The most common forms are:
  - Semiconductor memory
  - Magnetic surface memory
  - Optical
  - Magneto-optical
- Several physical characteristics of data storage are important:
  - Volatile memory
    - Information decays naturally or is lost when electrical power is switched off
  - Nonvolatile memory
    - Once recorded, information remains without deterioration until deliberately changed
    - No electrical power is needed to retain information

Memory

- Magnetic-surface memories
  - Are nonvolatile
- Semiconductor memory
  - May be either volatile or nonvolatile
- Nonerasable memory
  - Cannot be altered, except by destroying the storage unit
  - Semiconductor memory of this type is known as read-only memory (ROM)

- For random-access memory the organization is a key design issue
  - Organization refers to the physical arrangement of bits to form words

Memory

- A variety of physical types of memory have been employed. The most common today are semiconductor memory, magnetic surface memory, used for disk and tape, and optical and magneto-optical.
- Several physical characteristics (summarized in last slide) of data storage are important. In a volatile memory, information decays naturally or is lost when electrical power is switched off. In a nonvolatile memory, information once recorded remains without deterioration until deliberately changed; no electrical power is needed to retain information. Magnetic-surface memories are nonvolatile. Semiconductor memory (memory on integrated circuits) may be either volatile or nonvolatile. Nonerasable memory cannot be altered, except by destroying the storage unit. Semiconductor memory of this type is known as read-only memory (ROM). Of necessity, a practical nonerasable memory must also be nonvolatile.
- For random-access memory, the organization is a key design issue. In this context, organization refers to the physical arrangement of bits to form words. The obvious arrangement is not always used, as is explained in Chapter 5.

Memory Hierarchy

- Design constraints on a computer's memory can be summed up by three questions:
  - How much, how fast, how expensive
- There is a trade-off among capacity, access time, and cost
  - Faster access time (↑), greater cost per bit (↑)
  - Greater capacity(↑), smaller cost per bit (↓)
  - Greater capacity(↑), slower access time (↓)
- The way out of the memory dilemma(problem) is not to rely on a single memory component or technology, but to employ a memory hierarchy (pyramid)

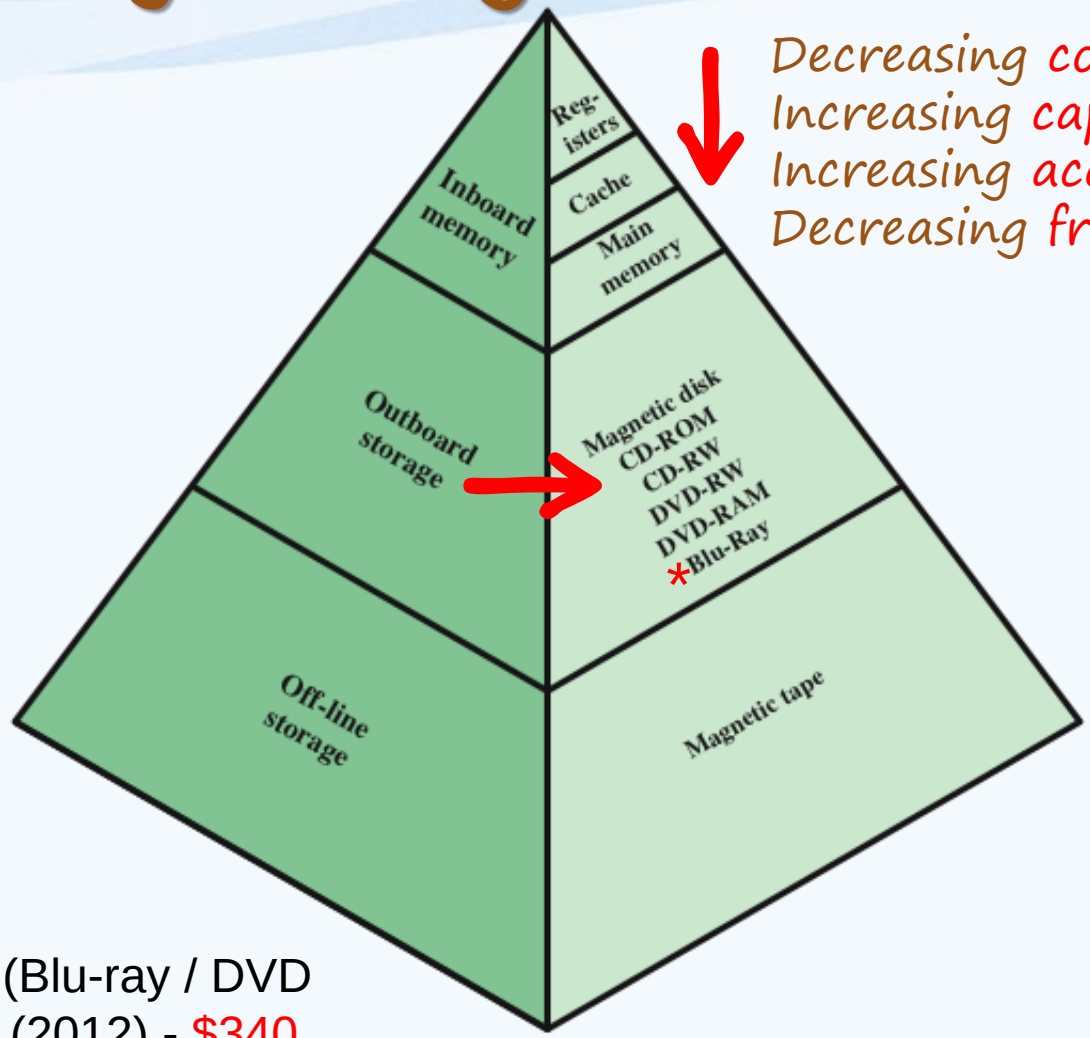
Memory Hierarchy

- The design constraints on a computer's memory can be summed up by three questions: How much? How fast? How expensive?
- The question of how much is somewhat open ended. If the capacity is there, applications will likely be developed to use it. The question of how fast is, in a sense, easier to answer. To achieve greatest performance, the memory must be able to keep up with the processor. That is, as the processor is executing instructions, we would not want it to have to pause waiting for instructions or operands. The final question must also be considered. For a practical system, the cost of memory must be reasonable in relationship to other components.
- As might be expected, there is a trade-off among the three key characteristics of memory: capacity, access time, and cost. A variety of technologies are used to implement memory systems, and across this spectrum of technologies,

Memory Hierarchy

- the following relationships hold:
  - Faster access time, greater cost per bit
  - Greater capacity, smaller cost per bit
  - Greater capacity, slower access time
- The dilemma facing the designer is clear. The designer would like to use memory technologies that provide for large-capacity memory, both because the capacity is needed and because the cost per bit is low. However, to meet performance requirements, the designer needs to use expensive, relatively lower-capacity memories with short access times.
- The way out of this dilemma is not to rely on a single memory component or technology, but to employ a memory hierarchy.

Memory Hierarchy – Diagram

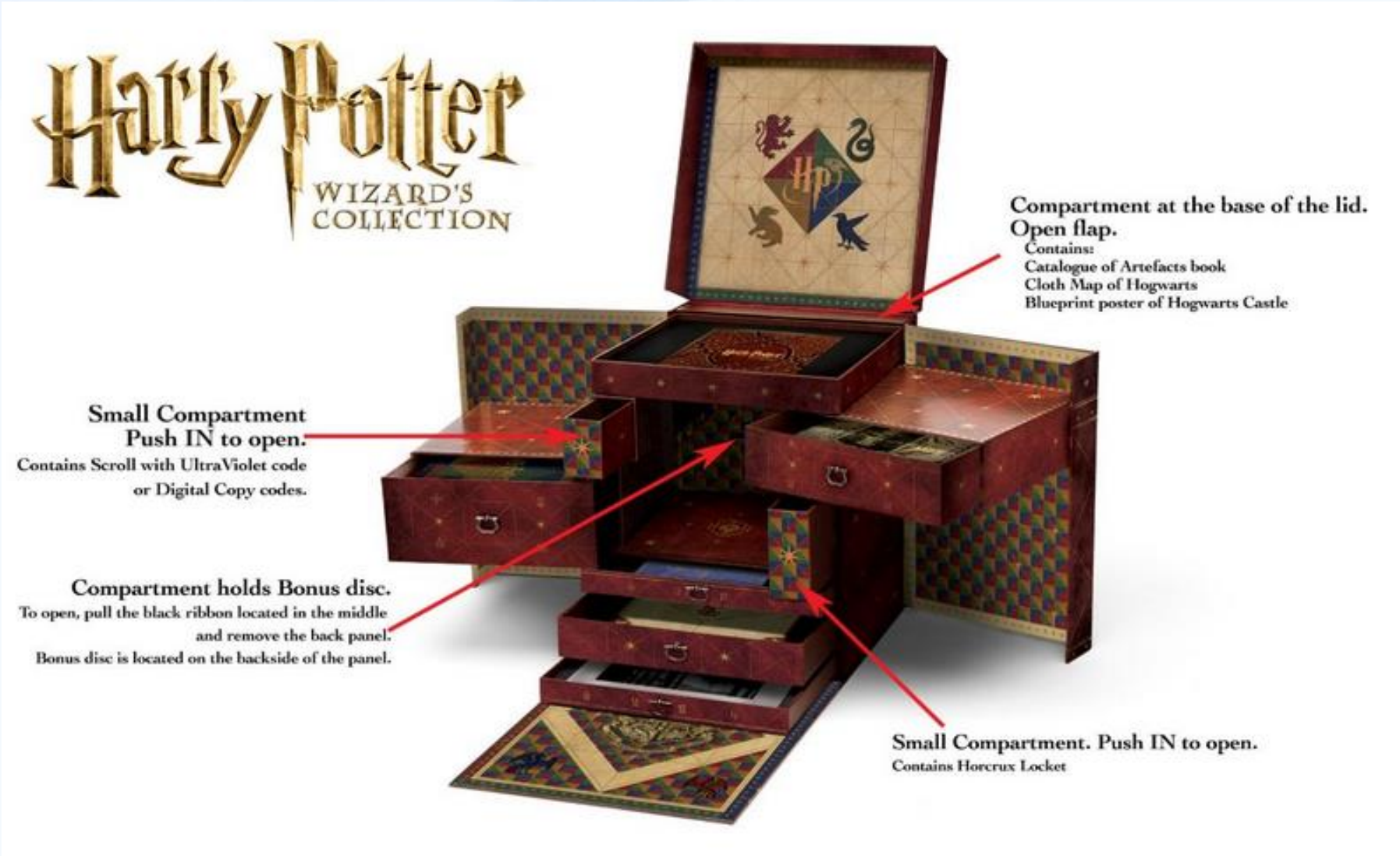


Decreasing cost per bit (↓)
Increasing capacity (↑)
Increasing access time (↑)
Decreasing frequency of access (↓)

* Harry Potter Wizard's Collection (Blu-ray / DVD Combo + UltraViolet Digital Copy) (2012) - $340

Figure 4.1 The Memory Hierarchy

Blu-ray / DVD Combo + UltraViolet Digital Copy



Memory Hierarchy – Diagram

- A typical hierarchy is illustrated in Figure 4.1.(S-25) As one goes down the hierarchy, the following occur:
  a) Decreasing cost per bit
  b) Increasing capacity
  c) Increasing access time
  d) Decreasing frequency of access of the memory by the processor

Memory Hierarchy – Diagram

- Thus, smaller, more expensive, faster memories are supplemented by larger, cheaper, slower memories. The key to the success of this organization is item (d) :decreasing frequency of access. We examine this concept in greater detail when we discuss the cache, later in this chapter, and virtual memory in Chapter 8. A brief explanation is provided at this point.
- The use of two levels of memory to reduce average access time works in principle, but only if conditions (a) through (d) apply.
- By employing a variety of technologies, a spectrum of memory systems exists that satisfies conditions (a) through (c).
- Fortunately, condition (d) is also generally valid.

Memory Hierarchy – Diagram

- The basis for the validity of condition (d) – [Decreasing frequency of access of the memory by the processor] is a principle known as *locality of reference*. During the course of execution of a program, memory references by the processor, for both instructions and data, tend to cluster. Programs typically contain a number of iterative loops and subroutines. Once a loop or subroutine is entered, there are repeated references to a small set of instructions. Similarly, operations on tables and arrays involve access to a clustered set of data words. Over a long period of time, the clusters in use change, but over a short period of time, the processor is primarily working with fixed clusters of memory references.

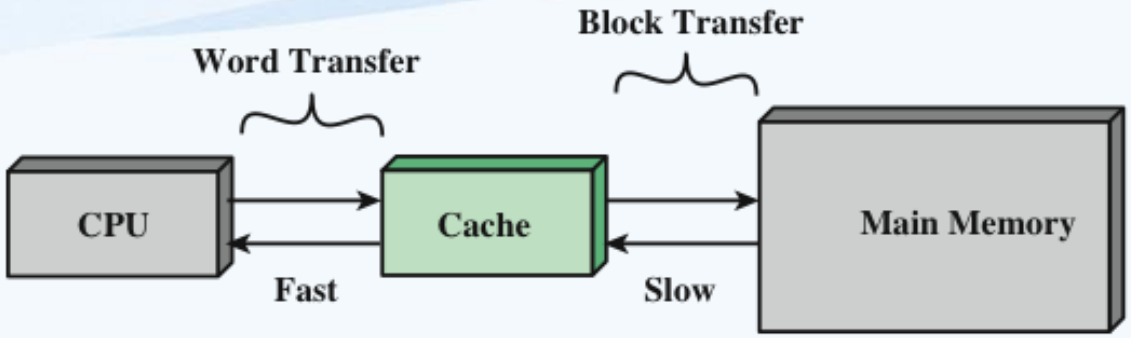
Memory Hierarchy – Diagram

- Accordingly, it is possible to organize data across the hierarchy such that the percentage of accesses to each successively lower level is substantially less than that of the level above.
- Consider the two-level example (S-25) already presented. Let level 2 memory contains all program instructions and data.
- The current clusters can be temporarily placed in level 1.
- From time to time, one of the clusters in level 1 will have to be swapped back to level 2 to make room for a new cluster coming in to level 1.
- On average, however, most references will be to instructions and data contained in level 1.

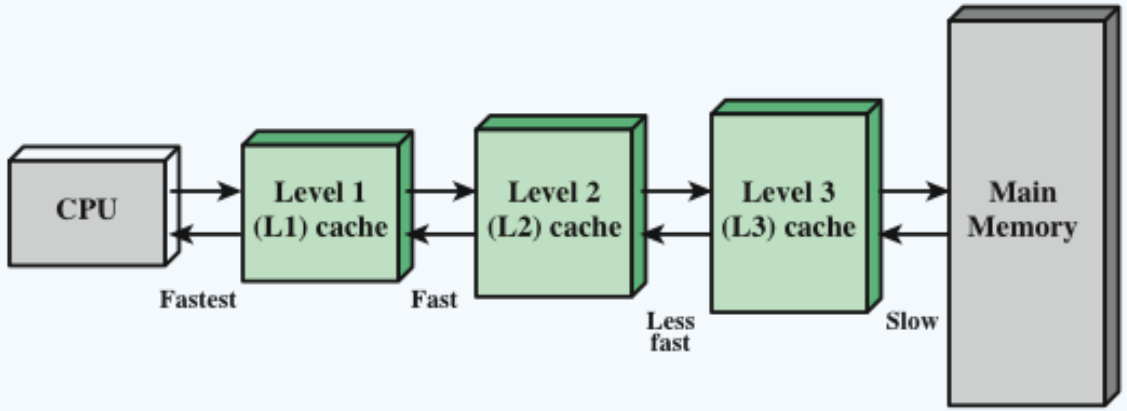
Memory Hierarchy – Diagram

- This principle can be applied across more than two levels of memory, as suggested by the hierarchy shown in Figure 4.1(S-25). The fastest, smallest, and most expensive type of memory consists of the registers internal to the processor. Typically, a processor will contain a few dozen such registers, although some machines contain hundreds of registers. Main memory is the principal internal memory system of the computer. Each location in main memory has a unique address. Main memory is usually extended with a higher-speed, smaller cache. The cache is not usually visible to the programmer or, indeed, to the processor. It is a device for staging the movement of data between main memory and processor registers to improve performance.

Cache and Main Memory



(a) Single cache



(b) Three-level cache organization

Figure 4.3 Cache and Main Memory

Cache and Main Memory

- Cache memory is designed to combine the memory access time of expensive, high-speed memory combined with the large memory size of less expensive, lower-speed memory.
- The concept is illustrated in Figure 4.3a(S-32). There is a relatively large and slow main memory together with a smaller, faster cache memory. The cache contains a copy of portions of main memory. When the processor attempts to read a word of memory, a check is made to determine if the word is in the cache. If so, the word is delivered to the processor. If not, a block of main memory, consisting of some fixed number of words, is read into the cache and then the word is delivered to the processor. Because of the phenomenon(occurrence) of locality of reference, when a block of data is fetched into the cache to satisfy a single memory reference, it is likely that there will be future references to that same memory location or to other words in the block.
- Figure 4.3b(S-32) depicts the use of multiple levels of cache. The L2 cache is slower and typically larger than the L1 cache, and the L3 cache is slower and typically larger than the L2 cache.

Cache/Main Memory Structure

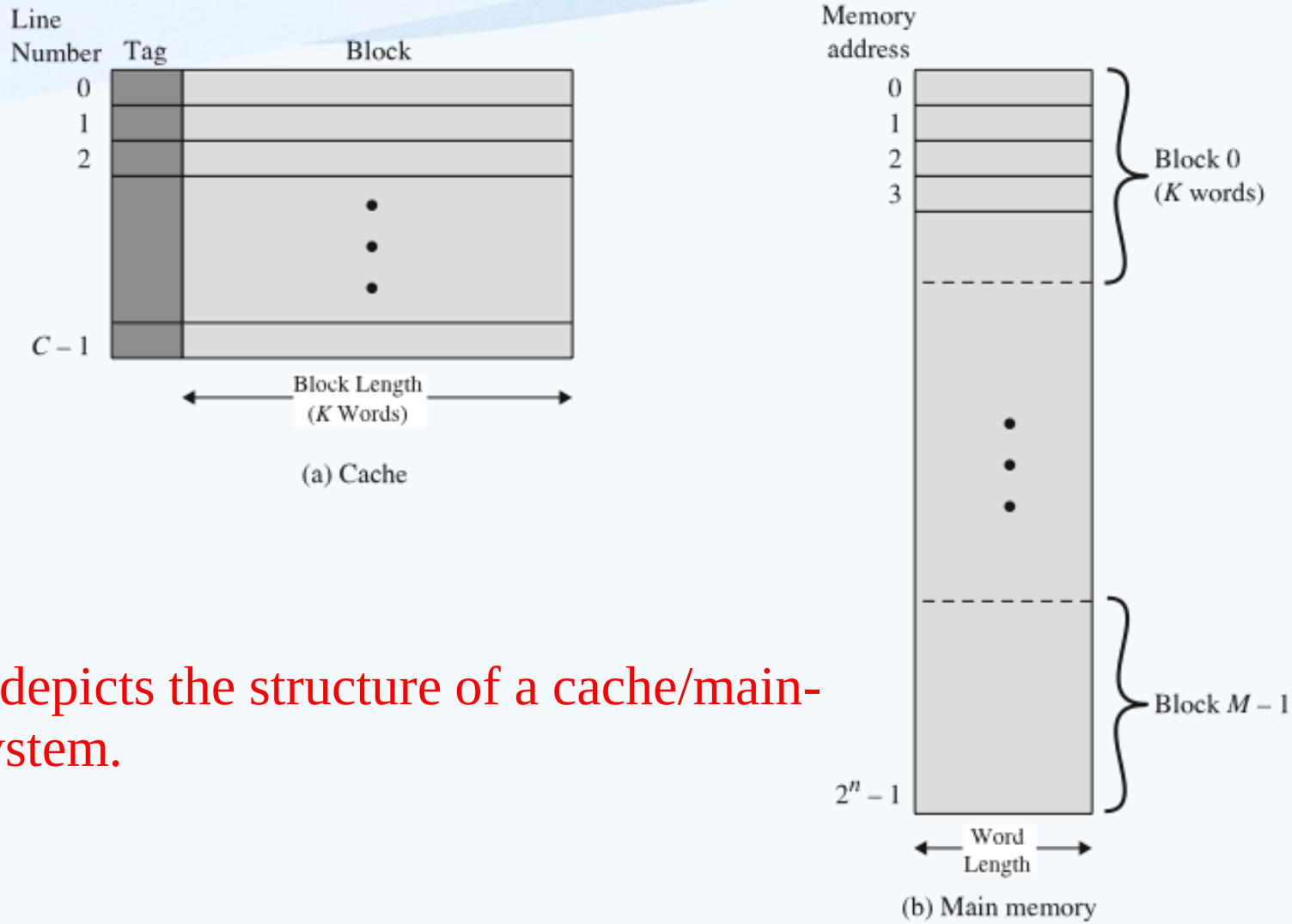


Figure 4.4 depicts the structure of a cache/main-memory system.

Figure 4.4 Cache/Main-Memory Structure

Cache Read Operation

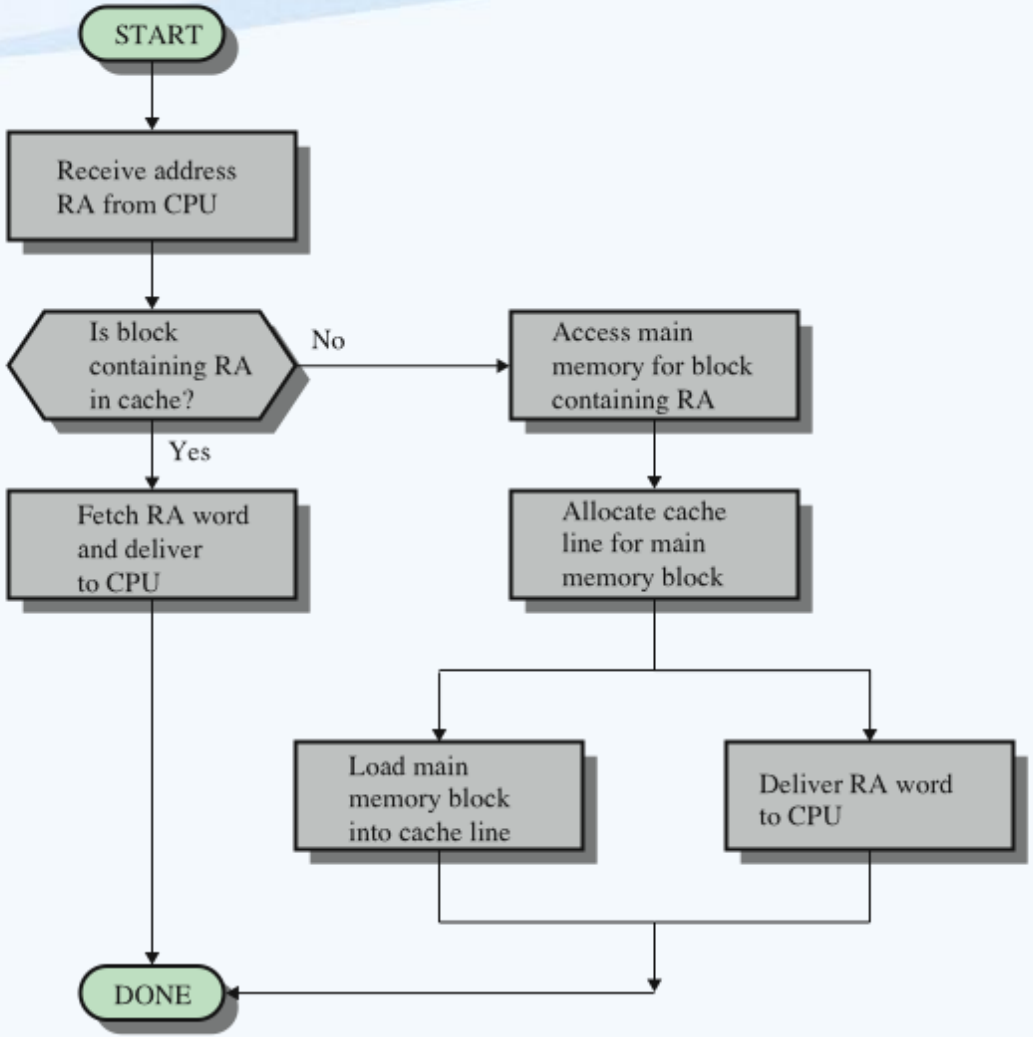


Figure 4.5 Cache Read Operation

Cache Read Operation

- Figure 4.5(S-35) illustrates the read operation. The processor generates the read address (RA) of a word to be read. If the word is contained in the cache, it is delivered to the processor. Otherwise, the block containing that word is loaded into the cache, and the word is delivered to the processor.

Typical Cache Organization

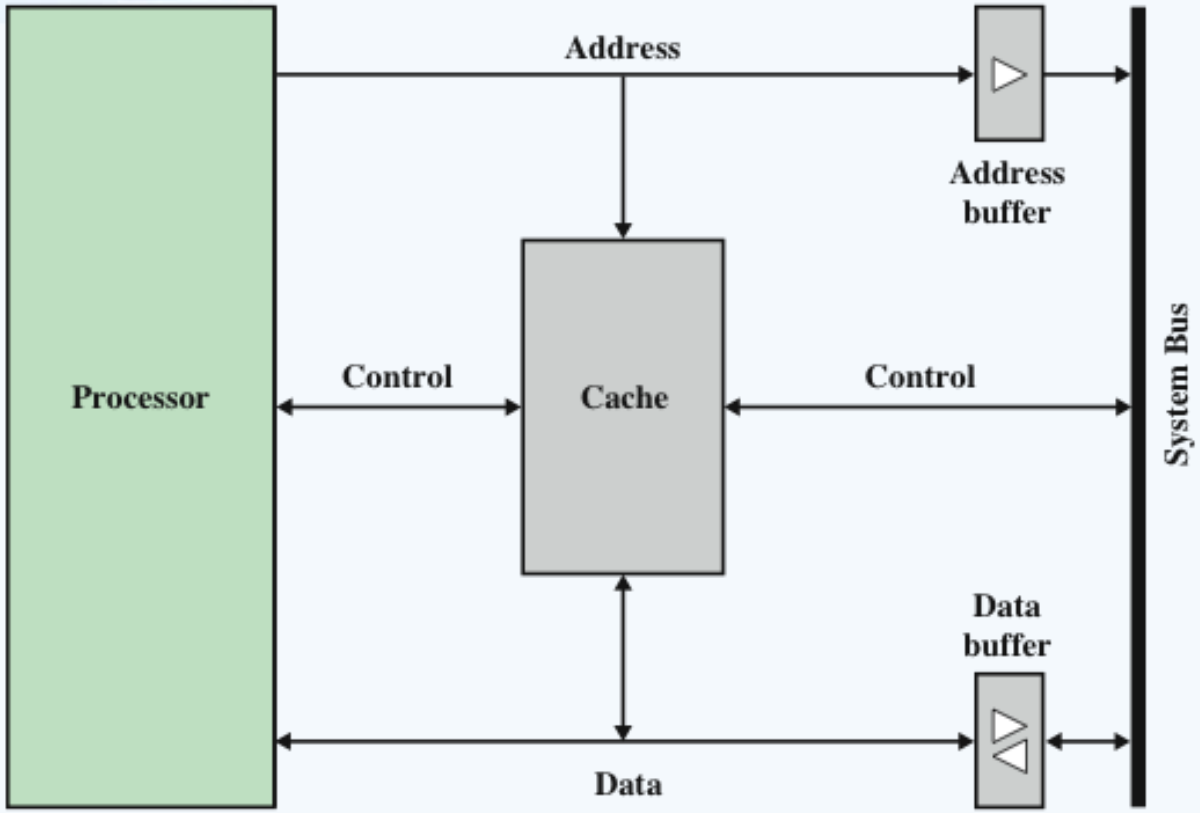


Figure 4.6 Typical Cache Organization

Typical Cache Organization

- Figure 4.5(S-35) shows these last two operations occurring in parallel and reflects the organization shown in Figure 4.6(S-37), which is typical of contemporary cache organizations. In this organization, the cache connects to the processor via data, control, and address lines. The data and address lines also attach to data and address buffers, which attach to a system bus from which main memory is reached. When a cache hit occurs, the data and address buffers are disabled and communication is only between processor and cache, with no system bus traffic. When a cache miss occurs, the desired address is loaded onto the system bus and the data are returned through the data buffer to both the cache and the processor. In other organizations, the cache is physically interposed between the processor and the main memory for all data, address, and control lines. In this latter case, for a cache miss, the desired word is first read into the cache and then transferred from cache to processor.

Elements of Cache Design

Cache Addresses	**Write Policy**
Logical	Write through
Physical	Write back
Cache Size	**Line Size**
Mapping Function	**Number of caches**
Direct	Single or two level
Associative	Unified or split
Set Associative	
Replacement Algorithm	
Least recently used (LRU)	
First in first out (FIFO)	
Least frequently used (LFU)	
Random	

Table 4.2 Elements of Cache Design

Elements of Cache Design

- This section provides an overview of cache design parameters and reports some typical results. We occasionally refer to the use of caches in high-performance computing (HPC). HPC deals with supercomputers and their software, especially for scientific applications that involve large amounts of data, vector* and matrix computation, and the use of parallel algorithms. Cache design for HPC is quite different than for other hardware platforms and applications. Indeed, many researchers have found that HPC applications perform poorly on computer architectures that employ caches. Other researchers have since shown that a cache hierarchy can be useful in improving performance if the application software is tuned to exploit the cache.

- Although there are a large number of cache implementations, there are a few basic design elements that serve to classify and differentiate cache architectures. Table 4.2(S-39) lists key elements.

*Vectors are line segments minimally defined as a starting point, a direction, and a length. They can, however, be much more complex and can include various sorts of lines, curves, and splines.

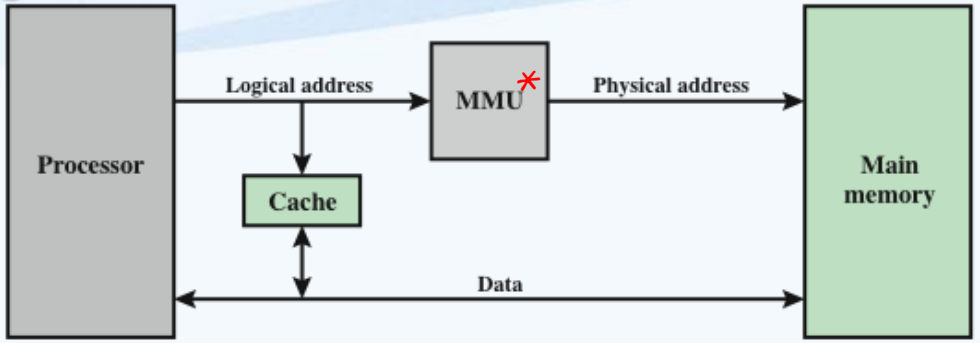
Cache Addresses

- Virtual memory
  - Facility that allows programs to address memory from a logical point of view, without regard to the amount of main memory physically available
  - When used, the address fields of machine instructions contain virtual addresses
  - For reads to and writes from main memory, a hardware memory management unit (MMU) translates each virtual address into a physical address in main memory

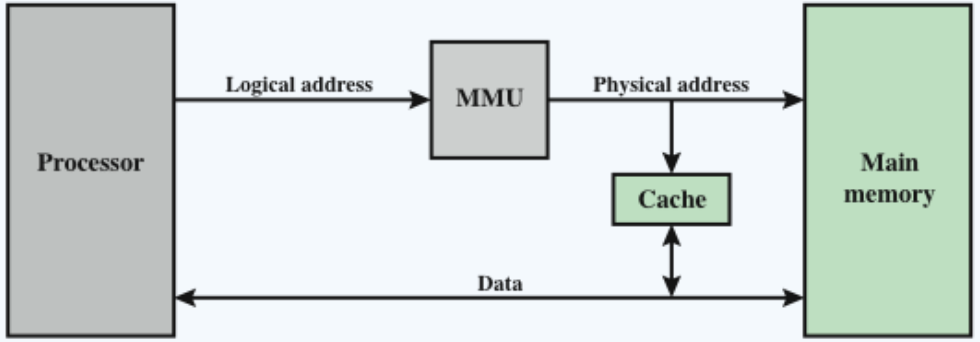
Cache Addresses

- Almost all non-embedded processors, and many embedded processors, support virtual memory, a concept discussed in Chapter 8. In essence, virtual memory is a facility that allows programs to address memory from a logical point of view, without regard to the amount of main memory physically available. When virtual memory is used, the address fields of machine instructions contain virtual addresses. For reads to and writes from main memory, a hardware memory management unit (MMU) translates each virtual address into a physical address in main memory.

Logical and Physical Caches



(a) Logical Cache



(b) Physical Cache

*Memory management unit (MMU)

Figure 4.7 Logical and Physical Caches

Logical and Physical Caches

- When virtual addresses are used, the system designer may choose to place the cache between the processor and the MMU or between the MMU and main memory (Figure 4.7)(S-43). A logical cache, also known as a virtual cache, stores data using virtual addresses. The processor accesses the cache directly, without going through the MMU. A physical cache stores data using main memory physical addresses.

- One obvious advantage of the logical cache is that cache access speed is faster than for a physical cache, because the cache can respond before the MMU performs an address translation. The disadvantage has to do with the fact that most virtual memory systems supply each application with the same virtual memory address space. That is, each application sees a virtual memory that starts at address 0. Thus, the same virtual address in two different applications refers to two different physical addresses. The cache memory must therefore be completely flushed(erased) with each application context switch, or extra bits must be added to each line of the cache to identify which virtual address space this address refers to.

Cache Sizes of Some Processors

Processor	Type	Year of Introduction	L1 Cache[a]	L2 cache	L3 Cache
IBM 360/85	Mainframe	1968	16 to 32 kB	—	—
PDP-11/70	Minicomputer	1975	1 kB	—	—
VAX 11/780	Minicomputer	1978	16 kB	—	—
IBM 3033	Mainframe	1978	64 kB	—	—
IBM 3090	Mainframe	1985	128 to 256 kB	—	—
Intel 80486	PC	1989	8 kB	—	—
Pentium	PC	1993	8 kB/8 kB	256 to 512 KB	—
PowerPC 601	PC	1993	32 kB	—	—
PowerPC 620	PC	1996	32 kB/32 kB	—	—
PowerPC G4	PC/server	1999	32 kB/32 kB	256 KB to 1 MB	2 MB
IBM S/390 G6	Mainframe	1999	256 kB	8 MB	—
Pentium 4	PC/server	2000	8 kB/8 kB	256 KB	—
IBM SP	High-end server/ supercomputer	2000	64 kB/32 kB	8 MB	—

[a] Two values separated by a slash refer to instruction and data caches.

[b] Both caches are instruction only; no data caches.

Cache Sizes of Some Processors

CRAY MTA[b]	Supercomputer	2000	8 kB	2 MB	—
Itanium	PC/server	2001	16 kB/16 kB	96 KB	4 MB
Itanium 2	PC/server	2002	32 kB	256 KB	6 MB
IBM POWER5	High-end server	2003	64 kB	1.9 MB	36 MB
CRAY XD-1	Supercomputer	2004	64 kB/64 kB	1MB	—
IBM POWER6	PC/server	2007	64 kB/64 kB	4 MB	32 MB
IBM z10	Mainframe	2008	64 kB/128 kB	3 MB	24-48 MB
Intel Core i7 EE 990	Workstaton/ server	2011	6 × 32 kB/32 kB	1.5 MB	12 MB
IBM zEnterprise 196	Mainframe/ Server	2011	24 × 64 kB/ 128 kB	24 × 1.5 MB	24 MB L3 192 MB L4

[a] Two values separated by a slash refer to instruction and data caches.

[b] Both caches are instruction only; no data caches.

Cache Sizes of Some Processors

- The first item in Table 4.2(S-39), cache size, has already been discussed. We would like the size of the cache to be small enough so that the overall average cost per bit is close to that of main memory alone and large enough so that the overall average access time is close to that of the cache alone. There are several other motivations for minimizing cache size. The larger the cache, the larger the number of gates involved in addressing the cache. The result is that large caches tend to be slightly slower than small ones—even when built with the same integrated circuit technology and put in the same place on chip and circuit board. The available chip and board area also limits cache size. Because the performance of the cache is very sensitive to the nature of the workload, it is impossible to arrive at a single "optimum" cache size. Table 4.3(S-45) lists the cache sizes of some current and past processors.

Mapping Function

- Because there are fewer cache lines than main memory blocks, an algorithm is needed for mapping main memory blocks into cache lines
- Three techniques can be used:

Direct

- The simplest technique
- Maps each block of main memory into only one possible cache line

Associative

- Permits each main memory block to be loaded into any line of the cache
- The cache control logic interprets a memory address simply as a Tag and a Word field
- To determine whether a block is in the cache, the cache control logic must simultaneously examine every line's Tag for a match

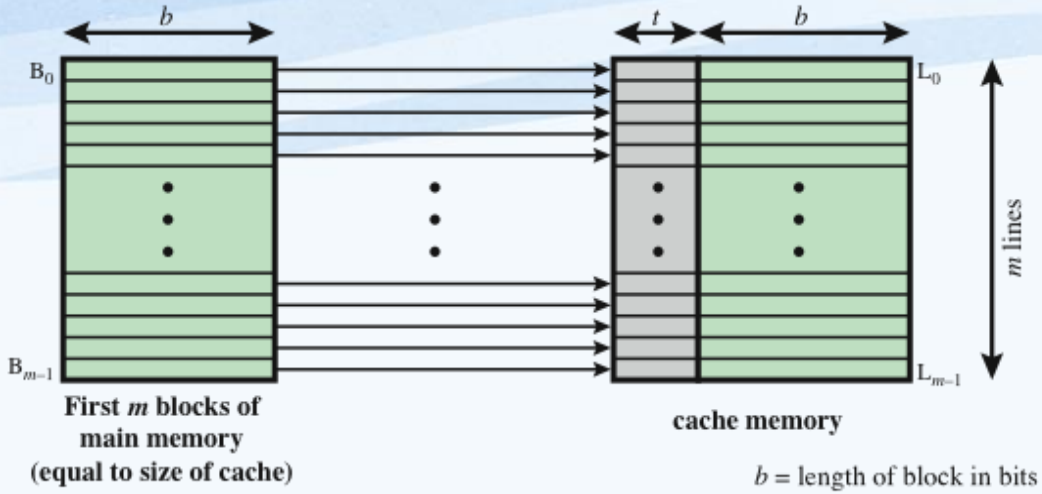
Set Associative

- A compromise that exhibits the strengths of both the direct and associative approaches while reducing their disadvantages

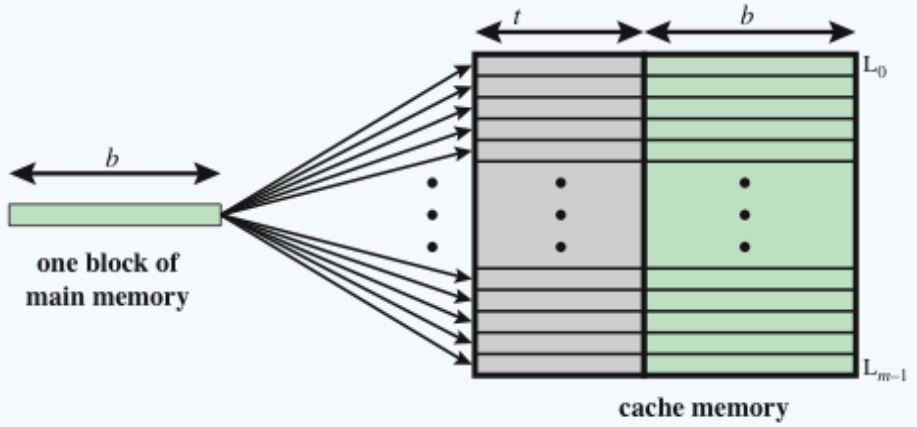
Mapping Function

- Because there are fewer cache lines than main memory blocks, an algorithm is needed for mapping main memory blocks into cache lines. Further, a means is needed for determining which main memory block currently occupies a cache line. The choice of the mapping function dictates how the cache is organized. Three techniques can be used: direct, associative, and set associative.
- Direct mapping: The simplest technique, known as direct mapping, maps each block of main memory into only one possible cache line.
- Associative mapping: Associative mapping overcomes the disadvantage of direct mapping by permitting each main memory block to be loaded into any line of the cache.
- Set-associative mapping: Set-associative mapping is a compromise that exhibits the strengths of both the direct and associative approaches while reducing their disadvantages.

Direct Mapping



(a) Direct mapping



(b) Associative mapping

Figure 4.8 Mapping From Main Memory to Cache: Direct and Associative

Direct Mapping

- The mapping is expressed as:

  $i = j \text{ modulo } m$

  (where i = cache line number, j = main memory block number, m = number of lines in the cache)

- Figure 4.8a(S-50) shows the mapping for the first m blocks of main memory. Each block of main memory maps into one unique line of the cache. The next m blocks of main memory map into the cache in the same fashion; that is, block B_m of main memory maps into line L_0 of cache, block B_{m+1} maps into line L_1, and so on.

Direct Mapping Cache Organization

The mapping function is easily implemented using the main memory address. Figure 4.9 illustrates the general mechanism.

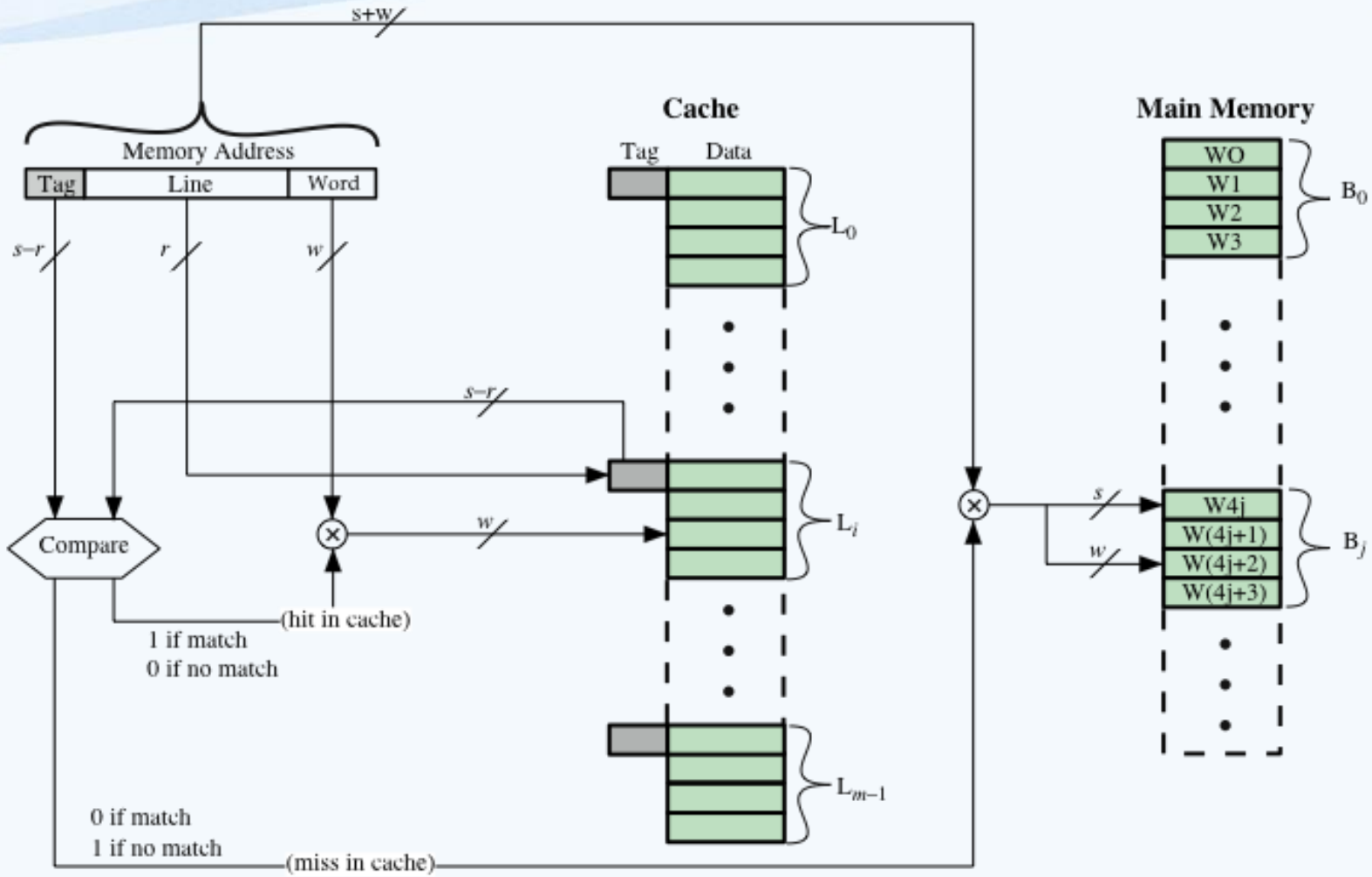


Figure 4.9 Direct-Mapping Cache Organization

Direct Mapping Example

Figure 4.10 Direct Mapping Example.

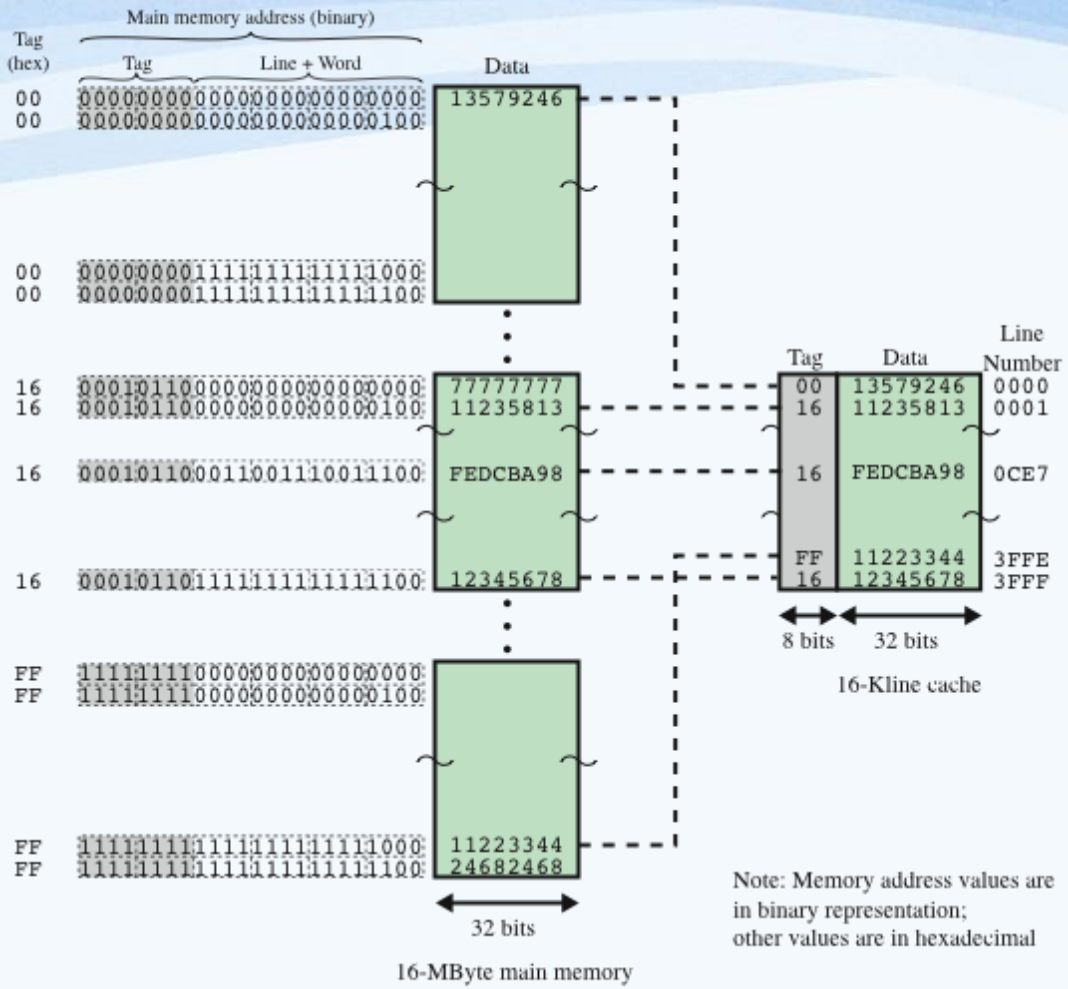


Figure 4.10 Direct Mapping Example

Example 4.2a Figure 4.10 shows our example system using direct mapping.[5] In the example, $m = 16K = 2^{14}$ and $i = j$ modulo 2^{14}. The mapping becomes

Cache Line	Starting Memory Address of Block
0	000000, 010000, ..., FF0000
1	000004, 010004, ..., FF0004
⋮	⋮
$2^{14} - 1$	00FFFC, 01FFFC, ..., FFFFFC

Note that no two blocks that map into the same line number have the same tag number. Thus, blocks with starting addresses 000000, 010000, ..., FF0000 have tag numbers 00, 01, ..., FF, respectively.

Referring back to Figure 4.5, a read operation works as follows. The cache system is presented with a 24-bit address. The 14-bit line number is used as an index into the cache to access a particular line. If the 8-bit tag number matches the tag number currently stored in that line, then the 2-bit word number is used to select one of the 4 bytes in that line. Otherwise, the 22-bit tag-plus-line field is used to fetch a block from main memory. The actual address that is used for the fetch is the 22-bit tag-plus-line concatenated with two 0 bits, so that 4 bytes are fetched starting on a block boundary.

Direct Mapping Summary

- Address length = (s + w) bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2w words or bytes
- Number of blocks in main memory = $2^{s+ w}/2^{w} = 2^{s}$
- Number of lines in cache = m = 2r
- Size of tag = (s – r) bits

Direct Mapping Summary

- The direct mapping technique is simple and inexpensive to implement. Its main disadvantage is that there is a fixed cache location for any given block. Thus, if a program happens to reference words repeatedly from two different blocks that map into the same line, then the blocks will be continually swapped in the cache, and the hit ratio will be low (a phenomenon known as thrashing).

Victim Cache

- Originally proposed as an approach to reduce the conflict misses of direct mapped caches without affecting its fast access time
- Fully associative cache
- Typical size is 4 to 16 cache lines
- Residing between direct mapped L1 cache and the next level of memory

Victim Cache

- One approach to lower the miss penalty is to remember what was discarded in case it is needed again. Since the discarded data has already been fetched, it can be used again at a small cost. Such recycling is possible using a victim cache. Victim cache was originally proposed as an approach to reduce the conflict misses of direct mapped caches without affecting its fast access time. Victim cache is a fully associative cache, whose size is typically 4 to 16 cache lines, residing between a direct mapped L1 cache and the next level of memory. This concept is explored in Appendix D.

Fully Associative Cache Organization

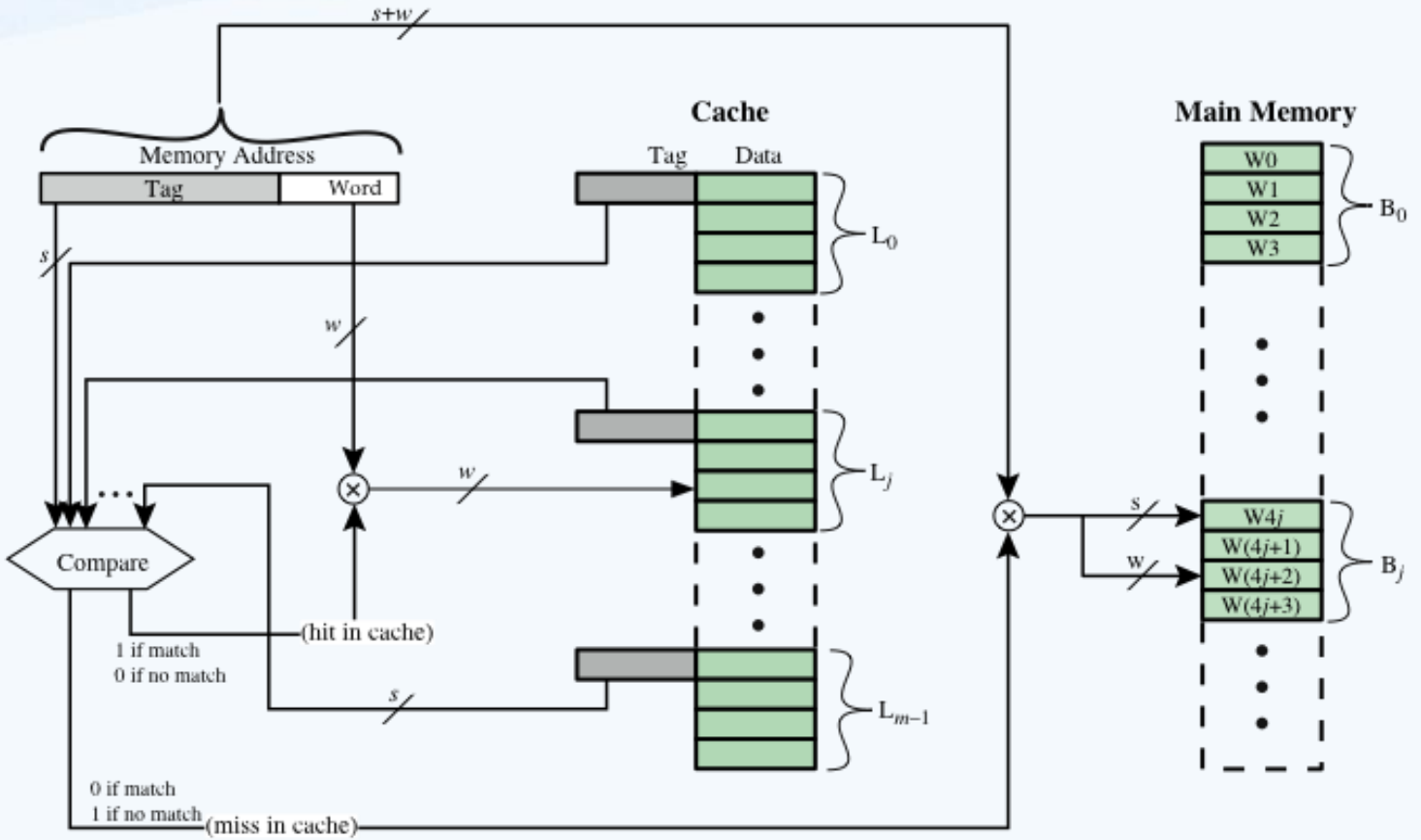


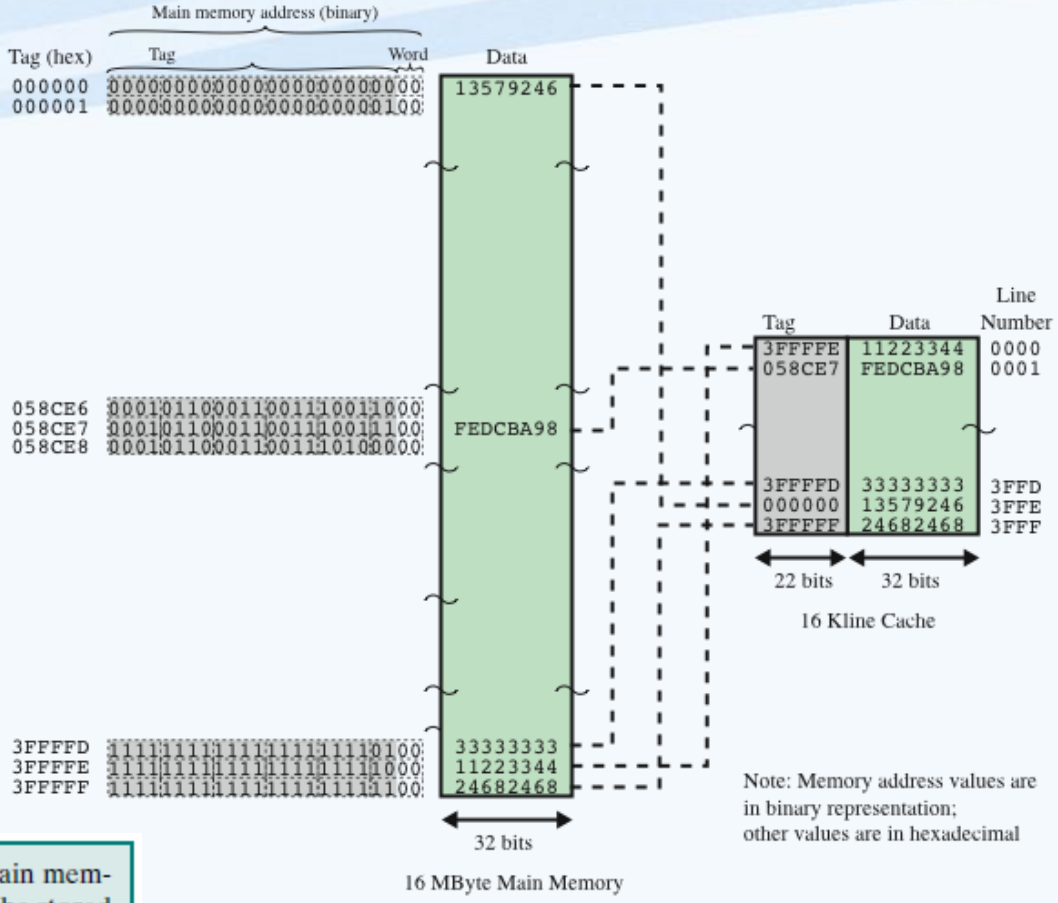
Figure 4.11 Fully Associative Cache Organization

Fully Associative Cache Organization

- Associative mapping overcomes the disadvantage of direct mapping by permitting each main memory block to be loaded into any line of the cache (Figure 4.8b). In this case, the cache control logic interprets a memory address simply as a Tag and a Word field. The Tag field uniquely identifies a block of main memory. To determine whether a block is in the cache, the cache control logic must simultaneously examine every line's tag (disadvantage) for a match. Figure 4.11(S-58) illustrates the logic.

Mapping Example

Figure 4.12 Associative Mapping Example.



Example 4.2b Figure 4.12 shows our example using associative mapping. A main memory address consists of a 22-bit tag and a 2-bit byte number. The 22-bit tag must be stored with the 32-bit block of data for each line in the cache. Note that it is the leftmost (most significant) 22 bits of the address that form the tag. Thus, the 24-bit hexadecimal address 16339C has the 22-bit tag 058CE7. This is easily seen in binary notation:

memory address	0001	0110	0011	0011	1001	1100	(binary)
	1	6	3	3	9	C	(hex)
tag (leftmost 22 bits)	00	0101	1000	1100	1110	0111	(binary)
	0	5	8	C	E	7	(hex)

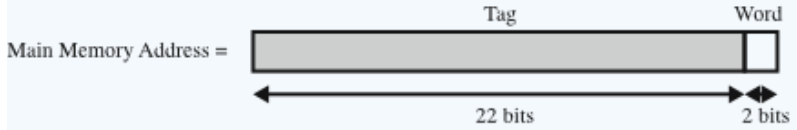


Figure 4.12 Associative Mapping Example

Associative Mapping Summary

Associative Mapping Summary

- Address length = (s + w) bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2w words or bytes
- Number of blocks in main memory = $2^{s+ w}/2^{w} = 2^{s}$
- Number of lines in cache = undetermined
- Size of tag = s bits

Direct Mapping Summary

- Address length = (s + w) bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2w words or bytes
- Number of blocks in main memory = $2^{s+ w}/2^{w} = 2^{s}$
- Number of lines in cache = m = 2r
- Size of tag = (s – r) bits

Associative Mapping Summary

- With associative mapping, there is flexibility as to which block to replace when a new block is read into the cache. Replacement algorithms, discussed later in this section, are designed to maximize the hit ratio. The principal disadvantage of associative mapping is the complex circuitry required to examine the tags of all cache lines in parallel.

Set Associative Mapping

- Compromise that exhibits the strengths of both the direct and associative approaches while reducing their disadvantages
- Cache consists of a number of sets
- Each set contains a number of lines
- A given block maps to any line in a given set
- e.g. 2 lines per set (e.g. means 'exempli gratia' or 'for example')
  - 2 way associative mapping
  - A given block can be in one of 2 lines in only one set

Set Associative Mapping

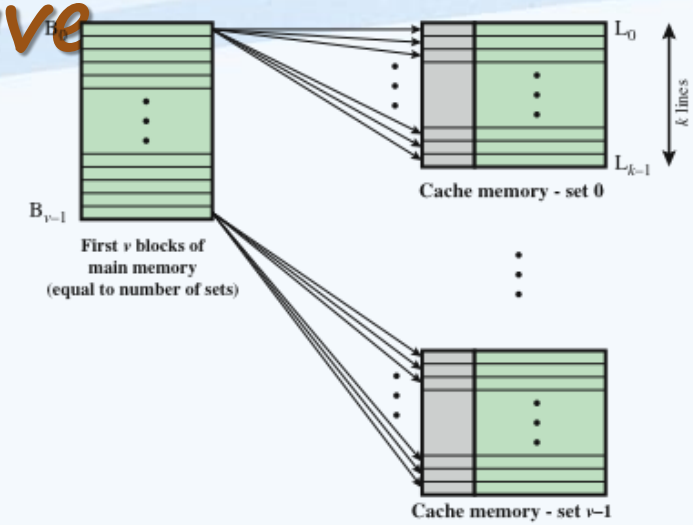
- Set-associative mapping is a compromise that exhibits the strengths of both the direct and associative approaches while reducing their disadvantages.
- In this case, the cache consists of a number sets, each of which consists of a number of lines. The relationships are:

  $m = v * k$

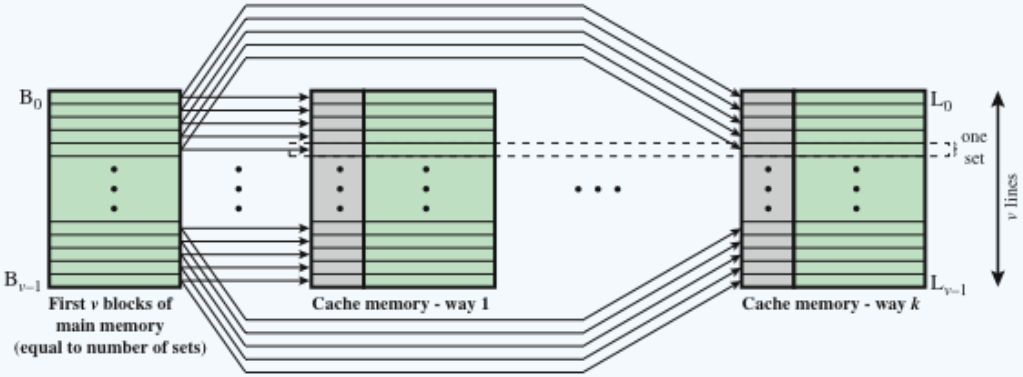
  $i = j \text{ modulo } v$

- (where i = cache set number, j = main memory block number, m = number of lines in the cache, v = number of sets k = number of lines in each set)
- This is referred to as k-way set-associative mapping.

Mapping From Main Memory to Cache : k-Way Set Associative



(a) *v* associative-mapped caches



(b) *k* direct-mapped caches

Figure 4.13 Mapping From Main Memory to Cache: *k*-way Set Associative

Mapping From Main Memory to Cache : k-Way Set Associative

- Figure 4.13a(S-65) illustrates this mapping for the first v blocks of main memory. As with associative mapping, each word maps into multiple cache lines. For set-associative mapping, each word maps into all the cache lines in a specific set, so that main memory block B_0 maps into set 0, and so on. Thus, the set-associative cache can be physically implemented as v associative caches.
- It is also possible to implement the set-associative cache as k direct mapping caches, as shown in Figure 4.13b(S-65). Each direct-mapped cache is referred to as a way, consisting of v lines. The first v lines of main memory are direct mapped into the v lines of each way; the next group of v lines of main memory are similarly mapped, and so on.
- The direct-mapped implementation is typically used for small degrees of associativity (small values of k) while the associative-mapped implementation is typically used for higher degrees of associativity.

k-Way Set Associative Cache Organization

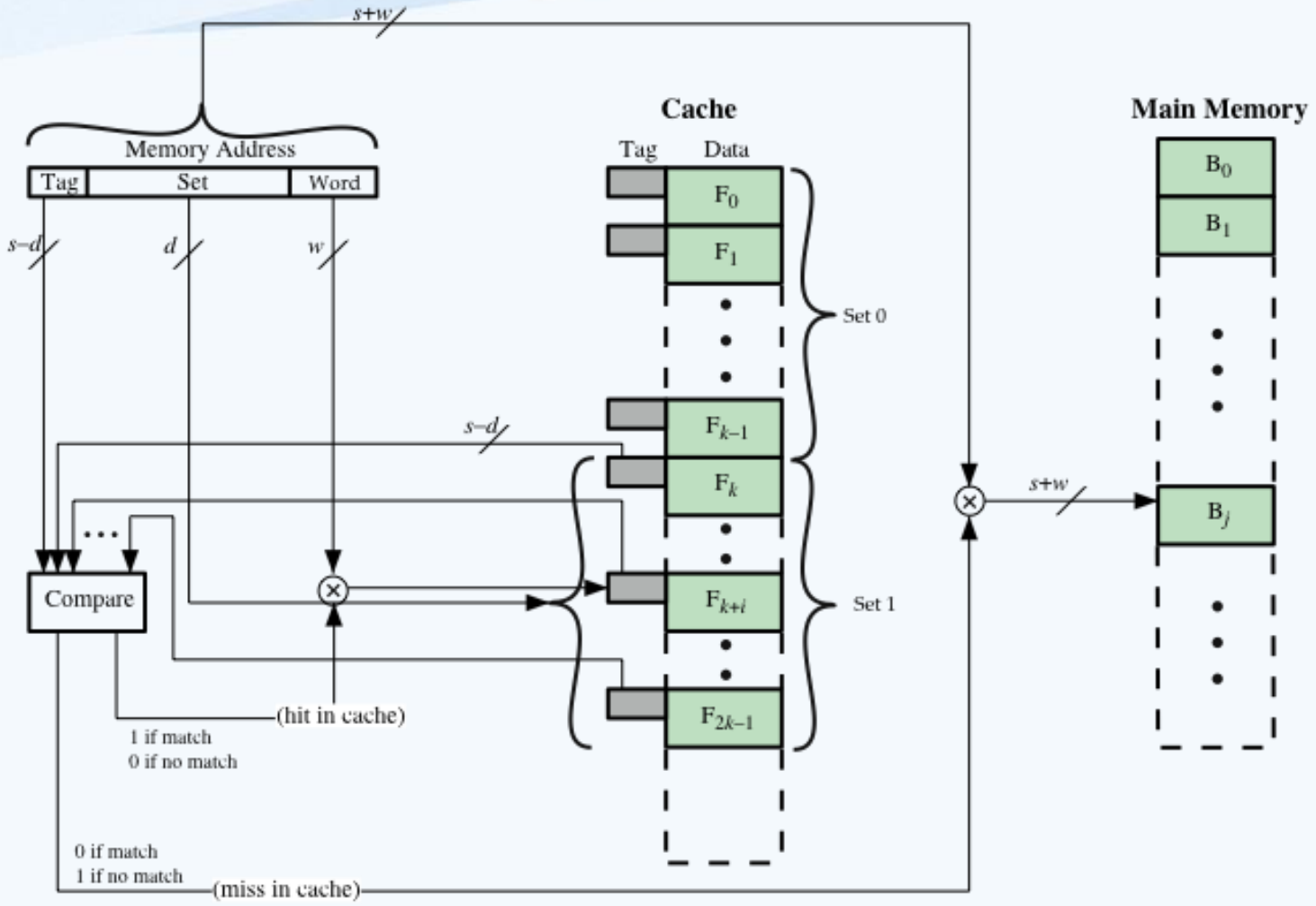


Figure 4.14 *k*-Way Set Associative Cache Organization

k-Way Set Associative Cache Organization

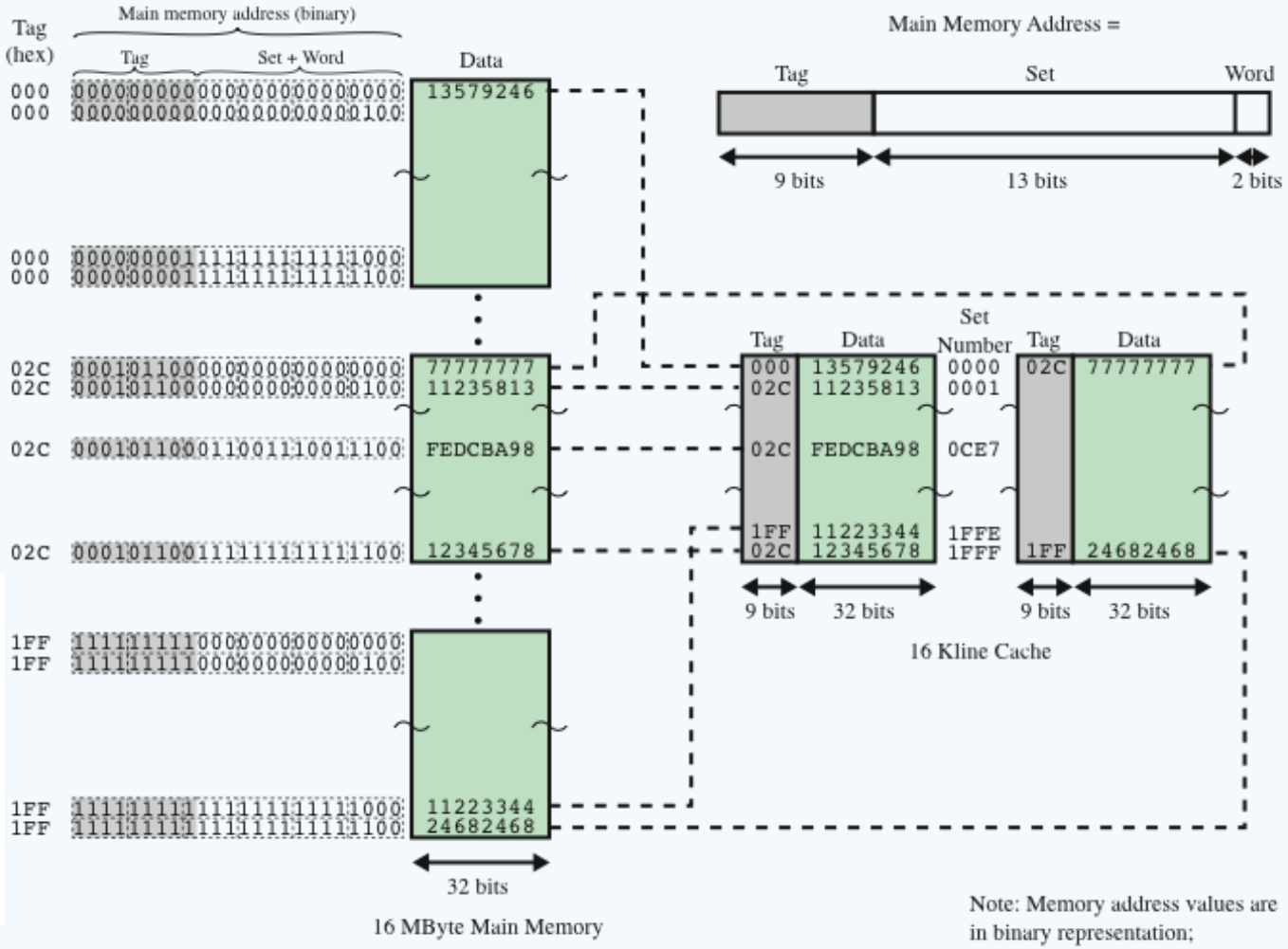
- For set-associative mapping, the cache control logic interprets a memory address as three fields: Tag, Set, and Word. The d set bits specify one of v = 2d sets. The s bits of the Tag and Set fields specify one of the 2s blocks of main memory. Figure 4.14(S-67) illustrates the cache control logic. With fully associative mapping, the tag in a memory address is quite large and must be compared to the tag of every line in the cache. With k-way set-associative mapping, the tag in a memory address is much smaller and is only compared to the k tags within a single set.

Set Associative Mapping Summary

- Address length = (s + w) bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^{w} words or bytes
- Number of blocks in main memory = $2^{s+w}/2^{w}=2^{s}$
- Number of lines in set = k
- Number of sets = v = 2^{d}
- Number of lines in cache = m=kv = k * 2^{d}
- Size of cache = k * 2^{d+w} words or bytes
- Size of tag = (s – d) bits

Two way set-associative mapping with two

Figure 4.15 shows an example using set-associative mapping with two lines in each set, referred to as two-way set-associative.



Example 4.2c Figure 4.15 shows our example using set-associative mapping with two lines in each set, referred to as two-way set-associative. The 13-bit set number identifies a unique set of two lines within the cache. It also gives the number of the block in main memory, modulo 2^{13}. This determines the mapping of blocks into lines. Thus, blocks 000000, 008000, ..., FF8000 of main memory map into cache set 0. Any of those blocks can be loaded into either of the two lines in the set. Note that no two blocks that map into the same cache set have the same tag number. For a read operation, the 13-bit set number is used to determine which set of two lines is to be examined. Both lines in the set are examined for a match with the tag number of the address to be accessed.

Figure 4.15 Two-Way Set Associative Mapping Example

Varying Associativity Over Cache Size

Beyond about 32 kB, increase in cache size brings no significant increase in performance

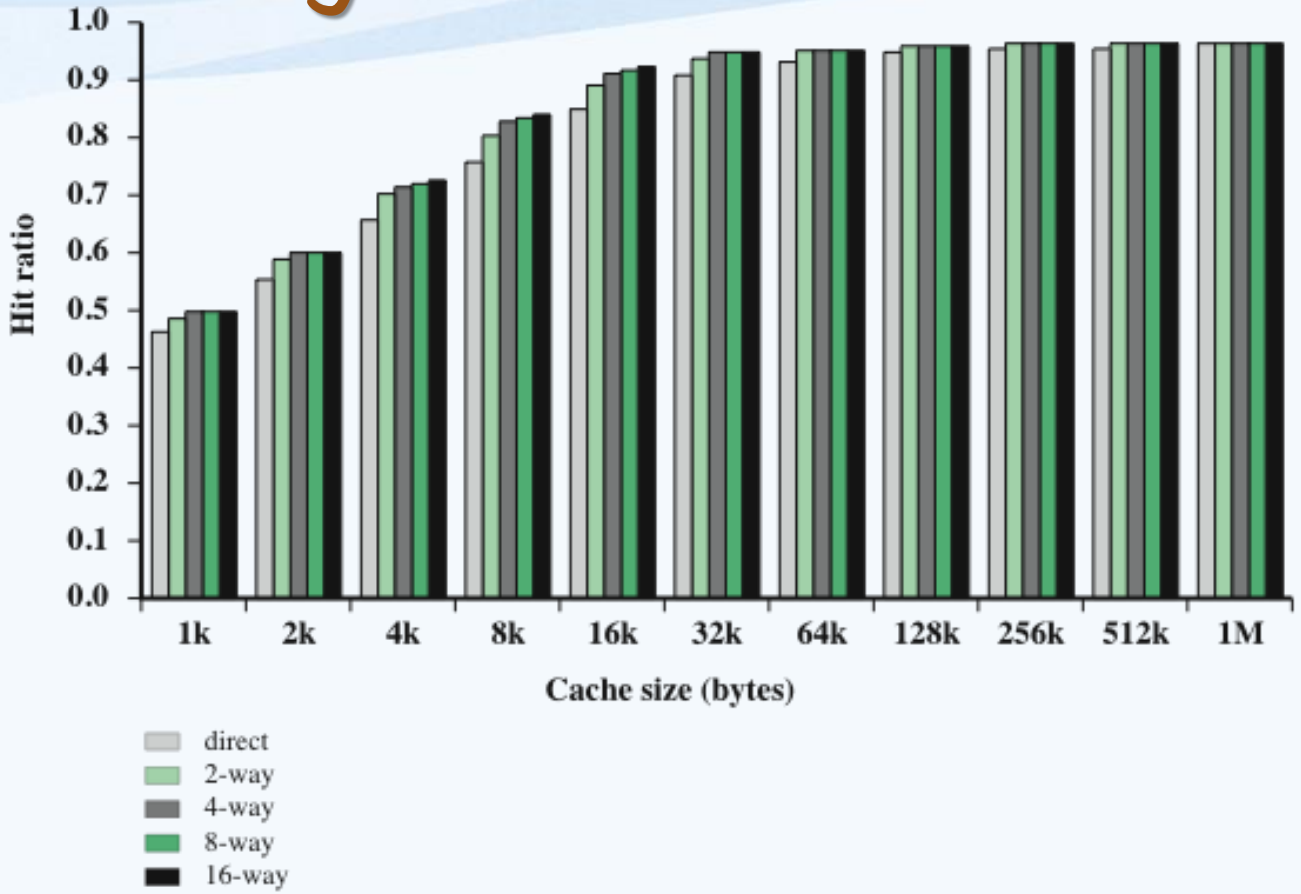


Figure 4.16 Varying Associativity over Cache Size

Varying Associativity Over Cache Size

- Figure 4.16(S-71) shows the results of one simulation study of set-associative cache performance as a function of cache size. The difference in performance between direct and two-way set associative is significant up to at least a cache size of 64 kB. Note also that the difference between two-way and four-way at 4 kB is much less than the difference in going from for 4 kB to 8 kB in cache size. The complexity of the cache increases in proportion to the associativity, and in this case would not be justifiable against increasing cache size to 8 or even 16 Kbytes. A final point to note is that beyond about 32 kB, increase in cache size brings no significant increase in performance.

- The results of Figure 4.16(S-71) are based on simulating the execution of a GCC compiler. Different applications may yield different results. For example, reports on the results for cache performance using many of the CPU2000 SPEC benchmarks. The results of in comparing hit ratio to cache size follow the same pattern as Figure 4.16(S-71), but the specific values are somewhat different.

Replacement Algorithms

- Once the cache has been filled, when a new block is brought into the cache, one of the existing blocks must be replaced
- For direct mapping there is only one possible line for any particular block and no choice is possible
- For the associative and set-associative techniques a replacement algorithm is needed
- To achieve high speed, an algorithm must be implemented in hardware

Replacement Algorithms

- Once the cache has been filled, when a new block is brought into the cache, one of the existing blocks must be replaced. For direct mapping, there is only one possible line for any particular block, and no choice is possible. For the associative and set-associative techniques, a replacement algorithm is needed. To achieve high speed, such an algorithm must be implemented in hardware.

Replacement Algorithms

- The four most common replacement algorithms are:
- Least recently used (LRU)
  - Most effective
  - Replace that block in the set that has been in the cache longest with no reference to it
  - Because of its simplicity of implementation, LRU is the most popular replacement algorithm
- First-in-first-out (FIFO)
  - Replace that block in the set that has been in the cache longest
  - Easily implemented as a round-robin or circular buffer technique
- Least frequently used (LFU)
  - Replace that block in the set that has experienced the fewest references
  - Could be implemented by associating a counter with each line

Replacement Algorithms

- A number of algorithms have been tried. We mention four of the most common. Probably the most effective is least recently used (LRU): Replace that block in the set that has been in the cache longest with no reference to it. For two-way set associative, this is easily implemented. Each line includes a USE bit. When a line is referenced, its USE bit is set to 1 and the USE bit of the other line in that set is set to 0. When a block is to be read into the set, the line whose USE bit is 0 is used. Because we are assuming that more recently used memory locations are more likely to be referenced, LRU should give the best hit ratio. LRU is also relatively easy to implement for a fully associative cache. The cache mechanism maintains a separate list of indexes to all the lines in the cache. When a line is referenced, it moves to the front of the list. For replacement, the line at the back of the list is used. Because of its simplicity of implementation, LRU is the most popular replacement algorithm.

Replacement Algorithms

- Another possibility is first-in-first-out (FIFO): Replace that block in the set that has been in the cache longest. FIFO is easily implemented as a round-robin or circular buffer technique. Still another possibility is least frequently used (LFU): Replace that block in the set that has experienced the fewest references. LFU could be implemented by associating a counter with each line. A technique not based on usage (i.e., not LRU, LFU, FIFO, or some variant) is to pick a line at random from among the candidate lines. Simulation studies have shown that random replacement provides only slightly inferior performance to an algorithm based on usage.

Write Policy

When a block that is resident in the cache is to be replaced there are two cases to consider:

If the old block in the cache has not been altered then it may be overwritten with a new block without first writing out the old block

If at least one write operation has been performed on a word in that line of the cache then main memory must be updated by writing the line of cache out to the block of memory before bringing in the new block

There are two problems to contend with:

More than one device may have access to main memory

A more complex problem occurs when multiple processors are attached to the same bus and each processor has its own local cache - if a word is altered in one cache it could conceivably invalidate a word in other caches

Write Policy

- When a block that is resident in the cache is to be replaced, there are two cases to consider. If the old block in the cache has not been altered, then it may be overwritten with a new block without first writing out the old block. If at least one write operation has been performed on a word in that line of the cache, then main memory must be updated by writing the line of cache out to the block of memory before bringing in the new block. A variety of write policies, with performance and economic trade-offs, is possible. There are two problems to contend with. First, more than one device may have access to main memory. For example, an I/O module may be able to read-write directly to memory. If a word has been altered only in the cache, then the corresponding memory word is invalid. Further, if the I/O device has altered main memory, then the cache word is invalid. A more complex problem occurs when multiple processors are attached to the same bus and each processor has its own local cache. Then, if a word is altered in one cache, it could conceivably invalidate a word in other caches.

Write Through and Write Back

- Write through
  - Simplest technique
  - All write operations are made to main memory as well as to the cache
  - The main disadvantage of this technique is that it generates substantial memory traffic and may create a bottleneck
- Write back
  - Minimizes memory writes
  - Updates are made only in the cache
  - Portions of main memory are invalid and hence accesses by I/O modules can be allowed only through the cache
  - This makes for complex circuitry and a potential bottleneck

Write Through and Write Back

- The simplest technique is called write through. Using this technique, all write operations are made to main memory as well as to the cache, ensuring that main memory is always valid. Any other processor–cache module can monitor traffic to main memory to maintain consistency within its own cache. The main disadvantage of this technique is that it generates substantial memory traffic and may create a bottleneck.

- An alternative technique, known as write back, minimizes memory writes. With write back, updates are made only in the cache. When an update occurs, a dirty bit, or use bit, associated with the line is set. Then, when a block is replaced, it is written back to main memory if and only if the dirty bit is set. The problem with write back is that portions of main memory are invalid, and hence accesses by I/O modules can be allowed only through the cache. This makes for complex circuitry and a potential bottleneck. Experience has shown that the percentage of memory references that are writes is on the order of 15%. However, for HPC applications, this number may approach 33% (vector-vector multiplication) and can go as high as 50% (matrix transposition).

Write Through and Write Back

- In a bus organization in which more than one device (typically a processor) has a cache and main memory is shared, a new problem is introduced. If data in one cache are altered, this invalidates not only the corresponding word in main memory, but also that same word in other caches (if any other cache happens to have that same word). Even if a write-through policy is used, the other caches may contain invalid data. A system that prevents this problem is said to maintain cache coherency. Possible approaches to cache coherency include the following:
  - Bus watching with write through: Each cache controller monitors the address lines to detect write operations to memory by other bus masters. If another master writes to a location in shared memory that also resides in the cache memory, the cache controller invalidates that cache entry. This strategy depends on the use of a write-through policy by all cache controllers.

Write Through and Write Back

- Hardware transparency: Additional hardware is used to ensure that all updates to main memory via cache are reflected in all caches. Thus, if one processor modifies a word in its cache, this update is written to main memory. In addition, any matching words in other caches are similarly updated.
- Non-cacheable memory: Only a portion of main memory is shared by more than one processor, and this is designated as non-cacheable. In such a system, all accesses to shared memory are cache misses, because the shared memory is never copied into the cache. The non-cacheable memory can be identified using chip-select logic or high-address bits.

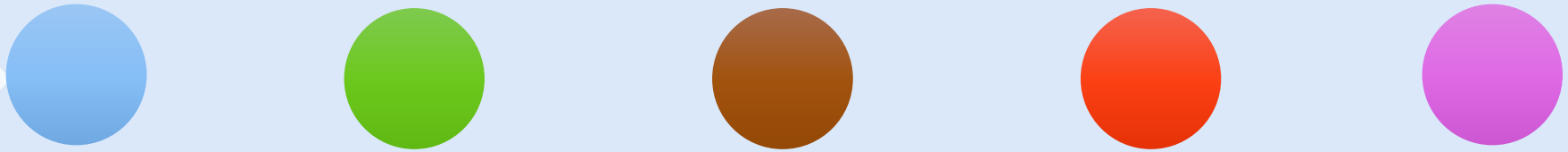
Line Size

When a block of data is retrieved and placed in the cache not only the desired word but also some number of adjacent words are retrieved

As the block size increases more useful data are brought into the cache

Two specific effects come into play:

- Larger blocks reduce the number of blocks that fit into a cache
- As a block becomes larger each additional word is farther from the requested word



As the block size increases the hit ratio will at first increase because of the principle of locality

The hit ratio will begin to decrease as the block becomes bigger and the probability of using the newly fetched information becomes less than the probability of reusing the information that has to be replaced

Line Size

- Another design element is the line size. When a block of data is retrieved and placed in the cache, not only the desired word but also some number of adjacent words are retrieved. As the block size increases from very small to larger sizes, the hit ratio will at first increase because of the principle of locality, which states that data in the vicinity of a referenced word are likely to be referenced in the near future. As the block size increases, more useful data are brought into the cache. The hit ratio will begin to decrease, however, as the block becomes even bigger and the probability of using the newly fetched information becomes less than the probability of reusing the information that has to be replaced. Two specific effects come into play:
  - Larger blocks reduce the number of blocks that fit into a cache. Because each block fetch overwrites older cache contents, a small number of blocks results in data being overwritten shortly after they are fetched.

Line Size

- As a block becomes larger, each additional word is farther from the requested word and therefore less likely to be needed in the near future.

- The relationship between block size and hit ratio is complex, depending on the locality characteristics of a particular program, and no definitive optimum value has been found. A size of from 8 to 64 bytes seems reasonably close to optimum. For HPC systems, 64- and 128-byte cache line sizes are most frequently used.

Multilevel Caches

- As logic density has increased it has become possible to have a cache on the same chip as the processor
- The on-chip cache reduces the processor's external bus activity and speeds up execution time and increases overall system performance
  - When the requested instruction or data is found in the on-chip cache, the bus access is eliminated
  - On-chip cache accesses will complete appreciably faster than would even zero-wait state bus cycles
  - During this period the bus is free to support other transfers

Multilevel Caches

- Two-level cache:
  - Internal cache designated as level 1 (L1)
  - External cache designated as level 2 (L2)
- Potential savings due to the use of an L2 cache depends on the hit rates in both the L1 and L2 caches
- The use of multilevel caches complicates all of the design issues related to caches, including size, replacement algorithm, and write policy

Multilevel Caches

- As logic density has increased, it has become possible to have a cache on the same chip as the processor: the on-chip cache. Compared with a cache reachable via an external bus, the on-chip cache reduces the processor's external bus activity and therefore speeds up execution times and increases overall system performance. When the requested instruction or data is found in the on-chip cache, the bus access is eliminated. Because of the short data paths internal to the processor, compared with bus lengths, on-chip cache accesses will complete appreciably faster than would even zero-wait state bus cycles. Furthermore, during this period the bus is free to support other transfers. The inclusion of an on-chip cache leaves open the question of whether an off-chip, or external, cache is still desirable. Typically, the answer is yes, and most contemporary designs include both on-chip and external caches.

Multilevel Caches

- The simplest such organization is known as a two-level cache, with the internal cache designated as level 1 (L1) and the external cache designated as level 2 (L2). The reason for including an L2 cache is the following: If there is no L2 cache and the processor makes an access request for a memory location not in the L1 cache, then the processor must access DRAM or ROM memory across the bus. Due to the typically slow bus speed and slow memory access time, this results in poor performance. On the other hand, if an L2 SRAM (static RAM) cache is used, then frequently the missing information can be quickly retrieved. If the SRAM is fast enough to match the bus speed, then the data can be accessed using a zero-wait state transaction, the fastest type of bus transfer.

Multilevel Caches

- Two features of contemporary cache design for multilevel caches are noteworthy. First, for an off-chip L2 cache, many designs do not use the system bus as the path for transfer between the L2 cache and the processor, but use a separate data path, so as to reduce the burden on the system bus. Second, with the continued shrinkage of processor components, a number of processors now incorporate the L2 cache on the processor chip, improving performance.
- The potential savings due to the use of an L2 cache depends on the hit rates in both the L1 and L2 caches. Several studies have shown that, in general, the use of a second-level cache does improve performance. However, the use of multilevel caches does complicate all of the design issues related to caches, including size, replacement algorithm, and write policy.

Hit Ratio (L1 & L2) For 8 Kbyte and 16 Kbyte L1

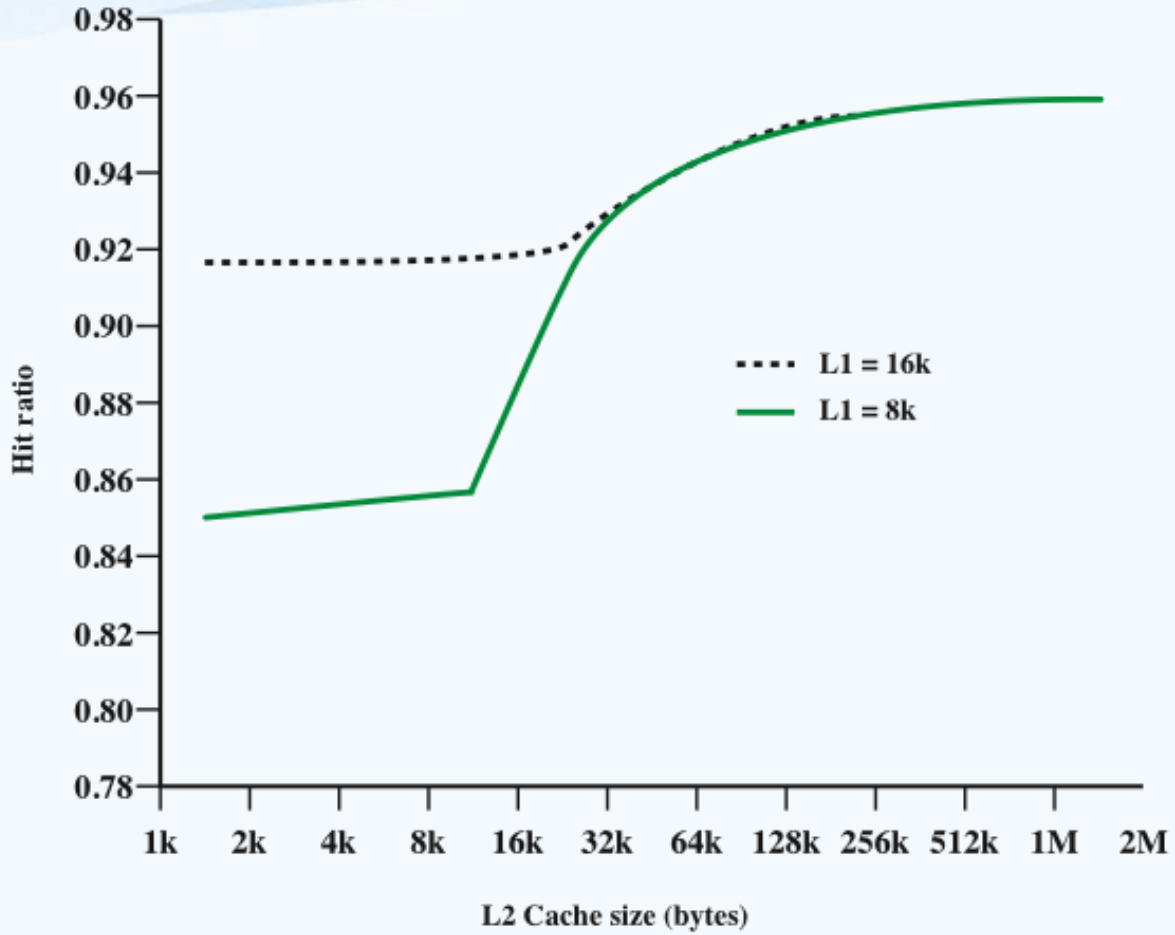


Figure 4.17 Total Hit Ratio (L1 and L2) for 8 Kbyte and 16 Kbyte L1

Hit Ratio (L1 & L2) For 8 Kbyte and 16 Kbyte L1

- Figure 4.17(S-92) shows the results of one simulation study of two-level cache performance as a function of cache size. The figure assumes that both caches have the same line size and shows the total hit ratio. That is, a hit is counted if the desired data appears in either the L1 or the L2 cache. The figure shows the impact of L2 on total hits with respect to L1 size. L2 has little effect on the total number of cache hits until it is at least double the L1 cache size. Note that the steepest part of the slope for an L1 cache of 8 Kbytes is for an L2 cache of 16 Kbytes. Again for an L1 cache of 16 Kbytes, the steepest part of the curve is for an L2 cache size of 32 Kbytes. Prior to that point, the L2 cache has little, if any, impact on total cache performance. The need for the L2 cache to be larger than the L1 cache to affect performance makes sense. If the L2 cache has the same line size and capacity as the L1 cache, its contents will more or less mirror those of the L1 cache.

Hit Ratio (L1 & L2) For 8 Kbyte and 16 Kbyte L1

- With the increasing availability of on-chip area available for cache, most contemporary microprocessors have moved the L2 cache onto the processor chip and added an L3 cache. Originally, the L3 cache was accessible over the external bus. More recently, most microprocessors have incorporated an on-chip L3 cache. In either case, there appears to be a performance advantage to adding the third level. Further, large systems, such as the IBM mainframe zEnterprise systems, now incorporate 3 on-chip cache levels and a fourth level of cache shared across multiple chips.

Unified Versus Split Caches

- Has become common to split cache:
  - One dedicated to instructions
  - One dedicated to data
  - Both exist at the same level, typically as two L1 caches
- Advantages of unified cache:
  - Higher hit rate
    - Balances load of instruction and data fetches automatically
    - Only one cache needs to be designed and implemented
- Trend is toward split caches at the L1 and unified caches for higher levels
- Advantages of split cache:
  - Eliminates cache contention between instruction fetch/decode unit and execution unit
    - Important in pipelining

Unified Versus Split Caches

- When the on-chip cache first made an appearance, many of the designs consisted of a single cache used to store references to both data and instructions. More recently, it has become common to split the cache into two: one dedicated to instructions and one dedicated to data.

- These two caches both exist at the same level, typically as two L1 caches. When the processor attempts to fetch an instruction from main memory, it first consults the instruction L1 cache, and when the processor attempts to fetch data from main memory, it first consults the data L1 cache.

- There are two potential advantages of a unified cache:

  - For a given cache size, a unified cache has a higher hit rate than split caches because it balances the load between instruction and data fetches automatically. That is, if an execution pattern involves many more instruction fetches than data fetches, then the cache will tend to fill up with instructions, and if an execution pattern involves relatively more data fetches, the opposite will occur.

Unified Versus Split Caches

- There are two potential advantages of a unified cache:
  - For a given cache size, a unified cache has a higher hit rate than split caches because it balances the load between instruction and data fetches automatically. That is, if an execution pattern involves many more instruction fetches than data fetches, then the cache will tend to fill up with instructions, and if an execution pattern involves relatively more data fetches, the opposite will occur.
  - Only one cache needs to be designed and implemented.
- The trend is toward split caches at the L1 and unified caches for higher levels, particularly for superscalar machines, which emphasize parallel instruction execution and the prefetching of predicted future instructions. The key advantage of the split cache design is that it eliminates contention for the cache between the instruction fetch/decode unit and the execution unit. This is important in any design that relies on the pipelining of instructions. Typically, the processor will fetch instructions ahead of time and fill a buffer, or pipeline, with instructions to be executed. Suppose now that we have a unified instruction/data cache. When the execution unit performs a memory access to load and store data, the request is submitted to the unified cache. If, at the same time, the instruction prefetcher issues a read request to the cache for an instruction, that request will be temporarily blocked so that the cache can service the execution unit first, enabling it to complete the currently executing instruction.

Unified Versus Split Caches

This cache contention can degrade performance by interfering with efficient use of the instruction pipeline. The split cache structure overcomes this difficulty. When the on-chip cache first made an appearance, many of the designs consisted of a single cache used to store references to both data and instructions. More recently, it has become common to split the cache into two: one dedicated to instructions and one dedicated to data. These two caches both exist at the same level, typically as two L1 caches. When the processor attempts to fetch an instruction from main memory, it first consults the instruction L1 cache, and when the processor attempts to fetch data from main memory, it first consults the data L1 cache.

Pentium 4 Cache

Problem	Solution	Processor on which Feature First Appears
External memory slower than the system bus.	Add external cache using faster memory technology.	386
Increased processor speed results in external bus becoming a bottleneck for cache access.	Move external cache on-chip, operating at the same speed as the processor.	486
Internal cache is rather small, due to limited space on chip	Add external L2 cache using faster technology than main memory	486
Contention occurs when both the Instruction Prefetcher and the Execution Unit simultaneously require access to the cache. In that case, the Prefetcher is stalled while the Execution Unit's data access takes place.	Create separate data and instruction caches.	Pentium
Increased processor speed results in external bus becoming a bottleneck for L2 cache access.	Create separate back-side bus that runs at higher speed than the main (front-side) external bus. The BSB is dedicated to the L2 cache.	Pentium Pro
	Move L2 cache on to the processor chip.	Pentium II
Some applications deal with massive databases and must have rapid access to large amounts of data. The on-chip caches are too small.	Add external L3 cache.	Pentium III
	Move L3 cache on-chip.	Pentium 4

Table 4.4 Intel Cache Evolution

Pentium 4 Cache

- The evolution of cache organization is seen clearly in the evolution of Intel microprocessors (Table 4.4)(S-99). The 80386 does not include an on-chip cache. The 80486 includes a single on-chip cache of 8 Kbytes, using a line size of 16 bytes and a four-way set-associative organization.
- All of the Pentium processors include two on-chip L1 caches, one for data and one for instructions. For the Pentium 4, the L1 data cache is 16 Kbytes, using a line size of 64 bytes and a four-way set-associative organization. The Pentium 4 instruction cache is described subsequently.
- The Pentium II also includes an L2 cache that feeds both of the L1 caches. The L2 cache is eight-way set associative with a size of 512 kB and a line size of 128 bytes. An L3 cache was added for the Pentium III and became on-chip with high-end versions of the Pentium 4.

Pentium 4 Block Diagram

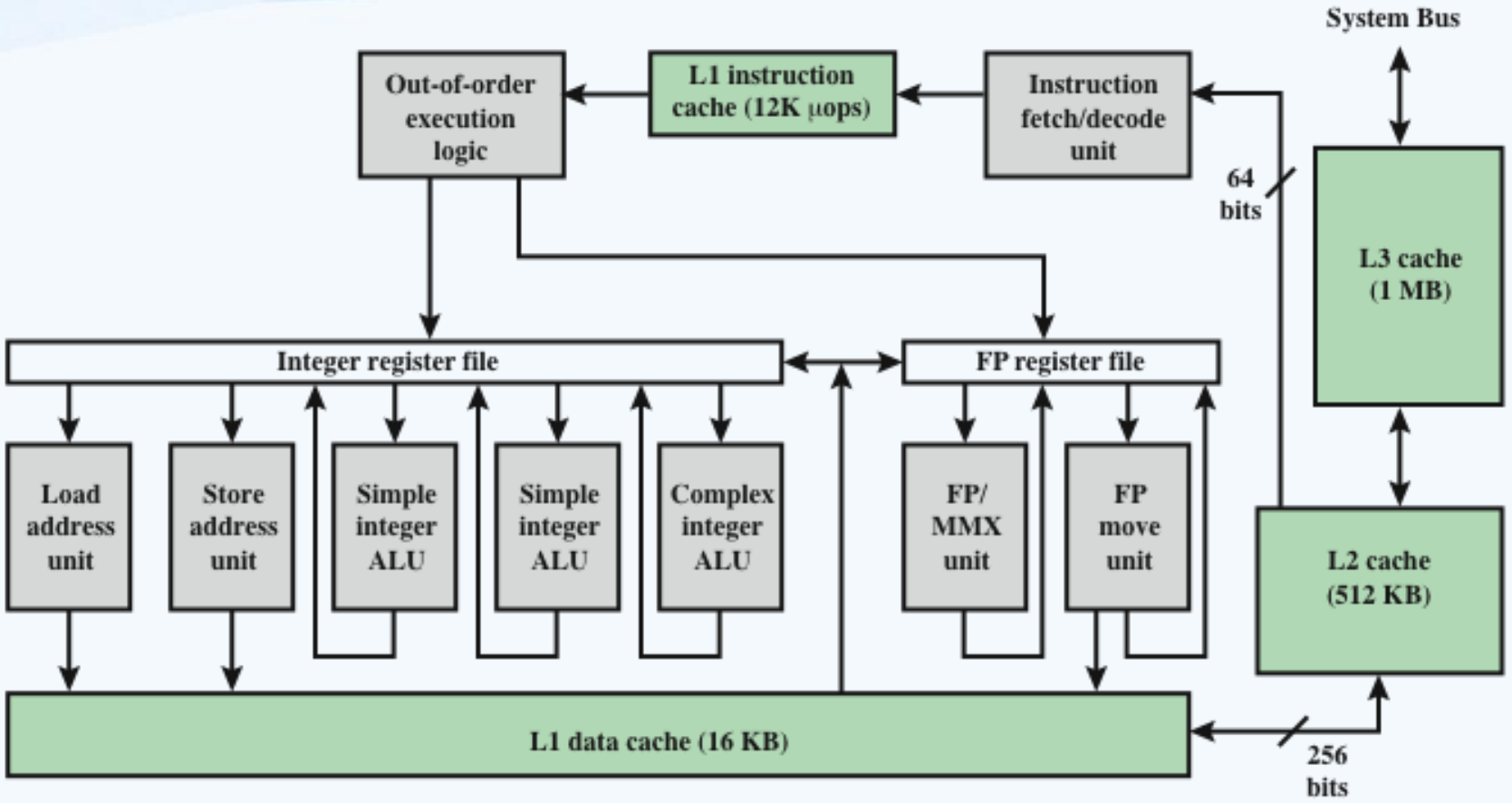


Figure 4.18 Pentium 4 Block Diagram

Pentium 4 Block Diagram

- Figure 4.18(S-101) provides a simplified view of the Pentium 4 organization, highlighting the placement of the three caches. The processor core consists of four major components:
  - Fetch/decode unit: Fetches program instructions in order from the L2 cache, decodes these into a series of micro-operations, and stores the results in the L1 instruction cache.
  - Out-of-order execution logic: Schedules execution of the micro-operations subject to data dependencies and resource availability; thus, micro-operations may be scheduled for execution in a different order than they were fetched from the instruction stream. As time permits, this unit schedules speculative execution of micro-operations that may be required in the future.

Pentium 4 Block Diagram

- Execution units: These units executes micro-operations, fetching the required data from the L1 data cache and temporarily storing results in registers.
- Memory subsystem: This unit includes the L2 and L3 caches and the system bus, which is used to access main memory when the L1 and L2 caches have a cache miss and to access the system I/O resources.

Pentium 4 Block Diagram

- Unlike the organization used in all previous Pentium models, and in most other processors, the Pentium 4 instruction cache sits between the instruction decode logic and the execution core. The reasoning behind this design decision is as follows: As discussed more fully in Chapter 16, the Pentium process decodes, or translates, Pentium machine instructions into simple RISC-like instructions called micro-operations. The use of simple, fixed-length micro-operations enables the use of superscalar pipelining and scheduling techniques that enhance performance. However, the Pentium machine instructions are cumbersome to decode; they have a variable number of bytes and many different options. It turns out that performance is enhanced if this decoding is done independently of the scheduling and pipelining logic. We return to this topic in Chapter 16.
- The data cache employs a write-back policy: Data are written to main memory only when they are removed from the cache and there has been an update. The Pentium 4 processor can be dynamically configured to support write-through caching.

Pentium 4 Cache Operating Modes

Control Bits	Operating Mode		
NW	Cache Fills	Write Throughs	Invalidates
0	Enabled	Enabled	Enabled
0	Disabled	Enabled	Enabled
1	Disabled	Disabled	Disabled

Note: CD = 0; NW = 1 is an invalid combination.

Table 4.5 Pentium 4 Cache Operating Modes

Pentium 4 Cache Operating Modes

- The L1 data cache is controlled by two bits in one of the control registers, labeled the CD (cache disable) and NW (not write-through) bits (Table 4.5). There are also two Pentium 4 instructions that can be used to control the data cache: INVD invalidates (flushes) the internal cache memory and signals the external cache (if any) to invalidate. WBINVD writes back and invalidates internal cache and then writes back and invalidates external cache.
- Both the L2 and L3 caches are eight-way set-associative with a line size of 128 bytes.

ARM Cache Features

Core	Cache Type	Cache Size (kB)	Cache Line Size (words)	Associativity	Location	Write Buffer Size (words)
ARM720T	Unified	8	4	4-way	Logical	8
ARM920T	Split	16/16 D/I	8	64-way	Logical	16
ARM926EJ-S	Split	4-128/4-128 D/I	8	4-way	Logical	16
ARM1022E	Split	16/16 D/I	8	64-way	Logical	16
ARM1026EJ-S	Split	4-128/4-128 D/I	8	4-way	Logical	8
Intel StrongARM	Split	16/16 D/I	4	32-way	Logical	32
Intel Xscale	Split	32/32 D/I	8	32-way	Logical	32
ARM1136-JF-S	Split	4-64/4-64 D/I	8	4-way	Physical	32

Table 4.6 ARM Cache Features

ARM Cache Features

- The ARM cache organization has evolved with the overall architecture of the ARM family, reflecting the relentless(persistent) pursuit of performance that is the driving force for all microprocessor designers.
- Table 4.6 shows this evolution. The ARM7 models used a unified L1 cache, while all subsequent models use a split instruction/data cache. All of the ARM designs use a set-associative cache, with the degree of associativity and the line size varying. ARM cached cores with an MMU use a logical cache for processor families ARM7 through ARM10, including the Intel StongARM and Intel Xscale processors. The ARM11 family uses a physical cache. The distinction between logical and physical cache is discussed earlier in this chapter (Figure 4.7).

ARM Cache and Write Buffer Organization

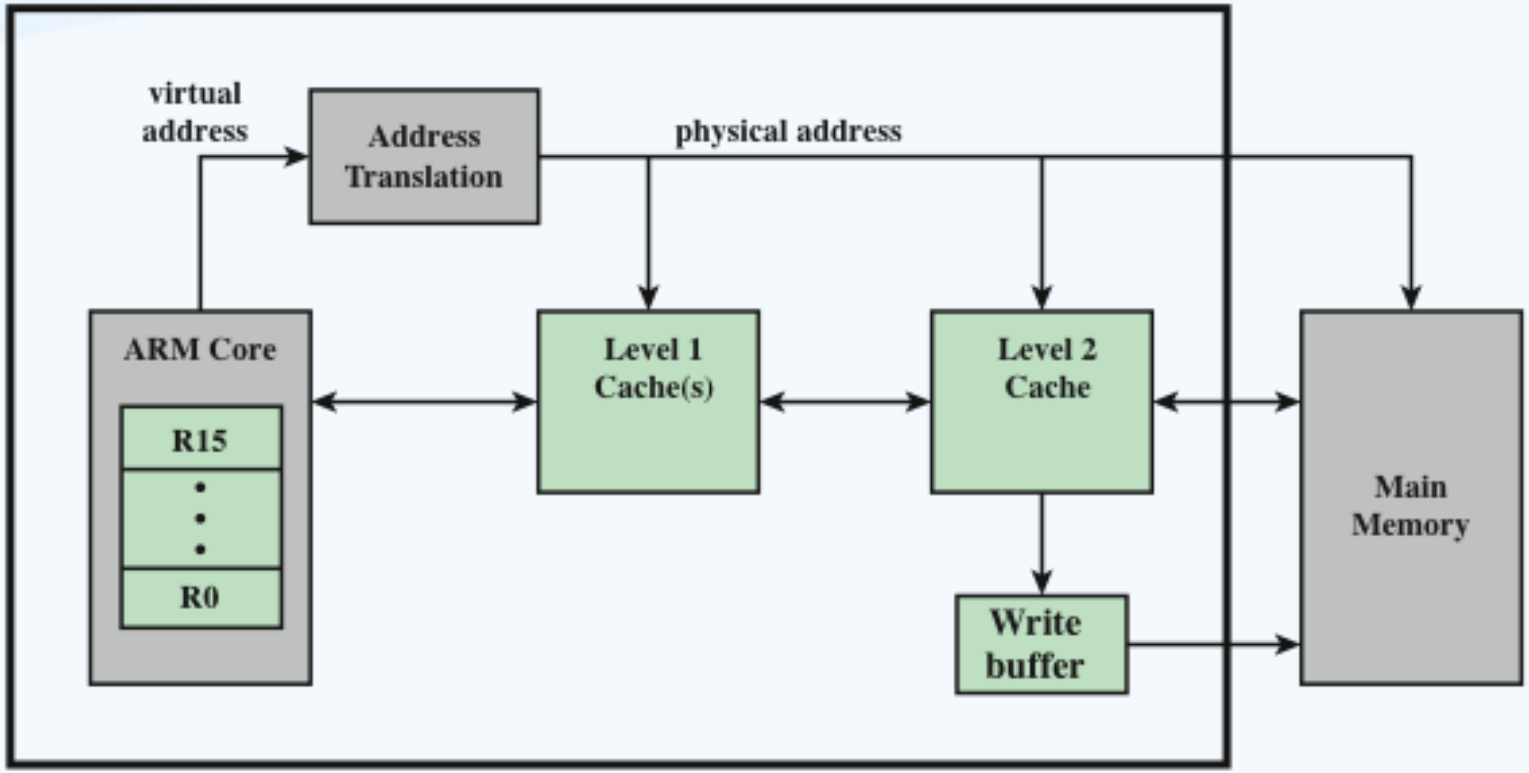


Figure 4.19 ARM Cache and Write Buffer Organization

ARM Cache and Write Buffer Organization

- An interesting feature of the ARM architecture is the use of a small first-in-first- out (FIFO) write buffer to enhance memory write performance. The write buffer is interposed between the cache and main memory and consists of a set of addresses and a set of data words. The write buffer is small compared to the cache, and may hold up to four independent addresses. Typically, the write buffer is enabled for all of main memory, although it may be selectively disabled at the page level. Figure 4.19(S-109), taken from, shows the relationship among the write buffer, cache, and main memory.

ARM Cache and Write Buffer Organization

- The write buffer operates as follows: When the processor performs a write to a bufferable area, the data are placed in the write buffer at processor clock speed and the processor continues execution. A write occurs when data in the cache are written back to main memory. Thus, the data to be written are transferred from the cache to the write buffer. The write buffer then performs the external write in parallel. If, however, the write buffer is full (either because there are already the maximum number of words of data in the buffer or because there is no slot for the new address) then the processor is stalled until there is sufficient space in the buffer. As non-write operations proceed, the write buffer continues to write to main memory until the buffer is completely empty.

- Data written to the write buffer are not available for reading back into the cache until the data have transferred from the write buffer to main memory. This is the principal reason that the write buffer is quite small. Even so, unless there is a high proportion of writes in an executing program, the write buffer improves performance.

Summary – Chapter 4 – Cache Memory

- Characteristics of Memory Systems
  - Location
  - Capacity
  - Unit of transfer
- Memory Hierarchy
  - How much?
  - How fast?
  - How expensive?
- Cache memory principles
- Elements of cache design
  - Cache addresses
  - Cache size
  - Mapping function
  - Replacement algorithms
  - Write policy
  - Line size
  - Number of caches
- Pentium 4 cache organization
- ARM cache organization

Assignment # 3 Question # 1 to 5 (Due next lecture)

4.1 What are the differences among sequential access, direct access, and random access?

4.2 What is the general relationship among access time, memory cost, and capacity?

4.3 How does the principle of locality relate to the use of multiple memory levels?

4.4 What are the differences among direct mapping, associative mapping, and set associative

mapping?

4.5 For a direct-mapped cache, a main memory address is viewed as consisting of three fields. List and define the three fields.

Assignment # 3 Question # 6 to 9 (Due next lecture)

4.6 For an associative cache, a main memory address is viewed as consisting of two fields.

List and define the two fields.

4.7 For a set-associative cache, a main memory address is viewed as consisting of three

fields. List and define the three fields.

4.8 What is the distinction between spatial locality and temporal locality?

4.9 In general, what are the strategies for exploiting spatial locality and temporal locality?

Sample Problems Chapter 4 Assignment # 3 Question # 1 (Due next lecture)

4.1 A certain memory system has a single level of cache, connected to a main memory. The time to access memory at an address that is present in the cache is t_h, whereas the time for the same access when the address is not present in the cache is t_m. Derive an equation for E, the effectiveness of the cache, defined as the ratio t_h / t_a, where t_a is the mean access time for all hits and all misses over the measurement interval.

Sample Problems Chapter 4 Assignment # 3 Question # 2 (Due next lecture)

4.2 Cache-Memory Mapping: Consider a memory of 32 blocks (labeled 0 through 31) and a cache of 8 blocks (labeled 0 through 7). In the questions below, list only correct results.

a. Under direct mapping, which blocks of memory contend for block 2 of the cache?

b. Under 4-way set associativity, to which blocks of cache may element 31 of memory go?

c. In the following sequence of memory block references from the CPU, beginning from an empty cache, on which reference must the cache layout for a direct-mapped and a 4-way set associative cache first differ?

order of reference	1	2	3	4	5	6	7	8
block referenced	0	15	18	5	1	13	15	26

d. Suppose a direct-mapped cache is equipped with a single-block victim cache. Out of the following sequence of memory block references from the CPU (R denotes read (or load) and W denotes write (or store)), on which references is the block retrieved from the victim cache? If no block is ever thus retrieved, so state. Show your reasoning.

order of reference	1	2	3	4	5	6	7	8	9	10	11	12
Type of reference	R	R	W	R	R	W	R	R	W	R	R	W
block referenced	0	15	18	5	1	13	15	26	6	8	15	3

Sample Problems Chapter 4 Assignment # 3 Question # 3 (Due next lecture)

4.3 A system with 350 MHz clock uses a separate data and instruction cache and a unified second-level cache. The first level cache is direct–mapped, write-through, and writes-allocate cache with 8kBytes of data total and 8 Byte blocks, and has a perfect write buffer (never causes and stalls). The first level instruction cache is a direct-mapped cache with 4kBytes of total data and 8 Bytes blocks. The second level cache is a two way set associative, write-back, write-allocate cache with 2Mbytes of total data and 32-Byte blocks.

The 1st level instruction cache has a miss rate of 2%. The first level data cache has a miss rate of 17%. The unified second level cache has a local miss rate of 12%. Assume that 30% of all instructions are data memory accesses; 50% of those are loads and 50% are stores. Assume that 50% of the blocks in the second-level cache are dirty at any time. Assume that there is no optimization for fast reads on L1 or L2 cache miss.

All the first-level cache hits cause no stalls. The second-level hit times is 10 cycles. (That means that the L1 miss penalty, assuming a hit in the L2 cache is, 10 cycles.) Main memory access time is 100 cycles to the first bus width of data; after that, the memory system can deliver consecutive bus widths of data on each following cycle. Outstanding non-consecutive memory requests cannot overlap; an access to one memory location must complete before an access to another location can begin. There is a 128-bit memory to the L2 cache and a 64-bit bus from both the L1 caches to the L2 cache. Assume a perfect TLB for this problem.

a. What percent of all data memory references cause a main memory access (main memory is accessed before the memory request is satisfied)?

b. How many bits are used to index each of the caches? Assume you can use physical addresses for cache.

c. How many cycles can the longest possible data memory access take?

d. What is the average memory access time in cycles (including instruction and data memory references)?

Sample Problems Chapter 4 Assignment # 3 Question # 4 to 6 (Due next lecture)

4.4 What are the differences between a write-allocate and no-write-allocate policy in a cache?

4.5 Can a direct mapped cache sometimes have a higher hit rate than a fully associative cache with an LRU replacement policy (on the same reference pattern and with the same cache size)? If so, give an example. If not, explain why not?

4.6 How does a data cache take advantage of spatial locality?

4.7 What is the advantage of using a logically-indexed physically-tagged L1 cache (as opposed to physically-indexed and physically-tagged)? What constraints does this design put on the size of the L1 cache?

4.8 Briefly describe the data access pattern of an application for which a cache with a random replacement policy performs worse than an LRU replacement policy.

4.9 Briefly describe the data access pattern of an application for which a cache with a random replacement policy performs better than an LRU replacement policy.

Sample Problems Solutions Chapter 3

4.1 Let p be the probability that it is a cache hit. So (1 – p) would be probability for cache miss. Now, $t_a = p \times t_h + (1-p)t_m$. So

$$E = \frac{t_h}{t_a} = \frac{t_h}{p \times t_h + (1-p)t_m} = \frac{1}{p + (1-p)\frac{t_m}{t_h}}$$

Sample Problems Solutions Chapter 3

4.2 **a.** Using the rule x mod 8 for memory address x, we get {2, 10, 18, 26} contending for location 2 in the cache.

b. With 4-way set associativity, there are just two sets in 8 cache blocks, which we will call 0 (containing blocks 0, 1, 2, and 3) and 1 (containing 4, 5, 6, and 7). The mapping of memory to these sets is x mod 2; i.e., even memory blocks go to set 0 and odd memory blocks go to set 1. Memory element 31 may therefore go to {4, 5, 6, 7}.

c. For the direct mapped cache, the cache blocks used for each memory block are shown in the third row of the table below

order of reference	1	2	3	4	5	6	7	8
block referenced	0	15	18	5	1	13	15	26
Cache block	0	7	2	5	1	5	7	2

For the 4-way set associative cache, there is some choice as to where the blocks end up, and we do not explicitly have any policy in place to prefer one location over another or any history with which to apply the policy. Therefore, we can imitate the direct-mapped location until it becomes impossible, as shown below on cycle 5:

order of reference	1	2	3	4	5	6	7	8
block referenced	0	15	18	5	1	13	15	26
Cache block	0	7	2	5	x			

It is required to map an odd-numbered memory block to one of {4, 5, 6, 7}, as shown in part (b).

Sample Problems Solutions Chapter 3

d. In this example, nothing interesting happens beyond compulsory misses until reference 6, before which we have the following occupancy of the cache with memory blocks :

Cache block	0	1	2	3	4	5	6	7	V
Memory block	0	1	18			5		15	

On reference 6, memory block 13 takes the place of memory block 5 in cache block 5 (conflict miss), so memory block 5 goes to the victim cache:

Cache block	0	1	2	3	4	5	6	7	V
Memory block	0	1	18			13		15	5

The 7th reference, to memory block 15, hits in cache. On reference 8, memory block 26 is mapped to cache block 2, displacing memory block 18 to the victim cache and overwriting memory block 5:

Cache block	0	1	2	3	4	5	6	7	V
Memory block	0	1	26			13		15	18

Cycle 9 is a routine compulsory miss. On reference 10, memory block 18 is retrieved on the read miss from the victim cache, changing places with 26:

Cache block	0	1	2	3	4	5	6	7	V
Memory block	0	1	18			13		15	26

On reference 11, when memory block 5 is desired, it must come from memory, putting 13 into the victim cache:

Cache block	0	1	2	3	4	5	6	7	V
Memory block	0	1	18			5		15	13

Finally, on reference 12, memory block 13 is written into cache block 5. This does not reuse the old value of memory block 13 in the victim cache, though it does interchange 5 and 13 in cache block 5 and the victim cache upon completion.

To summarize, only reference 10 retrieves a victim cache block.

Sample Problems Solutions Chapter 3

4.3 **a.** Here a main memory access is a memory store operation. So we will consider both cases i.e. all stores are L1 miss and all stores are not L1 miss.

All stores are not L1 miss = (L1 miss rate) × (L2 miss rate)
= (0.17) × (0.12) = 2.04%
All stores are L1 miss = (% data references that read ops) × (L2 miss rate) + (% data ref that are writes) × (L1 miss rate) × (L2 miss rate)
= (0.50) x (0.12) + (0.50) x (0.17) × (0.12)
= .06 + .042 = .102 = 10.2%

b. Data = 8Kbytes/8= 1024 bytes = 2^{10} = 10 bits
Instruction = 4Kbytes/8 = 512 bytes = 2^9 = 9 bits
L2 = 2M/32 = 64Kbytes = 32K sets = 2^{15} = 15 bits

c. Longest possible memory access will be when L1 miss + L2 miss + write back to main memory
So, total cycles = 1 + 10 + 2X101 = 213 cycles.

d. If you did not treat all stores as L1 miss then
(Avg memory access time)total = (1/1.3)(avg mem access time)inst + (.5/1.3)(avg mem access time)data
Now,
avg mem access time = (L1 hit time) + (L1 miss rate) x [(L2 hit time)+(L2 miss rate) x (mem transfer time)]
(Avg memory access time)inst = 1 + 0.02(10+(0.10) x 1.5 x 101)=1.503
(Avg memory access time)data = 1 + .17(10+(0.10) x 1.5 x 101)= 5.276
(Avg memory access time)total = (1/1.3)1.503 + (.5/1.3)5.276 = 1.156 + 2.02 = 3.18

Sample Problems Solutions Chapter 3

Note: We have to multiply the mean transfer time by 1.5 to account for the write backs in L2 cache. Now, let us treat all stores as L1 misses

(Avg memory access time)total = (1/1.3)(avg mem access time)inst + (.15/1.3)(avg mem access time)load + (.15/1.3)(avg mem access time)store

avg mem access time = (L1 hit time) + (L1 miss rate) x [(L2 hit time)+(L2 miss rate)X(mem transfer time)]

(Avg memory access time)inst = 1 + 0.02(10+(0.10) x 1.5 x 101)=1.503

(Avg memory access time)load = 1 + .17(10+(0.10) x 1.5 x 101)= 5.276

(Avg memory access time)store = 1 + 1(10+(0.10) x 1.5X101)= 26.15

(Avg memory access time)total = (1/1.3)1.503 + (.15/1.3)5.276 + (.15/1.3)26.15 = 1.156 + .608 + 3.01 = 4.774

Note: We have to multiply the mean transfer time by 1.5 to account for the write backs in L2 cache.

4.4 Write-allocate: When a write (store) miss occurs, the missed cache line is brought into the first level cache before actual writing takes place.
No Write-allocate: When a write miss occurs, the memory update bypasses the cache and updates the next level memory hierarchy where there is a hit.

4.5 Imagine a 4-word cache with an access pattern of 0, 1, 2, 3, 4, 0, 1, 2, 3, 4. The directed mapped cache will have a 30% hit rate while the LRU fully associative cache will have a 0% hit rate.

Sample Problems Solutions Chapter 3

4.6 When a word is loaded from main memory, adjacent words are loaded into the cache line. Spatial locality says that these adjacent bytes are likely to be used. A common example is iterating through elements in an array.

4.7 The cache index can be derived from the virtual address without translation. This allows the cache line to be looked up in parallel with the TLB access that will provide the physical cache tag. This helps keep the cache hit time low in the common case where of a TLB hit.

For this to work, the cache index cannot be affected by the virtual to physical address translation. This would happen if the any of the bits from the cache index came from the virtual page number of the virtual address instead of the page offset portion of the address.

4.8 A case where LRU would be expected to outperform random replacement is a randomly-accessed tree structure that is too large to fit in the cache. The LRU policy will do a better job of keeping the root of the tree in the cache and those nodes of the tree are accessed more often.

Sample Problems Solutions Chapter 3

4.9 One pathological case for LRU would be an application that makes multiple sequential passes through a data set that is slightly too large to fit in the cache. The LRU policy will result in a 0% cache hit rate, while under random replacement the hit rate will be high.

4.10 **a.** Can't be determined. *(ptr + 0x11 A538) is in the same set but has a different tag, it may or may not be in another way in that set

b. HIT. *(ptr + 0x13 A588) is in same the line as x (it is in the same set and has the same tag), and since x in present in the cache, it must be also

c. Can't be determined. It has the same tag, but is in a different set, so it may or may not be in the cache.

4.11 EAT = .9 (10) + .1 [.8 (10 + 100) + .2 (10 + 100 + 10000)] = 220ns

OR

EAT = 10 + .1 (100) + .02 (10000) = 220ns

We shall InshaAllah continue…

PART TWO – THE COMPUTER SYSTEM

Chapter 5 – Internal Memory