UNIVERSITY OF RIZAL SYSTEM

Computer Architecture and Organization

Nurturing Tomorrow's Noblest

UNIVERSITY OF RIZAL SYSTEM

Memory System Organization and Architectures

Nurturing Tomorrow's Noblest

Principle Of Locality

One of the most important concepts related to computer systems is **principle of locality**, also referred to as the **locality of reference**.

The **principle of locality** (also called **locality of reference**) is an idea in computers that helps them run faster and more efficiently. It basically means that when a computer is working on something, it tends to use the same information (data or instructions) over and over again, or it uses information that is close to the one it just used.

Two types of locality:

1. Temporal locality: This means that if a computer **uses a piece** of **information** now, it's likely to **need** that **same information again soon**. Imagine doing homework and frequently using the same math formula—if you just used it, you'll probably use it again in a few minutes.

2. Spatial locality: This means that if a computer **uses** a **certain piece** of **information**, it's also **likely** to **use other information stored near it**. It's like reading a book—after you read one page, you're likely to read the next page soon, because they're next to each other.

Computers use these patterns to **store frequently** or **nearby needed information in faster memory (like cache)** so that the **system runs quicker**, without having to go back and forth to slower memory.

An analogy may help illuminate the distinction between these two concepts.

Suppose that Bob is working in an office and spends much of his time dealing with documents in file folders. Thousand of folders are stored in file cabinets in the next room, and for convenience Bob has a file organizer on his desk that can hold a few dozen files. When Bob is working on a file and temporarily is finished, it may be likely that he will **need** to **read** or **write one** of the **documents** in that **file** in the **near future**, so he **keeps it** in his **desk organizer**. This is an example of exploiting **temporal locality**.

Bob also observes that when he retrieves a folder from the filing cabinets, it is likely that in the **near future** he will **need access** to **some** of the **nearby folders** as **well**, so he retrieves the folder he needs plus a few folders on either side at the same time. This is an example of exploiting **spatial locality**.

Characteristics Of Memory Systems

Characteristics of Computer Memory Systems

Location

Internal (e.g., processor registers, cache, main memory)

External (e.g., optical disks, magnetic disks, tapes)

Capacity

Number of words

Number of bytes

Unit of Transfer

Word

Block

Access Method

Sequential

Direct

Random

Associative

Performance

Access time

Cycle time

Transfer rate

Physical Type

Semiconductor

Magnetic

Optical

Magneto-optical

Physical Characteristics

Volatile/nonvolatile

Erasable/nonerasable

Organization

Memory modules

The term **location** in **Table** refers to whether **memory** is **internal** or **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**. The **control unit portion** of the **processor may also require** its **own internal memory**.

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** or **words (1 byte= 8 bits)**. Common **word lengths** are **8, 16,** and **32 bits**. External memory capacity is typically expressed in terms of **bytes**.

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**.

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. For example, the CRAY C90 (an older model CRAY supercomputer) has a **64-bitword length**. The **Intel x86 architecture** has a wide variety of **instruction lengths**, expressed as multiples of bytes, and a word size of **32 bits**.

Addressable units: In **some systems**, the **addressable unit** is the word. 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.

Characteristics Of Memory Systems

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

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**. Tape units 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**.

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

- **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. **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**.

Characteristics Of Memory Systems

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

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.

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

Three performance **parameters** are used:

- **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).**

For **non-random-access memory**

$$T_n = T_A + \frac{n}{R}$$

T_n	=	Average time to read or write n bits
T_A	=	Average access time
n	=	Number of bits
R	=	Transfer rate , in bits per second (bps)

Characteristics Of Memory Systems

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

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** 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. **Semiconductor memory** of this type is **known** as **read-only memory (ROM)**. Of necessity, a practical **nonerasable memory** must also be nonvolatile.

In this context, **organization** refers to the **physical arrangement** of bits to form words.

Memory Hierarchy

The **design constraints** on a **computer's memory** can be **summed** up by **three questions: How much (capacity)? 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**, to achieve **greatest performance**, the memory must be able to keep up with the processor. As the **processor** is **executing instructions**, we would **not want** it to have to **pause waiting** for instructions or operands. 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**.

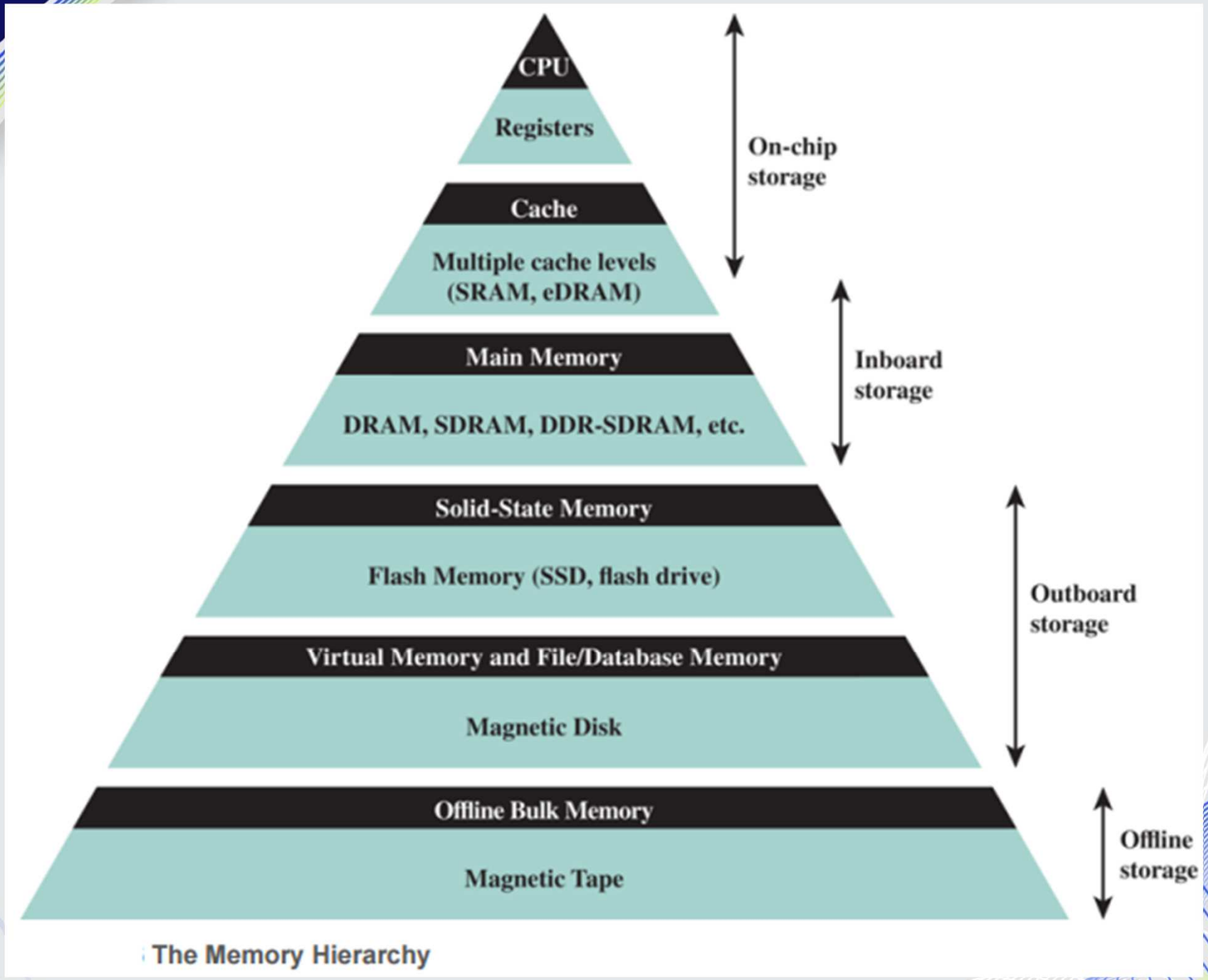
Open ended means that there isn't a clear maximum limit to how much memory is needed. As more memory becomes available, applications will likely evolve to take advantage of it, so the demand for capacity will continuously grow without a specific cap. There's always potential for more memory to be useful, depending on future needs and developments.

A variety of technologies are used to implement **memory systems**, and across this spectrum of technologies, 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 memor technologies** that **provide** for large-capacity memory, **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.

Memory Hierarchy

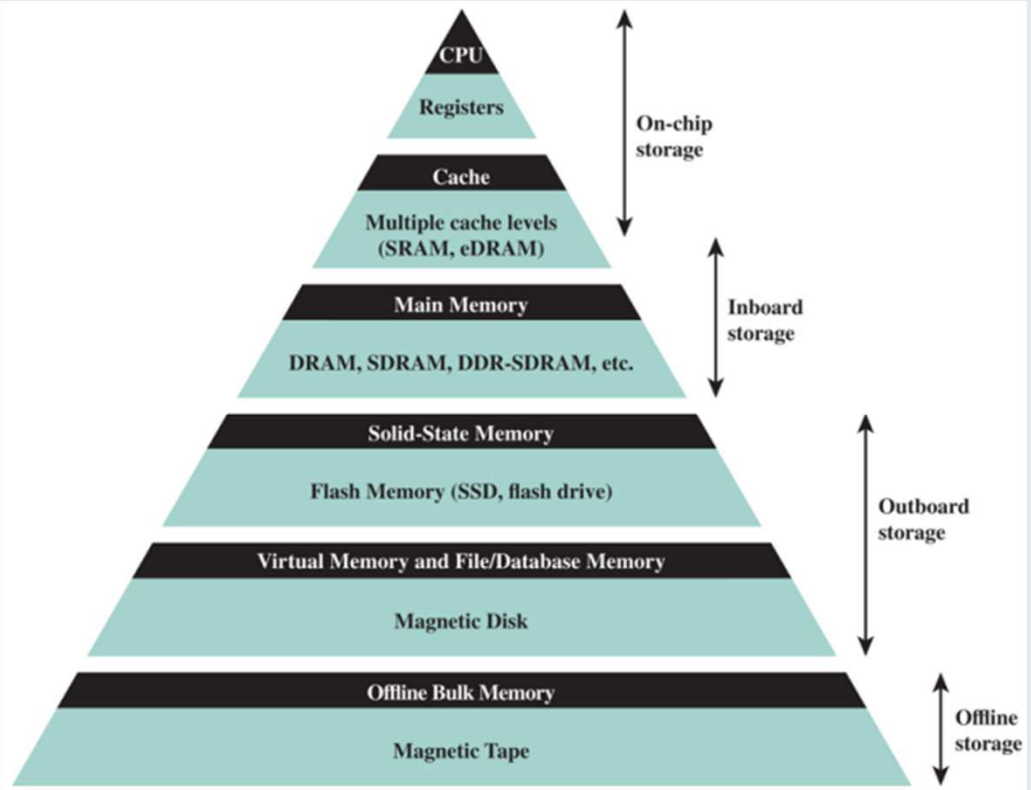


The Memory Hierarchy

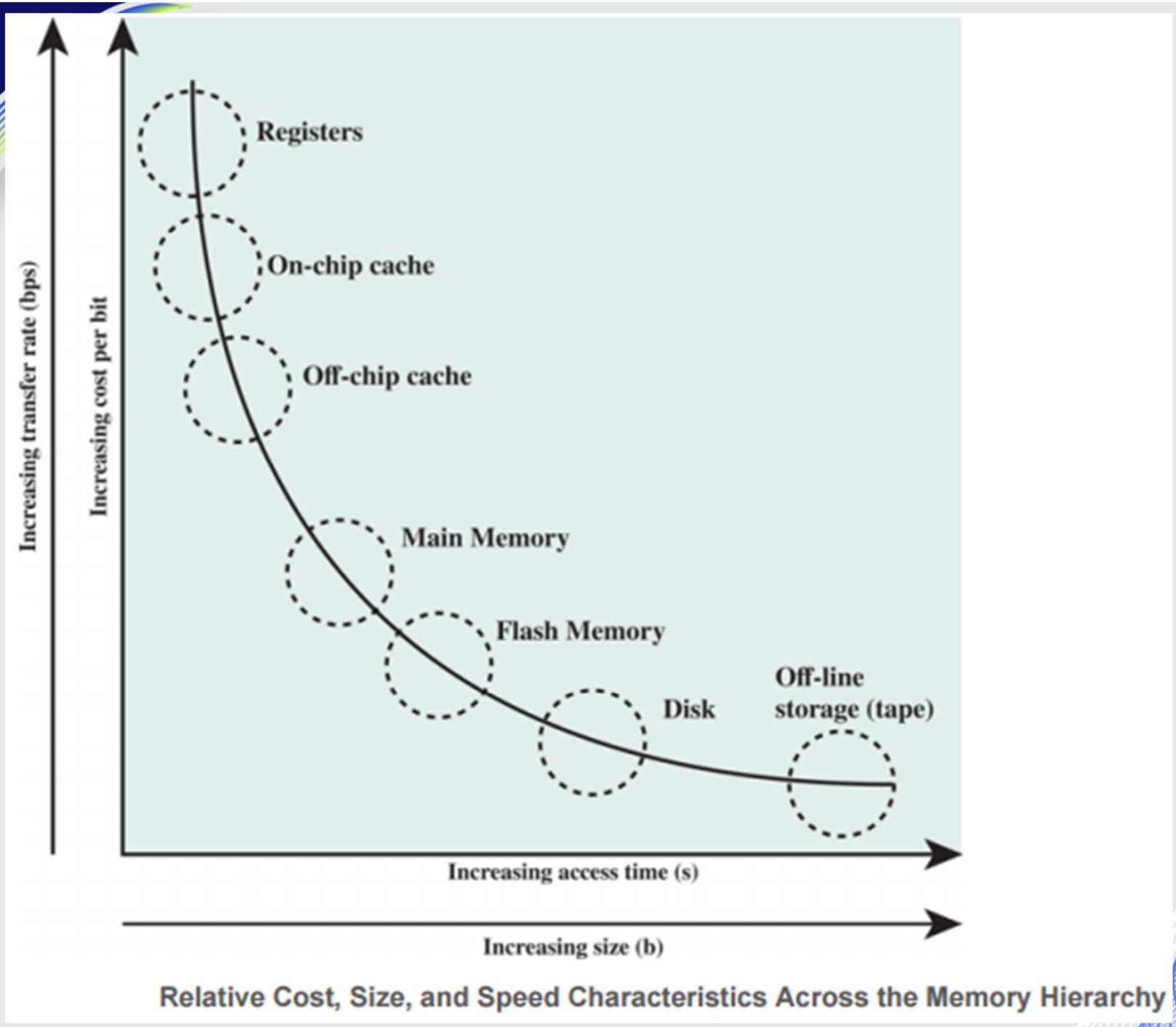
Cost and Performance Characteristics

As one goes down the **hierarchy**, the following occur:

- Decreasing cost per bit
- Increasing capacity
- Increasing access time
- Decreasing frequency of access of the memory by the processor

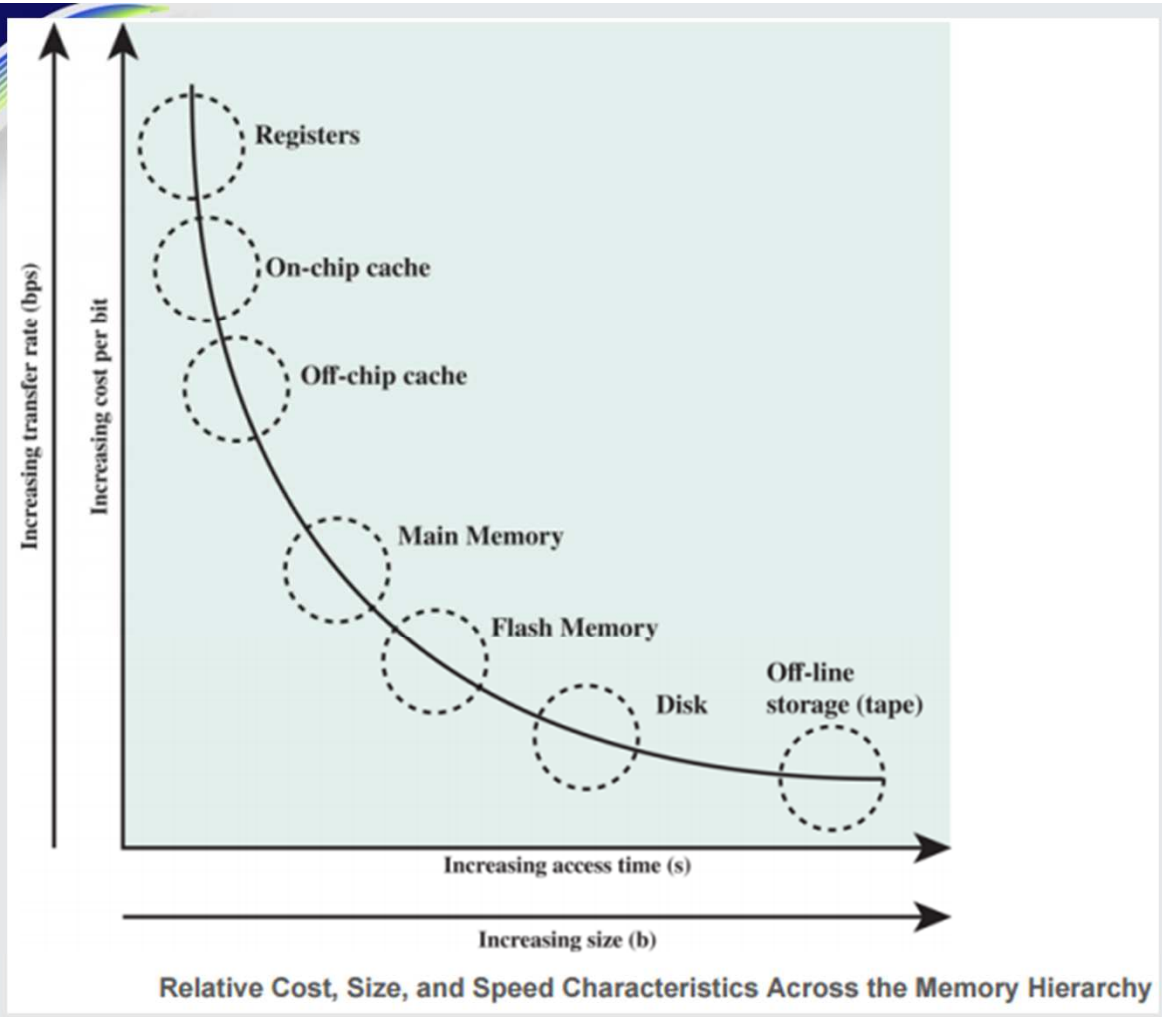


The Memory Hierarchy



Relative Cost, Size, and Speed Characteristics Across the Memory Hierarchy

In the figure (in a general way and not to scale), illustrates **relationships** across the memory hierarchy.



Relative Cost, Size, and Speed Characteristics Across the Memory Hierarchy

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**, which can be **achieved** by **exploiting** the **principle** of **locality**.

supplement – **make addition** to something: to **increase, extend**, or **improve** something by adding something to it

Characteristics of Memory Devices in a Memory Architecture

Memory level	Typical technology	Unit of transfer with next larger level (typical size)	Managed by
Registers	CMOS	Word (32 bits)	Compiler
Cache	Static RAM (SRAM); Embedded dynamic RAM (eDRAM)	Cache block (32 bytes)	Processor hardware
Main memory	DRAM	Virtual memory page (1 kB)	Operating system (OS)
Secondary memory	Magnetic disk	Disk sector (512 bytes)	OS/user
Offline bulk memory	Magnetic tape		OS/User

Main memory is the principal internal memory system of the **computer**. Each location in main memory has a unique address. Main memory is visible to the **programmer**, whereas cache memory is not. The various levels of cache are **controlled** by **hardware** and are **used** for **staging** the **movement** of **data** between **main memory** and **processor registers** to improve performance.

Data are **stored** more permanently on external mass storage devices, of which the most common are **hard disk** and **removable media**, such as removable magnetic disk, tape, and optical storage. External, nonvolatile memory is also referred to as **secondary memory** or **auxiliary memory**. These are **used** to **store** program and data files and are usually **visible** to the **programmer only** in terms of files and records, as opposed to individual bytes or words. Disk is also **used** to **provide** an extension to main memory known as **virtual memory.** Other forms of **secondary memory** include optical disks and flash memory.

Design Principles for a Memory Hierarchy

Three principles guide the **design** of a **memory hierarchy** and the **supporting memory management hardware** and **software**:

1. **Locality**: Locality is the principle that makes effective use of a memory hierarchy possible.

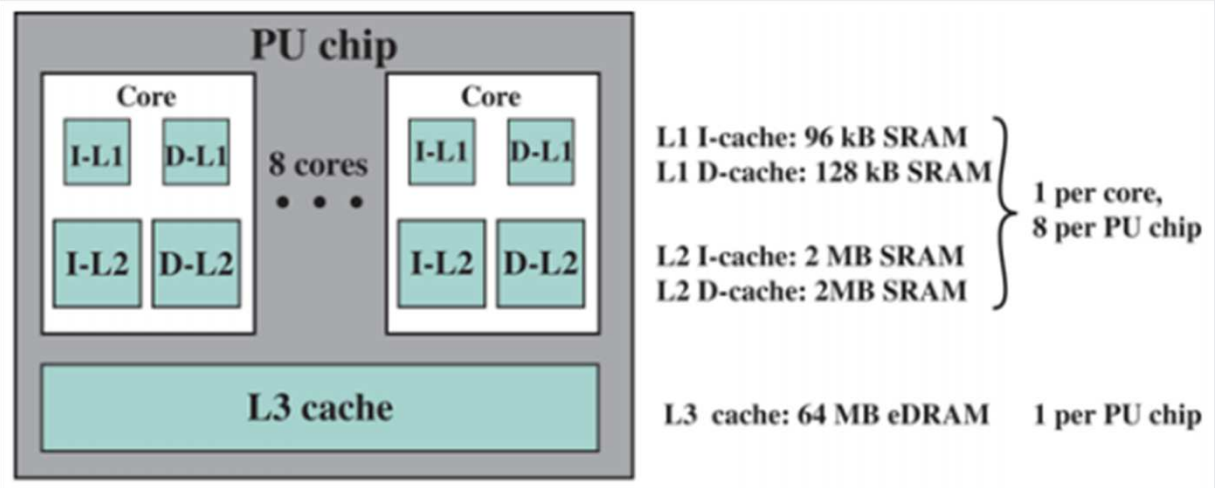
2. **Inclusion**: This principle dictates that all information items are originally stored in level $\boldsymbol{M_n}$, where $\boldsymbol{n}$ is the level most remote from the processor. During the processing, subsets of $\boldsymbol{M_n}$ are copied into$\boldsymbol{M_{n-1}}$, similarity, subsets of $\boldsymbol{M_{n-1}}$ are copied into $\boldsymbol{M_{n-2}}$ and so on.

3. **Coherence**: Copies of the same data unit in adjacent memory levels must be **consistent**. If a **word** is modified in the **cache**, **copies** of that **word** must be updated immediately or eventually at **all higher levels**.

Coherence has **both vertical** and **horizontal** implications, and is required because multiple memories at one level may share the same memory at the next higher level. This leads to **two requirements**:

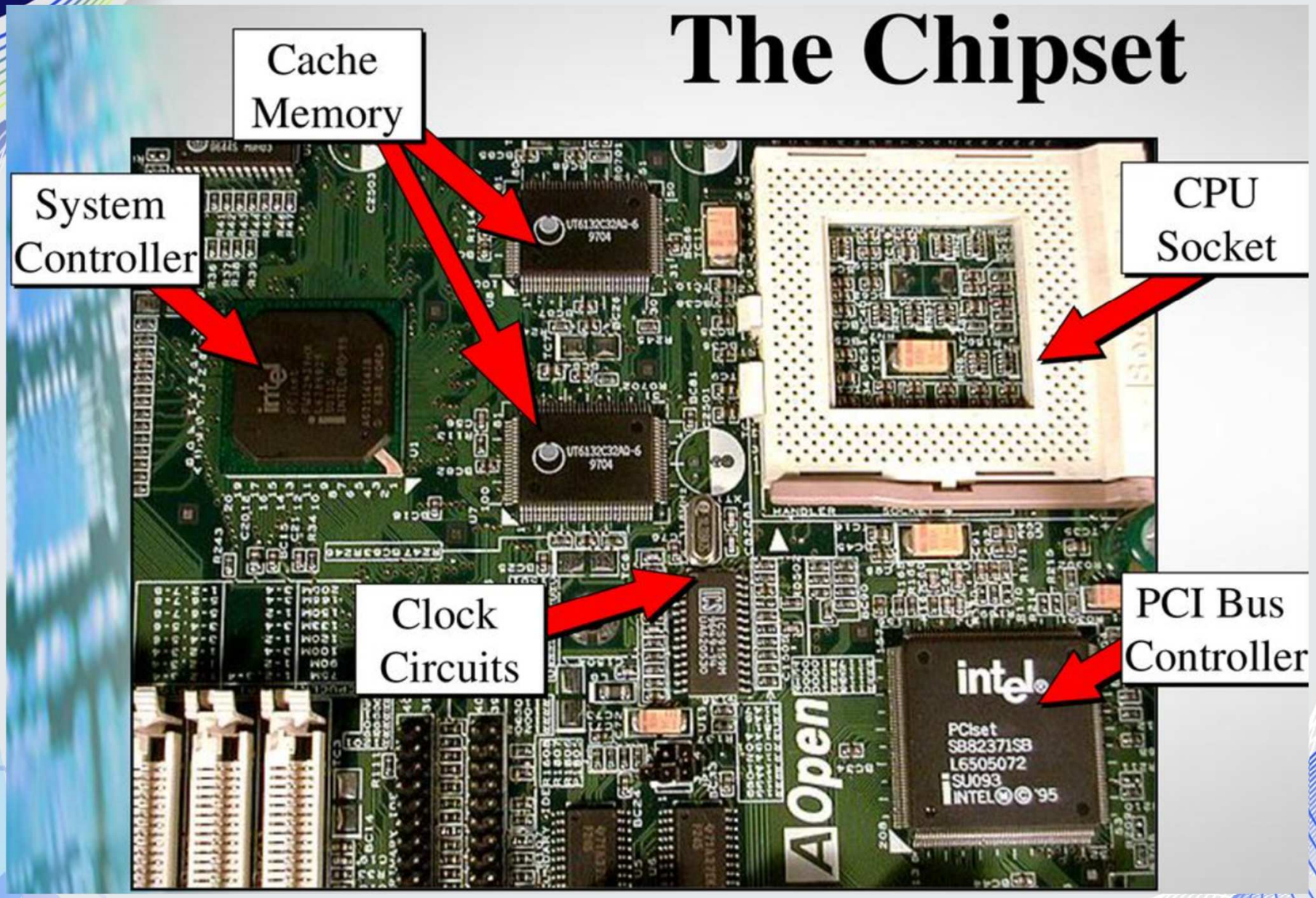
- Vertical coherence: If one core makes a change to a cache block of data at L2, that update must be returned to L3 before another L2 retrieves that block.

- Horizontal coherence: If two L2 caches that share the same L3 cache have copies of the same block of data, then if the block in one L2 cache is updated, the other L2 cache must be alerted that its copy is obsolete.

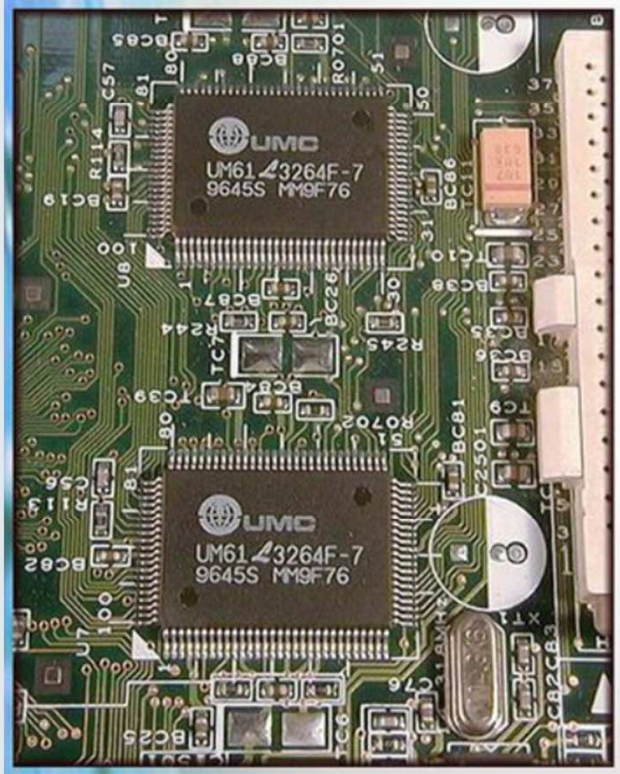


URS

The Chipset

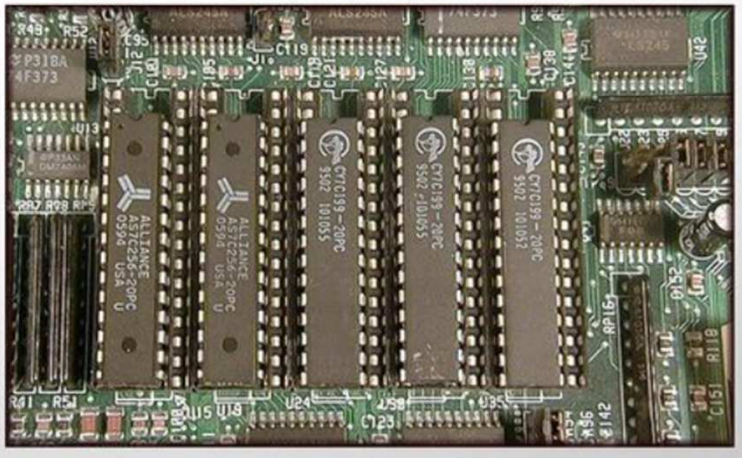


Cache Memory



Pentium I Cache

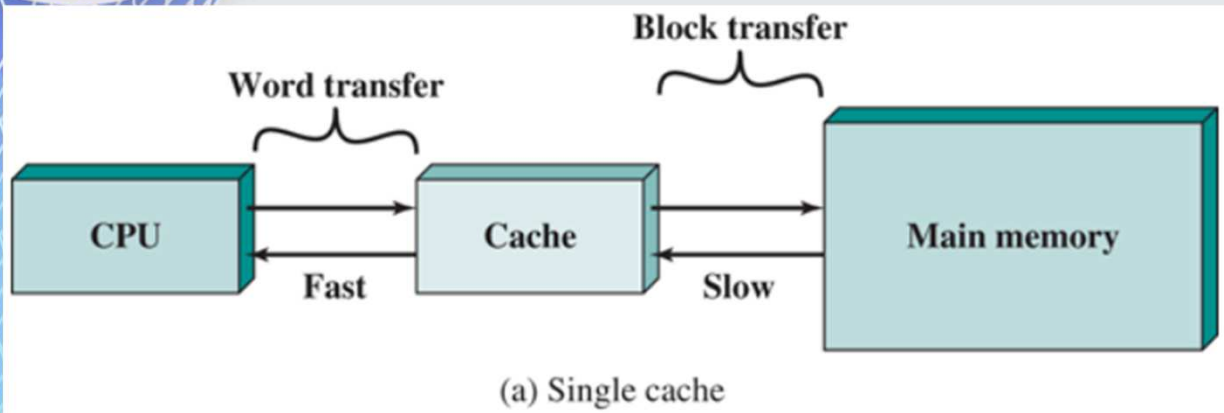
Pentium II/III Cache



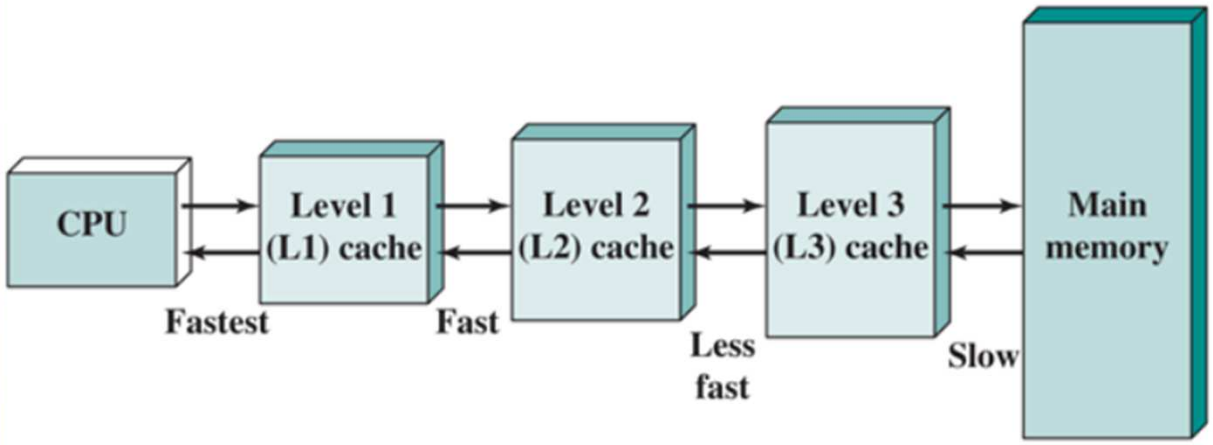
486 Cache

Cache Memory Principles

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 (a)**.



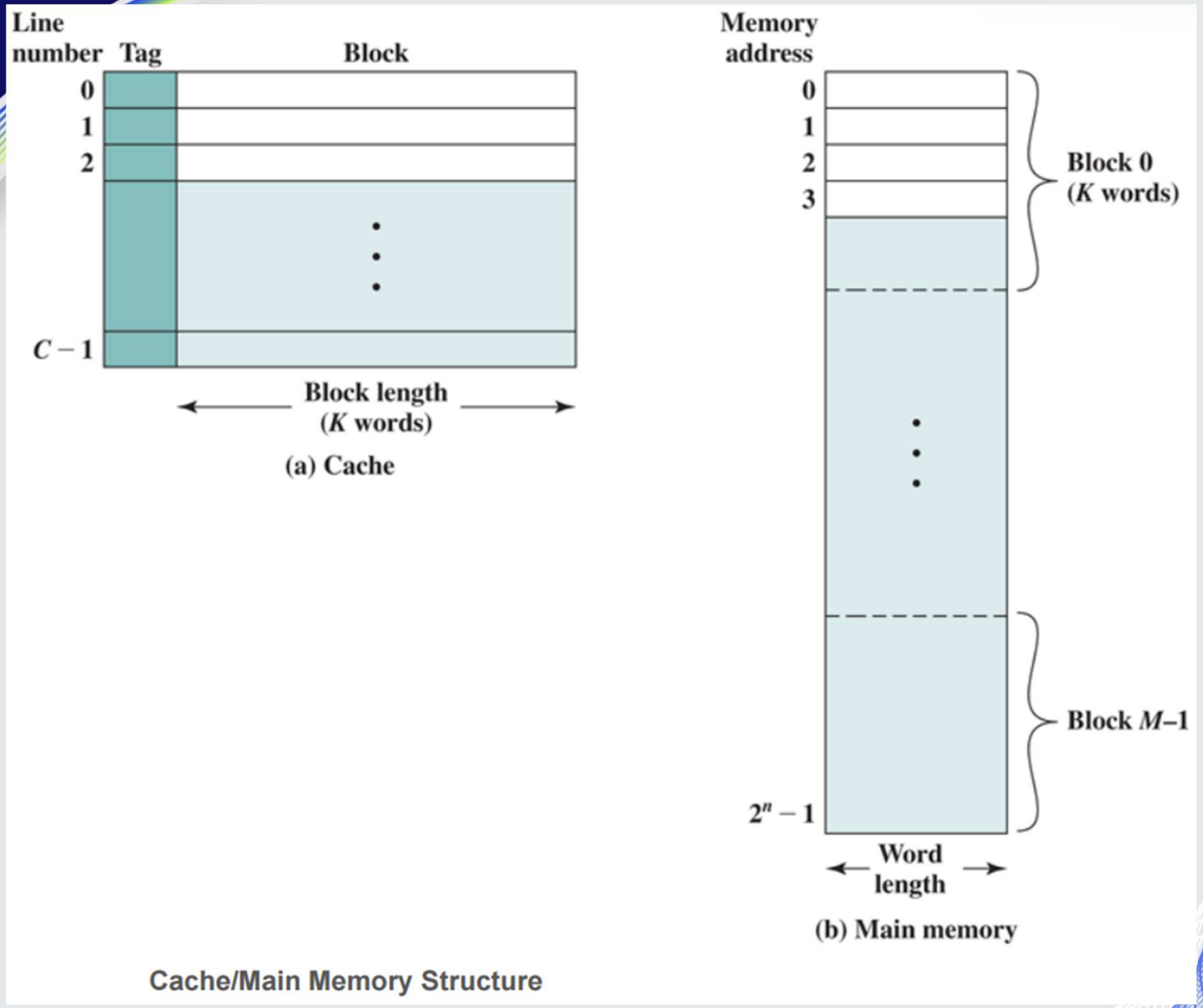
(a) Single cache



(b) Three-level cache organization

Figure (b) 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 and Main Memory



Cache/Main Memory Structure

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

Several **terms** are introduced:

- **Block:** The minimum unit of transfer **between cache** and **main memory**. In most of the literature, the term **block** refers both to the unit of data transferred and to the physical location in main memory or cache.
- **Frame:** To distinguish between the data transferred and the chunk of physical memory, the term **frame** or **block frame**, is sometimes used with **reference** to **caches**. Some texts and some literature use the term with **reference** to the **cache** and some with **reference** to **main memory**.
- **Line**: A portion of cache memory capable of holding one block, so-called because it is usually drawn as a horizontal object (i.e., all bytes of the line are typically drawn in one row). A line **also includes** control information.
- **Tag:** A portion of a cache line that is **used** for addressing purposes. A **cache line** may also include other control bits.

The **length** of a **line, not including tag** and **control bits**, is the **line size**. That is, the term line size refers to the number of data bytes or block size, contained in a line. They may be as small as 32 bits, with each "word" being a single byte.

If a **word** in a **block** of **memory** is **read**, that **block** is **transferred** to **one** of the **lines** of the **cache**. Because there are **more blocks than lines**, an **individual line** cannot be **uniquely** and **permanently dedicated** to a **particular block**. Thus, **each line includes** a **tag** that **identifies** which **particular block** is **currently being stored**. The **tag** is usually a **portion** of the **main memory address.**

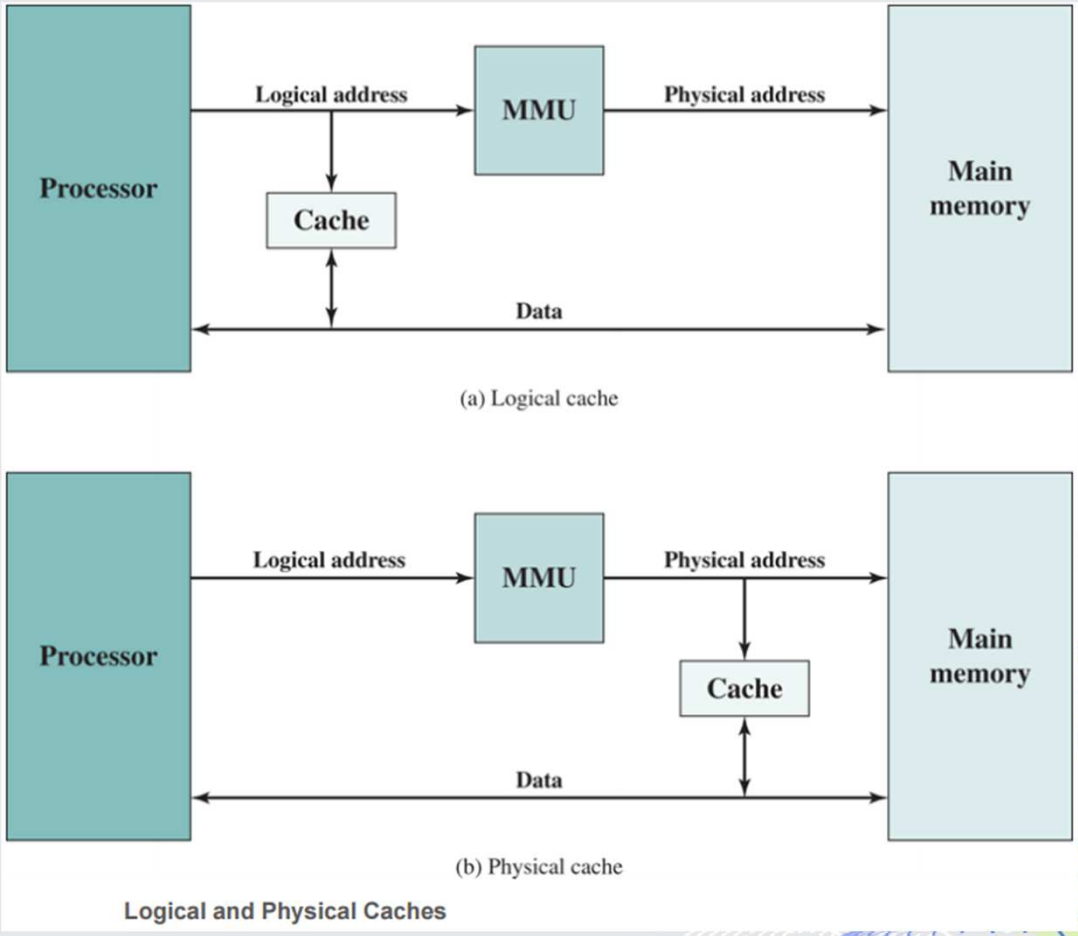
Elements of Cache Design

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 lists **key elements**.

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	

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.**

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**. 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**.



Logical and Physical Caches

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** 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 Size

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 **a 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 lists the cache sizes of some current and past processors.

Cache Sizes of Some Processors

Processor	Type	Year of Introduction	L1 Cachea	L2 cache	L3 Cache
IBM 360/85	Mainframe	1968	16 to 32 kB	—	—
PDP-11/70	Minicomputer	1975	1 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 620	PC	1996	32 kB/32 kB	—	—
IBM S/390 G6	Mainframe	1999	256 kB	8 MB	—
Pentium 4	PC/server	2000	8 kB/8 kB	256 kB	—
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 × 32kB / 32kB	6 × 1.5MB	12 MB
IBM zEnterprise 196	Mainframe/Server	2011	24 × 64kB / 128kB	24 × 1.5MB	24 MB L3 192 MB L4
IBM z13	Mainframe/server	2015	24 × 96kB / 128kB	24 × 2MB / 2MB	64 MB L3 480 MB L4
Intel Core i0-7900X	Workstation/server	2017	8 × 32kB / 32kB	8 × 1MB	14 MB

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

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	

The **choice** of the **mapping function** dictates how the cache is logically organized. **Three techniques** can be used: **direct, associative,** and **set-associative**.

Table provides a summary of key characteristics of the three approaches.

Cache Access Methods

Method	Organization	Mapping of Main Memory Blocks to Cache	Access using Main Memory Address
Direct Mapped	Sequence of *m* lines	Each block of main memory maps to one unique line of cache.	*Line* portion of address used to access cache line; *Tag* portion used to check for hit on that line.
Fully Associative	Sequence of *m* lines	Each block of main memory can map to any line of cache.	*Tag* portion of address used to check every line for hit on that line.
Set Associative	Sequence of *m* lines organized as *v* sets of *k* lines each $(m = v \times k)$	Each block of main memory maps to one unique cache set.	*Line* portion of address used to access cache set; *Tag* portion used to check every line in that set for hit on that line.

Example:

For all **three cases**, the example includes the following elements:

- The **cache** can hold **64 kB**.

- **Data** are transferred between **main memory** and the **cache** in **blocks** of **4 bytes each**. This means that the cache is **organized** as $\mathbf{16}\boldsymbol{k} = \mathbf{2}^{\mathbf{14}}$ **lines** of **4 bytes each**.

- The **main memory** consists of **16 MB**, with each byte **directly addressable** by a **24-bit address** $(\mathbf{2}^{\mathbf{24}} = \mathbf{16}\boldsymbol{M})$. Thus, for mapping purposes, we can consider **main memory** to consist of **4M blocks** of **4 bytes each**.

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**.

The **cache** consists of **number sets**, each of which **consists** of a **number** of **lines.**

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**.

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**.

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.

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	

Write Policy

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**.

There is another dimension to the **write policy** when a **miss occurs** at a cache level. There are **two alternatives** in the event of a write miss:

- **Write Allocate:** The block containing the word to be **written** is **fetched from main memory** (or **next level cache**) **into** the **cache** and the **processor** proceeds with the **write cycle**.

- **No Write Allocate:** The block containing the word to be written is **modified** in the **main memory** and not **loaded** into the **cache**.

Line Size

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.
- 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.

Number of Caches

When caches were originally introduced, the typical system had a single cache. More recently, the use of multiple caches has become the **norm**. Two aspects of this design issue concern the **number** of **levels** of **caches** and the use of unified versus split caches.

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**.

Most contemporary designs include both on-chip and external caches. The simplest such organization is known as a **two-level cache**, with the internal 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.

Two features of contemporary cache design for multilevel caches area: 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.

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.**

Unified Versus Split Cache

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 **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**.

Inclusion Policy

The **inclusive policy** dictates that a piece of data in one cache is guaranteed to be also found in all lower levels of caches. The advantage of the **inclusive policy** is that it **simplifies searching** for **data** when there are **multiple processors** in the computing system. For example, if one processor wants to know whether another processor has the data it needs, it does not **need** to **search all levels** of **caches** of that other processor but only the lowest-level cache.

The **exclusive policy** dictates that a piece of data in one cache is guaranteed not to be found in all lower levels of caches. The advantage of the exclusive policy is that it does not waste cache capacity since it does not **store multiple copies** of the **same data** in **all** of the **caches**. The **disadvantage** is the **need** to **search multiple cache levels** when invalidating or updating a block. To minimize the search time, the higher-level tag sets are typically duplicated at the lowest cache level to centralize searching.

With the **noninclusive policy**, a piece of data in one cache may or may not be found in lower levels of caches.

This can be contrasted with the other two policies with the following examples. Suppose that the **L2 line size** is a **multiple** of the **L1 line size**. For the inclusive policy, if a block is **evicted from** the L2 cache, the corresponding **multiple blocks** will be **evicted** from the L1 cache. In contrast, with a noninclusive policy, the L1 cache may **retain** portions of a **block** recently **evicted** from the **L2 cache**.

For the same difference in block size, if a portion of a block is **promoted** from the L2 cache **to** the L1 cache, the exclusive policy requires the entire L2 block be **evicted**. In contrast, the noninclusive policy does not **require** this **eviction**. As with the exclusive policy, a noninclusive policy will generally **maintain** all higher-level cache sets at the lowest cache level.


ຂໍຂອບໃຈທ່ານ
សូមអរគុណអ្នក
ધન્યવાદ
ви благодариме
شكرا
təşəkkür
köszönöm
gràcies
спасибо
gratias agimus tibi
谢谢
kiitos
grazas
nirringrazzjak
dank je
di ou mèsi
dankon
danke
gracias
tack
aitäh
thank you
dank u paldies
cảm ơn bạn
dankie
eskerrik asko
ありがとう
bakka þér
mochchakkeram
teşekkür ederim
σας ευχαριστώ
ngiyabonga
grazie
merci
благодаря
תודה
આભાર
дзякуй
go raibh maith agat
감사합니다
dziękuję
děkuji
hvala
ačiū
धन्यवाद
sukriya
terima kasih
takk
хвала
kop khun krap
obrigado
감사합니다
(gamsahamnida)
buíochas a ghabháil leat
diolch i chi
მადლობა
salamat
d'akujem
falemnderit
multumesc
তোমাকে ধন্যবাদ
շնորհակալություն

UNIVERSITY OF RIZAL SYSTEM

Management System
ISO 9001:2015

TÜVRheinland CERTIFIED

www.tuv.com
ID 9108653929

Thank you!

Nurturing Tomorrow's Noblest

www.facebook.com/UniversityofRizalSystem

University of Rizal System - Official

www.urs.edu.ph