

Memory organization

Hui Chen ^a

^aCUNY Brooklyn College, Brooklyn, NY, USA

December 5, 2023

Outline

- 1 Lesson Objectives
- 2 Type of Memories
- 3 Memory Hierarchy
- 4 Cache Memory
 - Direct Mapped Cache
 - Fully Associative Cache
 - Cache Replacement Policy
 - Effective Access Time
- 5 Virtual Memory

Acknowledgement

The content of most slides come from the authors of the textbook:

Null, Linda, & Lobur, Julia (2018). The essentials of computer organization and architecture (5th ed.). Jones & Bartlett Learning.

Table of Contents

- 1 Lesson Objectives
- 2 Type of Memories
- 3 Memory Hierarchy
- 4 Cache Memory
 - Direct Mapped Cache
 - Fully Associative Cache
 - Cache Replacement Policy
 - Effective Access Time
- 5 Virtual Memory

Lesson Objectives

Students are expected to be able to

1. Master the concepts of hierarchical memory organization.
2. Understand how each level of memory contributes to system performance, and how the performance is measured.
3. Master the concepts behind cache memory, virtual memory, memory segmentation, paging, and address translation.

Table of Contents

- 1 Lesson Objectives
- 2 Type of Memories**
- 3 Memory Hierarchy
- 4 Cache Memory
 - Direct Mapped Cache
 - Fully Associative Cache
 - Cache Replacement Policy
 - Effective Access Time
- 5 Virtual Memory

Types of Memories

There are two kinds of *main memory*:

- ▶ Random Access Memory (RAM)
- ▶ Read-Only-Memory (ROM).

RAM

There are two types of RAM:

- ▶ Dynamic RAM (DRAM)
- ▶ Static RAM (SRAM).

DRAM

Consists of capacitors that slowly leak their charge over time.

- ▶ Thus, they must be refreshed every few milliseconds to prevent data loss.
- ▶ However, DRAM is inexpensive owing to its simple design.

SRAM

Consists of circuits similar to the D flip-flop that we studied before.

- ▶ SRAM is very fast memory, and it doesn't need to be refreshed like DRAM does.
- ▶ It is used to build cache memory (discuss later).

ROM

To store permanent, or semi-permanent data that persists even while the system is turned off.

- ▶ ROM also does not need to be refreshed, either, and
- ▶ It needs very little charge to retain its memory.

Table of Contents

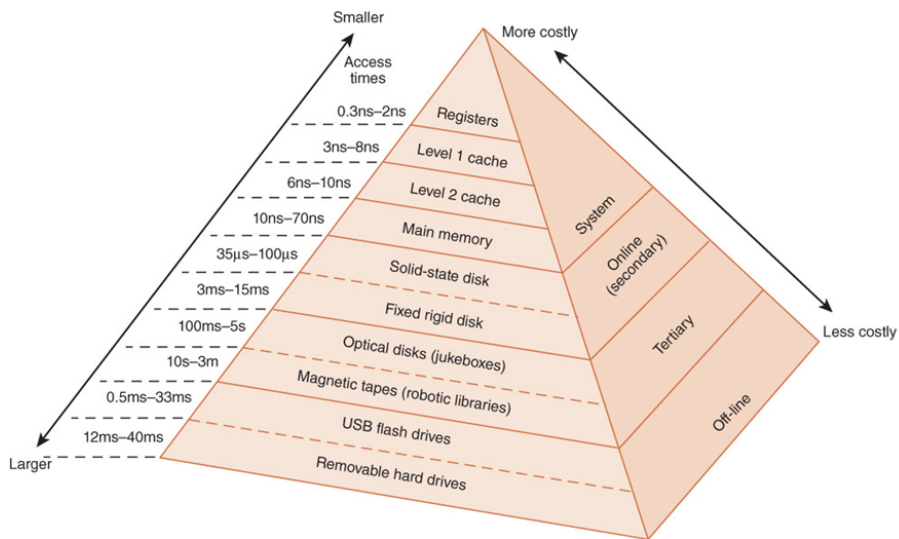
- 1 Lesson Objectives
- 2 Type of Memories
- 3 Memory Hierarchy**
- 4 Cache Memory
 - Direct Mapped Cache
 - Fully Associative Cache
 - Cache Replacement Policy
 - Effective Access Time
- 5 Virtual Memory

Implication: Memory Hierarchy

Generally speaking, faster memory is more expensive than slower memory, which is a “constant”.

- ▶ To provide the best performance at the lowest cost, memory is organized in a *hierarchical* fashion.
- ▶ Small, fast storage elements are kept in the CPU, larger, slower main memory is accessed through the data bus.
- ▶ Larger, (almost) permanent storage in the form of disk and tape drives is still further from the CPU.

Illustrating Memory Hierarchy



From Register to Virtual Memory

We are most interested in the memory hierarchy that involves registers, cache, main memory, and virtual memory.

- ▶ Registers are storage locations available on the processor itself.
- ▶ Virtual memory is typically implemented using a backing store (e.g., hard disk drive); it extends the address space from RAM to the hard drive.
- ▶ Virtual memory provides more space: Cache memory provides speed.

Operations

1. To access a particular piece of data, the CPU first sends a request to its nearest memory, usually cache.
2. If the data is not in cache, then main memory is queried. If the data is not in main memory, then the request goes to disk.
3. Once the data is located, then the data and a number of its nearby data elements are fetched into cache memory.

Operation Scenarios

This leads us to some scenarios.

- ▶ A hit is when data is found at a given memory level.
- ▶ A miss is when it is not found.
- ▶ The hit rate, or the hit ratio is the percentage of time data is found at a given memory level.
- ▶ The miss rate, or the miss ratio is the percentage of time it is not.
miss rate = $1 - \text{hit rate}$.
- ▶ The hit time is the time required to access data at a given memory level.
- ▶ The miss penalty is the time required to process a miss, including the time that it takes to replace a block of memory plus the time it takes to deliver the data to the processor.

Examples and analysis ...

Cache Hit and Miss: Simplified Scenario

Consider the simplified scenario where a system has a cache for the main memory:

- ▶ hit time for the main memory is 10 ns;
- ▶ hit time for the cache is 1 ns; and
- ▶ ignore anything else

Scenario 1: what's the average latency when the hit ratio is 10%?

$$\bar{t}_1 = 10\% \times 1 + (1 - 10\%) \times (1 + 10) = 0.1 + 0.9 * 11 = 0.1 + 9.9 = 10.0$$

Scenario 2. What's the average latency when the hit ratio is 90%

$$\bar{t}_2 = 90\% \times 1 + (1 - 90\%) \times (1 + 10) = 0.9 + 0.1 * 11 = 0.9 + 1.1 = 2.0$$

We shall discuss these again.

Principle of Locality

Meaning: Once a byte is accessed, it's likely that a nearby data element will be needed soon. There are three forms of locality:

- ▶ Temporal locality: Recently-accessed data elements tend to be accessed again.
- ▶ Spatial locality: Accesses tend to cluster.
- ▶ Sequential locality: Instructions tend to be accessed sequentially (a special case of spatial locality).

To take advantage of spatial locality, cache memory is organized in blocks.

Principle of Locality: Example

Assuming all variables are allocated in the main memory, can you identify access locality in the code snippet below:

```
double sum = 0;
for (int i=0; i<SIZE; i++) {
    sum = sum + data[i];
}
```

Table of Contents

- 1 Lesson Objectives
- 2 Type of Memories
- 3 Memory Hierarchy
- 4 Cache Memory**
 - Direct Mapped Cache
 - Fully Associative Cache
 - Cache Replacement Policy
 - Effective Access Time
- 5 Virtual Memory

Cache Memory

- ▶ It is to speed up accesses by storing recently used data closer to the CPU, instead of storing it in main memory.
- ▶ Although cache is much smaller than main memory, its access time is a fraction of that of main memory.
- ▶ Unlike main memory, which is accessed by address, cache is typically accessed by content; hence, it is often called content addressable memory.
 - ▶ Because of this, a single large cache memory isn't always desirable—it takes longer to search.

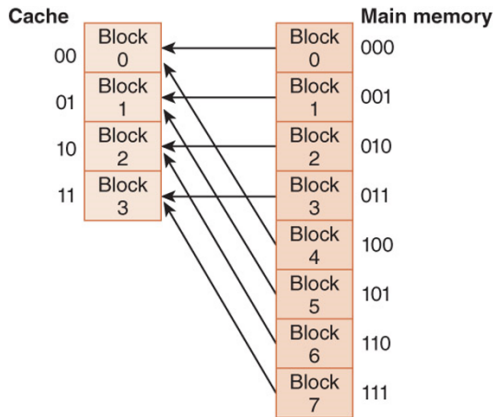
Direct Mapped Cache

The simplest cache mapping scheme is direct mapped cache.

- ▶ In a direct mapped cache consisting of N blocks of cache, block X of main memory maps to cache block $Y = X \bmod N$.
- ▶ Thus, if we have 10 blocks of cache, block 7 of cache may hold blocks 7, 17, 27, 37, ... of main memory.

Q: why cache are organized in blocks?

Example: Direct Mapping of Main Memory Blocks to Cache Blocks



(a) 8 memory blocks to 4 cache blocks

Main Memory Address Organization

To perform direct mapping, the binary main memory address is partitioned into the fields shown below.

Tag	Block	Offset
bits in main memory address		

- ▶ The offset field uniquely identifies an address within a specific block.
- ▶ The block field selects a unique block of cache.
- ▶ The tag field is whatever is left over.

The sizes of these fields are determined by characteristics of both memory and cache. (See examples)

Main Memory Address Organization: Example 1

Consider a byte-addressable main memory consisting of 4 blocks, and a cache with 2 blocks, where each block is 4 bytes. Using Tag, Block, and Offset, illustrate how main memory maps to cache. Particularly, which cache block is mapped to main memory address A ?

Main Memory Address Organization: Example 1

Consider a byte-addressable main memory consisting of 4 blocks, and a cache with 2 blocks, where each block is 4 bytes.

- ▶ Using the tag, block, and offset fields, we can represent the mapping:
 - ▶ We need 4 bits for main memory address ($4 \times 4 = 16 = 2^4$)
 - ▶ Each block is 4 bytes, so the offset field must contain 2 bits ($2^2 = 4$);
 - ▶ There are 2 blocks in cache, so the block field must contain 1 bit ($2^1 = 2$);
 - ▶ This leaves 1 bit for the tag.

Tag	Block	Offset
1 bit	1 bit	2 bits
4 bits in main memory address		

Main Memory Address Organization: Example 1

Suppose we need to access main memory address 3_{16} (0011_2 in binary). We can compute the mapping from the address:

$$\text{Memory Block} = 3/4 = 0$$

$$\text{Cache Block} = \text{Memory Block} \bmod 2 = 0 \bmod 2 = 0$$

$$\text{Tag} = \text{Memory Block}/2 = 0/2 = 0$$

However, we can conveniently look them up from the address:

Tag	Block	Offset
0	0	11
4 bits in main memory address		

Thus, the main memory address 0011_2 maps to cache block 0 with tag 0.

Main Memory Address Organization: Example 1

Suppose we need to access main memory address $A_{16} = 10_{10}$ (1010_2 in binary). We can compute the mapping from the address:

$$\text{Memory Block} = 10/4 = 2$$

$$\text{Cache Block} = \text{Memory Block} \bmod 2 = 2 \bmod 2 = 0$$

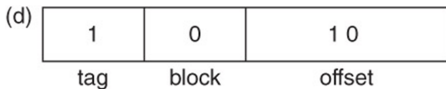
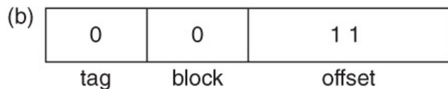
$$\text{Tag} = \text{Memory Block}/2 = 2/2 = 1$$

However, we can conveniently look them up from the address:

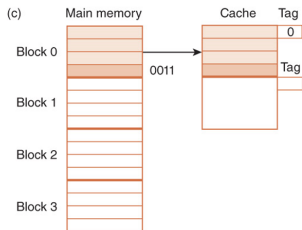
Tag	Block	Offset
1	0	11
4 bits in main memory address		

Thus, the main memory address 0011_2 maps to cache block 0 with tag 1.

Main Memory Address Organization: Example 1

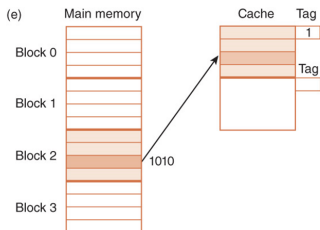


The address 0011 partitioned into fields



Mapping of block containing
Address 0011 = 0x3

The address 1010 partitioned into fields



Mapping of Block Containing
Address 1010 = 0xA

Main Memory Address Organization: Example 2

Assume a byte-addressable memory consists of 2^{14} bytes, cache has $16 = 2^4$ blocks, and each block has $8 = 2^3$ bytes. How does the main memory map to the cache?

Main Memory Address Organization: Example 2

Assume a byte-addressable memory consists of 2^{14} bytes, cache has $16 = 2^4$ blocks, and each block has $8 = 2^3$ bytes.

- ▶ The number of memory blocks is $2^{14}/2^3 = 2^{11}$
- ▶ Using the tag, block, and offset fields, we can represent the mapping:
 - ▶ We need 14 bits for main memory address (2^{14} bytes memory)
 - ▶ Each block is 8 bytes, so the offset field must contain 3 bits ($2^3 = 8$);
 - ▶ There are 16 blocks in cache, so the block field must contain 4 bit ($2^4 = 16$);
 - ▶ This leaves $14 - 3 - 4 = 7$ bits for the tag.

Tag	Block	Offset
7 bits	4 bits	3 bits
14 bits in main memory address		

Main Memory Address Organization: Example 3

Assume a byte-addressable memory consisting of 16 bytes divided into 8 blocks. Cache contains 4 blocks. What is the main memory organization for direct mapped cache?

Main Memory Address Organization: Example 3

Assume a byte-addressable memory consisting of 16 bytes divided into 8 blocks. Cache contains 4 blocks. What is the main memory organization for direct mapped cache?

- ▶ Memory address: 4 bits ($2^4 = 16$)
- ▶ Block size: $16/8 = 2$
- ▶ Offset field: 1 bit ($2^1 = 2$)
- ▶ block field: 2 bits ($2^2 = 4$)
- ▶ tag field: $4 - 1 - 2 = 1$ bit

Tag	Block	Offset
1 bit	2 bits	1 bit
4 bits in main memory address		

Main Memory Address Organization: Example 4

Consider 16-bit memory addresses and 64 blocks of cache where each block contains 8 bytes. What is the main memory organization for direct mapped cache?

Main Memory Address Organization: Example 4

Consider 16-bit memory addresses and 64 blocks of cache where each block contains 8 bytes. What is the main memory organization for direct mapped cache?

- ▶ 3 bits for the offset
- ▶ 6 bits for the block
- ▶ 7 bits for the tag

What are the values of the offset, block, and tag for memory reference at address 0x0404?

Tag	Block	Offset
0 0 0 0 0 1 0	0 0 0 0 0 0	1 0 0
7 bits	6 bits	3 bits
4 bits in main memory address		

Direct Mapped Cache: Summary

Direct mapped cache maps main memory blocks in a modular fashion to cache blocks. The mapping depends on:

- ▶ The number of bits in the main memory address (how many addresses exist in main memory).
- ▶ The number of blocks in cache (which determines the size of the block field).
- ▶ How many addresses (either bytes or words) are in a block (which determines the size of the offset field)?

However, direct mapped cache has a shortcoming: memory block and cache block mapping is fixed.

Full Associative Cache

A memory block can go anywhere in cache.

- ▶ Cache would have to fill up before any blocks are evicted.
- ▶ Memory address organization:
 - ▶ Memory address is partitioned into only two fields: the tag and the offset.
 - ▶ Example: 14-bit memory addresses and a cache with 16 blocks, each block of size $8 = 2^3$. The format of a memory reference is:

Tag	Offset
11 bits	3 bits
14 bits in main memory address	

When the cache is searched, all tags are searched in parallel to retrieve the data quickly.

- ▶ This requires special, costly hardware.

Cache Replacement

What if a cache block is already occupied?

- ▶ Direct mapped cache evicts a block whenever another memory reference needs that block.
- ▶ With fully associative cache, we have no such mapping, thus we must devise an algorithm to determine which block to evict from the cache.
 - ▶ The block that is evicted is the *victim block*.
 - ▶ There are a number of ways to pick a victim (replacement policy that we will discuss them shortly).

Set Associative Cache

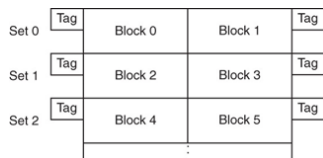
Combines the ideas of direct mapped cache and fully associative cache. Instead of mapping anywhere in the entire cache, a memory reference can map only to the subset of cache slots.

- ▶ N-way set associative cache mapping
 - ▶ Like direct mapped cache in that a memory reference maps to a particular location in cache.
 - ▶ Unlike direct mapped cache, a memory reference maps to a set of N several cache blocks, similar to the way in which fully associative cache works.

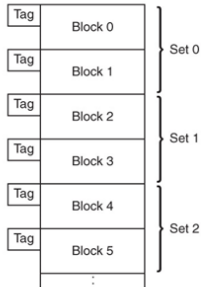
Set Associative Cache: Example

The number of cache blocks per set in set associative cache varies according to overall system design.

- ▶ For example, a 2-way set associative cache can be conceptualized as shown in the schematic below.
- ▶ Each set contains two different memory blocks.



(a) Logical view of 2-way set associative cache



(b) Linear view of 2-way set associative cache

Memory Address Organization

- ▶ In set associative cache mapping, a memory reference is divided into three fields: tag, set, and offset.
- ▶ As with direct-mapped cache, the offset field chooses the byte within the cache block, and the tag field uniquely identifies the memory address.
- ▶ The set field determines the set to which the memory block maps.

Set Associative Mapping: Example 1

Suppose we are using 2-way set associative mapping with a byte-addressable main memory of 2^{14} bytes and a cache with 16 blocks, where each block contains 8 bytes. How does the main memory map to the cache?

Set Associative Mapping: Example 1

Suppose we are using 2-way set associative mapping with a byte-addressable main memory of 2^{14} bytes and a cache with 16 blocks, where each block contains 8 bytes.

- ▶ Cache has a total of 16 blocks, and each set has 2 blocks, then there are $16/2 = 8$ sets in cache.
- ▶ Set field: 3 bits ($2^3 = 8$ sets)
- ▶ Offset field: 3 bits ($2^3 = 8$ bytes)
- ▶ Tag field: $14 - 3 - 3 = 8$ bits

Tag	Set	Offset
8 bits	3 bits	3 bits
14 bits in main memory address		

Comparison of Cache Mapping: Example 1

Suppose a byte-addressable memory contains 1 MB and cache consists of 32 blocks, where each block contains 16 bytes. Using direct mapping, fully associative mapping, and a 4-way set associative mapping, determine where the main memory address $0x326A0$ maps to in cache.

Comparison of Cache Mapping: Example 1

Suppose a byte-addressable memory contains 1 MB and cache consists of 32 blocks, where each block contains 16 bytes. Using direct mapping, fully associative mapping, and a 4-way set associative mapping, determine where the main memory address $0x326A0$ maps to in cache.

- ▶ Memory address: 20 bits ($1 \text{ MB} = 2^{20}$ bytes)

Direct Mapping	Tag 11 bits	Block 5 bits	Offset 4 bits	$16 = 2^4$, $32 = 2^5$, $20 - 4 - 5 = 11$
Fully Associative	Tag 16 bits		Offset 4 bits	$16 = 2^4$, $20 - 4 = 16$
4-way Set Associative	Tag 13 bits	Set 3 bits	Offset 4 bits	$16 = 2^4$, $32/4 = 8 = 2^3$, $20 - 3 - 4 = 13$
20 bits in main memory address				

Comparison of Cache Mapping: Example 1

Suppose a byte-addressable memory contains 1 MB and cache consists of 32 blocks, where each block contains 16 bytes. Using direct mapping, fully associative mapping, and a 4-way set associative mapping, determine where the main memory address $0x326A0$ maps to in cache.

$$0x326A0 = 00110010011010100000_2$$

Direct Mapping	Tag 0011 0010 011	Block 0 1010	Offset 0000
Fully Associative	Tag 0011 0010 0110 1010		Offset 0000
4-way Set Associative	Tag 0011 0010 0110 1	Set 0 10	Offset 0000
20 bits in main memory address			

Comparison of Cache Mapping: Example 2

A byte-addressable computer with an 8-block cache of 4 bytes each, trace memory accesses: 0x01, 0x04, 0x09, 0x05, 0x14, 0x21, and 0x01 for each mapping approach: direct mapping, fully associative, and 2-way set associative.

Comparison of Cache Mapping: Example 2: Answer Part 1

A byte-addressable computer with an 8-block cache of 4 bytes each, trace memory accesses: 0x01, 0x04, 0x09, 0x05, 0x14, 0x21, and 0x01 for each mapping approach: direct mapping, fully associative, and 2-way set associative. The total memory is $2^8 = 256$ bytes.

Direct Mapping	Tag 3 bits	Block 3 bits	Offset 2 bits	$4 = 2^2$, $8 = 2^3$, $8 - 3 - 2 = 3$
Fully Associative	Tag 6 bits		Offset 2 bits	$4 = 2^2$, $8 - 2 = 6$
4-way Set Associative	Tag 4 bits	Set 2 bits	Offset 2 bits	$4 = 2^2$, $8/2 = 4 = 2^2$, $8 - 2 - 2 = 4$
8 bits in main memory address				

Comparison of Cache Mapping: Example 2: Answer Part 2(a)

For direct mapping scheme:

Address Reference	Binary Address (divided into fields)	Hit or Miss	Comments
0x01	000 000 01	Miss	If we check cache block 000 for the tag 000, we find that it is not there. So we copy the data from addresses 0x00, 0x01, 0x02, and 0x03 into cache block 0 and store the tag 000 for that block.
0x04	000 001 00	Miss	We check cache block 001 for the tag 000, and on finding it missing, we copy the data from addresses 0x04, 0x05, 0x06, and 0x07 into cache block 1 and store the tag 000 for that block.
0x09	000 010 01	Miss	A check of cache block 010 (2) for the tag 000 reveals a miss, so we copy the data from addresses 0x08, 0x09, 0x0A, and 0x0B into cache block 2 and store the tag 000 for that block.
0x05	000 001 01	Hit	We check cache block 001 for the tag 000, and we find it. We then use the offset value 01 to get the exact byte we need.
0x14	000 101 00	Miss	We check cache block 101 (5) for the tag 000, but it is not present. We copy addresses 0x14, 0x15, 0x16, and 0x17 to cache block 5 and store the tag 000 with that block.
0x21	001 000 01	Miss	We check cache block 000 for the tag 001; we find tag 000 (which means this is not the correct block), so we overwrite the existing contents of this cache block by copying the data from addresses 0x20, 0x21, 0x22, and 0x23 into cache block 0 and storing the tag 001.
0x01	000 000 01	Miss	Although we have already fetched the block that contains address 0x01 once, it was overwritten when we fetched the block containing address 0x21 (if we look at block 0 in cache, we can see that its tag is 001, not 000). Therefore, we must overwrite the contents of block 0 in cache with the data from addresses 0x00, 0x01, 0x02, and 0x03, and store a tag of 000.

Comparison of Cache Mapping: Example 2: Answer Part 2(b)

For fully associate scheme:

Address Reference	Binary Address (divided into fields)	Hit or Miss	Comments
0x01	000000 01	Miss	We search all of cache for the tag 000000, and we don't find it. So we copy the data from addresses 0x00, 0x01, 0x02, and 0x03 into cache block 0 and store the tag 000000 for that block.
0x04	000001 00	Miss	We search all of cache for the tag 000001, and on finding it missing, we copy the data from addresses 0x04, 0x05, 0x06, and 0x07 into cache block 1 and store the tag 000001 for that block.
0x09	000010 01	Miss	We don't find the tag 000010 in cache, so we copy the data from addresses 0x08, 0x09, 0x0A, and 0x0B into cache block 2 and store the tag 000010 for that block.
0x05	000001 01	Hit	We search all of cache for the tag 000001, and we find it stored with cache block 1. We then use the offset value 01 to get the exact byte we need.
0x14	000101 00	Miss	We search all of cache for the tag 000101, but it is not present. We copy addresses 0x14, 0x15, 0x16, and 0x17 to cache block 3 and store the tag 000101 with that block.
0x21	001000 01	Miss	We search all of cache for the tag 001000; we don't find it, so we copy the data from addresses 0x20, 0x21, 0x22, and 0x23 into cache block 4 and store the tag 001000.
0x01	000000 01	Hit	We search cache for the tag 000000 and find it with cache block 0. We use the offset of 1 to find the data we want.

Comparison of Cache Mapping: Example 2: Answer Part 2(c)

For 2-way set associate scheme:

Address Reference	Binary Address (divided into fields)	Hit or Miss	Comments
0x01	0000 00 01	Miss	We search in set 0 of cache for a block with the tag 0000, and we find it is not there. So we copy the data from addresses 0x00, 0x01, 0x02, and 0x03 into set 0 (so now set 0 has one used block and one free block) and store the tag 0000 for that block. It does not matter which set we use; for consistency, we put the data in the first set.
0x04	0000 01 00	Miss	We search set 1 for a block with the tag 0000, and on finding it missing, we copy the data from addresses 0x04, 0x05, 0x06, and 0x07 into set and store the tag 0000 for that block.
0x09	0000 10 01	Miss	We search set 2 (10) for a block with the tag 0000, but we don't find one, so we copy the data from addresses 0x08, 0x09, 0x0A, and 0x0B into set 2 and store the tag 0000 for that block.
0x05	0000 01 01	Hit	We search set 1 for a block with the tag 0000, and we find it. We then use the offset value 01 within that block to get the exact byte we need.
0x14	0001 01 00	Miss	We search set 1 for a block with the tag 0001, but it is not present. We copy addresses 0x14, 0x15, 0x16, and 0x17 to set 1 and store the tag 0001 with that block. Note that set 1 is now full.
0x21	0010 00 01	Miss	We search cache set 0 for a block with the tag 0010; we don't find it, so we copy the data from addresses 0x20, 0x21, 0x22, and 0x23 into set 0 and store the tag 0010. Note that set 0 is now full.
0x01	0000 00 01	Hit	We search cache set 0 for a block with the tag 0000, and we find it. We use the offset of 1 within that block to find the data we want.

Cache Replacement Policy

With fully associative and set associative cache, a replacement policy is invoked when it becomes necessary to evict a block from cache.

- ▶ An optimal replacement policy would be able to look into the future to see which blocks won't be needed for the longest period of time.
- ▶ An optimal replacement is instructive to serve as a benchmark for assessing the efficiency of any other scheme we come up with.
- ▶ The replacement policy that we choose depends upon the locality that we are trying to optimize

LRU

Interested in temporal locality:

- ▶ The least recently used (LRU) algorithm keeps track of the last time that a block was accessed and evicts the block that has been unused for the longest period of time.
- ▶ The disadvantage of this approach is its complexity
 - ▶ LRU has to maintain an access history for each block, which ultimately slows down the cache.

FIFO

First-in, first-out (FIFO) is a popular cache replacement policy.

- ▶ In FIFO, the block that has been in the cache the longest, regardless of when it was last used.

FIFO

- ▶ A random replacement policy does what its name implies: It picks a block at random and replaces it with a new block.
- ▶ Random replacement can certainly evict a block that will often be needed or needed soon, but it never thrashes.

Performance of Hierarchical Memory

Effective access time (EAT) is a weighted average that takes into account the hit ratio and relative access times of successive levels of memory.

- ▶ Under the condition that access to cache and main memory occurs concurrently (the accesses overlap), the EAT for a two-level memory is given by:

$$\text{EAT} = H \times \text{Access}_C + (1 - H) \times \text{Access}_{MM}$$

where H is the cache hit rate and Access_C and Access_{MM} are the access times for cache and main memory, respectively.

Effective Access Time: Example 1

Consider a system with a main memory access time of 200 ns supported by a cache having a 10 ns access time and a hit rate of 99%. What is EAT?

Effective Access Time: Example 1

Consider a system with a main memory access time of 200 ns supported by a cache having a 10 ns access time and a hit rate of 99%.

- ▶ Suppose access to cache and main memory occurs concurrently (the accesses overlap). The EAT is:

$$0.99 \times 10 + 0.01 \times 200 = 9.9 + 2 = 11\text{ns}$$

Effective Access Time: Example 2

Consider a system with a main memory access time of 200 ns supported by a cache having a 10 ns access time and a hit rate of 99%. What is EAT?

Effective Access Time: Example 2

Consider a system with a main memory access time of 200 ns supported by a cache having a 10 ns access time and a hit rate of 99%.

- ▶ If the accesses do not overlap, the EAT is:

$$0.99 \times 10 + 0.01 \times (10 + 200) = 9.9 + 2.1 = 12\text{ns}$$

This analysis for determining the effective access time can be extended to any number of memory levels.

Discussion: Locality

Caching is depends upon programs exhibiting good locality.

- ▶ Some object-oriented programs have poor locality owing to their complex, dynamic structures.
- ▶ Arrays stored in column-major rather than row-major order can be problematic for certain cache organizations.

With poor locality, caching can actually cause performance degradation rather than performance improvement.

Discussion: Cache Replacement Policies

Cache replacement policies must take into account dirty blocks, those blocks that have been updated while they were in the cache.

- ▶ Dirty blocks must be written back to memory. A write policy determines how this will be done.
- ▶ There are two types of write policies, write through and write back.
 - ▶ Write through updates cache and main memory simultaneously on every write.
 - ▶ Write back (also called copyback) updates memory only when the block is selected for replacement.

Write Through vs. Write Back

The disadvantage of write through:

- ▶ Memory must be updated with each cache write, which slows down the access time on updates.
- ▶ This slowdown is usually negligible, because the majority of accesses tend to be reads, not writes.

The advantage of write back:

- ▶ Memory traffic is minimized

Its disadvantage:

- ▶ Memory does not always agree with the value in cache, causing problems in systems with many concurrent users.

Integrated Cache or Separated Cache

- ▶ The cache we have been discussing is called a unified or integrated cache where both instructions and data are cached.
- ▶ Many modern systems employ separate caches for data and instructions. This is called a Harvard cache.
 - ▶ The separation of data from instructions provides better locality, at the cost of greater complexity.
 - ▶ Simply making the cache larger provides about the same performance improvement without the complexity.

Victim Cache

- ▶ Cache performance can also be improved by adding a small associative cache to hold blocks that have been evicted recently.
- ▶ This is called a victim cache.

Trace Cache

- ▶ A trace cache is a variant of an instruction cache that holds decoded instructions for program branches, giving the illusion that noncontiguous instructions are really contiguous.

Multi-Level Cache

- ▶ Most of today's small systems employ multilevel cache hierarchies.
- ▶ The levels of cache form their own small memory hierarchy.

Example: Two-Level Cache

- ▶ Level 1 cache (8KB to 64KB) is situated on the processor itself.
 - ▶ Access time is typically about 4ns.
- ▶ Level 2 cache (64KB to 2MB) may be on the motherboard, or on an expansion card.
 - ▶ Access time is usually around 15 - 20ns.

Three-Level Cache

- ▶ In systems that employ three levels of cache, the Level 2 cache is placed on the same die as the CPU (reducing access time to about 10ns).
- ▶ Accordingly, the Level 3 cache (2MB to 256MB) refers to cache that is situated between the processor and main memory.
- ▶ Once the number of cache levels is determined, the next thing to consider is whether data (or instructions) can exist in more than one cache level.

Inclusive vs. Exclusive Cache

If the cache system used an inclusive cache, the same data may be present at multiple levels of cache.

- ▶ Strictly inclusive caches guarantee that all data in a smaller cache also exists at the next higher level.

Exclusive caches permit only one copy of the data.

- ▶ The trade-offs in choosing one over the other involve weighing the variables of access time, memory size, and circuit complexity

Table of Contents

- 1 Lesson Objectives
- 2 Type of Memories
- 3 Memory Hierarchy
- 4 Cache Memory
 - Direct Mapped Cache
 - Fully Associative Cache
 - Cache Replacement Policy
 - Effective Access Time
- 5 Virtual Memory

Virtual Memory

- ▶ Cache memory enhances performance by providing faster memory access speed.
- ▶ Virtual memory enhances performance by providing greater memory capacity, without the expense of adding main memory.
- ▶ Instead, a portion of a disk drive serves as an extension of main memory.
- ▶ If a system uses paging, virtual memory partitions main memory into individually managed page frames, that are written (or paged) to disk when they are not immediately needed.

Basic Concepts

- ▶ A physical address is the actual memory address of physical memory.
- ▶ Programs create virtual addresses that are mapped to physical addresses by the memory manager.
- ▶ Page faults occur when a logical address requires that a page be brought in from disk.
- ▶ Memory fragmentation occurs when the paging process results in the creation of small, unusable clusters of memory addresses.

Paging

Main memory and virtual memory are divided into equal sized pages.

- ▶ The entire address space required by a process need not be in memory at once. Some parts can be on disk, while others are in main memory.
- ▶ Further, the pages allocated to a process do not need to be stored contiguously—either on disk or in memory.
- ▶ In this way, only the needed pages are in memory at any time, the unnecessary pages are in slower disk storage.

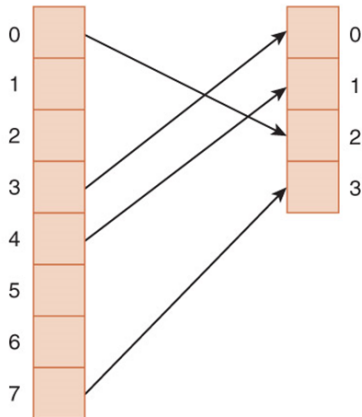
Page Table

- ▶ Information concerning the location of each page, whether on disk or in memory, is maintained in a data structure called a page table (shown below).
- ▶ There is a page table for each active process.

Page Table: Example

Virtual memory

Physical memory



Page

Page table

	Frame #	Valid Bit
0	2	1
1	-	0
2	-	0
3	0	1
4	1	1
5	-	0
6	-	0
7	3	1

Virtual Address to Physical Memory Address

- ▶ When a process generates a virtual address, the operating system translates it into a physical memory address.
- ▶ To accomplish this, the virtual address is divided into two fields: A page field, and an offset field.
- ▶ The page field determines the page location of the address, and the offset indicates the location of the address within the page.
- ▶ The logical page number is translated into a physical page frame through a lookup in the page table.

Page Fault

- ▶ If the valid bit is zero in the page table entry for the logical address, this means that the page is not in memory and must be fetched from disk.
- ▶ This is a page fault.
- ▶ If necessary, a page is evicted from memory and is replaced by the page retrieved from disk, and the valid bit is set to 1.
- ▶ If the valid bit is 1, the virtual page number is replaced by the physical frame number.
- ▶ The data is then accessed by adding the offset to the physical frame number.

Example

Suppose a system has a virtual address space of 8K and a physical address space of 4K, and the system uses byte addressing.

- ▶ We have $2^{13}/2^{10} = 2^3$ virtual pages.
- ▶ A virtual address has 13 bits ($8K = 2^{13}$) with 3 bits for the page field and 10 for the offset, because the page size is 1024.
- ▶ A physical memory address requires 12 bits, the first 2 bits for the page frame and the trailing 10 bits the offset.

Virtual Address
13 bits

Page	Offset
3 bits	10 bits

Physical Address
12 bits

Frame	Offset
2 bits	10 bits

Virtual Memory: Example

Suppose we have the page table shown below. What happens when the CPU generates address $5459_{10} = 1010101010011_2 = 0x1553$?

B Page table

Page	Frame	Valid bit
0	-	0
1	3	1
2	0	1
3	-	0
4	-	0
5	1	1
6	2	1
7	-	0

C Addresses

	Base 10	Base 16
Page 0 :	0 - 1023	0 - 3FF
1 :	1024 - 2047	400 - 7FF
2 :	2048 - 3071	800 - BFF
3 :	3072 - 4095	C00 - FFF
4 :	4096 - 5119	1000 - 13FF
5 :	5120 - 6143	1400 - 17FF
6 :	6144 - 7167	1800 - 1BFF
7 :	7168 - 8191	1C00 - 1FFF

13

Virtual Memory: Example

What happens when the CPU generates address
 $5459_{10} = 1010101010011_2 = 0x1553$?

Virtual Address
 13 bits

Page	Offset
3 bits	10 bits

The high-order 3 bits of the virtual address, $101_2 = 5_{10}$, provide the page number in the page table.

Virtual Memory: Example

The address 1010101010011_2 is converted to physical address $010101010011_2 = 0x1363$ because the page field 101 is replaced by frame number 01 through a lookup in the page table.

B Page table

		Frame	Valid bit
Page	0	-	0
	1	3	1
	2	0	1
	3	-	0
	4	-	0
	5	1	1
	6	2	1
	7	-	0

C Addresses

	Base 10	Base 16
Page 0 :	0 - 1023	0 - 3FF
1 :	1024 - 2047	400 - 7FF
2 :	2048 - 3071	800 - BFF
3 :	3072 - 4095	C00 - FFF
4 :	4096 - 5119	1000 - 13FF
5 :	5120 - 6143	1400 - 17FF
6 :	6144 - 7167	1800 - 1BFF
7 :	7168 - 8191	1C00 - 1FFF

13

Virtual Memory: Example

What happens when the CPU generates address 1000000000100_2 ?

B Page table

		Frame	Valid bit
Page	0	-	0
	1	3	1
	2	0	1
	3	-	0
	4	-	0
	5	1	1
	6	2	1
	7	-	0

C Addresses

	Base 10	Base 16
Page 0 :	0 - 1023	0 - 3FF
1 :	1024 - 2047	400 - 7FF
2 :	2048 - 3071	800 - BFF
3 :	3072 - 4095	C00 - FFF
4 :	4096 - 5119	1000 - 13FF
5 :	5120 - 6143	1400 - 17FF
6 :	6144 - 7167	1800 - 1BFF
7 :	7168 - 8191	1C00 - 1FFF

12

Effective Access Time

Effective access time (EAT) takes all levels of memory into consideration.

- ▶ Thus, virtual memory is also a factor in the calculation, and we also have to consider page table access time.

Suppose a main memory access takes 200ns, the page fault rate is 1%, and it takes 10ms to load a page from disk. We have:

$$\text{EAT} = 0.99(200\text{ns} + 200\text{ns}) + 0.01(10\text{ms}) = 100,396\text{ns}$$

TLB

Even if we had no page faults, the EAT would be 400ns because memory is always read twice:

- ▶ First to access the page table, and second to load the page from memory.

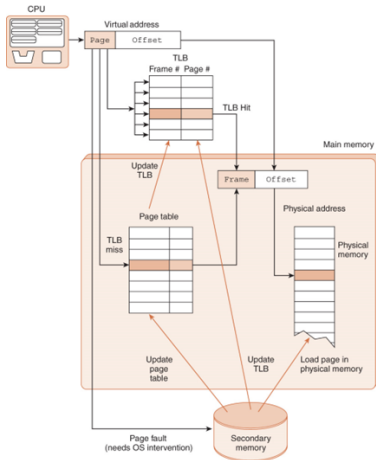
Because page tables are read constantly, it makes sense to keep them in a special cache called a translation look-aside buffer (TLB).

- ▶ TLBs are a special associative cache that stores the mapping of virtual pages to physical pages.

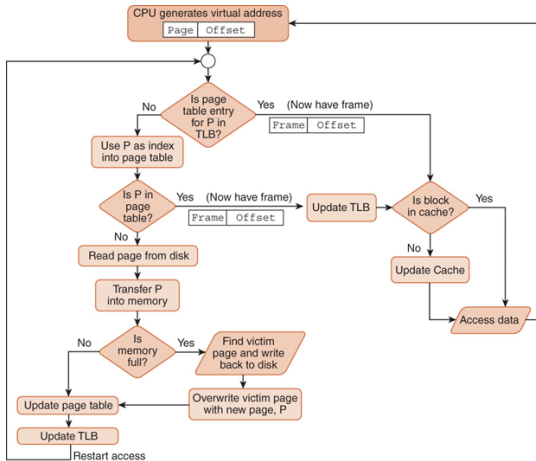
TLB Lookup Process

- ▶ Extract the page number from the virtual address.
- ▶ Extract the offset from the virtual address.
- ▶ Search for the virtual page number in the TLB.
- ▶ If the (virtual page #, page frame #) pair is found in the TLB, add the offset to the physical frame number and access the memory location.
- ▶ If there is a TLB miss, go to the page table to get the necessary frame number. If the page is in memory, use the corresponding frame number and add the offset to yield the physical address.
- ▶ If the page is not in main memory, generate a page fault and restart the access when the page fault is complete.

TLB Lookup Process



TLB, Page Table, and Main Memory



Segmentation

Another approach to virtual memory is the use of segmentation.

- ▶ Instead of dividing memory into equal-sized pages, virtual address space is divided into variable-length segments, often under the control of the programmer.
- ▶ A segment is located through its entry in a segment table, which contains the segment's memory location and a bounds limit that indicates its size.
- ▶ After a page fault, the operating system searches for a location in memory large enough to hold the segment that is retrieved from disk.

Fragmentation

Both paging and segmentation can cause fragmentation.

- ▶ Paging is subject to internal fragmentation because a process may not need the entire range of addresses contained within the page. Thus, there may be many pages containing unused fragments of memory.
- ▶ Segmentation is subject to external fragmentation, which occurs when contiguous chunks of memory become broken up as segments are allocated and deallocated over time.

Internal Fragmentation: Example

- ▶ Consider a small computer having 32K of memory.
- ▶ The 32K memory is divided into 8 page frames of 4K each.
- ▶ A schematic of this configuration is shown at the right.
- ▶ The numbers at the right are memory frame addresses.

	0
	4K
	8K
	12K
	16K
	20K
	24K
	28K

Internal Fragmentation: Example

- ▶ Suppose there are four processes waiting to be loaded into the system with memory requirements as shown in the table.
- ▶ We observe that these processes require 31K of memory.

Process Name	Memory Needed
P1	8K
P2	10K
P3	9K
P4	4K

Internal Fragmentation: Example

- ▶ When the first three processes are loaded, memory looks like this:
- ▶ All the frames are occupied by three of the processes.

Process Name	Memory Needed
P1	8K
P2	10K
P3	9K
P4	4K

P1	0
P1	4K
P2	8K
P2	12K
P2	16K
P3	20K
P3	24K
P3	28K

Internal Fragmentation: Example

- ▶ Despite the fact that there are enough free bytes in memory to load the fourth process, P4 has to wait for one of the other three to terminate, because there are no unallocated frames.
- ▶ This is an example of internal fragmentation.

External Fragmentation: Example

- ▶ Suppose that instead of frames, our 32K system uses segmentation.
- ▶ The memory segments of two processes is shown in the table at the right.
- ▶ The segments can be allocated anywhere in memory.

Process Name	Segment	Memory Needed
P1	S1	8K
	S2	10K
	S3	9K
P2	S1	4K
	S2	11K

External Fragmentation: Example

- ▶ All the segments of P1 and one of the segments of P2 are loaded as shown at the right.
- ▶ Segment S2 of process P2 requires 11K of memory, and there is only 1K free, so it waits.

External Fragmentation: Example

- ▶ Eventually, Segment 2 of Process 1 is no longer needed, so it is unloaded giving 11K of free memory.
- ▶ But Segment 2 of Process 2 cannot be loaded because the free memory is not contiguous.

External Fragmentation: Example

- ▶ Over time, the problem gets worse, resulting in small unusable blocks scattered throughout physical memory.
- ▶ This is an example of external fragmentation.
- ▶ Eventually, this memory is recovered through compaction, and the process starts over.

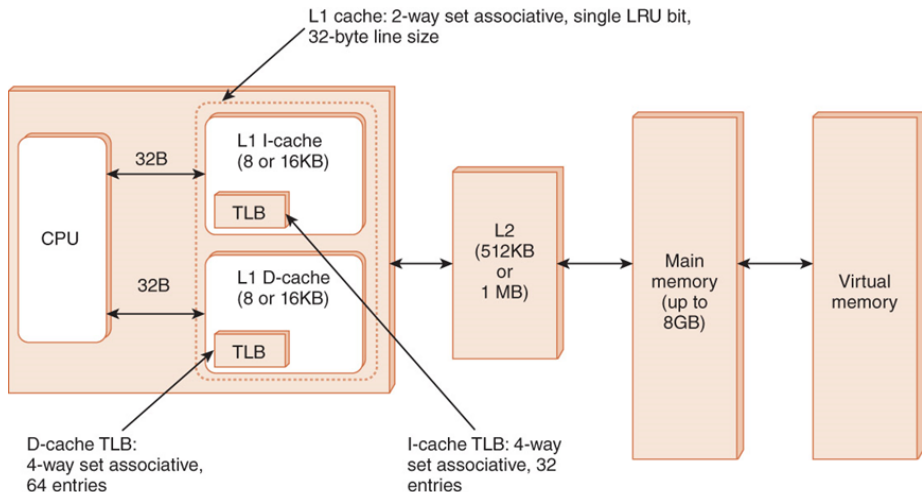
Paging vs. Fragmentation

- ▶ Large page tables are cumbersome and slow, but with its uniform memory mapping, page operations are fast. Segmentation allows fast access to the segment table, but segment loading is labor-intensive.
- ▶ Paging and segmentation can be combined to take advantage of the best features of both by assigning fixed-size pages within variable-sized segments.
- ▶ Each segment has a page table. This means that a memory address will have three fields, one for the segment, another for the page, and a third for the offset.

Virtual Memory: Real-World Example

- ▶ The Pentium architecture supports both paging and segmentation, and they can be used in various combinations including unpagged unsegmented, segmented unpagged, and unsegmented pagged.
- ▶ The processor supports two levels of cache (L1 and L2), both having a block size of 32 bytes.
- ▶ The L1 cache is next to the processor, and the L2 cache sits between the processor and memory.
- ▶ The L1 cache is in two parts: and instruction cache (I-cache) and a data cache (D-cache).

Virtual Memory: Real-World Example



Summary and Q&A

- ▶ Computer memory is organized in a hierarchy, with the smallest, fastest memory at the top and the largest, slowest memory at the bottom.
- ▶ Cache memory gives faster access to main memory, while virtual memory uses disk storage to give the illusion of having a large main memory.
- ▶ Cache maps blocks of main memory to blocks of cache memory. Virtual memory maps page frames to virtual pages.
- ▶ There are three general types of cache: direct mapped, fully associative, and set associative.

Summary and Q&A

- ▶ With fully associative and set associative cache, as well as with virtual memory, replacement policies must be established.
- ▶ Replacement policies include LRU, FIFO, or LFU. These policies must also take into account what to do with dirty blocks.
- ▶ All virtual memory must deal with fragmentation, internal for paged memory, external for segmented memory.