6.4 Cache Memory

Purpose: The purpose of cache is to speed up memory accesses time.

Basic operation: The cache scans memory addresses as they appear on the CPU-Memory bus. When the cache matches an address, the data are read from the cache memory instead of the main memory. When the cache fails to make a match, the cache copies the data from main memory and stores the new address and data in the cache, possibly, replacing an existing address and corresponding data.

Cache stores both data and addresses.

To find a value in cache, the incoming virtual addresses is compared to all addresses in the cache. If a match is found, corresponding data are delivered to the CPU. For this reason, cache is called a **content addressable memory**. The contents are searched to find the corresponding value (data).

There are three types of caches:

- Direct
- Fully Associative
- Set Associative

Typical access times for cache memory are 5 times faster than main memory. In 2014, the approximate cache access time was 10ns and for main store 50ns.

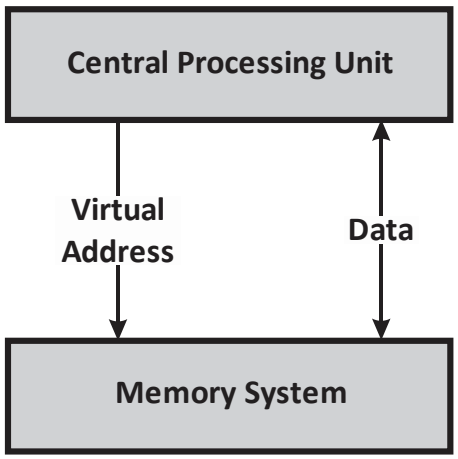


Figure 1. The CPU – Virtual Memory System Interface

Read

1. The CPU assigns a virtual address to the Memory Address Register (MAR).
2. The CPU asserts the $\text{Read}/\overline{\text{Write}}$ signal indicating that data are to be read from memory.
3. After a few clock cycles, data are assigned to the Memory Buffer Register (MBR).

Write

1. The CPU assigns data to the Memory Buffer Register (MBR).
2. The CPU assigns the virtual address of the data in the MBR to the Memory Address Register (MAR).
3. The CPU asserts the $\text{Read}/\overline{\text{Write}}$ signal indicating that data are to be written to the address given by the MAR.
4. After a few clock cycles, data in the (MBR) are assigned to memory at the address in the MAR.

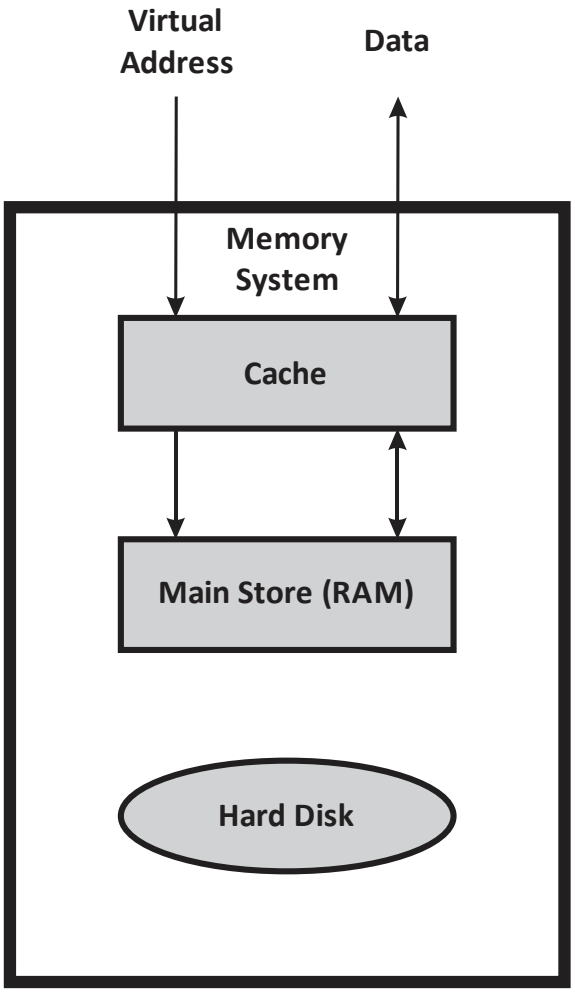


Figure 2. Anatomy of a Virtual Memory System

1. Data are sought in the cache, first.
2. The event that occurs when data cannot be found in the cache is called a **cache miss**.
3. The event that occurs when data are found in the cache is called a **cache hit**.
4. In the event of a cache miss, data are sought in the main store.
5. The event that occurs when data cannot be found in the main store is called a **page fault.**

6. If data are found in the main store, they are copied to the cache, along with the tag portion of the corresponding virtual address. Then, the cache makes the data available to the CPU. Accessing main store requires about five (5) times as much time as accessing cache. For example, if a cache access costs 10ns, then a main store access costs about 50ns.
7. In the event of a page fault, data are sought at their home location, on disk. Please recall, that cache and main store contain only copies of what is stored on hard disk. The virtual address is translated to a disk address consisting of the triple, surface, cylinder, and sector. It is advantageous to match the sector size to the page size. For example, it may happen that one sector occupies $2^{12} = 4096$ bytes making it advantageous to define a page to occupy the same storage. Accessing hard disk requires about 10ms or about six orders of magnitude longer to access disk than to access main store.

6.4.1.1 Direct Mapped Cache

Term	Definition
Tag	• The ***tag*** is the most significant portion of the virtual address. • Tags are stored in the cache. • A cache hit occurs when the tag portion of the virtual address tag **matches** the tag stored in the cache. • A cache miss occurs when the tag portion of the virtual address tag **does not match** the tag stored in the cache.
Block	• The ***block*** is the next most significant portion of the virtual address. • The block is ***not*** stored in the cache. • For a direct mapped cache the block contains the index of tag. • For a direct mapped cache the block is used to find the location where the tag is stored in the cache. The tag portion of the virtual memory address is compared against the tag found in the cache.
Offset	• The ***offset*** is the least significant portion of the virtual address. • The offset is ***not*** stored in the cache. • The offset can be used to determine how many bytes of data are stored in the cache for each cache entry. For example, if the offset occupies 4 bits, then $2^4 = 16$ bytes of data are stored in each cache entry. • The specific value of the offset identifies the particular byte in a byte-addressable memory.

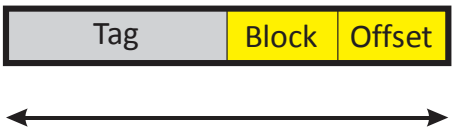


Figure 3 The Format of a Virtual Address Using Direct Mapping

- The tag in the virtual memory address is matched against the tag stored in the cache.
- The block is used by the cache to find the tag.

- The offset is used by the cache to find the specific byte in the data stored in the cache.

Example 1. Consider a byte-addressable memory having $2^{10} = 1024$ bytes and a cache having eight (8) cache entries. Each cache entry contains four (4) bytes of data and one tag consisting of the most significant five (5) bits of the 10-bit virtual address.

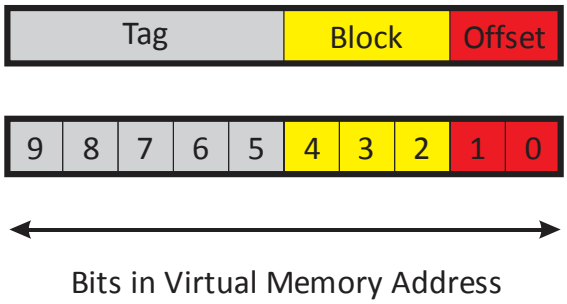


Figure 4. Example 1. The Format of a Virtual Address Using Direct Mapping

- In this example there are $\frac{2^{10}}{2^3} = 2^7 = 128$ blocks of 4 bytes that are served by each cache entry. Divide the number of bytes of main store by the number of cache entries to determine how many blocks are served by a single cache entry. This computation is valid only for a direct mapped cache.

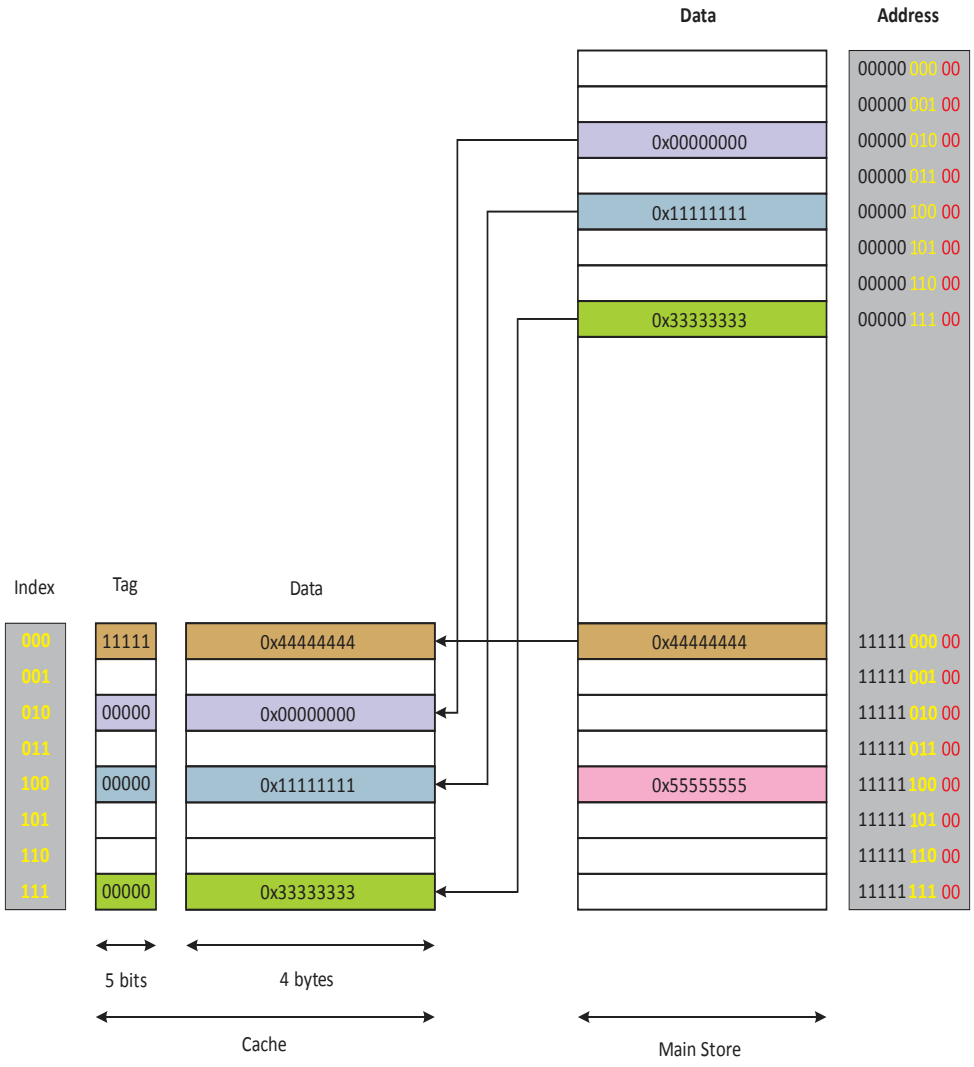


Figure 5. Example 1. Direct Cache Mapping

- Data at address 11111 **000** 00 is mapped to the cache entry at index position **000**.
- Data at address 00000 **010** 00 is mapped to the cache entry at index position **010**.
- Data at address 00000 **100** 00 is mapped to the cache entry at index position **100**.
- Data at address 00000 **111** 00 is mapped to the cache entry at index position **111**.
- Data at address 11111 **100** 00 was mapped to the cache entry at index position **100** but was, later, overwritten by data at address 00000 **100** 00. **A direct mapped cache cannot store two values having the same index**.

1. Extract the index from the virtual address. The index is stored in bit positions 2, 3 and 4 as shown on the previous page.
2. Use the index to find the cache entry.
3. Extract the tag from the cache entry.
4. Compare the tag in the virtual address with the tag in the cache.
   4.1. If the tags match, deliver the corresponding data stored in the cache to the CPU.

4.2. If the tags do not match, assign the tag from the virtual memory address to the cache entry. Assign data from the main store to the cache entry. Deliver the new data to the CPU.

Exercise[1]: A computer has a 32-bit address and a direct-mapped cache. Addressing is to the byte level. The cache has a capacity of 1 KB and uses lines that are 32 bytes. It uses write-through and so does not require a dirty bit.

Solution:

- **A line is data in a cache entry. Since data occupy 32 bytes, 5 bits are required to address data in the cache entry. The offset is used to address the bytes in the cache entry. Five (5) bits are required to address memory occupying 32 bytes.**

(a) How many bits are in the index for the cache?

Solution:
The index is the block. To determine the number of bits in the block, we need to divide the size of the cache by the size of each entry. $\frac{1\,KB}{32\,B} = \frac{2^{10}B}{2^5B} = 2^5$**. There are** $2^5 = 32$ **blocks requiring 5 bits for the block field.**

(b) How many bits are in the tag for the cache?

Solution:
The tag can be computed by subtracting the number of bits required for the block and the offset from the number of bits in the virtual address. Recall that the virtual address occupies 32 bits. Thus
Tag= $32 - 5 - 5 = 22$ **bits.**

(c) What is the total number of bits of storage in the cache, including valid bits, the tags, and the cache lines?

Solution:
For every cache entry there is:

- **1 valid bit**
- **1 Tag occupying 22 bits**
- **1 Cache entry occupying 32 bytes or 256 bits**
- **The total number of bits is:** $32 \times (1 + 22 + 32 \times 8) = 8928$

[1] Exercise 13-3, page 659 Mano and Kime "Logic and Computer Design Fundamentals, 4th Ed." Pearson Education, Inc, 2008 ISBN 0-13-198926-X

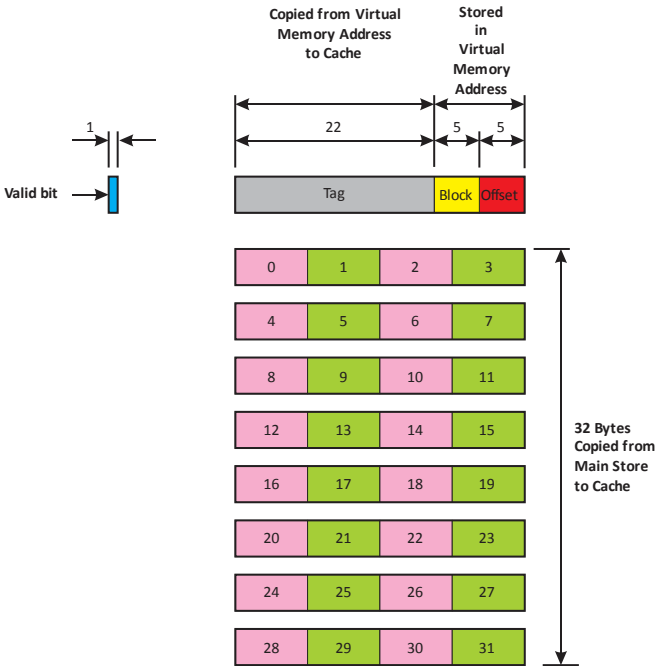


Figure 6. Direct Cache Entry Exercise

6.4.1.2 Fully Associative Cache

Term	Definition
Tag	• The ***tag*** is the most significant portion of the virtual address. • Tags are stored in the cache. • A cache hit occurs when the tag portion of the virtual address tag **matches** the tag stored in the cache. • A cache miss occurs when the tag portion of the virtual address tag **does not match** the tag stored in the cache.
Block	There is no block in a fully associative cache. The block field is appended to the tag field.
Offset	• The ***offset*** is the least significant portion of the virtual address. • The offset is ***not*** stored in the cache. • The offset can be used to determine how many bytes of data are stored in the cache for each cache entry. For example, if the offset occupies 4 bits, then $2^4 = 16$ bytes of data are stored in each cache entry. • The specific value of the offset identifies the particular byte in a byte-addressable memory.

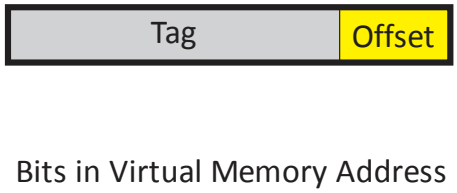


Figure 7. The Format of a Virtual Address Using Fully Associative Mapping

- The tag in the virtual memory address is matched against the tag stored in the cache.
- The offset is used by the cache to find the specific byte in the data stored in the cache.

Example 2. Consider a byte-addressable memory having $2^{10} = 1024$ bytes and a cache having eight (8) cache entries. Each cache entry contains four (4) bytes of data and one tag consisting of the most significant eight (8) bits of the 10-bit virtual address.

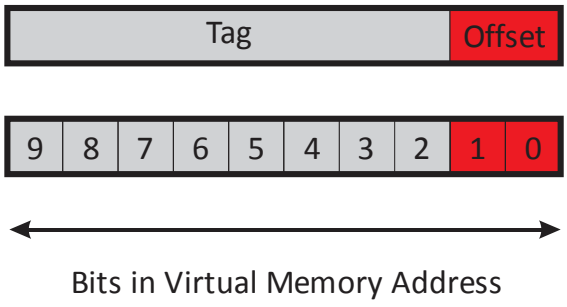


Figure 8. Example 2. The Format of a Virtual Address Using Fully Associative Mapping

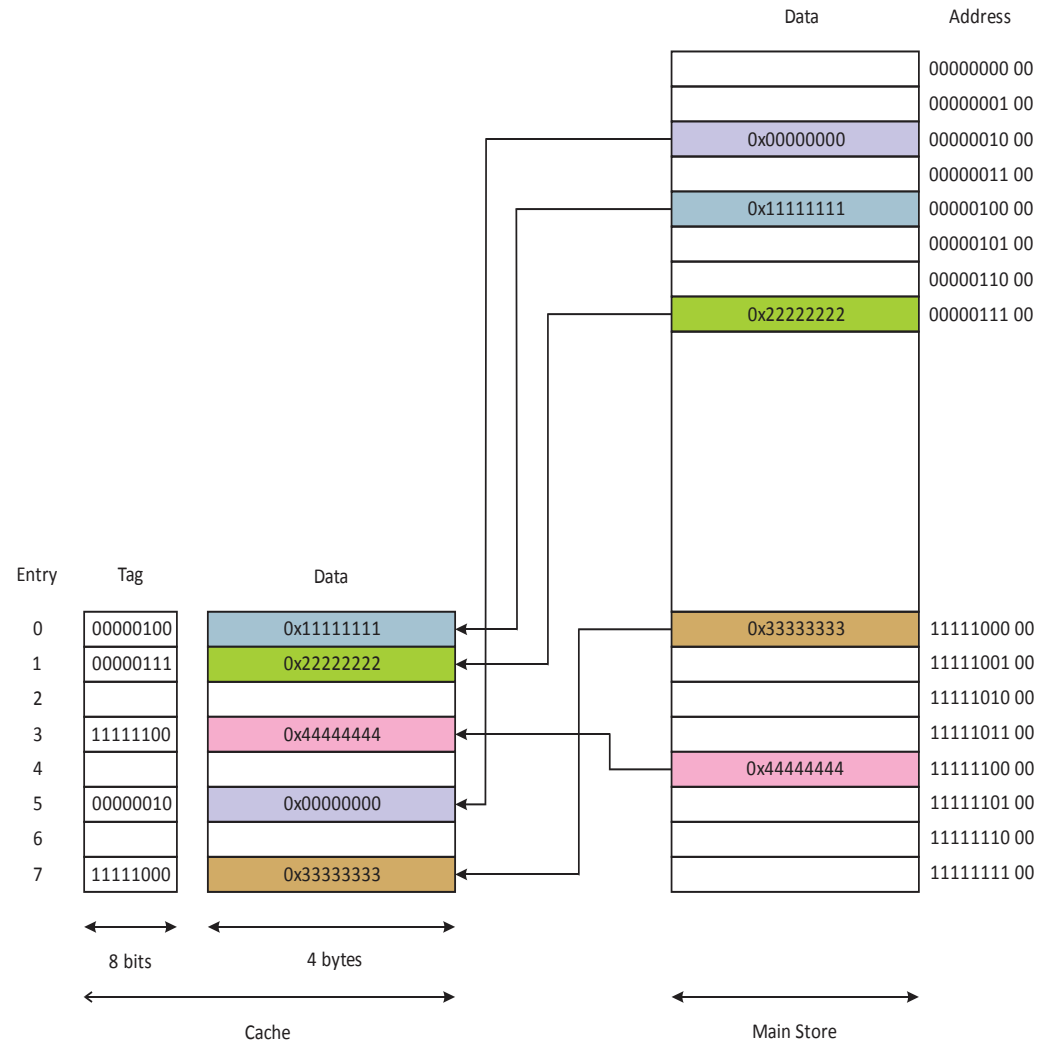


Figure 9. Example 2. Fully Associative Cache Mapping

- Data at address 0000 0010 is mapped to cache entry 5. Please note that the entry identifier is not part of the virtual address and is not used to either store or retrieve values from the cache.
- Data at address 0000 0100 is mapped to cache entry 0.
- Data at address 0000 0111 is mapped to cache entry 1.
- Data at address 1111 1000 is mapped to cache entry 7.
- Data at address 1111 1100 is mapped to cache entry 3.

1. Extract the tag from the virtual address.
2. Compare the tag simultaneously to all tags in the cache.
   2.1. If the virtual address tag matches a tag in the cache, deliver the corresponding data stored in the cache to the CPU.

2.2. If the tags do not match, assign the tag from the virtual memory address to the cache entry. Assign data from the main store to the cache entry. Deliver the new data to the CPU.

6.4.1.3 Set Associative Cache

Term	Definition
Tag	• The ***tag*** is the most significant portion of the virtual address. • Tags are stored in the cache. • A cache hit occurs when the tag portion of the virtual address tag **matches** the tag stored in the cache. • A cache miss occurs when the tag portion of the virtual address tag **does not match** the tag stored in the cache.
Block	• The ***block*** is the next most significant portion of the virtual address. • The block is ***not*** stored in the cache. • For a set associative cache, the block identifies a **set** of entries where the tag portion of the virtual address and corresponding data may be found. The block is employed as an index, similar to that used in a direct mapped cache. The difference here is that the **set-associative block** identifies a row containing two or more tags and corresponding data. • The tag portion of the virtual address is matched against all the tags in the set simultaneously. The set is shown as a row in the diagram below.
Offset	• The ***offset*** is the least significant portion of the virtual address. • The offset is ***not*** stored in the cache. • The offset can be used to determine how many bytes of data are stored in the cache for each cache entry. For example, if the offset occupies 4 bits, then $2^4 = 16$ bytes of data are stored in each cache entry. • The specific value of the offset identifies the particular byte in a byte-addressable memory.

Bits in Virtual Memory Address

Figure 10. The Format of a Virtual Address Using Set Associative Mapping

- The tag in the virtual memory address is matched against the tags stored in the cache.
- The block is used by the cache to find the set of tags and corresponding data that the tag portion of the virtual address is compared against.
- The offset is used by the cache to find the specific byte in the data stored in the cache.

Example 3. Consider a byte-addressable memory having $2^{10} = 1024$ bytes and a cache having eight (8) cache entries. Each cache entry contains four (4) bytes of data and one tag consisting of the most significant six (6) bits of the 10-bit virtual address.

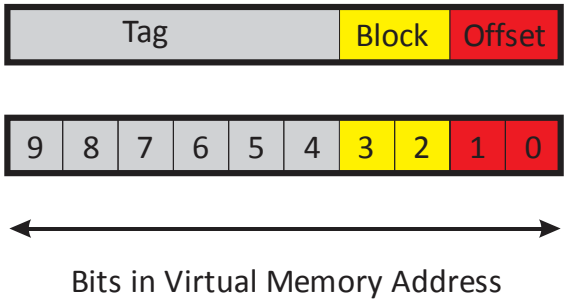


Figure 11. Example 3. The Format of a Virtual Address Using Set Associative Mapping

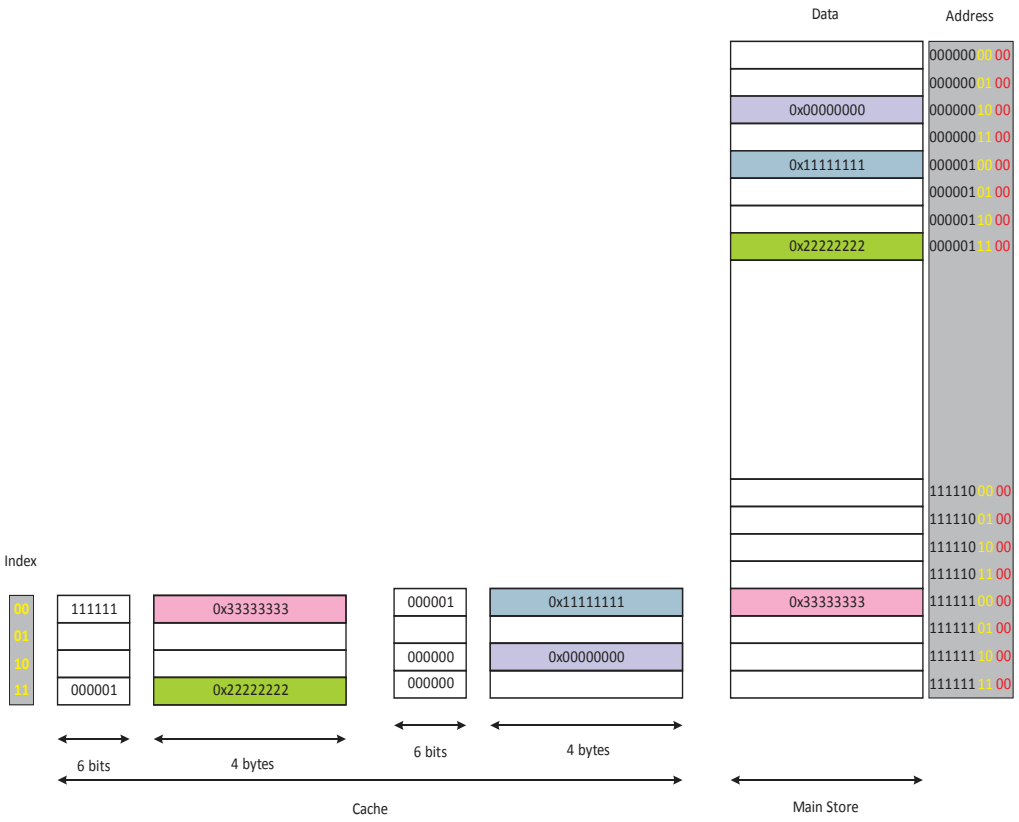


Figure 12. Example 3. Set Associative Cache Mapping

1. The index is extracted from the virtual address.
2. The tag is extracted from the virtual address.
3. The index is used to find the row defining the set of tags that are simultaneously compared to find a match.
   3.1 If a match is found, then corresponding data are delivered to the CPU.
   3.2 If a match is NOT found then a cache entry in the row is replaced by the tag portion of the virtual address and corresponding data retrieved from memory. Data are then delivered to the CPU.

Exercise: A two-way set-associative cache in a system with 24-bit addresses has four 4-byte words per line and a capacity of 1MB. Addressing is to the byte level.

(a) How many bits are there is the index and the tag?

Solution:

1. Compute the number of rows in the cache. A row consists of two (2) tags and two (2) corresponding entries, each occupying 8 bytes. The capacity of a cache excludes the space required to store tags, valid bits, or dirty bits. The capacity of cache only includes that storage copied from main store. For this exercise each row has two (2) 8-byte entries. Index in this exercise has the same meaning as block. The size of the block (or index) is the cache capacity divided by the size of all entries in a row. Let r be the number rows.

$$r = \frac{1\ MB}{(8+8)B} = \frac{2^{20}B}{2^4 B} = 2^{16}$$

Let i be the number of bits in the block (or index).

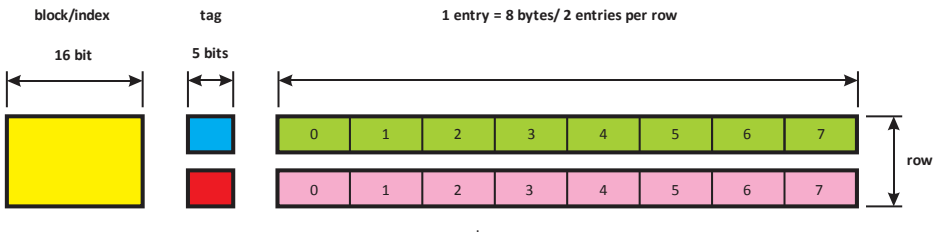
$$i = \log_2 r = \log_2 2^{16} = 16$$

To compute the size of the tag, we must subtract the size of the block (index) and the size of the offset. Let o be the size of the offset and e be the size of a cache entry.

$$e = 8 = 2^3$$
$$o \log_2 e = \log_2 2^3 = 3$$

Let v be the size of the virtual address and t be the size of the tag.

$$v = 24$$
$$t = v - i - o = 24 - 16 - 3 = 5$$



A row in the two-way set-associative cache

(b) Indicate the value of the index in hexadecimal for cache entries from the following main store addresses in hexadecimal: F8C00F, 14AC89, 48CF0F, and 3ACF05.

Solution:

F	8	C	0	0	F		1	1	1	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1
												0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1			
												1				**8**				**0**				**1**						

1	4	A	C	8	9		0	0	0	1	0	1	0	0	1	0	1	0	1	1	0	0	1	0	0	0	1	0	0	1
												1	0	0	1	0	1	0	1	1	0	0	1	0	0	0	1			
												9				**5**				**9**				**1**						

4	8	C	F	0	F		0	1	0	0	1	0	0	0	1	1	0	0	1	1	1	1	0	0	0	0	1	1	1	1
												0	0	0	1	1	0	0	1	1	1	1	0	0	0	0	1			
												1				9				E				1						

3	A	C	F	0	5		0	0	1	1	1	0	1	0	1	1	0	0	1	1	1	1	0	0	0	0	0	1	0	1
												0	1	0	1	1	0	0	1	1	1	1	0	0	0	0	0			
												5				9				E				0						

(c) Can all of the cache entries from part (b) be in the cache simultaneously?

6.4.2 Replacement Policies

- **In the event of a cache-miss, a cache entry replaced.**
- **Optimal –**
  - All entries that will be needed again soon are kept.
  - All entries that will not be needed again soon are discarded.
  - An algorithm that could look into the future to determine the precise entries to keep or discard based on the foregoing criteria would be *optimal*.
  - An optimal replacement policy is impossible to implement because we cannot know the future.
- **Least recently used (LRU)**
  - LRU is based on temporal locality.
  - Record the time each block was most recently accessed.
  - Sort the blocks from most recently used to least recently used.
  - Update the list after every memory reference.
  - When a cache entry must be replaced, discard the least recently used entry.
  - LRU is too costly to implement.
- **First-in, first-out (FIFO)**
  - The cache entry that has been in the cache for the longest time is replaced.
- **Random**
  - Degenerate situations occur employing LRU and FIFO cause the cache to thrash.
  - Thrashing is the state where a cache entry is discarded only to be restored again. This cycle is repeated to the detriment of better performance.
  - A random cache replacement policy may cure thrashing but at the cost of poorer overall performance compared to FIFO.
  - Random is difficult to do.

6.4.3 Effective Access Time and Hit Ratio

Symbol	Definition
$\boldsymbol{P_C}$	Probability that the address referenced is in *cache*
$\boldsymbol{P_M}$	Probability that the address referenced is in *main* memory.
$\boldsymbol{T_C}$	Cache access time
$\boldsymbol{T_M}$	Main store access time
$\boldsymbol{T_D}$	Hard disk access time
$\boldsymbol{T_{eff}}$	Memory system effective access time

Example 1. Find the Effective Access Time (EAT or $\boldsymbol{T_{eff}}$) for a memory system consisting of a cache having an access time of 10ns and a main store access time of 200ns and the cache hit ratio is 0.99.

$$\boldsymbol{T_{eff}} = \boldsymbol{P}(\boldsymbol{Cache\ Hit}) \times \boldsymbol{T_C} + \boldsymbol{P}(\boldsymbol{Cache\ Miss}) \times \boldsymbol{T_M}$$
$$\boldsymbol{T_{eff}} = 0.99 \times 10ns + (1 - 0.99) \times 200ns = 11.9ns$$

Example 2.

Symbol	Typical Value	Definition
$\boldsymbol{P_C}$	0.95	Probability that the address referenced is in *cache*
$\boldsymbol{P_M}$	$1 - 5 \times 10^{-7}$	Probability that the address referenced is in *main* memory.
$\boldsymbol{T_C}$	2 ns	Cache access time
$\boldsymbol{T_M}$	10 ns	Main store access time

T_D	13 ms	Hard disk access time
T_{eff}		Memory system effective access time

1. Simple: assume that the memory system only includes cache and main store. Assume that $P_M = 0.05$

$$T_{eff} = P_C T_C + (1 - P_C) T_M$$
$$T_{eff} = 0.95 \times 2ns + (1 - 0.95) \times 10ns$$
$$T_{eff} = 1.9ns + 0.5ns = 2.4ns$$

2. Complex: assume that the memory system includes cache, main store and disk.

$$T_{eff} = P_C T_C + (1 - P_C) T_{Meff}$$
$$T_{Meff} = P_M T_M + (1 - P_M) T_D$$

$$T_{Meff} = (1 - 5 \times 10^{-7}) \times 10ns + 5 \times 10^{-7} \times 13ms$$
$$T_{Meff} = (1 - 5 \times 10^{-7}) \times 10ns + 5 \times 10^{-7} \times 13ms = 16.5ns$$

$$T_{eff} = P_C T_C + (1 - P_C) T_{Meff}$$
$$T_{eff} = 0.95 \times 2ns + 0.05 \times 16.5ns = 2.725ns$$

6.4.4 When Does Caching Break Down?

- Object oriented programming
  - Many small member functions calling each other. Each time a function is called, a new locality is created.
- Two dimensional array access

6.4.5 Cache Write Policies

- **Write-through**
  - A write-through policy updates both the cache and the main store simultaneous on every write.
  - Slower than write-back
- **Write-back**
  - A write-back policy only updates main store when a cache entry must be discarded and its dirty-bit has been set.
  - A dirty bit is assigned to every cache entry such that if the data in the entry is assigned a value (overwritten) the dirty bit is set signaling that the new data must be copied back to main store.

6.4.6 Instruction and Data Caches

- **Unified Cache or Integrated Cache**
  - A single cache that is used to store both instructions and data
- **Harvard Cache**
  - Two caches –
    - Data Cache
    - Instruction Cache
  - Separating the caches increases the probability of a cache hit (at least in the instruction cache).

6.4.7 Levels of Cache

- L1 Cache –
  - Access time: 4ns
  - Size: 8 - 64 Kbytes
  - Resident on the processor chip
- L2 Cache
  - Access time: 15 – 20ns
  - Size: 64 – KB to 2MB.
  - External to processor chip