# CSCI 4717: Direct Mapping Cache Assignment

As you will recall, we discussed three cache *mapping functions*, i.e., methods of addressing to locate data within a cache.

* Direct
* Full Associative
* Set Associative

Each of these depends on two facts:

* RAM is divided into blocks of memory locations. In other words, memory locations are grouped into blocks of 2n locations where n represents the number of bits used to identify a word within a block. These n bits are found at the least-significant end of the physical address. The image below has n=2 indicating that for each block of memory, there are 22 = 4 memory locations.



Therefore, for this example, the least two significant bits of an address indicate the location within a block while the remaining bits indicate the block number. The table below shows an example with a 20 bit address with four words per block. Notice that for each group of four words, the word bits take on each of the four possible values allowed with 2 bits while the block identification bits remain constant.,

|  |  |  |  |
| --- | --- | --- | --- |
| Block | Address | Block identification bits | Word bits |
| Block 0 | 0x00000 | 0000 0000 0000 0000 00 | 00 |
| 0x00001 | 0000 0000 0000 0000 00 | 01 |
| 0x00002 | 0000 0000 0000 0000 00 | 10 |
| 0x00003 | 0000 0000 0000 0000 00 | 11 |
| Block 1 | 0x00004 | 0000 0000 0000 0000 01 | 00 |
| 0x00005 | 0000 0000 0000 0000 01 | 01 |
| 0x00006 | 0000 0000 0000 0000 01 | 10 |
| 0x00007 | 0000 0000 0000 0000 01 | 11 |
| Block 2 | 0x00008 | 0000 0000 0000 0000 10 | 00 |
| 0x00009 | 0000 0000 0000 0000 10 | 01 |
| 0x0000A | 0000 0000 0000 0000 10 | 10 |
| 0x0000B | 0000 0000 0000 0000 10 | 11 |
| Block 3 | 0x0000C | 0000 0000 0000 0000 11 | 00 |
| 0x0000D | 0000 0000 0000 0000 11 | 01 |
| 0x0000E | 0000 0000 0000 0000 11 | 10 |
| 0x0000F | 0000 0000 0000 0000 11 | 11 |
| And so on...until we get to the last row |
| Block 2n-1 | 0xFFFFC | 1111 1111 1111 1111 11 | 00 |
| 0xFFFFD | 1111 1111 1111 1111 11 | 01 |
| 0xFFFFE | 1111 1111 1111 1111 11 | 10 |
| 0xFFFFF | 1111 1111 1111 1111 11 | 11 |

* The cache is organized into *lines*, each of which contains enough space to store exactly one block of data and a tag uniquely identifying where that block came from in memory.

As far as the mapping functions are concerned, the book did an okay job describing the details and differences of each. I, however, would like to describe them with an emphasis on how we would model them using code.

## Direct Mapping

Remember that direct mapping assigned each memory block to a specific line in the cache. If a line is all ready taken up by a memory block when a new block needs to be loaded, the old block is trashed. The figure below shows how multiple blocks are mapped to the same line in the cache. This line is the only line that each of these blocks can be sent to. In the case of this figure, there are 8 bits in the block identification portion of the memory address.



The address for this example is broken down something like the following:

|  |  |  |
| --- | --- | --- |
| Tag | 8 bits identifying line in cache | word id bits |

Once the block is stored in the line of the cache, the tag is copied to the tag location of the line.

## Direct Mapping Summary

The address is broken into three parts: (s-r) MSB bits represent the tag to be stored in a line of the cache corresponding to the block stored in the line; r bits in the middle identifying which line the block is always stored in; and the w LSB bits identifying each word within the block. This means that:

* The number of addressable units = 2s+w words or bytes
* The block size (cache line width not including tag) = 2w words or bytes
* The number of blocks in main memory = 2s (i.e., all the bits that are not in w)
* The number of lines in cache = m = 2r
* The size of the tag stored in each line of the cache = (s - r) bits

Direct mapping is simple and inexpensive to implement, but if a program accesses 2 blocks that map to the same line repeatedly, the cache begins to thrash back and forth reloading the line over and over again meaning misses are very high.

## Full Associative Mapping

In full associative, any block can go into any line of the cache. This means that the word id bits are used to identify which word in the block is needed, but the tag becomes all of the remaining bits.

|  |  |
| --- | --- |
|                 Tag                 | word id bits |

## Full Associative Mapping Summary

The address is broken into two parts: a tag used to identify which block is stored in which line of the cache (s bits) and a fixed number of LSB bits identifying the word within the block (w bits). This means that:

* The number of addressable units = 2s+w words or bytes
* The block size (cache line width not including tag) = 2w words or bytes
* The number of blocks in main memory = 2s (i.e., all the bits that are not in w)
* The number of lines in cache is not dependent on any part of the memory address
* The size of the tag stored in each line of the cache = s bits

## Set Associative Mapping

This is the one that you really need to pay attention to because this is the one for the homework. Set associative addresses the problem of possible thrashing in the direct mapping method. It does this by saying that instead of having exactly one line that a block can map to in the cache, we will group a few lines together creating a ***set***. Then a block in memory can map to any one of the lines of a specific set. There is still only one set that the block can map to.



Note that blocks 0, 256, 512, 768, etc. can only be mapped to one set. Within the set, however, they can be mapped associatively to one of two lines.

The memory address is broken down in a similar way to direct mapping except that there is a slightly different number of bits for the tag (s-r) and the set identification (r). It should look something like the following:

|  |  |  |
| --- | --- | --- |
| Tag (s-r bits) | set identifier (r bits) | word id (w bits) |

Now if you have a 24 bit address in ***direct mapping*** with a block size of 4 words (2 bit id) and 1K lines in a cache (10 bit id), the partitioning of the address for the cache would look like this.

Direct Mapping Address Partitions

|  |  |  |
| --- | --- | --- |
| Tag (12 bits) | line identifier (10 bits) | word id (2 bits) |

If we took the exact same system, but converted it to 2-way ***set associative mapping*** (2-way meaning we have 2 lines per set), we'd get the following:

|  |  |  |
| --- | --- | --- |
| Tag (13 bits) | line identifier (9 bits) | word id (2 bits) |

Notice that by making the number of sets equal to half the number of lines (i.e., 2 lines per set), one less bit is needed to identify the set within the cache. This bit is moved to the tag so that the tag can be used to identify the block within the set.

## Homework Assignment

Your assignment is to simulate a 4K ***direct mapping cache*** using C. The memory of this system is divided into 8-word blocks, which means that the 4K cache has 4K/8 = 512 lines. I've given you two function declarations in C. In addition, I've given you two arrays, one representing chars stored in main memory and one representing the lines of the cache. Each line is made up of a structure called "cache\_line". This structure contains all of the information stored in a single line of the cache including the tag and the eight words (one block) contained within a single block.

typedef struct
{
    int tag;
    char block[8];
}cache\_line;

An array is made up of these lines called "cache" and it contains 512 lines.

cache\_line cache[512];

Next, a memory has been created called "memory". It contains 64K bytes.

char memory[65536];

There will also be two global variables, int number\_of\_requests and int number\_of\_hits. The purpose of these variables will be discussed later.

Using this information, you should be able to see that there are 16 bits in a memory address (216 = 65,536), 3 bits of which identify a word (char) within a block, 9 bits identify the line that a block should be stored in, and the remaining 16-3-9=4 bits to store as the tag. Each of these arrays are global so your routines will simply modify them.

You will be creating two functions with the following prototypes. (Be sure to use the exact prototypes shown below.)

int requestMemoryAddress(unsigned int address);
unsigned int getPercentageOfHits(void);

requestMemoryAddress() takes as its parameter "address" a 16-bit value (0 to 65,535) and checks if the word it points to in memory exists in the cache. If it does, return the value found in the cache. If it doesn't, load the appropriate line of the cache with the requested block from memory[] and return a -1 (negative one).

Each time a request is made to requestMemoryAddress(), increment number\_of\_requests, and each time one of those requests results in a TRUE returned, increment number\_of\_hits. These values will be used in ***getPercentageOfHits()***. This routine is called to see how well the cache is performing. It should return an integer from 0 to 100 representing the percentage of successful hits in the cache. The equation should be something like:

                          number\_of\_hits
    Percentage of hits = -------------------- X 100%
                          number\_of\_requests

The data that ends up in cache[] from memory[] won't affect the returned values of either function, but I will be examining it to see if the correct blocks of data were moved into the cache[] array.

Well, have fun! Don't hesitate to read the book. There's a great deal of useful information in there including examples and descriptions of the different mapping functions.

## Notes on Cache Memory

### Basic Ideas

The cache is a small mirror-image of a portion (several "lines") of main memory.

* cache is faster than main memory ==> so we must maximize its utilization
* cache is more expensive than main memory ==> so it is much smaller

How do we keep that portion of the current program in cache which maximizes cache utilization.

### Locality of reference

The principle that the instruction currently being fetched/executed is very close in memory to the instruction to be fetched/executed next. The same idea applies to the data value currently being accessed (read/written) in memory.

### So...

If we keep the most active segments of program and data in the cache, overall execution speed for the program will be optimized.  Our strategy for cache utilization should maximize the number of cache read/write operations, in comparison with the number of main memory read/write operations.

### Example

A *line* is an adjacent series of bytes in main memory (that is, their addresses are contiguous). Suppose a line is 16 bytes in size.  For example, suppose we have a 212 = 4K-byte cache with 28 = 256 16-byte lines; a 224 = 16M-byte main memory, which is 212 = 4K times the size of the cache; and a 400-line program, which will not all fit into the cache at once.



Each active cache line is established as a *copy* of a corresponding memory line during execution. Whenever a memory write takes place in the cache, the "Valid" bit is reset (marking that line "Invalid"), which means that it is no longer an exact image of its corresponding line in memory.

### Cache Dynamics

When a memory *read* (or *fetch*) is issued by the CPU:

1. If the line with that memory address is in the cache (this is called a cache *hit*), the data is read from the cache to the MDR.
2. If the line with that memory address is not in the cache (this is called a *miss*), the cache is updated by replacing one of its active lines by the line with that memory address, and then the data is read from the cache to the MDR.

When a memory *write* is issued by the CPU:

1. If the line with that memory address is in the cache, the data is written from the MDR to the cache, and the line is marked "invalid" (since it no longer is an image of the corresponding memory line).
2. If the line with that memory address is not in the cache, the cache is updated by replacing one of its active lines by the line with that memory address. The data is then written from the MDR to the cache and the line is marked "invalid."

Cache updating is done in the following way.

1. A candidate line is chosen for replacement using an algorithm that tries to minimize the number of cache updates throughout the life of the program run. Two algorithms have been popular in recent architectures:

- Choose the line that has been least recently used - "LRU" for short (e.g., the PowerPC)

- Choose the line randomly (e.g., the 68040)

1. If the candidate line is "invalid," write out a copy of that line to main memory (thus bringing the memory up to date with all recent writes to that line in the cache).
2. Replace the candidate line by the new line in the cache.

### Mapping Memory Lines to Cache Lines - Three Strategies

As a working example, suppose the cache has 27 = 128 lines, each with 24 = 16 words. Suppose the memory has a 16-bit address, so that 216 = 64K words are in the memory's address space.



#### Direct Mapping

Under this mapping scheme, each memory line j maps to cache line j mod 128 so the memory address looks like this:



Here, the "Word" field selects one from among the 16 addressable words in a line. The "Line" field defines the cache line where this memory line should reside. The "Tag" field of the address is is then compared with that cache line's 5-bit tag to determine whether there is a hit or a miss. If there's a miss, we need to swap out the memory line that occupies that position in the cache and replace it with the desired memory line.

E.g., Suppose we want to read or write a word at the address 357A, whose 16 bits are 0011010101111010. This translates to Tag = 6, line = 87, and Word = 10 (all in decimal). If line 87 in the cache has the same tag (6), then memory address 357A is in the cache. Otherwise, a miss has occurred and the contents of cache line 87 must be replaced by the memory line 001101010111 = 855 before the read or write is executed.

Direct mapping is the most efficient cache mapping scheme, but it is also the least effective in its utilization of the cache - that is, it may leave some cache lines unused.

#### Associative Mapping

This mapping scheme attempts to improve cache utilization, but at the expense of speed. Here, the cache line tags are 12 bits, rather than 5, and any memory line can be stored in any cache line. The memory address looks like this:



Here, the "Tag" field identifies one of the 2 12 = 4096 memory lines; all the cache tags are searched to find out whether or not the Tag field matches one of the cache tags. If so, we have a hit, and if not there's a miss and we need to replace one of the cache lines by this line before reading or writing into the cache. (The "Word" field again selects one from among 16 addressable words (bytes) within the line.)

For example, suppose again that we want to read or write a word at the address 357A, whose 16 bits are 0011010101111010. Under associative mapping, this translates to Tag = 855 and Word = 10 (in decimal). So we search all of the 128 cache tags to see if any one of them will match with 855. If not, there's a miss and we need to replace one of the cache lines with line 855 from memory before completing the read or write.  The search of all 128 tags in the cache is time-consuming. However, the cache is fully utilized since none of its lines will be unused prior to a miss (recall that direct mapping may detect a miss even though the cache is not completely full of active lines).

#### Set-associative Mapping

This scheme is a compromise between the direct and associative schemes described above. Here, the cache is divided into sets of tags, and the set number is directly mapped from the memory address (e.g., memory line j is mapped to cache set j mod 64), as suggested by the diagram below:



The memory address is now partitioned to like this:



Here, the "Tag" field identifies one of the 26 = 64 different memory lines in each of the 26 = 64 different "Set" values. Since each cache set has room for only two lines at a time, the search for a match is limited to those two lines (rather than the entire cache). If there's a match, we have a hit and the read or write can proceed immediately. Otherwise, there's a miss and we need to replace one of the two cache lines by this line before reading or writing into the cache. (The "Word" field again select one from among 16 addressable words inside the line.)

In set-associative mapping, when the number of lines per set is *n*, the mapping is called *n*-way associative.  For instance, the above example is 2-way associative.

E.g., Again suppose we want to read or write a word at the memory address 357A, whose 16 bits are 0011010101111010. Under set-associative mapping, this translates to Tag = 13, Set = 23, and Word = 10 (all in decimal). So we search only the two tags in cache set 23 to see if either one matches tag 13. If so, we have a hit. Otherwise, one of these two must be replaced by the memory line being addressed (good old line 855) before the read or write can be executed.

## A Detailed Example

Suppose we have an 8-word cache and a 16-bit memory address space, where each memory "line" is a single word (so the memory address need not have a "Word" field to distinguish individual words within a line).  Suppose we also have a 4x10 array a of numbers (one number per addressible memory word) allocated in memory column-by-column, beginning at address 7A00. That is, we have the following declaration and memory allocation picture for the array a:

float [][] a = new float [4][10];



Here is a simple equation that recalculates the elements of the first row of a:



This calculation could have been implemented directly in C/C++/Java as follows:

Sum = 0;

for (j=0; j<=9; j++)

    Sum = Sum + a[0][j];

Ave = Sum / 10;

for (i=9; i>=0; i--)

    a[0][i] = a[0][i] / Ave;

The emphasis here is on the underlined parts of this program which represent memory read and write operations in the array a. Note that the 3rd and 6th lines involve a memory read of a[0][j] and a[0][i], and the 6th line involves a memory write of a[0][i].  So altogether, there are 20 memory reads and 10 memory writes during the execution of this program.  The following discussion focusses on those particular parts of this program and their impact on the cache.

### Direct Mapping

Direct mapping of the cache for this model can be accomplished by using the rightmost 3 bits of the memory address. For instance, the memory address 7A00 = 0111101000000 000, which maps to cache address 000. Thus, the cache address of any value in the array a is just its memory address modulo 8.

Using this scheme, we see that the above calculation uses only cache words 000 and 100, since each entry in the first row of a has a memory address with either 000 or 100 as its rightmost 3 bits.

The hit rate of a program is the number of cache hits among its reads and writes divided by the total number of memory reads and writes.  There are 30 memory reads and writes for this program, and the following diagram illustrates cache utilization for direct mapping throughout the life of these two loops:



Reading the sequence of events from left to right over the ranges of the indexes i and j, it is easy to pick out the hits and misses. In fact, the first loop has a series of 10 misses (no hits). The second loop contains a read and a write of the same memory location on each repetition (i.e., a[0][i] = a[0][i]/Ave; ), so that the 10 writes are guaranteed to be hits.  Moreover, the first two repetitions of the second loop have hits in their read operations, since *a*09 and *a*08 are still in the cache at the end of the first loop. Thus, the hit rate for direct mapping in this algorithm is 12/30 = 40%

### Associative Mapping

Associative mapping for this problem simply uses the entire address as the cache tag. If we use the least recently used cache replacement strategy, the sequence of events in the cache after the first loop completes is shown in the left-half of the following diagram. The second loop happily finds all of *a* 09 - *a*02 already in the cache, so it will experience a series of 16 hits (2 for each repetition) before missing on *a* 01 when i=1. The last two steps of the second loop therefore have 2 hits and 2 misses.



The hit rate for associative mapping the cache for this algorithm is therefore 18/30 = 60%. We see that this is an improvement over direct mapping, since all 8 words in the cache are now being utilized to achieve a higher hit rate. However, associative mapping takes more time per read/write because every cache address must be searched to see if its current tag matches the desired memory address.

### Set-Associative Mapping

Set associative mapping tries to compromise these two. Suppose we divide the cache into two sets, distinguished from each other by the rightmost bit of the memory address, and assume the least recently used strategy for cache line replacement. Cache utilization for our program can now be pictured as follows:



Notice here that all entries in a that are referenced in this algorithm have even-numbered addresses (their rightmost bit = 0), so only the top half of the cache is utilized. The hit rate is therefore slightly worse than associative mapping and slightly better than direct. That is, set-associative cache mapping for this program yields 14 hits out of 30 read/writes for a hit rate of 46%.

With the help of direct access you can access data store on your storage media or RAM directly from that memory address.
In case of sequential access head search or reach at the memory address from start address of your storage device.
Direct access search given address fast compared to sequential access.
In Direct access head do to direct to that location.

Read more: <http://wiki.answers.com/Q/What_is_the_difference_between_random_access_and_direct_access#ixzz1J4rSwCYs>

File constructed in a manner in which records may be placed in a random order; also called *direct access file.* Each record in a random access file has associated with it a relative index number. Whenever a record is read from a random access file, a computer program must produce a relative index number for this record in order to locate the record in the file. This type of file design offers the following
advantages:
(1) it provides rapid access to the desired information. In a decision-making environment where information is needed quickly, random access is a requisite to rapid retrieval;
(2) it is efficient for retrieving a relatively few records at a time; and
(3) it provides a method of keeping files up to date as transactions or events occur

Read more: <http://wiki.answers.com/Q/What_is_difference_between_random_file_access_and_sequencial_flle_access_in_c#ixzz1J4t8h3sh>

During sequential access, if you want to access the nth byte of a file, you must read all bytes 0 through n. During random file access, you may skip directly to the nth byte without having to read through all previous bytes.

Direct access means going straight to the record you want,and random access means pick data randomly and then find that data which you required.

Read more: <http://wiki.answers.com/Q/What_are_the_differences_between_direct_access_and_random_access#ixzz1J54hVchv>





Sequential access compared to [random access](http://www.answers.com/topic/random-access).

In [computer science](http://www.answers.com/topic/computer-science), **sequential access** means that a group of elements (e.g

Read more: <http://www.answers.com/topic/sequential-access#ixzz1J57wLSE9>

With the help of direct access you can access data store on your storage media or RAM directly from that memory address. In case of sequential access head search or reach at the memory address from...

Read more: <http://wiki.answers.com/Q/What_is_difference_between_random_file_access_and_sequential_file_access_in_java#ixzz1J58KYyIU>

|  |
| --- |
| Principle of locality relate to the use of multiple memory levels? |

] 

**Temporal Locality**: Concept that a resource will be referenced at one point in time will be referenced again. Cache miss traffic decreases fast when cache size increases and temporal locality determines sensitivity to cache size.
**Spatial Locality**: Concept that likelihood of referencing a resource is higher if a resource near it was referenced. Cache miss traffic does not increase much when line size increases. Spatial locality determines sensivity to line size.

Read more: <http://wiki.answers.com/Q/Principle_of_locality_relate_to_the_use_of_multiple_memory_levels#ixzz1J5Amq5G4>