LSU EE 4720 Cache Notes and Study Guide Updated F 05, Cor F 07

David M. Koppelman

1.1 Introduction

The relationship between cache structure and the binary representation of an effective address is simple and when understood the ≈ third problem on the final exams will be a big chunk of easy credit (except, perhaps for the part d or e's). These notes are intended to help everyone (in EE 4720) get their big chunk of credit. They consist of a description of cache structure, a note-sheet-ready diagram, and a discussion of exam problem types.

1.2 Cache Structure

A typical direct-mapped cache is illustrated below, control logic and connections needed to write the cache are omitted. The illustration uses the same symbols that appear in the class lecture notes, homeworks, and exams. Further below the relationship between cache structure and the address bits is shown.

The **address space size** is the number of bits in an address, denoted a. For older systems, and most examples in class, $a = 32$, for modern systems, $a = 64$.

The `Addr` output port of the illustrated CPU is used for load and store instructions and for instruction fetch. The discussion below is for load and store instructions in which the `Addr` lines will carry the **effective address**, that is, the address to load or store from.

On a load the index bits (see diagrams) of the effective address are used to look up a tag in the **Tag Store** (labeled "Tag" in the illustration) while the combined index and offset bits are used to retrieve the data from the **Data Store**. If the **valid bit** retrieved from the Tag Store is `1` then the data retrieved from the Data Store is valid. (Uninitialized entries are invalid, as are entries removed from the cache [for reasons not covered in class].) The retrieved data might be valid but for a different address. To verify that it's for the address appearing on the CPU `Addr` port the **tag** of the stored data (retrieved from the Tag Store) is compared to the tag bits of `Addr`. If they are the same and the valid bit is `1` there is a **cache hit**. The **alignment network** (labeled `Align`) is used when the size of the data being retrieved is smaller than the width of the bus, w. It puts the requested data in the least significant bits of w and sets the other bits of w to 0 (unsigned request) or 1 (signed request and retrieved data is negative).

Data is managed in units called **blocks** or **lines** (both terms are used). There is one tag for each line in the cache. Symbol l denotes $\log_2$ of the line size. Since each Tag Store entry covers a line and each line is 2^l characters, the lowest address bit used for the Tag Store lookup is at position l.

Symbol w denotes the **bus width**, the number of bits that the CPU can read at once. (A computer may have many busses, w only applies to this particular connection.) Symbol c denotes the **character size**, in bits. (Each address stores one character, on many systems a character is eight bits.) On most examples given in class $c = 8$ and $w = 32$, this allows the CPU to read a 32-bit integer in one step. In the illustrated example $w = 16c = 128$ bits, probably larger than most real systems. The number of alignment bits, d, is $\log_2 \frac{w}{c}$, in the example $d = 4$. The **alignment bits** are the low-order d bits of the address, they are used by the alignment network to determine how much to shift the data by. The lowest bit position used for the Data Store lookup is d, and bits 0 to $d - 1$ are used by the alignment network.

Symbol s denotes $\log_2$ of the number of **sets** in the cache. Sets are numbered from 0 to $2^s - 1$ and the set number for an address is the value of its index bits. In a direct-mapped cache (illustrated) each set holds one line; in an x-way set-associative cache each set holds x lines. All the lines in a set have the same index (by definition of a set) and different tags.

The number of bits used for the Tag Store lookup is s, the index bits. (If this is not obvious re-read the preceding paragraph.) The number of bits used for the Data Store lookup is $s + l - d$. Each entry in the data store is 2^d characters (to match the bus width) and therefore the storage capacity of the Data Store is $2^{s+l-d}2^d = 2^{s+l}$ characters. For a direct-mapped cache, this is the **cache capacity**. For an x-way set-associative cache, which by definition has x Data Stores, the capacity is $x2^{s+l}$.

A special case: If the line size equals the bus width then the bits used for lookup in the Tag and Data store will be identical. (In the illustration, if the line size were changed to 16 characters the bits used for both the Data and Tag Store would be 10:4.) (If you need to memorize this special case you haven't studied enough.)

In an x**-way, set-associative cache** the Tag Store, Data Store, and the logic generating the `Hit` signal are duplicated x times, each duplicate is called a **way**. For a particular access at most one way can hit, a multiplexor selects the data from that way. The cache described earlier is a special case, a **direct-mapped cache**, and might also be called 1-way set-associative. ("Might" and not "is" because since it's 1-way it's not really set-associative, in the same way a 1-way superscalar processor isn't really superscalar.) As stated above, in a set-associative cache each set can hold x lines, one line per way.

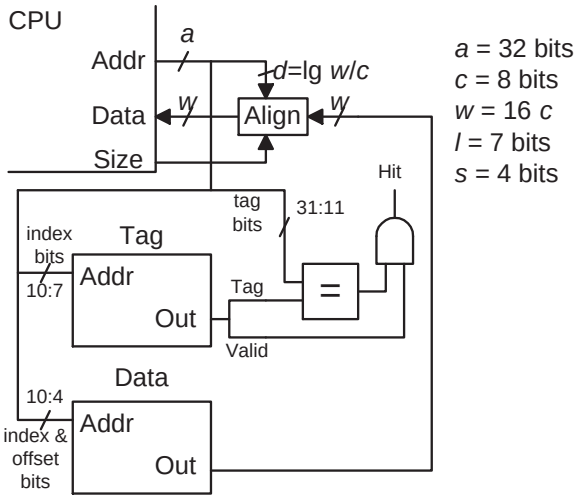
1.3 Cache Illustration, Program, Address Bit Categorization

A simple program, a cache, and an address bit categorization are illustrated below. It is important that the relationship between these is thoroughly understood. The better understood, the easier it is to remember. For more on the program see the *Stride or Two-Dimensional Array Access* section below.

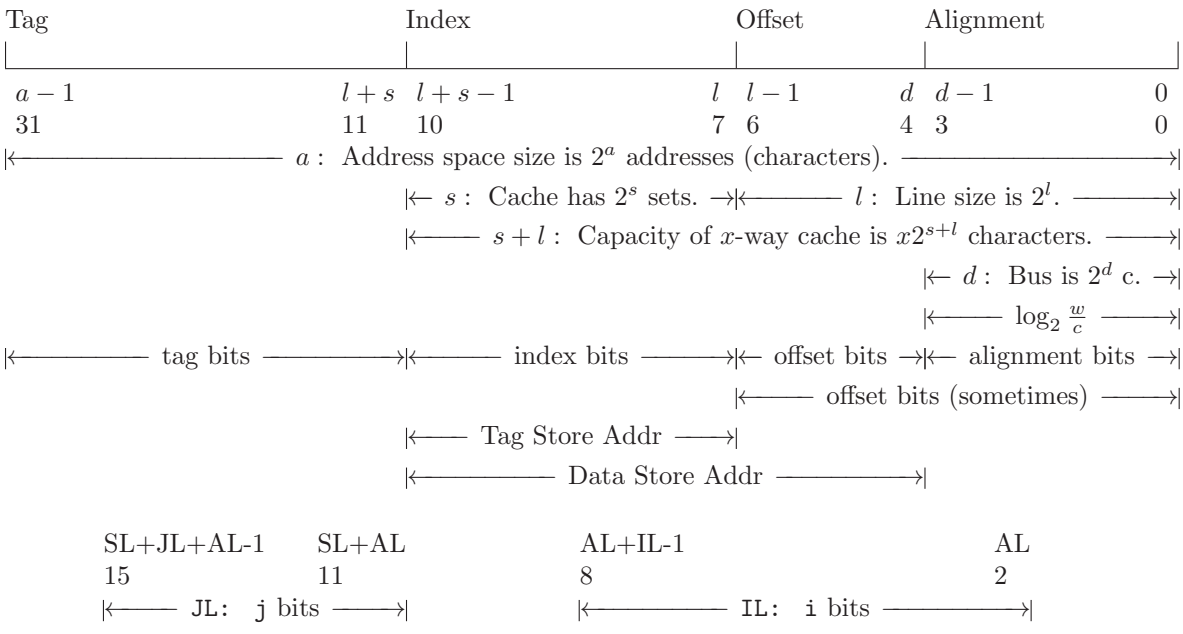
The cache illustrated is direct mapped (1-way set associative). (The address bit categorization shows the capacity of an x-way cache.) There are two definitions of offset bits, the larger one is usually used, an exception is the diagram. The program accesses something which is not quite a two dimensional array. (It would be a 2-D array if `SL` were the same as `IL`.)

```
// sizeof(int) = 4 characters
extern int *a;
int AL = 2; // log-base-2( sizeof(int) )
int SL = 9; // Stride log_2
int STRIDE = 1 << SL; // = 2^9 = 512;
int IL = 7;
int ICOUNT = 1 << IL; // = 2^7 = 128;
int JL = 5;
int JCOUNT = 1 << JL; // = 2^5 = 32;

for ( int i=0; i<ICOUNT; i++ )
   for ( int j=0; j<JCOUNT; j++ )
      sum += a[ i + j * STRIDE ];
```



Address Bits:



1.4 Common Exam Problem Types

Nearly every final exam will have some type of cache question, parts of these questions fall into a few types. The types are explained below along with a list of final exams where they appear.

1.4.1 Cache Structure

Given some information about the cache structure, determine other information. This is usually parts (a) and (b). If these notes are understood such questions can be answered in under a minute. Really, and you don't have to be that smart either. Concentrate on the address bit categorization.

1.4.2 Different Cache, Same Capacity

Cache B is like cache A except for X. Find the Y for cache B. For example: *call the cache illustrated here A and consider a new cache in which the line size is four times as long but the associativity and capacity are the same. Determine s and l and show the bits used for the Tag and Data Store lookups.*

These problems are also fairly easy, the solution usually involves changing some combination of s, l, and x. If the cache capacity of both caches is to be the same then choose x_B, s_B, and l_B such that $x_A 2^{s_A+l_A} = x_B 2^{s_B+l_B}$, where x_A is the associativity of cache A, etc.

Problems: Spring 2002 4a (last part), Fall 2001 3a (last part), Spring 2000 4c Fall 2000, 4 (last part)

1.4.3 Find the Hit Ratio

Find the hit ratio for a cache (usually from an earlier part) while running the given program, usually in C. This type of question is easy if the program reads memory sequentially. To solve it determine the size of the array element (it's given in the comments) and from that determine how many will fit on a line. Access to the first element on a line will miss, the rest will hit.

Problems: A question of this type is asked on almost every final exam. Spring 2003 4b, Fall 2002 4b, Fall 2001 3b (first part), Spring 1999 3b

In one variation on this question the array access is in a nested loop. The hit ratio for the first iteration of the outer loop can be found using the procedure above. For subsequent iterations of the outer loop you need to determine if data brought in on the first iteration is still there.

Problems: Spring 2002 4b (first part)

1.4.4 Stride or Two-Dimensional Array Access

In some problems a two-dimensional array is accessed. (In some cases it is not called a two-dimensional array.) Though the array is logically two dimensional it is declared as an ordinary (one-dimensional) array in C. For example, if `a` is to be a 32×16 array it would be declared as a 512-element array; to access element `i`,`j` the code `a[ i + j * 32 ]` would be used. (In some problems the 32 is replaced with a symbol, often called something like "STRIDE.") In these problems the array dimensions will usually be a power of 2, greatly facilitating the solution. The key to solving a problem of this type is to find which bit positions of the address are affected by `i` and `j`. In the above example, if `i` ranges from 0 to 31 and `a` is a character array then `i` affects bits 0 to 4. If `a` were an integer array and integers were four characters then `i` would affect bits 2 to 6. (BTW, the same thing holds for one-dimensional arrays.) If `a` is a character array and variable `j` ranges from 0 to 15, then `j` would affect bits 5 to 8 (because multiplying `j` by 32 shifts it five bit positions to the left). How the cache behaves depends upon how `i` and `j` overlap the tag, index, and offset bits.

The cache illustration (the big program/cache/bits diagram above) shows an example of how `i` and `j` relate to the address bits, including of course the tag, index, and offset. To make the program example the same as the example just discussed set `AL=0` (for a character array), and `SL=5`. (`IL` and `JL` don't matter.) For the Spring 2001 Problem 3b set `AL=0`, `SL=10`, `IL=5`, and `JL=8` (`i` and `j` are reversed).

Problems: Spring 2001 3b, Fall 1999 2a

1.4.5 Find a Cache Configuration the Maximizes the Hit Ratio

Find an alternate cache configuration, usually of the same capacity, that maximizes the hit ratio (or achieves some other goal). To solve it determine which bits of the address are affected by the loop indices (see the multiple-dimension stuff above), and "move" the index, offset, and tag bits so their overlap achieves the desired result.

Problems: Fall 2002 4c, Fall 1999 2b.

1.4.6 Fill The Cache

In this question write a program or modify a given one to fill the cache with the minimum number of accesses. If it's an ordinary cache this type of question is easy. Set the stride to the line size (taking into account the array element size), the number of iterations should be the number of lines in the cache, $x2^s$.

Problems: Fall 2001 3b (second part), Spring 2000 4b, Spring 1999 3c

1.4.7 Maximize (or Minimize) the Hit Ratio

Modify a given program following some constraint to maximize or minimize the hit ratio. The constraint might be keeping a certain value as small as possible. These problems can be moderately difficult.

Problems: Spring 2002 4b (second part)

1.4.8 Miscellaneous

Modify a program so that a particular array will be evicted from the cache with the minimum number of accesses. Spring 2003 Final, 4c.

Determine line size, associativity and other items from number of misses: Spring 2005 Final problem 4c and 4d.

Other Problems: Spring 2001 3