A Short Tutorial on Addressing of Cache
(Course Material CS_M33, Autumn 2000)

Anton Setzer
30 November 2000

1 Addressable Units, Address Space

On different architectures, the unit which is addressable is different. We introduce the terminology addressable unit for this. By saying an architecture which is addressable up to byte level, we mean for instance that the addressable unit is 1 byte.

The address space is the amount of memory, which can be addressed by the bus. Usually the memory is smaller than the address space. This allows for instance to expand memory later. In all our examples however, the address space coincides with the main memory given.

Since addresses are given by binary codes, the ratio address space/addressable unit will always be a power of two. If it is $2^k$, then addresses consist of $k$ bits.

Examples:

- If the address space is 16 Mbyte (ie. $2^4 \times 2^{10} \times 2^{10}$ byte = $2^{24}$ byte), the word length is 4 byte, and the addressable unit is a word, then this ratio is $2^{22}$, so 22 bits are needed for the addresses.

- If the address space is 1 Gbyte (ie. $2^{10} \times 2^{10} \times 2^{10}$ byte), and the addressable unit is a byte, the address has length 30 bits.

2 Blocks

Since cache is much smaller than main memory, we have to assign to every element of the cache a tag, which gives information about which of the main memory locations, which could be stored at at this position in cache, is actually stored.

If we do this for every addressable unit, then a large part of the data stored in cache will be tags. In order to keep the size of the cache small, we store in cache larger blocks of data. Then we have only to assign a tag to each block. A block will consists of $2^k$ addressable unit for some $k$. The block size depends on the cache, if there is more than one cache involved (for instance for multiple
processors), the block size of each might be different. Observe, that the word block occurs in other contexts as well, where it means something different. Eg. when I/O sends data, this is often done in blocks of variable size.

If we stored arbitrary blocks in cache, we would obtain overlaps, which is difficult to administrate. Instead we divide main memory evenly into blocks. If the block size is 8 addressable units, memory locations 0-7 form the first block, memory locations 8-#15 form the next one (#15 indicates that we refer to a decimal number), etc. Now we can assign to every block an address: The first block gets number 0, the next one block 1 etc. Since the block size in our example is 8 addressable unit, this means that the block 0 is at memory locations 0 - 7, block 1 at memory locations 8 - #15, etc. We see now that block k is at memory location k x 8 + 0 to k x 8 + 7. If we divide an address k x 8 + i (i ∈ {0, ..., 7}) by 8 with remainder, then we obtain k. Since division with remainder is the same as a shift to the right by 3 bits, we obtain that the block address can be obtained in this example from the memory address by taking all but the least significant 3 bits. For instance memory address 11111111111111111010 has block address 11111111111111111111. The least significant 3 bits determine the number of the addressable unit within that block.

In general, if the address is built from k bits and the block size is 2^l addressable units, then we have that the k - l most significant bits form the block address and the remaining l bits determine the number of the addressable unit within that block: The address is divided as follows

\[
\begin{array}{ccc}
\text{block address} & \text{addressable unit within block} \\
\hline
k - l & l & k \\
\end{array}
\]

In the slides, we called the part referring to the addressable unit “word”, since often (but not always!!!) the addressable unit is a word.

3 Direct Mapped Cache

In case of a direct mapped cache, for every block in main memory there is a unique cache line in which it can be mirrored. Since we have locality of variables (references to memory locations tend to be close to each other), it is not efficient to associate consecutive blocks in main memory with the same cache line (each cache line stores one block): We will often refer to variables in consecutive blocks within a small time interval, which means that we have to load one block into cache line, replace it by the other, replace it again by the other, etc. - we don’t save much by using cache memory.

Instead we associate memory blocks with an as large as possible distance to the same cache line. If we have as a toy example a cache with only 16 cache lines, then we would assign therefore memory blocks 0 to 15 consecutively with the cache lines 0 to 15, then 16 to 31 again consecutively with the cache lines 0 to 15, etc.
Now as for the division of the blocks before we can see that in our toy example we obtain the cache line by taking the 4 least significant bits \((2^4 = 16)\). The rest of the address, namely all but the least significant 4 bits, are needed in order to determine for a cache line, which address block it refers to. (If we concatenate the binary representation of this address with the binary representation of a cache line, we obtain the original block address). So we have to attach to a block in memory this information, which is called tag.

In general, if we have \(2^i\) blocks in main memory, and \(2^k\) cache lines (so the size of the cache is \(2^k\) blocks), then the tag will have size \(l - k\) bits.

If we take a memory address, then we can first divide it in block address and “word”-address \(w\), and then divide the block address into tag \(t\) and cache line \(l\). When this address is loaded into main memory, then first the corresponding block is loaded into cache line \(l\) which obtains tag \(t\).

4 Fully-Associative Cache

In case of fully-associative cache all memory blocks can be mapped to any line in cache, so the tag is the block address.

5 Associative Cache

In case of associative cache, every block is associated with a set of cache lines. The size of this set is fixed, if it is \(2^k\) we say that we have a \(2^k\)-way associative cache. If we have again our toy example with cache size 16 blocks, but now 2-way set associative, than we have 8 sets of 2 cache lines. Block 0 can be mapped into one line of set 0, block 1 can be mapped into one line of set 1, etc. and block 8 can be mapped into one line in set 0, again. So the set number in which a block can be mapped is obtained from the block number by taking in this case the 3 least significant bits. In the general situation, if we have cache size \(2^l\), and a \(2^k\)-way set-associative cache, we have \(2^{l-k}\) sets, and the \(l - k\) least significant bits of a binary block address determine the set number. The remaining bits of the block address determine again the tag associated with the block, if it is stored in cache.