I will assume that you are familiar with what caches are and why they are used. This paper will contain an explanation of the types of organizational schemas of caches.

We begin with a very simple view of the cache: caches are divided into **blocks** and each block may be of a various size. The number of blocks in a cache is usually a power of 2. For now, we will assume that each block of the cache contains only one byte.

Thus, if we were to depict what I have just said, our current cache would look like this:

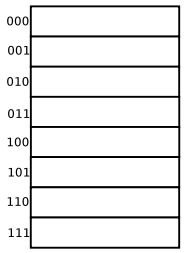

Figure 1: A cache example

As you can see, this cache contains eight blocks and each block can accomodate, as stated, 1 byte (or 8 bits) of information.

The next question is very important: *Where should we put data in the cache?* This question and its answers relates directly to the organizational schemas of the caches.

I will begin with the simplest approach: direct mapping — each main memory address maps to exactly one cache block. The following figure depicts the idea:

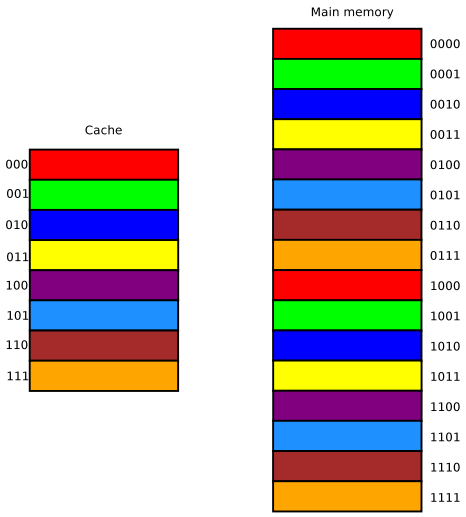

Figure 2: Direct mapping

Here we had a mapping of a 16 byte memory to an 8 byte cache. The direct

mapped cache can store specific lines of memory into the same cache block. For example, the address 'b0000 will always go into the cache block with index 'b000. (the colouring may help you see the designated cache blocks for every main memory line) This organizational schema directly maps a memory line into a specific cache block, hence the name **direct mapping**.

How can we compute this mapping? Well, if you think about it, what we have done is that we have logically split the main memory into chunks, each chunk having the size of the whole cache. (in our example we have divided the 16 byte main memory into 2 chunks of 8 bye — the size of the cache). We usually call this chunks **cache pages** and the size of such a page varies from one organizational schema to another. In the case of direct mapping, the cache page size is equal to the cache size. You can see that the mapping is cyclic: the first 8 bytes are mapped in a unique way, then the 9th one is mapped to the place of the first one, the 10th one is mapped to the place of the second one and so forth. One way to figure out which cache block a particular memory address should go to is to use the mod operator. Thus, if the cache contains 2^k blocks, then the data at memory address i would go to the cache block with index $(i \mod 2^k)$.

More easily, if we want to do this operation directly in the hardware it is enough to take the least significant k bits of a binary value to compute the value $\mod 2^k$. Just look in the example: memory address 'b0000 maps to cache block 'b000 and so does memory address 'b1000. It is all about the least significant bits.

One problem that arrises is that other addresses might also map to the same cache block. As we have seen, that is the case of main memory addresses 'b0000 and 'b1000. How can we distinguish between them? The answer is fairly simple: notice that the lest significant *3* bits are the same in the case of these 2 addresses so we can distinguish between them using the remaining bits: the first one. This is the concept of **tag**. For a cache which has 2^k blocks, each containg 1 byte of data, each main memory address of length m bits will be split in 2 parts: **the index** part which is the least significant k bits of the address which will help us locate the cache block this memory address corresponds to and **the tag** part which is the most significant m-k bits which will help us distinguish between the memory addresses that map to the exact same cache block.

Thus, a cache block entry would look like this:

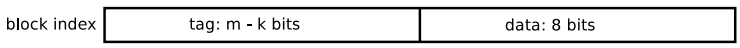

Figure 3: Cache block entry

This is what happens when the processor request an address: it will first look at the cache. The least significant k bits of the address requested are used as an index to find the appropiate cache block and then the most significant m-k bits (**the tag**) are compared to the tag value that is already present in the tag part of the cache block. If these 2 values are equal, we have a **cache hit**, otherwise a **cache miss**.

It is obviously clear that this organizational schema requires some additional information to be kept inside a cache block: the tag. Also, this schema is very inflexible when it comes to cache block replacement policy: by having a single location for a cache line, we do not exploit *time locality* — no matter how much a

cache block has been used, if another memory address is requested that maps to the same cache block, the previous data that was held there will be overwritten.

It is now time to also talk about *space locality*. Think about our initial assumption: that each cache block will store just 1 byte of data. This defeats the purpose of *spatial locality* since we do not copy in the cache values that are next to the value requested — we only copy the value requested. Solution? Again, simple: what we can do is make the cache block retain more than 1 byte. Let's say that the each cache block can retain n bytes. The following figure depicts this:

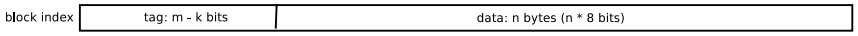

Figure 4: Cache block entry 2

What exactly have we done here? We can now logically split the main memory into chunks, each chunk being of size n *bytes*. Usually, n is a power of two and this size of n *bytes* is called **cache line**. It is time for **block addresses**. Recall that we have split the main memory into **blocks** of size n bytes. How many such blocks are there? The answer is simple: the total size of the memory divided by the size of the cache line. How do we find out the block address of a main memory address i? The answer is again simple: we do the integer division i / n. The following figure depicts an example:

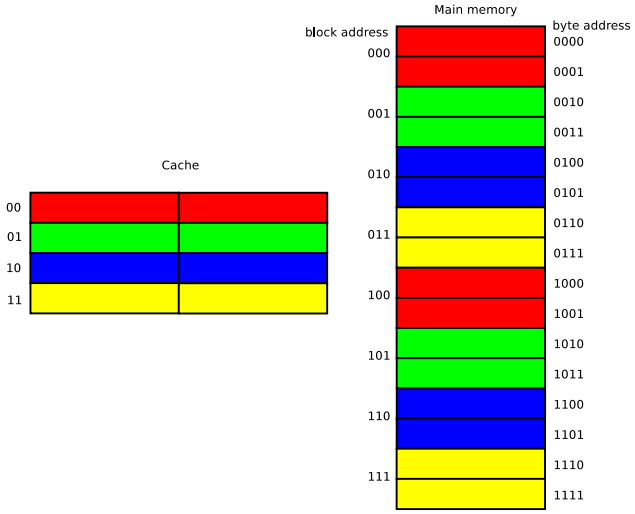

Figure 5: Direct mapping with bigger cache line

Here we have a cache that has 4 blocks and can accomodate 2 bytes in each block. (the tag is left out for simplicity purposes). This means that the main memory can be divided into *blocks* of size 2 bytes each. What is different here is that we do not do direct mapping on byte addresses, but on block addresses. Given a main memory address, for example 'b1101 (that is 13 in decimal) we can find its block address by dividing 13 with 2, which gives us 6 ('b110 in binary). If n (the size of the cache line) is a power of two, let's say 2^b then the integer division would just mean to ignore the first b bits in the address. In our

case n was equal to 2 so b was 1 and you can see clearly that the set of the block addresses are obtained from erasing the trailing bit of the set of the byte addresses. Once we know the block address we can map it as before: by doing the modulo division with the number of cache blocks. (or cache entries if you prefer). Thus, to continue our example, if we have block address 'b110 (6 in decimal) and 4 cache entries, the remainder will be 2. ('b10 — this can be also obtained by taking the least significant 2 bits out of the block address).

So: a short recap — how to arrive from byte address to cache entry index? Given a cache with 2^k entries, each entry consisting of n bytes where n is a power of 2 of the form 2^b, and a main memory address width of m bits, we can obtain the main memory block address by taking the most significant m - b bits from the original address, and then we can find the index of the cache entry by taking the least significant k bits of of these m - b.

Thus, when we access one byte of data in memory, we will copy its entire block(cache line) into the cache, to hopefully take advantage of spatial locality. But now we store more than one byte in each cache line! How can we refer to a single byte (the one requested)? Remember that the cache line size is n or 2^b. We can use b bits to address each individual byte in the cache line. Luckily for us, those b are the lease significant bits of the original address. We call this the **offset** and we use it to locate the requested byte inside the cache line. To make things simpler, byte i of a memory block is always stored in byte i of the corresponding cache line. We should not forget about the **tag** as well. Thus, when an address arrives at the cache controller it is split as follows:

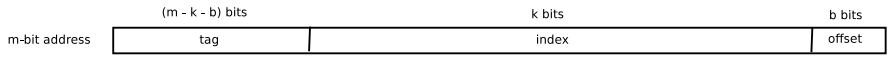

Figure 6: Logical split of a memory address

Remember that 2^b is the size (in bytes) of the cache line, 2^k is the number of cache entries and m is the size in bits of the memory address.

Now: what I have just explained is not exactly accurate. We can not address individual bytes in a machine. We can only address words: and the word size may vary. If, for example a words contains 4 bytes and a cache line contains 64 bytes (which, by the way, is usual nowadays), then you do not need 6 bits to address the words. You only need 4. Thus, the trailing 2 bits of the address are unused. Very sad for them. This is the final figure which depicts accurately how an address is logically split in direct mapping.

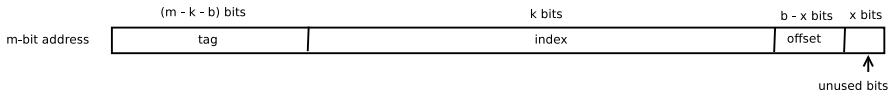

Figure 7: Logical split of a memory address 2

Here, the meanings are just the same, but the new introduced variable x has the property that 2^x = *sizeof(word)*.

We have finally solved the problem of **spatial locality**! What about the **temporal locality**? What happens if a program uses addresses (in decimal) 2, 6, 2, 6, ...? We would constantly need to rewrite the cache because there is only

one location in the cache that corresponds to each address. We need a solution to address the **temporal locality** as well.

Another proposed organizational schema is a type of cache called**fully associative**. Here, we permit the data block to be placed in any available cache block instead of forcing the data to be placed at a specific cache block, thus we remove the index.

But what are the prices of full associativity?

- Because there is no index in the address any more, the entire address must be used as the tag, thus we require more space for the cache.
- A piece of data could be anywhere in the cache, so we must check every cache block. To do this in parallel (because if we do it sequentally would defeat the purpose of caching since it would take a lot of time) we would need a lot of comparators, thus the complexity of the hardware would be enourmous for large caches with many cache blocks.

There is also, an intermediate possibility: **set associative** cache. This types of cache combines the fully associative schema with the direct mapped schema. The idea is quite simple: the cache is divided into groups of blocks, called **sets** and each memory address maps to exactly one set in the cache, but data may be placed in any block within that set. If each set has 2^x blocks, then we call this type of cache **2^x-way associative cache**.

How do we determine where a memory address belongs in an associative cache? The answer is simple: in a similar way. Take, for example, this figure:

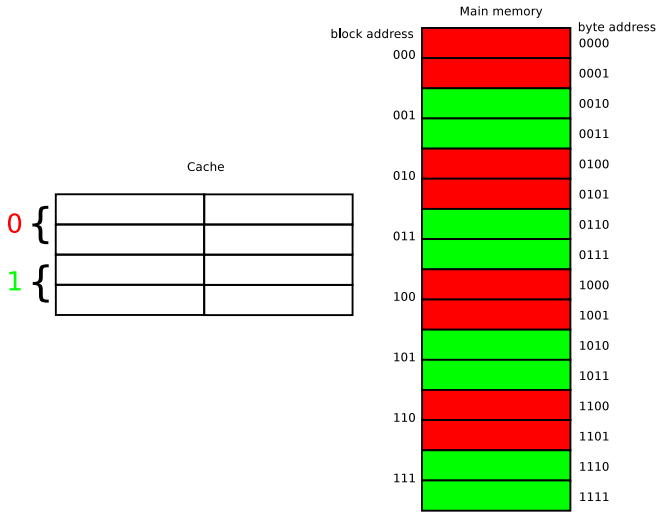

Figure 8: 2-way set associative cache

Here we have a cache that has 4 blocks, each block containing 2 bytes. These 4 blocks are divided into 2 sets, each set containing 2 blocks. Again, because the cache line size is 2 bytes, the main memory is divided into chunks of 2 bytes. Coressponding to these chunks, we have 8 conceptual blocks in the main memory. Each of these 8 blocks are directly mapped to one of the two sets, but the data(the conceptual blocks of the main memory — recall that we copy in

the cahce the whole block, not individual bytes) can be placed anywhere in the 2 blocks of each set. For example, main memory address 'b0001 has block address 'b000. This block directly maps to set 0, but block 'b000 can be placed either in the first block of the cache or the second block of the cache. To deal with this, we need to have 2 comparators. Similarly, to deal with 2^x*-way associativity* we need 2^x comparators.

Our arithmetic computations now compute a *set index* — we are mapping to sets, not directly to cache blocks. Thus, we have:

- *block address = memory address / size of cache line*
- *set index = block address mod / number of sets*
- *block offset(or cache line offset) = memory address / size of cache line*

Finally, a main memory address can be partinioned as follows:

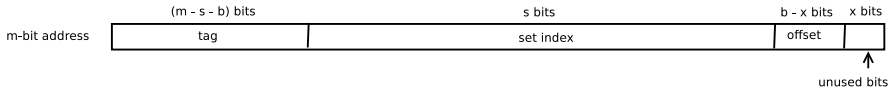

Figure 9: Logical split of a memory address 3

Remember: m is the number of bits of the main memory address, 2^s is the number of sets (2^s is equal to *size of cache in bytes / size of cache line in bytes / number of ways of associativity*), 2^x is the size of a word in bytes and 2^b is the size of the cache line.

To illustrate this, I will give you an example: my machine has a L1 cache of 32KB, 8-way associative and a cache line size of 64 bytes. The number of sets must be, therefore:

$$\frac{32*2^{10}bytes}{64bytes}\frac{1}{8ways} = 64sets$$

This is accurate, as it can be found from */sys/devices/system/cpu* (works only on Linux).