The simplest cache hardware consists of an SRAM with a few additional pieces of logic.

The cache hardware is designed so that each memory location in the CPU's address space maps to a particular cache line, hence the name "direct-mapped (DM) cache".

There are, of course, many more memory locations then there are cache lines, so many addresses are mapped to the same cache line and the cache will only be able to hold the data for one of those addresses at a time.

The operation of a DM cache is straightforward.

We'll use part of the incoming address as an index to select a single cache line to be searched.

The "search" consists of comparing the rest of the incoming address with the address tag of the selected cache line.

If the tag matches the address, there's a cache hit and we can immediately use the data in the cache to satisfy the request.

In this design, we've included an additional "valid bit" which is 1 when the tag and data fields hold valid information.

The valid bit for each cache line is initialized to 0 when the cache is powered on, indicating that all cache lines are empty.

As data is brought into the cache, the valid bit is set to 1 when the cache line's tag and data fields are filled.

The CPU can request that the valid bit be cleared for a particular cache line - this is called "flushing the cache".

If, for example, the CPU initiates a read from disk, the disk hardware will read its data into a block of main memory, so any cached values for that block will out-of-date.

So the CPU will flush those locations from the cache by marking any matching cache lines as invalid.

Let's see how this works using a small DM cache with 8 lines where each cache line contains a single word (4 bytes) of data.

Here's a CPU request for the location at byte address 0xE8.

Since there 4 bytes of data in each cache line, the bottom 2 address bits indicate the appropriate byte offset into the cached word.

Since the cache deals only with word accesses, the byte offset bits aren't used.

Next, we'll need to use 3 address bits to select which of the 8 cache lines to search.

We choose these cache index bits from the low-order bits of the address.

Why?

Well, it's because of locality.

The principle of locality tells us that it's likely that the CPU will be requesting nearby addresses and for the cache to perform well, we'd like to arrange for nearby locations to be able to be held in the cache at the same time.

This means that nearby locations will have to be mapped to different cache lines.

The addresses of nearby locations differ in their low-order address bits, so we'll use those bits as the cache index bits - that way nearby locations will map to different cache lines.

The data, tag and valid bits selected by the cache line index are read from the SRAM.

To complete the search, we check the remaining address against the tag field of the cache.

If they're equal and the valid bit is 1, we have a cache hit, and the data field can be used to satisfy the request.

How come the tag field isn't 32 bits, since we have a 32-bit address?

We could have done that, but since all values stored in cache line 2 will have the same index bits (0b010), we saved a few bits of SRAM and chose not save those bits in the tag.

In other words, there's no point in using SRAM to save bits we can generate from the incoming address.

So the cache hardware in this example is an 8-location by 60 bit SRAM plus a 27-bit comparator and a single AND gate.

The cache access time is the access time of the SRAM plus the propagation delays of the comparator and AND gate.

About as simple and fast as we could hope for.

The downside of the simplicity is that for each CPU request, we're only looking in a single cache location to see if the cache holds the desired data.

Not much of "search" is it?

But the mapping of addresses to cache lines helps us out here.

Using the low-order address bit as the cache index, we've arranged for nearby locations to be mapped to different cache lines.

So, for example, if the CPU were executing an 8-instruction loop, all 8 instructions can be held in the cache at the same time.

A more complicated search mechanism couldn't improve on that.

The bottom line: this extremely simple "search" is sufficient to get good cache hit ratios for the cases we care about.

Let's try a few more examples, in this case using a DM cache with 64 lines.

Suppose the cache gets a read request for location 0x400C.

To see how the request is processed, we first write the address in binary so we can easily divide it into the offset, index and tag fields.

For this address the offset bits have the value 0, the cache line index bits have the value 3, and the tag bits have the value 0x40.

So the tag field of cache line 3 is compared with the tag field of the address.

Since there's a match, we have a cache hit and the value in the data field of cache line can be used to satisfy the request.

Would an access to location 0x4008 be a cache hit?

This address is similar to that in our first example, except the cache line index is now 2 instead of 3.

Looking in cache line 2, we that its tag field (0x58) doesn't match the tag field in the address (0x40), so this access would be a cache miss.

What are the addresses of the words held by cache lines 0, 1, and 2, all of which have the same tag field?

Well, we can run the address matching process backwards!

For an address to match these three cache lines it would have look like the binary shown here, where we've used the information in the cache tag field to fill in the high-order address bits and low-order address bits will come from the index value.

If we fill in the indices 0, 1, and 2, then convert the resulting binary to hex we get 0x5800, 0x5804, and 0x5808 as the addresses for the data held in cache lines 0, 1, and 2.

Note that the complete address of the cached locations is formed by combining the tag field of the cache line with the index of the cache line.

We of course need to be able to recover the complete address from the information held in the cache so it can be correctly compared against address requests from the CPU.