# Read-Only Caches
## A series of Verilog designs.

**G.J. Van Loo Fen Logic Ltd.**

**Rev 1.2**

On my webpage www.verilog.pro you will find a number of cache designs. The designs vary from simple to more complex. At the moment all the cache are read-only but maybe in due time I will add a read-write cache. (But you can add a write-around function.) All design are free, as in totally free to use, abuse or change. All I ask is that the credits stay in the header of the files.

Versions:

1.0 Original
1.1 Text about line cache was missing: added.
1.2 Added fast flush cache.

## Designs.

At the moment of writing the website holds the following cache designs:

- async_ro_cache
- ro_cache
- ro_cache2
- ro_linecache
- ro_ff_cache

All the memories used in the design can be found in the 'memories' section on the same website.

### async_ro_cache

The async_ro_cache is a fully functional but not very realistic cache design. The reason it is not very realistic is because it uses asynchronous memories. That type of memory is no longer used in ASIC designs. Asynchronous memories still exists in FPGAs but there they use a lot of resources.

The appeal and the reason why it is present, is that the asynchronous cache is simple and as such it is ideal in the process of understanding how a cache works. It also was the basis upon which all other cache design where built.

### ro_cache

The ro_cache design is the simplest realistic read-only cache. It uses synchronous single ported memories for both the tag and the data memories. These are the smallest high-density memories available and as such give a cache with the smallest area.

I think I can shave another cycle of on a miss and maybe two if I used dual ported memories but I leave it for now.

## ro_cache2

The ro_cache2 design is an example of a two-way cache. For the rest it is identical to ro_cache.v. Because of the two way operation it use two memoires in both tag and cache section but the memories are only half as deep.

## ro_linecache

The ro_linecache is an again a variant on the ro_cache but it uses a four word cache line. The TAG logic is identical to the one in ro_cache.v but it is four times smaller. This is because it needs a tag per line, not per unit. Using a smaller tag is done though the parameters, there is no code change in the tag module.

## ro_ff_cache

The ro_ff_cache is another variant on the ro_cache. It uses a separate valid memory which allows it to perform a cache flush much faster. In the given example the valid memory is 16 bits wide thus a flush operation is sixteen times faster: 16 clock cycle instead of 256 clock cycles. The different code is only in the tag module. The top level module stays the same, apart from a valid-width parameter which is passed on to the tag module.

# Principle of operation.

Originally I was planning to refer to the WWW to find an explanation of how a cache works. I soon realised that every explanation I found does not tell anything about how it ***really*** works. All are missing details which are important if you want to design or understand the hardware which makes up a cache. Therefore I was obliged to write this text and make up all the diagrams which tell you how a cache ***really, really*** works.

**Overview.**
The main components of the cache are two memories: a data memory and a tag memory.
The cache data memory holds a copy of data from the main memory. Because the cache data memory is much smaller then the main memory it can only hold part of the main memory data. The cache data can come from anywhere of the main memory thus if you store it you need remember where it came from. Thus you add a 'tag' which tells what the original address of that data in that location was. This means every entry in the cache data memory has a related tag.

**CAM**
There are several ways of building a TAG memory. In the old days the address was stored in a CAM (Content Addressable Memory). That was a memory with a build-in comparator for ***every*** memory location. You present the memory with a value (For a cache that would be the main memory address you want to read) which is then compared against every entry and it outputs the location in the CAM where that value is stored (or a 'fail' if the value can not be found). Very nice but ***extremely expensive*** in silicon area. Also it does not scale very well. It becomes bulky and slow for bigger CAMs.

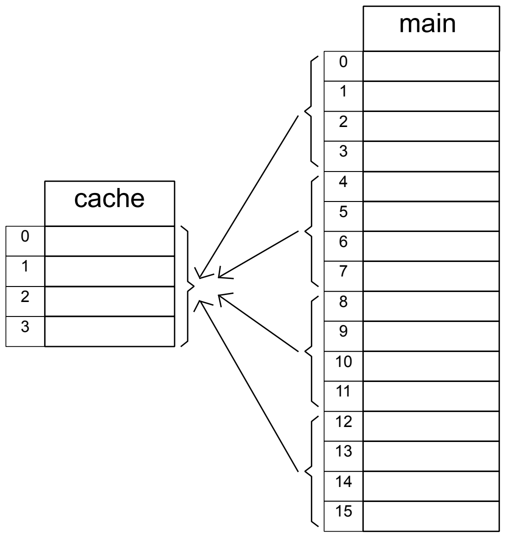
**TAG**
Then some clever cookie had another idea. It is a bit more convoluted which makes it a bit difficult to explain but I will give it a try using a 4 entry cache and a 16 entry main memory as example.

One entry of the cache data memory can not hold data from 'any' location of the main memory. It can only hold a copy of ***certain*** locations of the main memory.

The data memory locations 0-3 are 'mapped' on the main memory locations 0-3. But they are also mapped on the main memory locations 4-7, 8-11 and 12-15.:

Thus the cache data in location 0 can hold a copy from the main memory addresses 0, 4, 8 or 12.
Location1 can hold data from the  main memory addresses 1, 5, 9 or 13
etc.

The following diagram shows this:



The following is a complete table for the example:

| **Main address** | **Cache address** | | **Main address** | **Cache address** |
|---|---|---|---|---|
| 0 | 0 | | 8 | 0 |
| 1 | 1 | | 9 | 1 |
| 2 | 2 | | 10 | 2 |
| 3 | 3 | | 11 | 3 |
| 4 | 0 | | 12 | 0 |
| 5 | 1 | | 13 | 1 |
| 6 | 2 | | 14 | 2 |
| 7 | 3 | | 15 | 3 |

The advantaged become clear if we write the addresses in binary:

| **Main address** | **Cache address** | | **Main address** | **Cache address** |
|---|---|---|---|---|
| 0000 | 00 | | 1000 | 00 |
| 0001 | 01 | | 1001 | 01 |
| 0010 | 10 | | 1010 | 10 |
| 0011 | 11 | | 1011 | 11 |
| 0100 | 00 | | 1100 | 00 |
| 0101 | 01 | | 1101 | 01 |
| 0110 | 10 | | 1110 | 10 |
| 0111 | 11 | | 1111 | 11 |

Thus the least significant address bits of the main memory and the tag memory are identical.

Coming back to the TAG functionality.

Entry 0 of the data cache can now hold a copy of ***four*** main memory locations: 0, 4, 8 or 12. To remember which of the four values it is we need only two tag bits. 00, 01, 10 or 11.

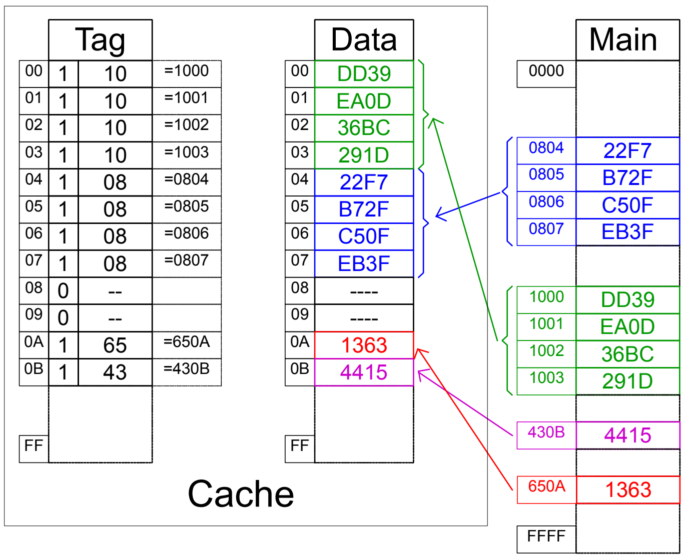
At this moment a nice digital miracle happens: if we take the two tag bits and put them in front of the data cache address bits we get: 0000, 0100, 1000 and 1100. Those are exactly the addresses of the main memory locations.
If you do the same with the other three data cache addresses the same thing happens: ***the tag contents plus the tag address form the main memory address***.

## Valid

Unfortunately there is one more situation we have to deal with. After starting, the cache will contain some random values. Thus none of the cache memory locations can be used until it has at least been written to. To handle this situation each cache entry needs a 'valid' bit. A bit which tells if that location contains valid data. After a reset all the valid bits will need to be set to zero. Once that has happened the cache is ready for usage. I will go into more details later because there are several ways in which valid bits can be implemented.

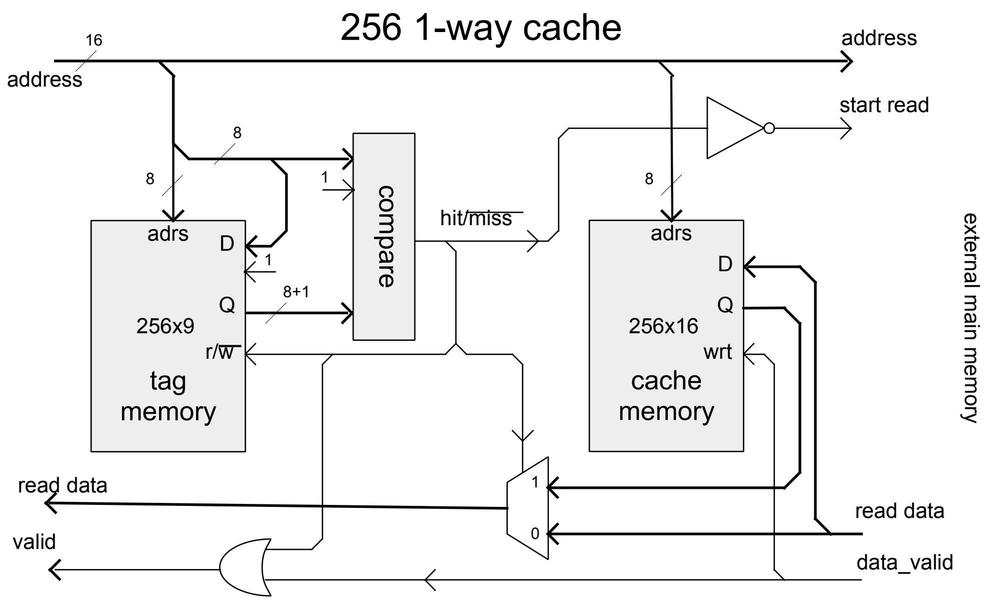
## Example.

As example I have a 16-bit wide and 64K deep main memory. For that we are going to make a 256 entry cache. Thus the cache data memory and the cache tag memory are both 256 entries deep. To address those we need an 8-bit address. (It is custom to draw the cache TAG memory and DATA memory separately). Below is a picture how the cache could look like after it has been used for a while:



At some time in the past the locations 1000-1003, 803-807, 430B and 650A have been read. Two of the TAG memory location still have their valid bit set to 0 so they are unused.

Here follows a diagram of the logic which implements such a cache. Not every details of the cache is shown as that would make the picture too complex. For example, logic to clear the tags after a reset has been omitted. If you want to see every detail just open the Verilog files I have provided and you can read a simulate every detail
The code which matches best with the diagram is in the "async_ro_cache" database.



**Implementation.**
The read address is split into two parts. Because the cache is 256 entries deep the LS 8 bits are used to address the tag and data memory. That leaves 8 more address bits which are not used. These have to be compared against. (See section TAG above). Therefore the16-bit wide address is split in 8+8 bits. The LS 8 bits go to the address of both cache memories. The top 8 bits are treated as 'compare data' in the tag logic. This means the TAG memory need to store 8 'compare' bits. But it also needs a valid bit. This gives a size of 8+1=9 bits wide and 256 entries deep. The cache data memory is also 256 entries deep and the width is whatever the width of the data bus is. (Here 16 bits).

**Operation.**
When an address arrives from the left hand side, it selects one entry from the tag memory and one entry from the cache memory. The output from the tag memory is compared against the MS 8 bits of the incoming address. There are two possibilities HIT or MISS.

**Hit.**
If the bits are the same we have a cache 'hit'. That means the data which is being addressed is present in the cache. The LS 8 address bits have also been going to the cache data memory. The data read from that memory is send back to the requester (Left hand side) via the mux. Also a 'valid' signal is returned to indicate that the data can be used.

**Miss. (or not hit)**
If the bits are different (or the valid is low) we have a cache 'miss'. That means the data which is being addressed is ***not*** in the cache. The miss signal is passed on to the logic which reads from the main memory. This starts a read request to the main memory. (How the main memory works is outside the

scope of this document.) It will take a while for the read from main memory to complete. After all if the main memory was very fast we would not need a cache in the first place.

After a while the read data returns from the main memory. Its arrival is marked by the data_valid signal on the right hand side. The data is passed back to the requester setting the 'valid' signal (left hand side).

Now we have new data we also need to update the cache. The newly arrived data is dutifully written to the cache data memory. But we also need to update the tag memory. Thus the tag memory is written with the MS 8 bits of the address. We also write a '1' to the valid bit.

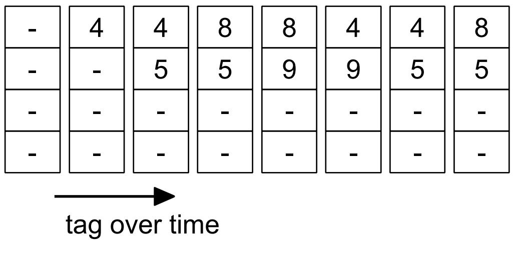
## 2-way, 4-way.

In cache technology there are so called 1-way, 2-way, 4-way or more general: N-way caches. Here I will explain what that means and why the exists. For this I will assume you have read an understood the 'TAG' section.

In the TAG section I explained how the cache data memory is 'repeatedly lined up' with the main memory. There I also explained how e.g. cache entry 0 is mapped on multiple main memory locations. I am going re-using the example here.

Lets assume a small program which runs in a loop and reads instructions from the memory. The read sequence is: 4, 5, 8, 9, 4, 5, 8, 9, 4.... When reading from the main addresses 4, and 5 it copies the read data into the cache data memory locations 0 and 1.
But when it read from main address 8 and 9 the data in the cache entries 0 and 1 are overwritten. This despite the fact that there are two more unused cache entries The program then loops to read from 4 and 5 again. But the cache does no longer hold the correct entries and thus again the main memory is accessed to get the data. In fact we have nothing but cache misses all the time.

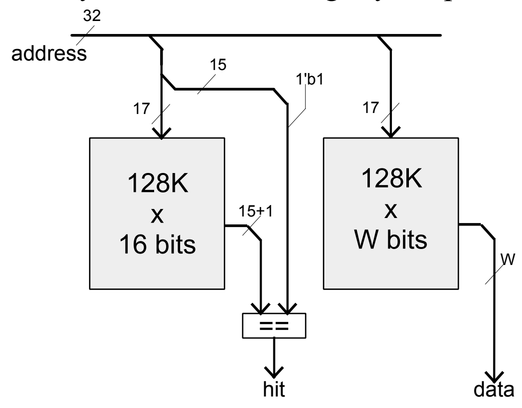
The following diagram shows this behaviour:



What we see is that the cache does not help at all. The data is continuously overwritten by new data. The example is a bit flawed in that we have an extremely small cache but the same holds for a bigger cache.
To help prevent this behaviour we can add a 'way' to a cache. That is, we double the tag memory giving us two 'ways'.

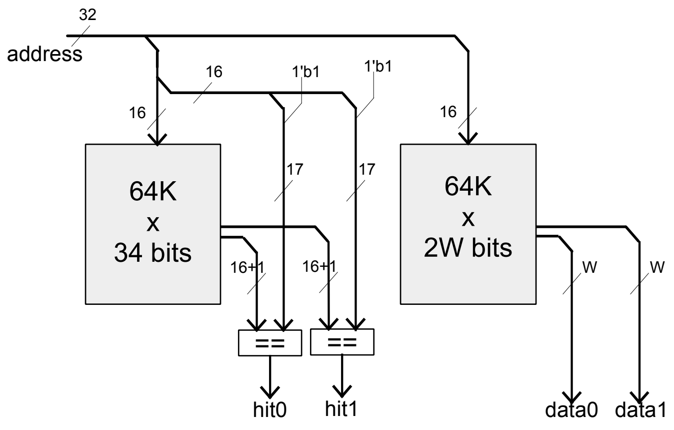
Example:
To explain an N-way cache I have to use a bigger example then our 4 entry cache. So I am going to use a 128K cache and a 32-bit wide address bus. The main memory data is 'W' bits wide.

First the ***old 1-way cache*** but in a slightly simplified diagram:



If there is a hit the data is returned to the requester.

This is a diagram of a ***two way cache*** but it has the same number of data words: 128K words:



The tag memory is half as deep and much wider. The same holds for the data memory.
Instead of one tag compare we perform two compare operations in parallel.
If there is a hit0 we pass data0 to the user.
If there is a hit1 we pass data1 to the user.
The circuit must be build in such a way that there never is a hit0 ***and*** a hit1 at the same time. Of course it is possible that neither hit0 nor hit1 is active in which case we have a cache miss and the data must be retrieved from main memory.
The trick lies in the fact that ***both*** TAG memories cover the ***whole*** of the memory. The penalty it that the TAG memory has to be one bit wider then for a one-way cache.

Writing of the tag memory become a bit more difficult. Instead of one entry, we now have two entries. After a reset the valid bits are used to select one of the two. Thus the entry which is not valid is always written. If both are valid we have to decide which if the two to replace. See 'replacement' further on.

Coming back to the simple 4-entry cache read-loop example earlier, this is what the tag system looks like for a four entry two way cache. The tag memory is half as deep but twice as wide:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| - | - | 4 | - | 4 | - | 4 | 8 | 4 | 8 | 4 | 8 |
| - | - | - | - | 5 | - | 5 | - | 5 | 9 | 5 | 9 |

$\longrightarrow$

tag over time

The read sequence is again 4, 5, 8, 9, 4, 5, 8, 9.... Now when the loop returns to reading from address 4 you get a cache hit as the data is still there. Also notice that we are now using the previously two unused locations and we have ***not*** doubled our data memory. In fact only the tag memory has grown slightly bigger. (From 4x3 bits to 4x4 bits)

**Replacement**
Replacement happens when a new cache entry must be written. We then have to decide which of the old ones to remove. In a one-way cache that is trivial as there is not choice. But in a 2, 4 or N way cache we can choose which of the 2,4 or N entries to replace.

There have been lots of studies about "cache replacement policy". Just look them up. In my code I use pseudo random replacement with a free running counter as pseudo random generator.
I have implemented pseudo LRU in the past but I am going to keep some knowledge to myself.

## Line cache

A line cache is very much like a standard cache. The only difference is that it reads 'lines' of code from the main memory. Thus instead of reading one data unit each time it reads a number of words all in one. This is called a 'line fetch'.

Line caches are likely to be a lot more efficient than normal caches for the following reasons:

- Memory reads tend to be sequential. Instructions are read and executed n-order until a branch or jump instruction. Even then most jumps are over a few (3-4) instructions so tend to end up in the vicinity of the last instruction.
  This means that if addrass X must be read it is very likely that the next reads are from location X+1, X+2 ...
- Dynamic memories forms the bulk of the main memory in most systems these days. To make those memories efficient you can only read a burst of data from them. The burst tends to be bigger in the latest versions (DDR2, DDR3). The size is often sequential 8 words. If you would read a single word the other 7 words would be discarded. Then a few cycles later e new words is read and again 7 words are unused.
  Note that first perfectly with the previous argument: the extra words are likely to be needed so lets get them to the cache.
- Less important but nice is that the TAG memory becomes smaller. You now have to tag only per line. This a 8-unit line cache requires an 8 time smaller TAG memory.

## Miscellaneous

**Combinations.**
I have kept every cache variant separately. However there is nothing which prevent the combination of features. With the given code examples it should not be too difficult to make a 4-way, 8 line, fast flushing cache.

**Writing.**
There is no read/write cache design (yet). The read-only caches can be used for instruction fetches or video processing etc. If a write cache of any type is needed: contact me.
There exists various types of write caches.

**Write past.**
Write data passes the cache completely and goes straight to main memory. It is trivial to add to the cache given here.

**Write through.**
Write data is written to the cache but also passed on the the main memory. This is a bit more difficult to implement. The advantage is that recently written data can be fetched quickly. But the penalty is that a lot of write to the main memory might be superfluous as the data will son be overwritten Think about `i` in : "`for (i=0; i<8;i++)`"..

**Write back.**
In a write back cache the data is written to the cache only. The write data is passed on to the main memory only when the line containing the data must be replaced or the cache is flushed. This requires extra bits in the TAG memory to see if a line in the cache holds newly written data (is "dirty") or if it can be discarded without writing. This type of cache is bit more challenging to implement. Its advantages are obvious looking at the example above.

**Valid bits.**
*Paragraph superseded.* There I now an cache version which has valid memory and can do fast flushing: ro_ff_cache.v
~~In my design you will find that the valid bits are stored alongside the tag bits. However they can also be store separately but then multiple valid bits in parallel. e.g. a 16-bit wide valid memory can read and write 16 bits at a time. This requires extra logic when reading to select the right bit from the set. It also requires extra logic when writing because one of the 16 bits must be set (OR-ed) and then all 16 bits have to be written back.~~
~~However the main advantage comes when the cache has to be 'flushed'. With 16 bits in parallel this goes 16 time faster. Thus a 256 entry cache can be flushed in 16 cycle instead of 256.~~

~~If I have time I might add a 'fast flush' cache example.~~