ADS2 Assessed Exercise: Modeling a Cache – Part I

Wim Vanderbauwhede

1 Aim

The final aim of the coursework is to create a Java model for a simple cache as used in modern computer systems. The aim of this part is to design a high-level solution, without writing any code.

Source code for a template and testbench will be provided for the second part of the assignment.

2 What to submit

For the first part of the assignment is the design of your solution. You have to write in your own words, in no more than 2 pages, what your solution will look like in terms of the datastructures and algorithms to be used. You can use code or pseudocode if you like.

This part of the exercise does not directly contribute to the marks of the coursework but you have to submit it and it has to be of sufficient quality, otherwise the coursework will not be marked. There is no particular syntax required, the main requirements are that the description is complete, clear and readable.

More detail on the API as well as source code skeletons and source code for the testbench will be provided for the second part of the assignment, when you will write the actual code.

You *must* submit this report in TXT format through the Moodle submission system, and the filename *must* be *< your matric + 1st char of your name in lowercase >. < txt >*, so for example if your matric number is *1107023m* then your file *must* be named *1107023m.jpg*.

3 Overview of the cache model to be implemented

3.1 What is a cache?

In an actual computer system, accessing DRAM memory requires many clock cycles. To limit the time spent in waiting for memory access, processors have a *cache*, a small but fast memory. For every memory read or write operation, first the processor checks if the data is present in the cache, and if so it uses the cache rather than accessing the DRAM. Otherwise it will fetch the data from memory and store it in the cache. The aim is that

access of the memory through the cache is on average much faster than direct access, but otherwise the behaviour is quite the same. Real-life caches are very complicated; for the purpose of this exercise we will create a simple conceptual model of a cache that illustrates the key points.

3.2 Managing used and free Locations

As the size of the cache is much smaller than the DRAM, how do we store portions of the DRAM content in it? We can model the DRAM memory and the storage part of the cache as arrays of fixed size. So if we want to store some data in the cache, we find a free location and write the data into it. At some later point, the data will be moved from the cache to the DRAM, freeing up this location. So we need a data structure to keep track of the used/free locations. The easiest way is to keep track of the free locations.

3.3 Hits and misses

When the CPU reads from or writes to a memory address, then the data can either be present in the cache (*hit*) or not (*miss*). When the data is not present, it needs to be fetched from the DRAM memory.

3.4 Eviction and replacement policy

So what happens when the cache is full? We need to free up space by *evicting* data from the cache, and writing it back to the DRAM memory. There are several possible policies to do this. The simplest one (which we will use, even if it is certainly not the best one) is to evict data from the most recently used location, because all it requires is that we keep track of that single location. This is called "Last In First Out" or LIFO. "Last used" means accessed for either read or write.

3.5 Associativity

When we evict data from the cache it needs to be written back to the DRAM memory. Conversely, the data that we put into the cache was read from an address location in the DRAM memory. Therefore the cache must not only keep track of the data but also of its original address. In other words, we need a lookup between the address in the DRAM and the corresponding address in the cache. A cache which allows to store any memory address at any cache location is called "fully associative", and this is what you should implement.

3.6 Cache lines

If we would have a one-on-on mapping between the addresses in the cache and those in the DRAM memory, then for a cache of a given size, we would need data structures for the

lookups and for the management of the used/free addresses of the same size. This would require a lot of silicon. Therefore, in practice the cache will not simply fetch the content of a single memory address, but a contiguous block of memory called a *cache line*. For example the Arm Cortex-A53 has a 64-byte cache line. Assuming that our memory stores 32-bit words, then the cache line is 16 words long, so the size of the lookup tables is 16x smaller than the actual cache size.

There is another reason for the use of cache lines: when a given address is accessed, subsequent memory accesses are frequently to neighbouring addresses. So fetching an entire cache line on a cache miss tends to reduce the number of subsequent cache misses.

3.7 Cache write behaviour

When the CPU writes to the cache, it can either immediately also write to the DRAM memory (*write-through*) or not (*write-back*). For this exercise the cache will use the *write-back* policy, so the writes are always to the cache. If the address the CPU is writing to is not yet in the cache, we need to update the cache first. The update policy used or this exercise is called *write allocate*: on a write miss, a cache line is loaded into the cache, followed by the write operation.

3.8 Cache line addressing

The only complication in the cache line-based model is that we need to manipulate the memory address to determine the start of the cache line and the location of the data inside the cache line. We do this using bit shift and bit mask operations: e.g. for a 16-word cache line, the first 4 bits of the address identify the position of the data in the cache line. We don't need to store these bits in the lookup tables of the cache because the cache stores only whole cache lines. In other words, from the perspective of the cache the memory consists of cache lines rather than individual locations. So we have the following formulas:

```
data_position_in_cache line = address & 0xF
cache_line_address = address >> 4

address = (cache_line_address << 4) + data_position_in_cache line
```

Methods implementing these formulas will be provided.

3.9 Summary

The purpose of this exercise is to implement a model for a Fully Associative Last-In-First-Out Cache with Write-Back/Write-Allocate, with variable DRAM memory size, cache size and cache line size.