Single Cycle MIPS Simulator with Cache

Nathaniel Vallen
32221415
Global SW Convergence

Introduction:

This project is made based on the single cycle MIPS simulator but with the addition of a cache system. The idea of a cache system is to further optimize the system's performance by utilizing the tradeoff between memory size and latency. Basically, smaller memory gives higher speed (lower latency), and on the contrary, larger memory results in lower speed (more latency), but naturally, provides ample space for storage. In real life practices, the tradeoff includes aspects other than the two mentioned, such as cost and bandwidth. However, to simplify our full software implementation of the cache system in a single cycle MIPS simulator, only the former two will be put in consideration.

A cache can be understood as a small-sized storage (as compared to a data/instruction memory) that holds frequently and/or recently used data to accelerate processing speed of memory accessing instructions. Cache storages are typically divided into several levels (two or three), with each of them offering slightly different benefits from the others.

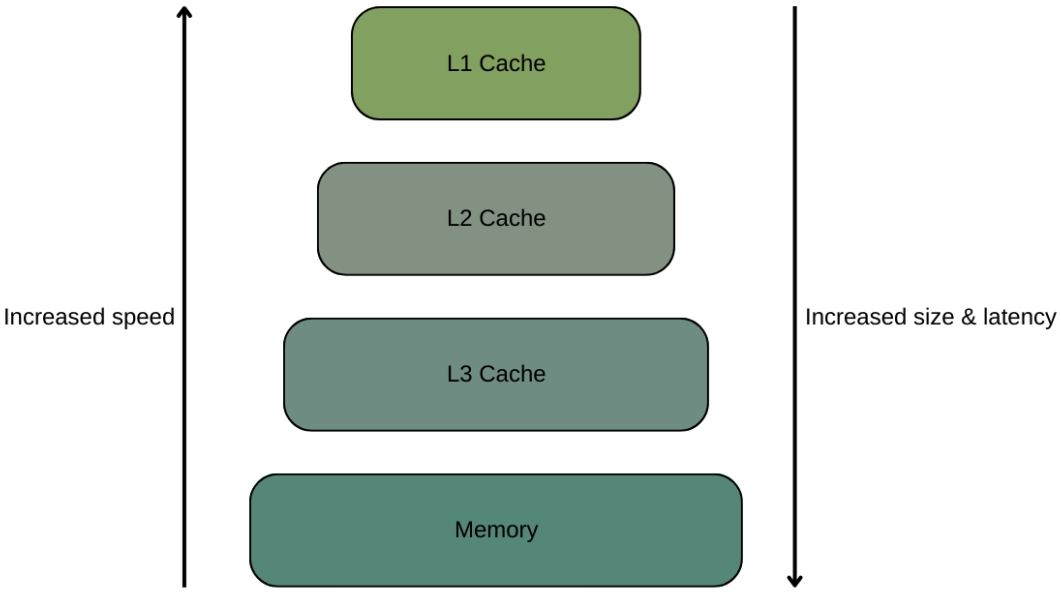

Figure 1. Different levels of cache

In general, L1 cache (level 1) is the closest one to the processor, yielding the best speed, or in other words, the shortest latency. However, as mentioned earlier, this is possible due to its small capacity of storage in comparison to the other cache levels. L2 cache, as illustrated above, has a bigger storage and in consequence, has reduced speed. L3 cache can also be described in the same manner in respect to size and speed comparison. Lastly, the largest storage that holds all available data is the main memory of the system itself.

Caches boost the system's performance by providing the required data with lower latency as compared to the memory. By using the same diagram, we can also infer how the data fetching hierarchy works in memory systems with cache. As caches are typically small in size, it can only fit so much data inside before having to replace the old data with the new ones. This implies that sometimes, the attempt to fetch a specific data from the cache can result in a miss, after which the system has to refer to next level caches or eventually the memory itself. From this, we can conclude that the optimization the cache provides gets higher as the cache miss rate gets lower. In other words, the main goal is to minimize the cache miss rate and increase its hit rate.

Another basic concept of the cache system is locality. The term "locality" refers to the likeliness of other data being accessed after a certain data is accessed. The first principle regarding locality is temporal locality. Temporal locality refers to the probability of recently accessed data to be accessed again in the near future. A cache can adopt this property by storing recently used data and replacing the ones that are rarely used. The second principle is called spatial locality. Spatial locality means that after a particular data is accessed, neighboring data in nearby memory addresses have a high likelihood of also being accessed in the future. A cache can utilize this property by keeping a certain range of data (also known as cache line), covering the accessed data and the other data around it.

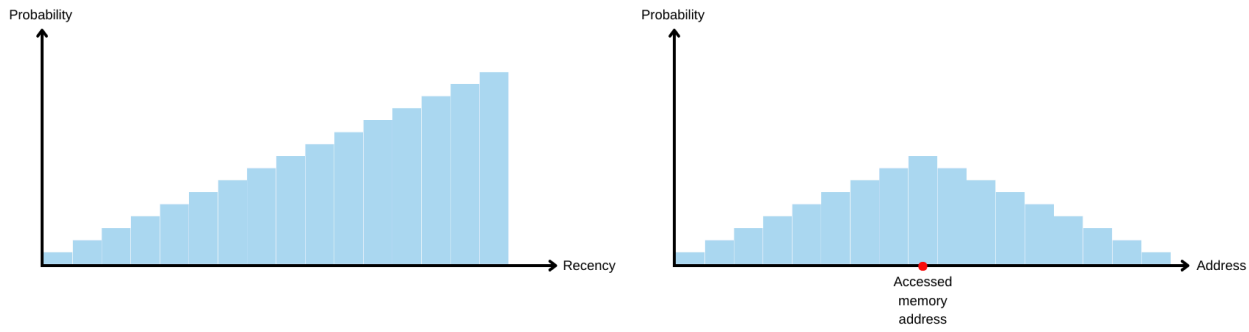

Figure 2. Simple graph of temporal (left) and spatial (right) locality

There are two fundamental parts of a cache, namely data store and tag store. As their names suggest, a data store keeps the relevant data, while a tag store keeps tags that correspond to data in the data store. Data is kept in the data store as a unit of cache line. In other words, larger cache line size will keep more data belonging to one tag. Thus, this can be used to minimize the required size of the tag store, although this will also bring in some tradeoffs which will be discussed later.

Types of cache also vary according to the needs. Primarily, there are three types of cache, direct mapped cache, fully associative cache, and set associative cache. In this project, I have decided to use a 4-way set associative cache for optimization reasons (further explained in the

next part). A 4-way set associative cache divides both tag store and data store into 4 ways. Upon receiving an instruction, the program divides it into three parts, which are tag, index, and (cache line) offset. These are the pieces of information that will help with marking where the data should be located in the cache. As we already know, a tag will serve as the identifier of the designated cache line. An index is useful to indicate the corresponding set (line), and the cache line offset marks the exact location in the cache line where the designated data starts from.

Last but not least, a crucial part of systems implementing cache is replacement policy. As time progresses, the cache will start replacing irrelevant data with new data. The process of selecting a victim cache line to evict and replacing it with a new cache line follows a chain of conditions which is also known as a replacement policy. Additionally, this policy also plays a pivotal role in determining where to place data in the cache memory before any data is moved there (when the cache is empty). There are several well-known replacement policies including LRU (Least Recently Used) and SCA (Second Chance Algorithm). The LRU replacement policy implements temporal locality by replacing the oldest data (least recently used/accessed) in the cache with the new ones. In order to do this, the system keeps a record of the last *n* memory accesses and tracks the access recency of each data (address). Unfortunately, this is also where the weakness of LRU policy lies. As the value of *n* grows, more memory accesses can be recorded, but also more complex management is required. Therefore, another option is available to use as a replacement policy, which is SCA (Second Chance Algorithm). In addition, SCA puts into factor both temporal and spatial locality. SCA considers recency by keeping a pointer that marks the oldest cache line in a set which will be evicted once a new cache line has to come in. However, that is not the only prerequisite for a cache line to be removed. Apart from being the oldest cache line, the said cache line also has to have no SCA bit in order to be evicted. An SCA bit is given to a cache line if it already exists in the cache and gets accessed one more time. This is where frequency is considered in the SCA policy. More frequently accessed data (cache lines) have a higher chance to stay in the cache even after being marked as the oldest cache line, while cache lines which are only accessed once in a long while will not have the opportunity to obtain the SCA bit, making them more vulnerable to being evicted by the entry of new cache lines. The main privilege of this replacement policy is the simplicity of its implementation. In order to use the SCA policy, we only need to add a pointer to mark the oldest cache line and an extra bit for each cache line, which is what we previously referred to as the SCA bit.

As the final addition, after completely applying all the requirements above, one other bit needs to be introduced in order to make the cache system work as intended. As of now, we have been considering the case of reading from cache. However, there are other instructions that write on a specific address instead of reading from it. In this case, assuming that the specified address is available in the cache, then the new data will be written in the cache instead of the memory. This behavior can be dangerous when the time comes for this updated cache line to be evicted and replaced, losing all the updated values forever. To overcome this, an updated cache line must

always have a mark called “dirty” (using a dirty bit). Dirty cache lines must have their whole cache line copied into the main memory before proceeding with the program. This way, we can assure that the updated cache line’s contents are always identical to the contents of the main memory in the address it is referring to, erasing the worry that the updated values may be lost for good.

Important Concepts & Considerations:

I. The following MIPS core instructions are excluded from this program:
   A. Load Byte Unsigned (LBU) [24]
   B. Load Halfword Unsigned (LHU) [25]
   C. Load Linked (LL) [30]
   D. Store Byte (SB) [28]
   E. Store Halfword (SH) [29]
   F. Store Conditional (SC) [38]

II. The full datapath of this program can be found here.

III. A closer look at the 4-way cache system is presented here.

IV. The accurate presentation of a single cache plane is shown below:

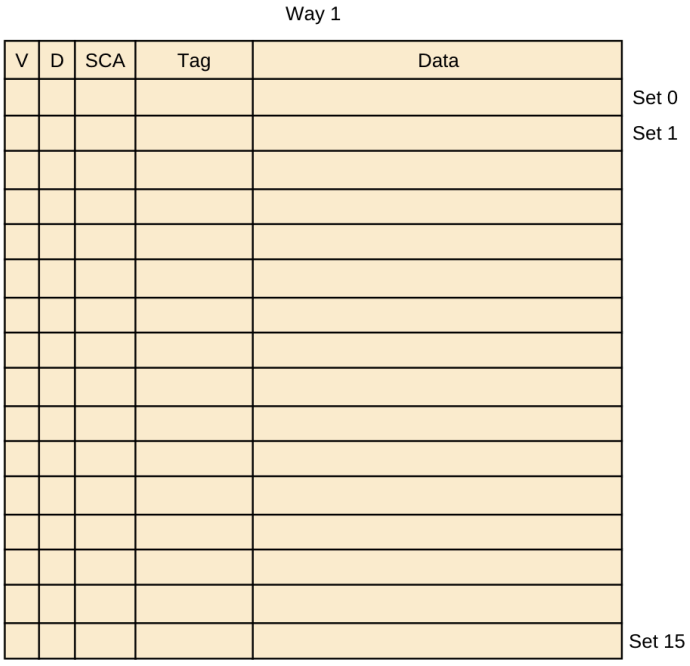

Figure 3. Cache plane

V. The rough datapath working inside the cache can be seen here.

VI. The simplified structure of the cache in the datapath is shown below:

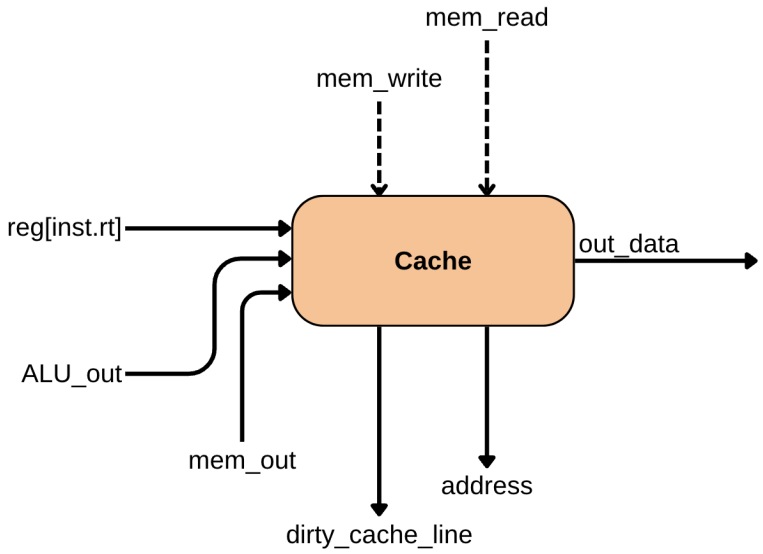

Figure 4. Cache datapath

VII. The specific properties of the cache displayed in this project are the following:

- Determined Values
  - Address space → 32-bit
  - Cache/data store size → 1024B
  - Cache line size → 16B $= 2^4 \rightarrow k = 4$
  - Number of cache planes (way) → 4
- $cache\ plane\ size = \frac{cache\ size}{number\ of\ cache\ planes} = \frac{1024}{4} = 256B$
- $total\ number\ of\ cache\ lines = \frac{cache\ size}{cache\ line\ size} = \frac{1024}{16} = 64$
- $number\ of\ cache\ lines\ per\ cache\ plane\ (sets\ in\ a\ way) = \frac{total\ number\ of\ cache\ lines}{number\ of\ cache\ lines} = \frac{64}{4} = 16 = 2^4 \rightarrow n = 4$

- $tag = address\ space - index - cache\ line\ offset = 32 - 4 - 4 = 24$
- $index = n$ or $index = \log_2(number\ of\ entries) = 4$
- $cache\ line\ offset = k$ or $cache\ line\ offset = \log_2(cache\ line\ size) = 4$

Figure 5. Cache specifications

VIII. The design of the 4-way set associative cache is based on the following considerations:

A. Set associative cache → A good middle point between speed and hit rate.
B. Relatively large cache size → Increased hit rate.

C. Relatively small cache line size → Reduced cache access latency and miss penalty.
D. Relatively high number of sets → Increased hit rate.
E. Relatively low number of ways → Reduced cache access latency.

These are highly relative and evaluated with the consideration that this program runs rather simple MIPS programs. The scale would naturally be much different to that of real life applications.

Build Configurations & Environment:

The code in this project was written in C programming language using Visual Studio Code IDE.

Screen Captures:

```
PC: 0x24
Jumped to 0xffffffff
Displaying all registers:
reg[00]: 0        0x0              reg[08]: 0        0x0              reg[16]: 0        0x0              reg[24]: 0        0x0
reg[01]: 0        0x0              reg[09]: 0        0x0              reg[17]: 0        0x0              reg[25]: 0        0x0
reg[02]: 100      0x64             reg[10]: 0        0x0              reg[18]: 0        0x0              reg[26]: 0        0x0
reg[03]: 0        0x0              reg[11]: 0        0x0              reg[19]: 0        0x0              reg[27]: 0        0x0
reg[04]: 0        0x0              reg[12]: 0        0x0              reg[20]: 0        0x0              reg[28]: 0        0x0
reg[05]: 0        0x0              reg[13]: 0        0x0              reg[21]: 0        0x0              reg[29]: 16777216 0x1000000
reg[06]: 0        0x0              reg[14]: 0        0x0              reg[22]: 0        0x0              reg[30]: 0        0x0
reg[07]: 0        0x0              reg[15]: 0        0x0              reg[23]: 0        0x0              reg[31]: -1       0xffffffff
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Number of instructions executed: 10
Number of R-type instructions executed: 3
Number of I-type instructions executed: 7
Number of J-type instructions executed: 1
Number of memory access instructions executed: 4
Number of branches taken: 0

Hit count: 3
Miss count: 1
```

Figure 6. Final result

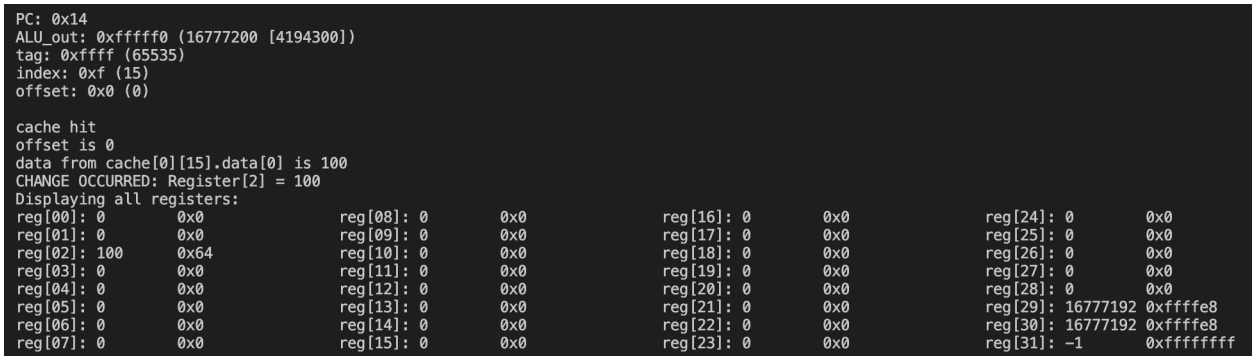

Figure 7. Cache proof