**Cache memory**

Cache memory is a high-speed memory region (compared to the speed of main memory) that temporarily stores program instructions or data for future use. Usually, these instructions or data items have been retrieved from main memory recently and are likely to be needed again shortly.

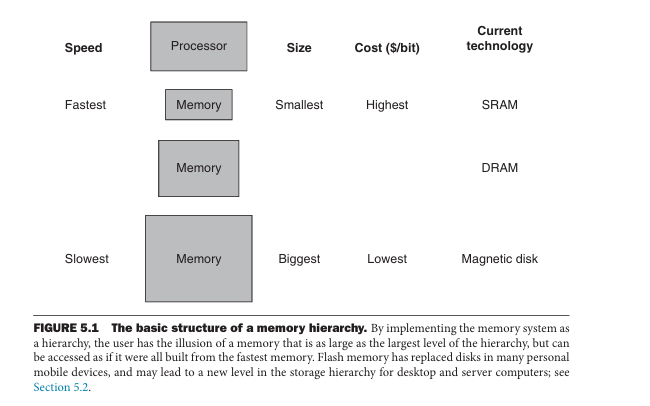
The primary purpose of cache memory is to increase the speed of repeatedly accessing the same memory location and nearby memory locations. To be effective, accessing the cached items must be significantly faster than accessing the original source of the instructions or data, referred to as the backing store.

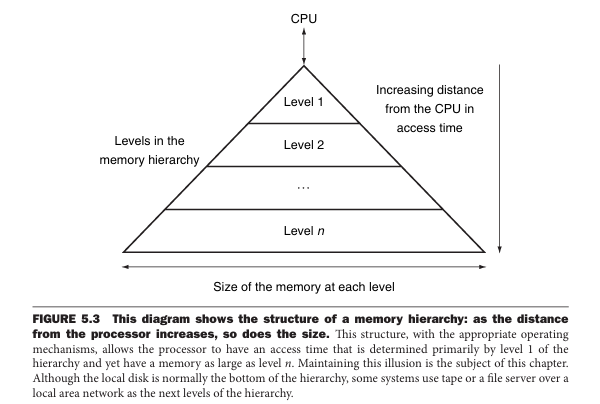
When caching is in use, each attempt to access a memory location begins with a search of the cache. If the requested item is present, the processor retrieves and uses it immediately. This is called a cache hit. If the cache search is unsuccessful (a cache miss), the instruction or data item must be retrieved from the backing store. In the process of retrieving the requested item, a copy is added to the cache for anticipated future use.

Cache memory improves computer performance because many algorithms executed by operating systems and applications exhibit locality of reference. Locality of reference refers to the reuse of data that has been accessed recently (this is referred to as temporal locality) and to accessing data in physical proximity to data that has been accessed previously (this is called spatial locality).

As a rule, cache memory regions are small in comparison to the backing store. Cache memory de vices are designed for maximum speed, which generally means they are more complex and costly per bit than the data storage technology used in the backing store. Due to their limited size, cache memory devices tend to fill quickly. When a cache does not have an available location to store a new entry, an older entry must be discarded. The cache controller uses a cache replacement policy to select which cache entry will be overwritten by the new entry.

The goal of processor cache memory is to maximize the percentage of cache hits over time, thus providing the highest sustained rate of instruction execution. To achieve this objective, the cach ing logic must determine which instructions and data will be placed into the cache and retained for future use.

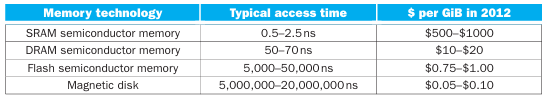
****

****

**Memory technologies**

There are four primary technologies used today in memory hierarchies. Main memory is implemented from DRAM (dynamic random access memory), while levels closer to the processor (caches) use SRAM (static random access memory). DRAM is less costly per bit than SRAM, although it is substantially slower. The price difference arises because DRAM uses significantly less area per bit of memory, and DRAMs thus have larger capacity for the same amount of silicon.

The third technology is flash memory. This nonvolatile memory is the secondary memory in Personal Mobile Devices. The fourth technology, used to implement the largest and slowest level in the hierarchy in servers, is magnetic disk. The access time and price per bit vary widely among these technologies, as the table below shows, using typical values for 2012.

****

**Level 1 cache**

In a multilevel cache architecture, the cache levels are numbered, beginning with 1. A level 1 cache (also referred to as an L1 cache) is the first cache the processor searches when requesting an in struction or data item from memory. Because it is the first stop in the cache search, an L1 cache is generally constructed using the fastest SRAM technology available and is physically located as close to the processor’s logic circuitry as possible.

The emphasis on speed makes the L1 cache costly and power-hungry, which means it must be quite small in comparison to main memory, particularly in cost-sensitive applications. Even when it is modestly sized, a fast level 1 cache can provide a substantial performance boost in comparison to an otherwise equivalent processor that does not employ caching.

The processor (or the MMU, if present) transfers data between DRAM and cache in fixed-size data blocks called cache lines. Computers using DDR DRAM modules generally use a cache line size of 64 bytes. The same cache line size is commonly used at all cache levels.

Modern processors often divide the L1 cache into two sections of equal size, one for instructions and one for data. This configuration is referred to as a split cache. In a split cache, the level 1 in struction cache is referred to as the L1 I-cache and the level 1 data cache is the L1 D-cache. The processor uses separate buses to access each of the caches, thereby implementing a significant aspect of the Harvard architecture. This arrangement speeds instruction execution by enabling access to instructions and data in parallel, assuming L1 cache hits for both.

**Direct-mapped cache**

A direct-mapped cache is a block of memory organized as a one-dimensional array of cache sets, where each address in the main memory maps to a single set in the cache. Each cache set consists of the following items:

• A cache line, containing a block of data read from main memory.

• A tag value, indicating the location in main memory corresponding to the cached data.

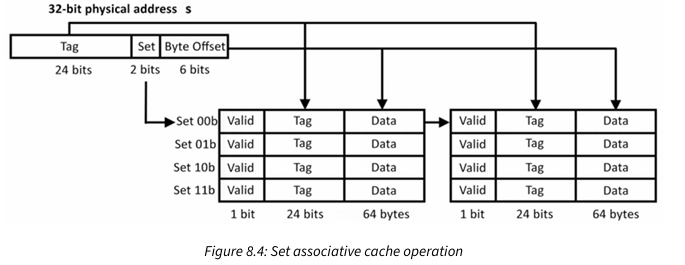
• A valid bit, indicating whether the cache set contains data.

This example represents an instruction cache, which is read-only. Data cache contain one more bit, this is called dirty bit. If set, the dirty bit indicates the data in a line is modified and the system must write the data to memory before its cache line can be freed for reuse.

There are times when the cache contains no data, such as immediately following processor pow er-on. The valid bit for each cache set is initially clear to indicate the absence of data in the set. When the valid bit is clear, use of the set for lookups is inhibited. Once data has been loaded into a cache set, the hardware sets the valid bit.

**Set associative cache**

In a two-way set associative cache, memory is divided into two equal-sized caches. Each of these has half the number of entries of a direct-mapped cache of the same total size. The hardware consults both caches in parallel on each memory access and a hit may occur in either one. The following diagram illustrates the simultaneous comparison of a 32-bit address tag against the tags contained in two L1 I-caches:

****

When a cache miss occurs, the memory management logic must select which of the two cache tables to use as the destination for the data being loaded from DRAM. A common method is to track which of the relevant sets in the two tables has gone the longest without being accessed and overwrite that entry. This replacement policy, called least-recently used (LRU), requires hardware support to keep track of which cache line has been idle the longest. The LRU policy relies on the temporal locality heuristic that says data that has not been accessed for an extended time is less likely to be accessed again soon.

Another method for choosing between the two tables is to simply alternate between them on successive cache insertions.

The hardware to implement this replacement policy is simpler than the LRU policy, but there may be a performance impact due to the arbitrary selection of the line being invalidated. Cache replacement policy selection represents another area of trade-off between increased hardware complexity and incremental performance improvements.

In a two-way set associative cache, two cache lines from different physical locations with match ing Set fields are present in the cache simultaneously, assuming both cache lines are valid. At the same time, in comparison to a direct-mapped cache of the same total size, each of the two-way set associative caches is half the size. This represents yet another design trade-off: the number of unique Set values that can be stored in a two-way set associative cache is reduced compared to direct mapping, but multiple cache lines with identical Set fields can be cached simultaneously.

The cache configuration that provides the best overall system performance will depend on the memory access patterns associated with the types of processing activity performed on the system.

Set associative caches can contain more than the two caches of this example. Modern processors often have 4, 8, or 16 parallel caches, referred to as 4-way, 8-way, and 16-way set associative caches. These caches are referred to as set associative because an address Tag field associates with all the cache lines in the set simultaneously. A direct-mapped cache implements a one-way set associative cache.

The advantage of multi-way set associative caches over direct-mapped caches is that they tend to have higher hit rates, resulting in better system performance than direct-mapped caches in most practical applications. If multi-way set associative caches provide better performance than one-way direct mapping, why not increase the level of associativity even further? Taken to the limit, this progression ends with fully associative caching.

**Processor cache write policies**

In processors with split caches, the L1 D-cache is similar in structure to the L1 I-cache, except the circuitry must permit the processor to write to memory as well as read from it. Each time the processor writes a data value to memory, it must update the L1 D-cache line containing the data item, and, at some point, it must also update the DRAM location in physical memory containing the cache line. As with reading DRAM, writing to DRAM is a slow process compared to the speed of writing to the L1 D-cache.

The most common cache write policies in modern processors are:

• Write-through: This policy updates DRAM immediately every time the processor writes data to memory.

• Write-back: This policy holds the modified data in the cache until the line is about to be evicted from the cache. A write-back cache must provide an additional status bit asso ciated with each cache line indicating whether the data has been modified since it was read from DRAM. This is called the dirty bit. If set, the dirty bit indicates the data in that line is modified and the system must write the data to DRAM before its cache line can be freed for reuse.

**Cache controller**

**To control cache, the most common method is use finite-state machines (FSM).**

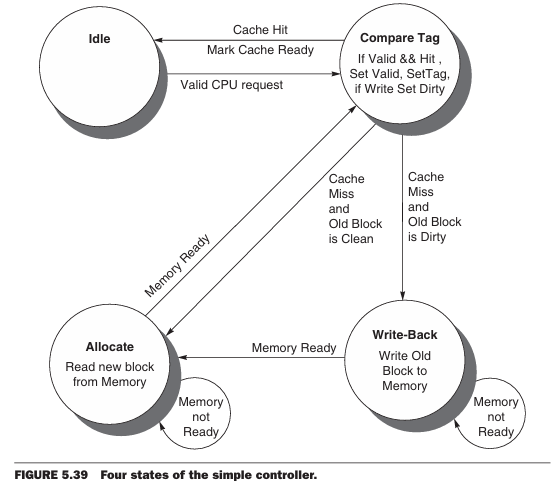
****

Figure 5.39 shows the four states of our simple cache controller:

■ Idle: This state waits for a valid read or write request from the processor, which moves the FSM to the Compare Tag state.

■ Compare Tag: As the name suggests, this state tests to see if the requested read or write is a hit or a miss. The index portion of the address selects the tag to be compared. If the data in the cache block referred to by the index portion of the address are valid, and the tag portion of the address matches the tag, then it is a hit. Either the data are read from the selected word if it is a load or written to the selected word if it is a store. The Cache Ready signal is then set. If it is a write, the dirty bit is set to 1. Note that a write hit also sets the valid bit and the tag field; while it seems unnecessary, it is included because the tag is a single memory, so to change the dirty bit we likewise need to change the valid and tag fields. If it is a hit and the block is valid, the FSM returns to the idle state. A miss first updates the cache tag and then goes either to the Write-Back state, if the block at this location has dirty bit value of 1, or to the Allocate state if it is 0.

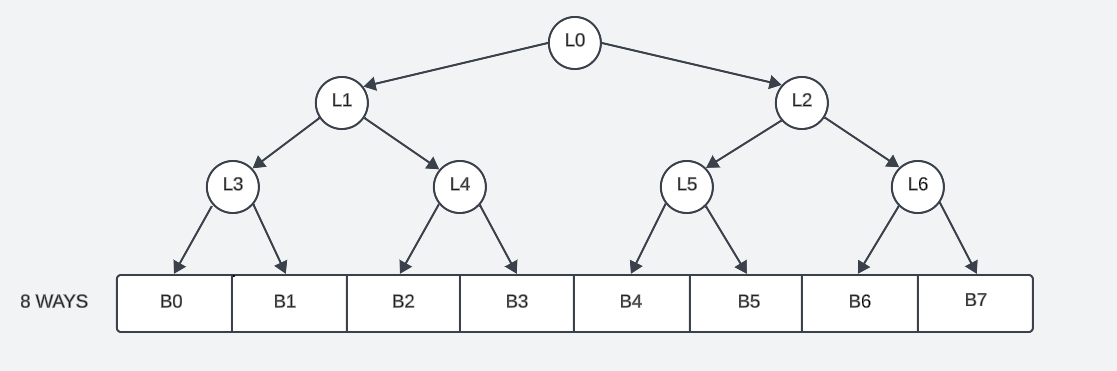
■ Write-Back: This state writes the 128-bit block to memory using the address composed from the tag and cache index. We remain in this state waiting for the Ready signal from memory. When the memory write is complete, the FSM goes to the Allocate state.

■ Allocate: The new block is fetched from memory. We remain in this state waiting for the Ready signal from memory. When the memory read is complete, the FSM goes to the Compare Tag state. Although we could have gone to a new state to complete the operation instead of reusing the Compare Tag state, there is a good deal of overlap, including the update of the appropriate word in the block if the access was a write.

**Least recently used (LRU) cache**

Cache replacement algorithms are efficiently designed to replace the cache when the space is full. The **Least Recently Used (LRU)** is one of those algorithms. As the name suggests when the cache memory is full, LRU picks the data that is least recently used and **removes**it in order to make space for the new data. The priority of the data in the cache changes according to the need of that data i.e. if some data is fetched or updated recently then the priority of that data would be changed and assigned to the **highest priority**, and the priority of the data **decreases**if it remains unused operations after operations.

(Phần thiết kế) Pseudo LRU 8 ways



Hình 1.1. Psuedo LRU tree

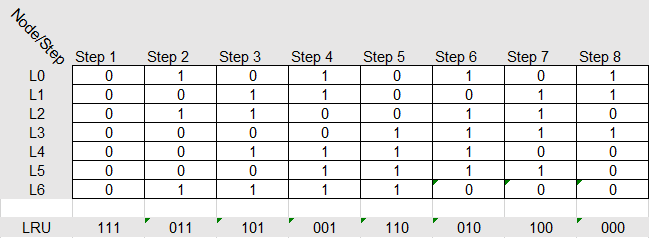
Trong thiết kế 8-way set associative cache, LRU tree có cấu trúc như hình trên. LRU tree có thể được mô tả như 3 tầng, tầng đầu tiên có L0, tầng thứ hai là L1 và L2, tầng thứ ba chứa L3, L4, L5, L6. Gọi mỗi way là một block (trong block chứa tag và data), ví dụ khi truy cập block B5, giá trị nhị phân là 101, lúc này LRU tree sẽ được cập nhật lại giá trị mỗi lần truy cập, như vậy L0 sẽ bằng 1, L2 bằng 0 và L5 sẽ bằng 1. Như quy tắc trên, mỗi lần truy cập cache, LRU tree sẽ cập nhật lại giá trị và lưu trữ trạng thái ở từng node (L0, L1, L2,…).

A diagram of a machine

Description automatically generated

Hình 1.2. Psudeo LRU circuit

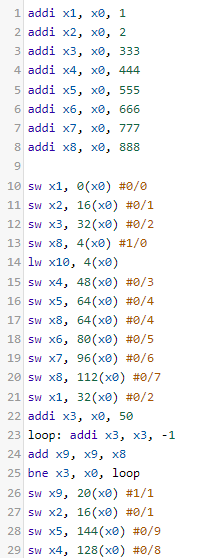
Từ những giá trị đã được lưu trong các node L0, L1, … giá trị LRU sẽ được tính như hình trên, Hình dưới đây sẽ giải thích trực quan hơn các trường hợp cụ thể cách hoạt động của bộ LRU.



Hình 1.3. Giá trị LRU tree ở mỗi lần truy cập

Ở hình trên, giá trị của mỗi node được lưu lại qua từng step (hoặc gọi là lần truy cập), ở step 1 giá trị của các node đều bằng 0, do đó giá trị LRU sẽ có giá trị là 111 như sơ đồ 1.2. Tiếp đến step 2, giá trị LRU ở step 1 sẽ được sử dụng để cập nhật lại LRU tree, ở bước này, giá trị LRU trước đó là 111, do đó L0 được cập nhật bằng 1, vì L0 bằng 1 nên L2 sẽ được cập nhật bằng LRU[1] và bằng 1, và vì L2 bằng 1 nên L6 sẽ được cập nhật bằng LRU[0] và bằng 1. Tương tự với các step sau đó.

Đoạn code assembly như hình dưới sẽ được dùng để kiểm tra hoạt động của bộ 8-way set associative cache.

Ban đầu, các giá trị thanh ghi từ x1 đến x8 được khởi tạo để thuận tiện cho quá trình kiểm tra kết quả.

Ở dòng 10, giá trị thanh ghi x1 sẽ được lưu vào set thứ nhất (mỗi set chứa 8 ways). Tiếp đến x2, x3 cũng được lưu vào set thứ nhất, vì mỗi set chứa 8 block nên lúc này 3 block chứa giá trị của x1, x2, x3 và còn 5 block chưa có giá trị.

Ở dòng 13, giá trị thanh ghi x8 được lưu và set thứ 2. Dòng 14, giá trị ở set thứ 2 sẽ được load vào x10.

Tương ứng các dòng sau đó, x4, x5, x8 (vì x8 được lưu vào địa chỉ 64 đã lưu trước đó nên block có địa chỉ 64 sẽ thay thế x5 bằng x8), x6, x7, x8 lưu vào set thứ nhất. Lúc này set thứ nhất đã lưu đủ 8 giá trị (set thứ nhất đầy).

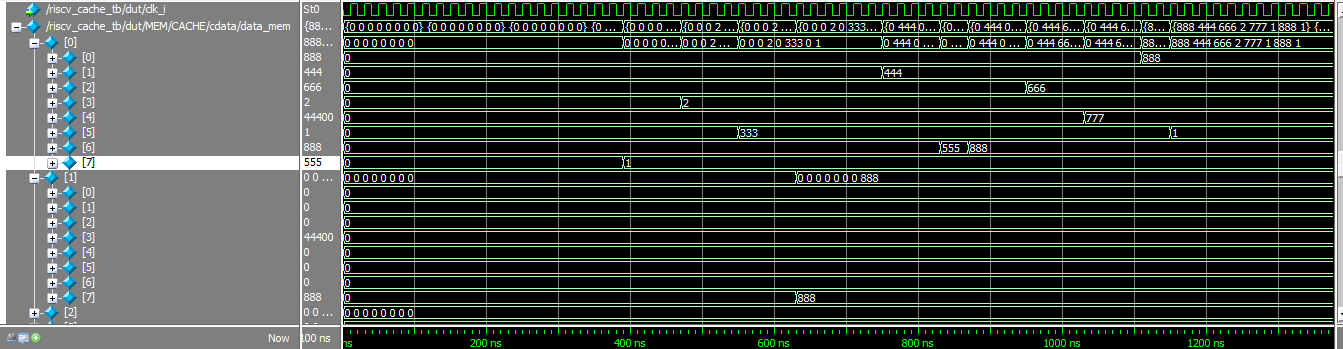
Do đó, lệnh ở dòng 21, ô có địa chỉ 32 chứa giá trị 333 (thanh ghi x3) sẽ được thay thế bằng 1 (thanh ghi x1).

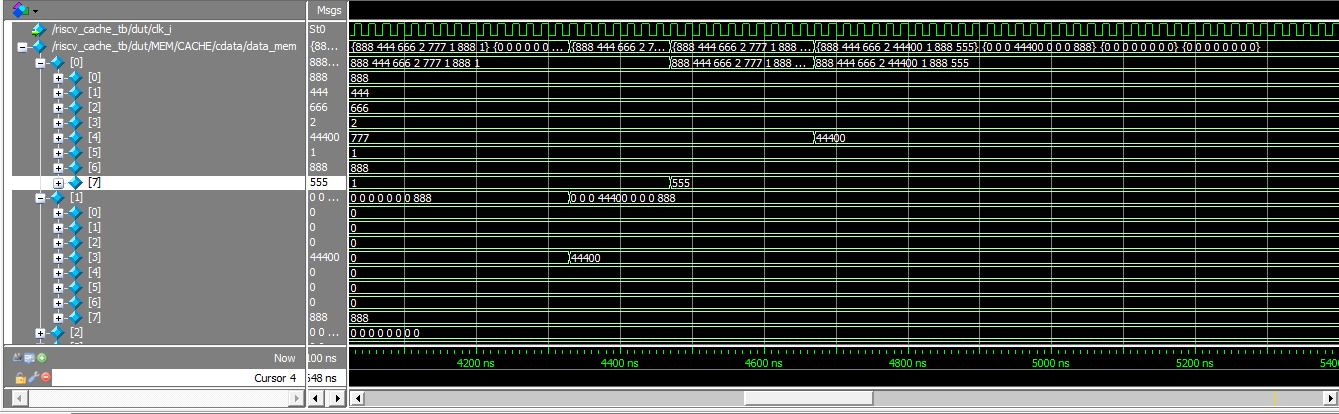
Từ dòng 23 đến dòng 25, giá trị x9 sẽ được tính toán và có giá trị sau cùng là 888 nhân 50 và bằng 44400. Sau đó x9 được lưu vào set thứ 2.

Dòng 27, ô có địa chỉ 16 sẽ cập nhật giá trị của x2.

Ở dòng 28, vì set thứ nhất đã đầy và không có block nào chứa địa chỉ khớp với địa chỉ 144, do đó 1 block nào đó (được quyết định bởi khối LRU) sẽ được lưu lại giá trị vào memory và cập nhật giá trị mới.. Tương tự với dòng lệnh 29.

Dưới đây là waveform mô tả đoạn code assembly phía trên.





Hình 1.4. Waveform cho đoạn test assembly

Như dữ liệu đã tính toán từ hình 1.3, giá trị ban đầu sẽ được lưu vào block 7 tiếp đến lần lượt là block 3, 5, 1, 6, 2, 4, 0. Và mỗi lần truy cập sau đó, LRU tree sẽ được cập nhật và đưa ra địa chỉ của block tiếp theo cần được ghi ở từng set.

**Multilevel Cache Organisation**

Link: <https://www.geeksforgeeks.org/multilevel-cache-organisation/>

**Multilevel Caches** is one of the techniques to improve Cache Performance by reducing the “MISS PENALTY”. Miss Penalty refers to the extra time required to bring the data into cache from the Main memory whenever there is a **“miss”** in the cache.

**Average access Time For Multilevel Cache:(Tavg)**

Tavg = H1 \* C1 + (1 – H1) \* (H2 \* C2 +(1 – H2) \*M )

where   
H1 is the Hit rate in the L1 caches.   
H2 is the Hit rate in the L2 cache.   
C1 is the Time to access information in the L1 caches.   
C2 is the Miss penalty to transfer information from the L2 cache to an L1 cache.   
M is the Miss penalty to transfer information from the main memory to the L2 cache.

The multilevel cache hierarchy in modern processors results in a dramatic performance improve ment in comparison to an otherwise equivalent system that lacks cache memory. Caching allows the speediest processors to run with minimal DRAM access delay a large percentage of the time. Although cache memory adds substantial complexity to processor designs and consumes a great deal of valuable die area and power, processor vendors have determined that the use of a multi level cache memory architecture is well worth these costs.

**Advantages of Multilevel Cache Organization**:

**Reduced access time:** By having multiple levels of cache, the access time to frequently accessed data is greatly reduced. This is because the data is first searched for in the smallest, fastest cache level and if not found, it is searched for in the next larger, slower cache level.

**Improved system performance:** With faster access times, system performance is improved as the CPU spends less time waiting for data to be fetched from memory.

**Lower cost:** By having multiple levels of cache, the total amount of cache required can be minimized. This is because the faster, smaller cache levels are more expensive per unit of memory than the larger, slower cache levels.

**Energy efficiency:** Since the data is first searched for in the smallest cache level, it is more likely that the data will be found there, and this reduces the power consumption of the system.

**Disadvantages of Multilevel Cache Organization:**

**Complexity:**The addition of multiple levels of cache increases the complexity of the cache hierarchy and the overall system. This complexity can make it more difficult to design, debug and maintain the system.

**Higher latency for cache misses:** If the data is not found in any of the cache levels, it will have to be fetched from the main memory, which has a higher latency. This delay can be especially noticeable in systems with deep cache hierarchies.

**Higher cost:** While having multiple levels of cache can reduce the overall amount of cache required, it can also increase the cost of the system due to the additional cache hardware required.

**Cache coherence issues**: As the number of cache levels increases, it becomes more difficult to maintain cache coherence between the various levels. This can result in inconsistent data being read from the cache, which can cause problems with the overall system operation.

**Level 2 cache**

Modern high-performance processors generally contain a substantial bank of L2 cache on-chip. Unlike an L1 cache, an L2 cache typically combines instructions and data in a single memory region, representing the von Neumann architecture rather than the Harvard architectural characteristic of the split L1 cache structure.

An L2 cache is generally slower than an L1 cache but still much faster than direct accesses to DRAM. Although an L2 cache uses SRAM bit cells with the same basic circuit configuration, the L2 circuit design emphasizes a reduction in the per-bit die area and power consumption relative to the L1 design. These modifications permit an L2 cache to be much larger than an L1 cache, but they also make it significantly slower.

**References:**

Modern Computer Architecture and Organization (book)

Computer Organization and Design (book)

<https://www.geeksforgeeks.org/lru-cache-implementation/>

**Todo work:**

Design i-cache, 2 level cache, multi words in line.

**Future work:**

Make cache configurable (direct mapped or set associative, write policy, replacement algorithm, …)

Implement protocol (AXI) to connect cache with memory or CPU with cache.