Shruti Iyer, Apurva Raman, Bonnie Ishiguro
Computer Architecture 2016

Everything is a Cache: Memory and Caching

This is a survey and synthesis of our learning about memory, caches, and virtual memory. We heavily referenced Computer Organization and Design, Fourth Edition (Hennessy and Patterson).

Memory Hierarchy

Memory is the storage space in a computer for both data and the instructions required to process this data. To optimize for both space and speed, this memory is organized into a hierarchy. If the memory hierarchy is well designed, it will provide the illusion of giving the processor access to memory as large as the largest level of the hierarchy and as fast as the fastest level of the hierarchy. The figure below illustrates a typical memory hierarchy for a processor.

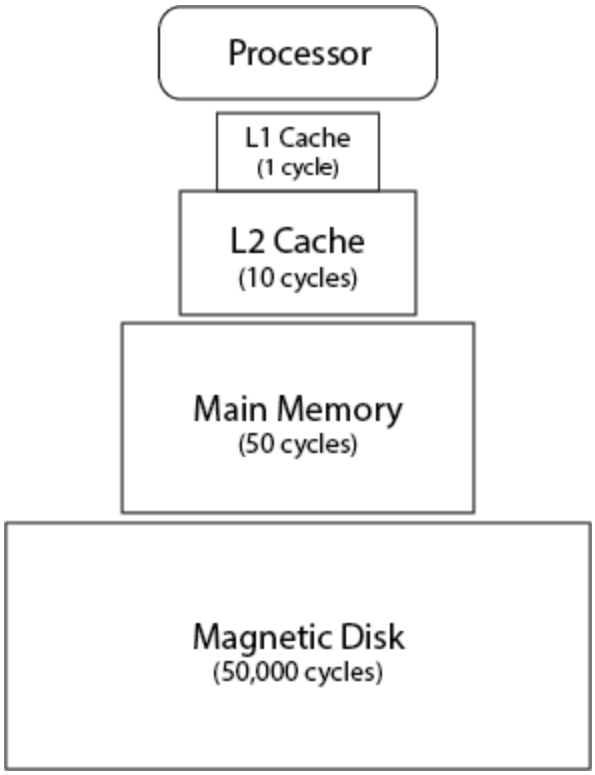

Figure 00000: Memory Hierarchy with Hit Times

Between any two levels of this hierarchy, the upper level is smaller, faster, and closer to the processor. The minimum unit of information that exists in this hierarchy is called a *block*. A processor's request for data results in either a *hit*, when the data exists in a block in the upper

level, or a *miss*. A miss results in the lower level being accessed to retrieve the block containing the data. The *hit time* is the time it takes to access the upper level, which includes the time needed to determine whether the access is a hit or miss. The *miss penalty* is the time to replace a block in the upper level with the corresponding block from the level level, plus the time it takes to deliver this block to the processor. The miss penalty is normally much more expensive than the hit time.

As shown in the figure above, a typical memory hierarchy is composed of a disk, main memory, and caches. *Magnetic disk* is a non-volatile type of memory that is used for more permanent storage, and it is therefore significantly slower to access than the other levels of the hierarchy. *Main memory* is a type of volatile storage known as random-access memory, where programs and data are stored when they are actively being used by the processor. A volatile type of memory requires power to retain stored information, while non-volatile type of memory can retrieve power even after having been power-cycled. *Virtual memory*, which will be described more in depth later on, allows main memory to be used as a cache for the magnetic disk. The L1 (on-chip) and L2 (off-chip) caches are used to reduce the time it takes the processor to access data from main memory by storing copies of data from frequently used locations.

The following figure demonstrates the process of requesting data from lower levels of the memory hierarchy. Program data (yellow) is requested from the L1 cache first, since L1 has the fastest access time. If it is not found there, the L2 cache is accessed. If the request results in a hit, the data found in L2 is stored in L1. Otherwise, main memory is accessed.

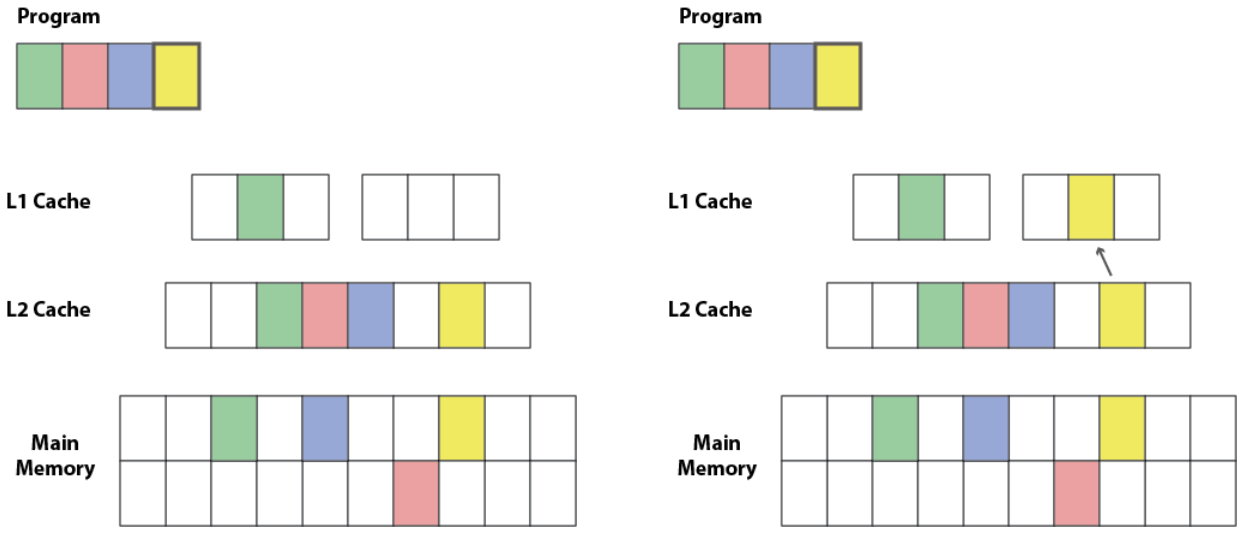

Figure 00001: Requesting Data from the L2 Cache

Cache Design

Shruti Iyer, Apurva Raman, Bonnie Ishiguro
Computer Architecture 2016

Cache memory makes for more efficient processing by reducing the overall data access time required to run programs. Caches take advantage of the principle of locality, which refers to access patterns of memory locations. *Temporal locality* occurs when data locations are re-referenced within a short time span. *Spatial locality* occurs when data locations with nearby addresses are referenced within a short time span. A program may exhibit more temporal locality or more spatial locality, depending on how it is written. The data that is likely to be most frequently accessed can be stored in a cache, which is smaller and faster than the level of the memory hierarchy that it serves. This data, therefore, will take fewer clock cycles to fetch, which can significantly reduce the overall runtime of the program.

Block placement for a cache can be implemented in a variety of ways. A *direct mapped* cache allows for a direct mapping from any block address in main memory to a single location in the cache. On the other hand, a *fully associative* cache allows a block address to be associated with any entry in the cache. A *set associative* cache is between these two extremes, in that it breaks the cache into n-sized sets, where a block address maps to a single set in the cache but can be associated with any entry within that set. The advantage of increasing associativity within a cache is that it decreases miss rate because it reduces the misses that compete for the same location in the cache. However, it increases hit time because either the full cache or a set within the cache needs to be searched to find a particular entry. Associative caches also require replacement rules to determine which block in the cache to replace when adding a new entry. The address in a block in memory includes its tag, which is used to identify the block within a particular set, its index, which is used to identify the set containing the address, and a block offset. A *valid bit* indicates whether the data in a block is in lower level of the hierarchy; if this bit is off, the block does not exist in the lower level. A *dirty bit* indicates whether the block has been modified but not yet updated in main memory.

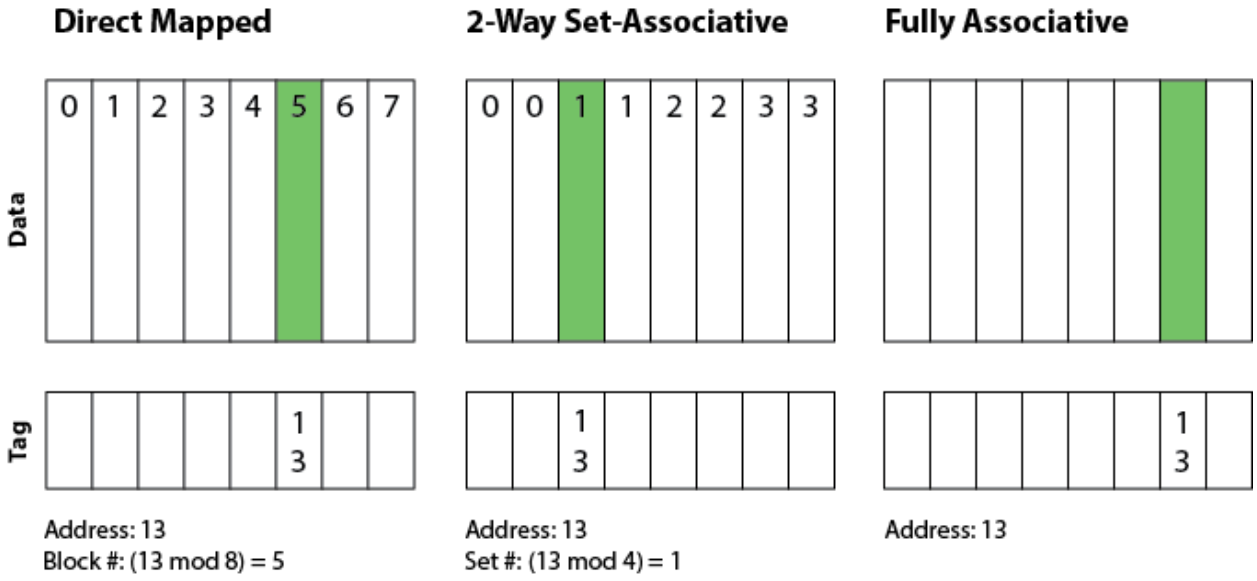

Figure 00010: Cache Block Placement Schemes

A cache miss is a request for data from the cache that cannot be filled, which results in a request to main memory or a lower level cache to access the data. In the case of a cache miss, the processor pipeline is stalled and the instruction is restarted after the requested data is fetched from the lower level memory into the cache.

There are two main strategies for writing to the cache, write-through and write-back. *Write-through* keeps main memory and the cache consistent by always writing data into both at the same time. This is costly, however, because writing to main memory would normally require the processor to stall while waiting for the write to finish. To improve performance, a write buffer can be used to store data while it is waiting to be written to memory, which frees the processor to continue instruction execution. *Write-back* is a different scheme that writes a modified block to the lower level of the memory hierarchy only when it is replaced in the cache. Virtual memory uses write-back because writing to the lower level of the hierarchy, the disk, takes significantly more clock cycles than main memory. It would be impractical to write to both levels at every write.

A *split cache* is a scheme in which a single level of the memory hierarchy is composed of two independent caches that operate in parallel, one that handles instructions and one that handles data. Although this scheme may result in a slightly increased miss rate due to dividing the instruction entries from the data entries, it also yields a doubled cache bandwidth by supporting both kinds of accesses simultaneously. L1 caches are normally split caches, while L2 caches are normally shared.

Multi-level caching is a technique that aims to reduce the miss penalty by adding additional levels to the hierarchy. The main memory cache uses this strategy, typically in a two-level structure composed of an L1 and an L2 cache. This allows the L1 cache to focus on minimizing hit time and the L2 cache to focus on reducing the miss rate. The second level of the cache is accessed whenever a miss occurs in the primary cache. Its access time is longer than that of the primary cache because it is not accessed as frequently, but its larger size reduces overall cache misses by allowing for more data to be stored in a faster level of the hierarchy.

As mentioned above, set associative and fully associative caches require replacement policy algorithms to choose which blocks to replace after they are full. Every cache has a number of sets that each hold a certain number of blocks. The memory addresses associated with a cache map to one of these sets. Direct-mapped caches have a set size of 1 block, so every memory address corresponds to a particular block in the cache. When the set size is greater than 1 however, such as with a set associative cache, a memory address can map to any block within the set. Therefore, when the cache is full, an algorithm is needed to choose which block that already lives in the cache to replace. First In First Out, Least Recently Used,

Most Recently Used, and Random Replacement are four examples of these cache replacement policies.

First In First Out (FIFO) replaces the block in the cache that was placed the earliest, with no reference to how frequently it has been accessed. This algorithm takes advantage of temporal locality.

Least Recently Used (LRU) replaces the block in the cache that was accessed the least recently. This algorithm keeps track of when each block was accessed using “age bits”, and tracking this can be expensive. When an item is accessed, LRU places it on the top of the cache. When the cache limit is reached, the items at the bottom of the cache are removed first.

Most Recently Used (MRU) replaces the block in the cache that was most recently accessed. This algorithm is designed for programs where older items are more likely to be accessed.

Random Replacement (RR) randomly chooses which block in the cache to replace. It is very simple as it does not keep track of anything, and is therefore used in ARM processors.

Virtual Memory

Virtual memory allocates space such that no two programs use the same locations in memory and allows the disk to be used as an extension of the main memory using caching principles; it is a specific application of caching in a different part of the hierarchy. While the L1 and L2 caches exist primarily to reduce access times, virtual memory also allow for extending the space in the main memory for a program using the disk and safely executing programs concurrently by using memory protection.

Virtual memory was originally conceived as a method of extending main memory with disk, essentially using main memory as a cache for disk on mainframes. This allowed a program to be longer than the amount of space in main memory. Before caching and virtual memory, this was done using overlays, which split the program into mutually exclusive modules that could be run separately. Caching became more widely used over time as the memory gap grew and there was an increasing need to speed up memory access.

Now, one of the primary uses for virtual memory is *memory protection*, which allows multiple programs to run at the same time without letting one program interfere with the memory other programs are using. Each program has its own allotted space in memory, called the *address space,* which is a range of locations that belongs to only that program.

It is useful to understand virtual memory in relation to caches, as many of the concepts in virtual memory are analogous to concepts in cache design. Some of the terminology for caching is also used for virtual memory, such as hit time, hit rate, miss penalty, and miss rate, but in other cases different terminology is used. For the remainder of this section, the virtual memory terminology will be used.

Cache Design Concept	**Virtual Memory Concept**
Block	Page
Cache miss	Page fault
Caching algorithm	Page replacement policy

Figure 00011: Cache and Virtual Memory Terminology

When using virtual memory, the processor produces a virtual address for the data, which is comprised of the page number (the index into the page table corresponding to the page, like the tag) and the page offset (the location difference between the address of the byte of the you want and the start of the page). The virtual address is translated into a physical address in the main memory. The way this translation or mapping works is called *relocation.* The mapping of virtual addresses to physical ones is done before they are actually used to access

the memory. The physical addresses do not have to be contiguous because pages are a fixed size. This does not hold true for some cases, for example when segmentation is used instead of paging, but we will not discuss implementation of segmentation.

The mapping is stored and tracked with a page table, often supplemented with a TLB, as discussed in the section below. The page table is essentially arrays in memory which are indexed by the *virtual page number* (which is like a tag for this case) and contain the valid bit (which indicates whether the virtual page is in main memory) the virtual page number, and the offset. This mapping structure supports full associativity.

Since virtual memory accesses this disk on a page fault, the time to process one is on the order of millions of clock cycles, which means it has a very large miss penalty. Thus, one of the main considerations in designing virtual memory is how to reduce the miss rate.

One way this is done is by replacing pages efficiently after a page fault. When a page fault occurs, the OS takes over using an exception mechanism. The page needs to be found and placed into the main memory. The OS makes space on the disk (swap space) for each page of a process when starting a process and also creates a data structure to record where each virtual page is stored on the disk. It also creates a data structure that tracks which addresses and processes use each page. When page faults happen, the page that is replaced is something that will not likely be used soon. Predicting what pages won't be necessary is done by the caching algorithm. Choosing a caching algorithm that has a low miss rate at the cost of some additional complexity is worth it for virtual memory because the miss penalty is so high.

Another way in which the miss rate is reduced is through full associativity between the main memory and disk mapping. The cost of the the extra delay and space of a larger table is outweighed by the miss penalty, so it makes sense to have virtual memory be fully associative. Large pages also help with this; a typical page size is about 4 to 16 KB. For desktop and server applications, page size are getting larger, whereas for embedded systems, pages are tending towards smaller sizes. Virtual memory also uses write-back instead of write-through to reduce the time spent accessing the disk.

Shruti Iyer, Apurva Raman, Bonnie Ishiguro
Computer Architecture 2016

Translation Lookaside Buffer (TLB)

Translation Lookaside Buffer (TLB) is a special cache that keeps track of recently used translations from virtual addresses to physical addresses for faster retrieval. It contains a subset of virtual to physical page mappings that are in the page table. Just like any regular cache, TLB contains a tag field along with other auxiliary values like Dirty bit and Valid bit.

Whenever the program needs to access a page in the physical memory, it first looks in the TLB for a translation from the virtual address to a physical address. If there is no matching entry in the TLB for a page, *TLB miss* occurs. The reason for a TLB miss could just be that the translation was absent from the TLB. In that case, the actual page table is examined. If the page does exist in main memory, the processor loads the translation into the TLB and tries to make that reference again. A TLB miss could also mean that the table doesn't contain an entry for the virtual address. *Page fault* occurs because the page lives on the disk and needs to be copied into the main memory.

[TLB in Memory Hierarchy Block Diagram]

A wide variety of associativity can be used to build TLB. Small sized, fully-associative TLBs are used because they have lower miss rate because of full associativity. Also, since the size of the TLB is small, the cost of full associativity is not too high. But, with full associativity, it becomes difficult and expensive to track entries and choose an entry to replace with. Hence, some systems use large sized, small-associative TLBs.

How TLB works with VM and Caches

Cache and main memory work together as a memory hierarchy. So a page cannot exist in cache if it doesn't exist in main memory. The Operating System removes any pages from cache if it decides to move a page from the main memory to disk. OS also removes that page entry from the TLB and page table. So any attempts to access that page will expectedly result in a page fault.

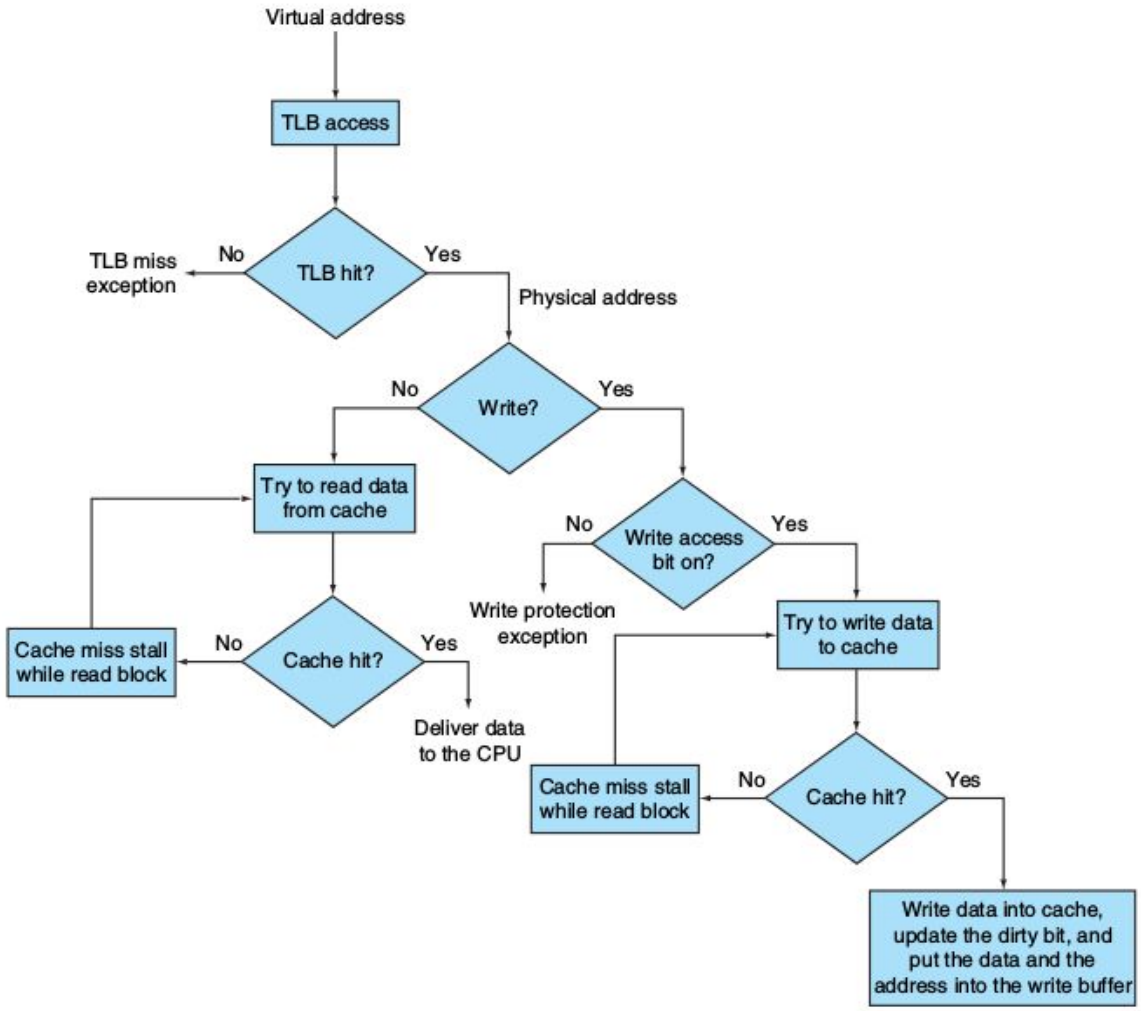

Figure 00100: Chapter 5, Computer Organization and Design, Pg 506

TLB MIsses and Page Faults

TLB Misses might indicate two things:

1. Page is present in memory and only need to create corresponding entry in TLB
2. Page is not in memory. Need to transfer control to the OS to deal with a page fault

Handling a TLB miss or a page fault requires using the exception mechanism to interrupt the active process, transferring control to the operating system, and later resuming execution of the interrupted process. A page fault will be recognized sometime during the clock cycle used to access memory.

To restart the instruction after the page fault is handled, the program counter of the instruction that caused the page fault must be saved. Additionally, normal instruction execution need to be stopped; a TLB miss or page fault exception must be asserted by the

end of the same clock cycle that the memory access occurs. This will result in the next clock cycle beginning the exception processing rather than continue normal instruction execution. If the page fault was not recognized in this clock cycle, an instruction (for eg. load) could overwrite a register which would create problems when we try to restart the instruction.

On a page fault, the OS

1. Looks up the page table using the virtual address and find the location of the referenced page on disk
2. Chooses a physical page to replace; if the chosen page is dirty, it must be written out to disk before we can bring a new virtual page into this physical page.
3. Starts a read to bring the referenced page from disk into the chosen physical page.

TLB Walk-through

Let's see what happens when the CPU requests for a block using its virtual address.

1. Initially, the TLB is empty because no address translations have been requested. When, the CPU asks for the 'red' block's physical address, the TLB is first checked. Because it is empty, the page table is checked next for the translation. Once the page table provides the physical address, the TLB is updated with the translation and the block is accessed.

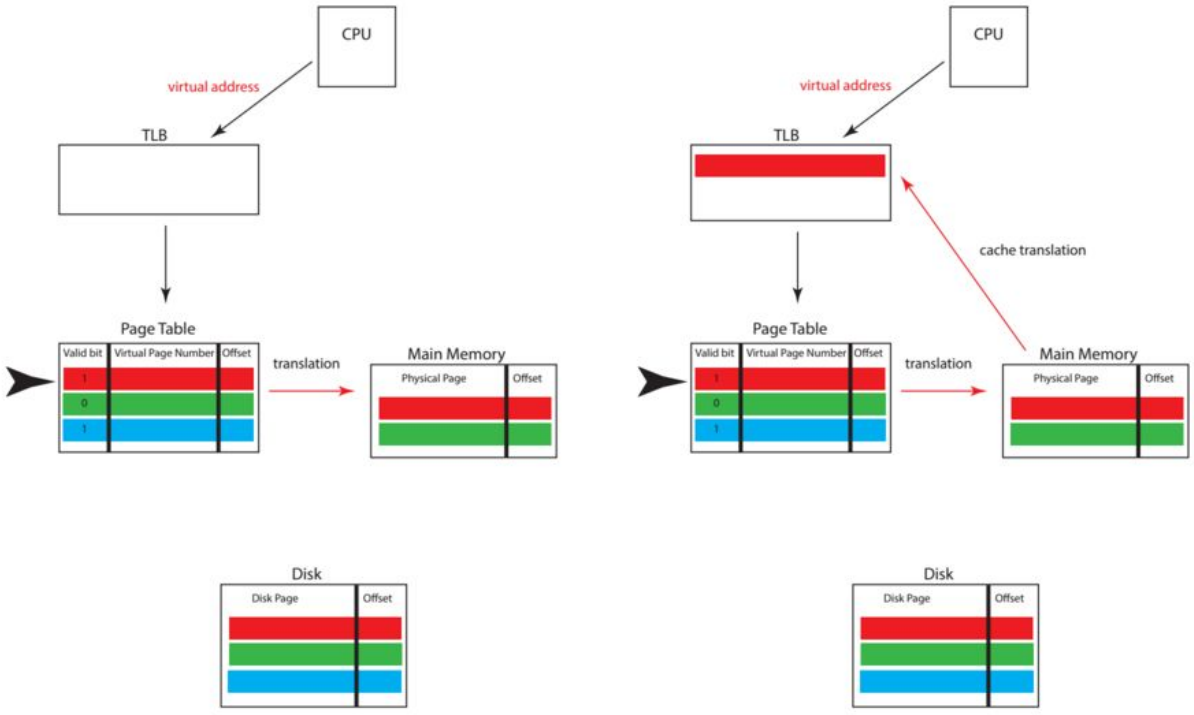

Figure 00101: TLB and Virtual Memory Walk-Through 1

2. Next, the 'green' block's translation is requested. The process is the same as the one for accessing the 'red' block. Now, the TLB is at capacity.

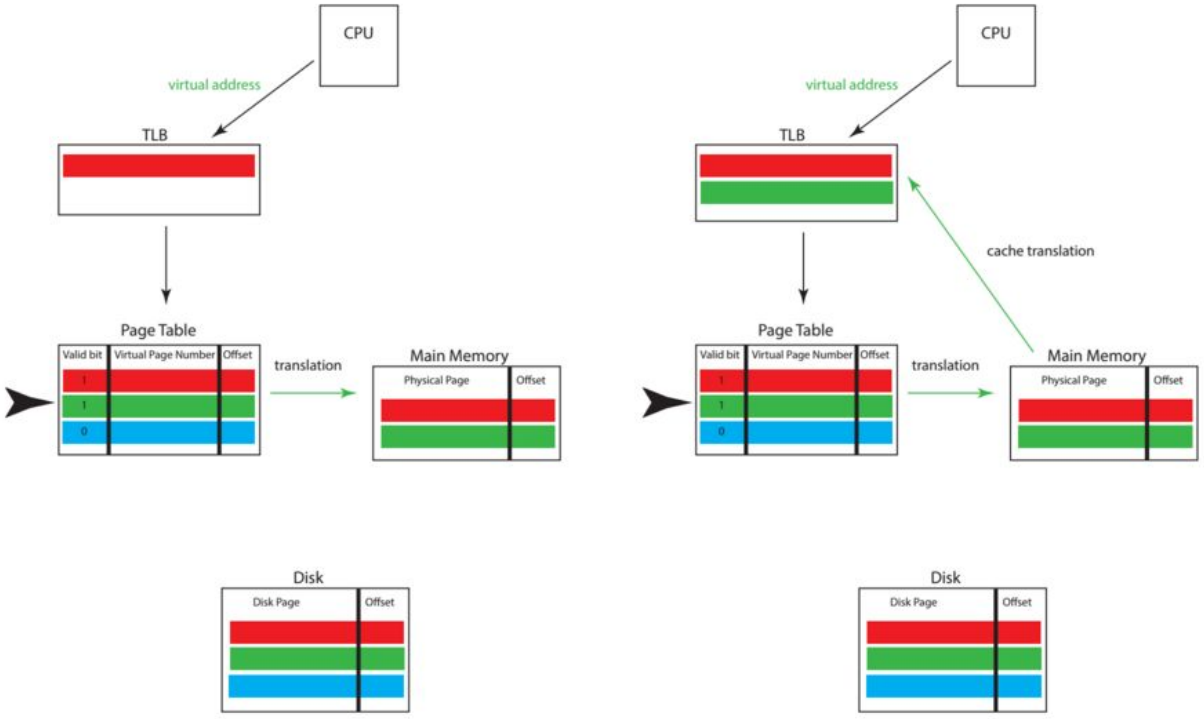

Figure 00110: TLB and Virtual Memory Walk-Through 2

3. Here, the 'blue' block's translation is requested. Once the page table returns the address translation. Since the TLB is full, one of the existing translation needs to be replaced. If we use LRU, we replace the least recently accessed block i.e. 'red' block's translation with the 'blue'.

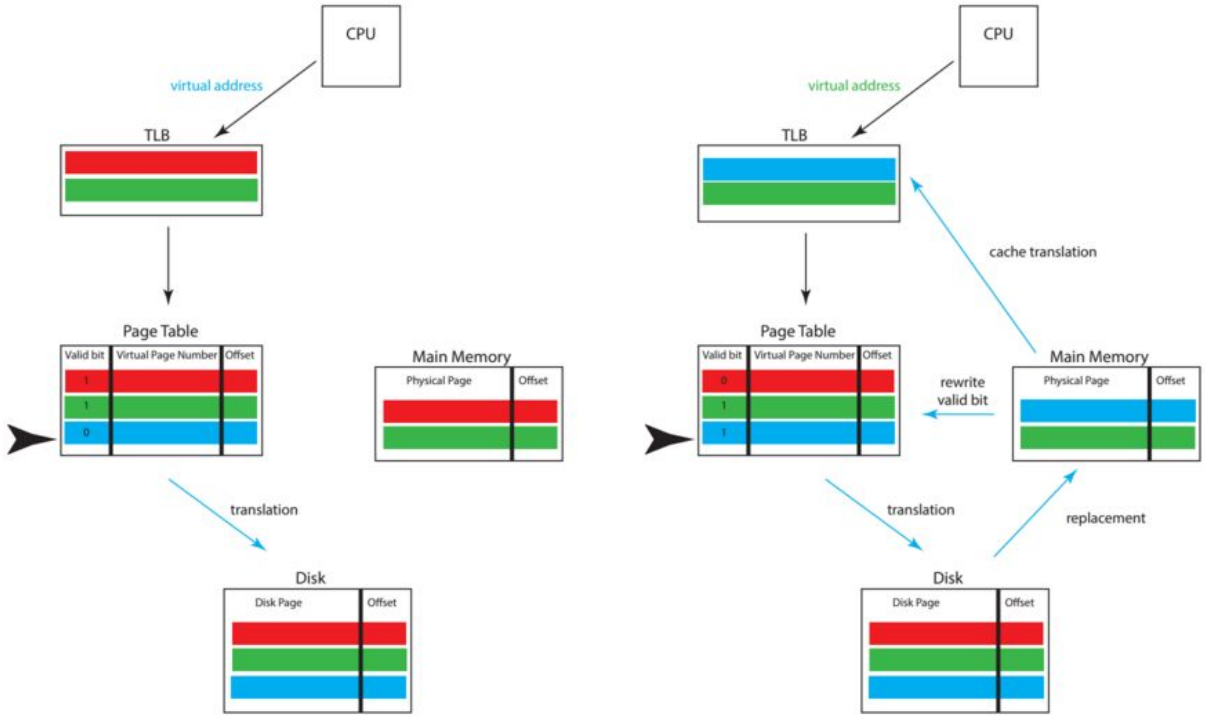

Figure 00111: TLB and Virtual Memory Walk-Through 3

Works consulted:

Bellis, Mary. "The Invention of the Intel 1103." *About.com*. About.com, 25 Aug. 2016. Web. 13 Dec. 2016.

"Caching in Theory And practice." Dropbox Tech Blog. N.p., n.d. Web. 13 Dec. 2016.

Kjell, Bradley. "Main Memory." *Main Memory*. Central Connecticut State University, n.d. Web. 13 Dec. 2016.

Li, Yamin. Computer Principles and Design in Verilog HDL. Hoboken: Wiley, 2015. Print.

Lin, Charles. "Fully Associative Cache." *Computer Organization*. University of Maryland, 2003. Web. 13 Dec. 2016.

"Most Efficient Cache Replacement Algorithm." Caching - Most Efficient Cache Replacement Algorithm - Software Engineering Stack Exchange. N.p., n.d. Web. 13 Dec. 2016.

Patterson, David A., and John L. Hennessy. “Computer Organization and Design”, Revised Fourth Edition the Hardware/Software Interface. Burlington: Elsevier Science, 2011. Print

"Virtual Memory Address Translation." *Virtual Memory Addressing*. University of Minnesota Duluth, n.d. Web. 13 Dec. 2016.