Chapter 5

Memory Hierarchy

Reading: The corresponding chapter in the 2nd edition is Chapter 7, in the 3rd edition it is Chapter 7 and in the 4th edition it is Chapter 5.

5.1 Overview

While studying CPU design in the previous chapter, we considered memory at a high level of abstraction, assuming it was a hardware component that consists of millions of memory cells, which can be individually addressed, for reading or writing, in a reasonable time (i.e., one CPU clock cycle). In this chapter, we will learn that memory is, in fact, built hierarchically, in different layers. This is because the ultimate goals in memory design are to

- have lots of it (gigabytes, terabytes, etc., enough to contain the entire address space),
- make it fast (as fast as CPU registers),
- make it affordable (not too expensive).

These goals are challenging to combine since fast memory, such as S-RAM, is very expensive, while cheaper memory, such as D-RAM, is slower and the cheapest memory, like, e.g., hard drive storage, is extremely slow compared to S-RAM or D-RAM. Building memory based on S-RAM only would make it too expensive though. Building memory based on D-RAM only would soften the price but slow down the overall performance significantly.

To achieve all of the three design goals, hardware designers combine a small amount of expensive, fast memory and large amounts of inexpensive, slow memory in such a way that the combination of the two behaves as if large amounts of fast memory were available (and that, at an affordable price). To create this **illusion** of lots of fast memory, we create a hierarchical memory structure, with multiple levels. An example of a structure with 4 levels is shown in Figure 5.1. Studying such hierarchical structure in more detail is the topic of this chapter.

Each level in the memory hierarchy contains a subset of the information that is stored in the level right below it:

$$\text{CPU} \subset \text{Cache} \subset \text{Main Memory} \subset \text{Disk}.$$

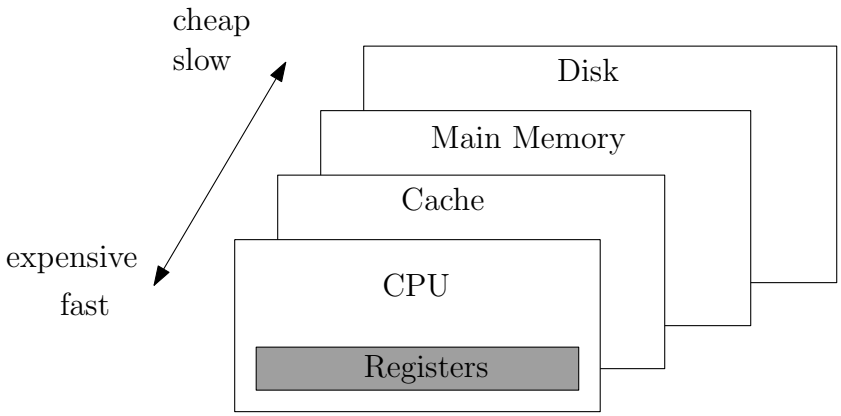

Figure 5.1: Memory hierarchy.

In the CPU, **registers** allow to store 32 words, which can be accessed extremely fast. If information is not present in one of the 32 registers, the CPU will request information from memory, by providing the address of the location where the required information is stored. First, the **cache** will verify whether it has the requested information available, or not. The cache is located close to the CPU and composed of a relatively small amount of fast and expensive memory (S-RAM). So, if the requested information is available in the cache, it can retrieved quickly. If not, **main memory**, which is significantly larger and composed of slower and cheaper D-RAM, is accessed. If the requested information is in the main memory, it is provided to the cache, which then provides it to the CPU. If not, the **hard drive**, which contains all information that is stored in the machine, is accessed. The hard drive offers a vast amount of storage space, at an affordable price, however, accessing it is slow. So, fundamentally, the closer to the CPU a level in the memory hierarchy is located, the faster, smaller, and more expensive it is.

In order to create the illusion of having lots of fast memory available, it is crucial that, with high probability, the cache contains the data the CPU is looking for, such that the main memory and especially the hard drive get accessed only sporadically. Fortunately, not all data in the entire address space is equally likely to be accessed: usually, only a small portion of the entire address space is being accessed over any range of 10-100 lines of code. This is because of **locality**:

- **Temporal locality**: recently accessed memory content tends to get accessed again soon. For example, most programs have simple loops which cause instructions and data to be referenced repeatedly. In

  ```
  for(i=0; i<100; i++)
  ```

  the memory location that stores the variable `i` will be referenced repeatedly, as well as the locations that contain the sequence of machine instructions that encode the loop.

- **Spatial locality**: memory content that is located nearby recently accessed memory content tends to be accessed as well, within the next 10-100 clock cycles. For example, program instructions are usually accessed sequentially, if no branches or jumps occur. Also, reading or writing arrays usually results in accessing memory sequentially.

Thus, by keeping the relatively small amount of data which is most likely to be accessed in the cache (i.e., small, fast memory, close to the CPU), memory access will occur rapidly most of the time, i.e., when the requested information is available in the cache: this is called a **hit**. If the requested information is not present in the cache, called a **miss**, it is copied from the next level down in the hierarchy (in this case, the main memory), in so-called **blocks**.

In general, for any two adjacent levels in memory hierarchy, a **block** is the minimum amount of information that is transferred between them, which can either be present or absent in the upper level (i.e., the level closest to the CPU). A **hit** occurs if the data required by the processor appears in some block in the upper level and a **miss** occurs if this is not the case and the lower level needs to be accessed to copy the block that contains the data requested by the CPU into the upper level (after finding the information at the lower level or an even lower level). Figure 5.2 shows this procedure, with the cache as upper level and the main memory as lower level. Since the data contained in one block are likely to get referenced soon (again), resulting in several cache hits, the average memory access time is likely to be low. Block size is usually larger for lower levels in the memory hierarchy. We introduce the following definitions to assess memory performance:

- Hit rate (h) = (# memory references found in the upper level) / (# memory references);
- Miss rate = $1 - h$;
- Hit time (T_h) : time to access the upper level (including the time to determine whether the access is a hit or a miss);
- Miss penalty (T_m) : time to replace a block in the upper level with the relevant block, retrieved from a lower level, plus the time the upper level takes to deliver the requested information (found somewhere in the block) to the CPU.

Usually, T_m is significantly larger than T_h. The average memory access time (AMAT) can be calculated as

$$\text{AMAT} = h \times T_h + (1 - h) \times T_m.$$

For example, if $h = 0.98, T_h = 1$, and $T_m = 10$, AMAT $= 0.98 \cdot 1 + (1 - 0.98) \cdot 10 = 1.18$. It is clear that making T_h and T_m as small as possible, while having h as close to 1 as possible, results in better memory performance.

5.2 Cache

Cache is the name that was chosen to represent the level of the memory hierarchy between the CPU and the main memory. When designing a cache, several choices have to be made, including block-size, how the cache stores information, how to handle writes, etc. Let's start with a simple example.

5.2.1 A simple direct-mapped cache with one-word blocks

Let's consider a cache with the simplest possible block size, i.e., one word (32 bits, 4 bytes) per block. When the CPU request the memory content at address A, the cache needs to find out whether

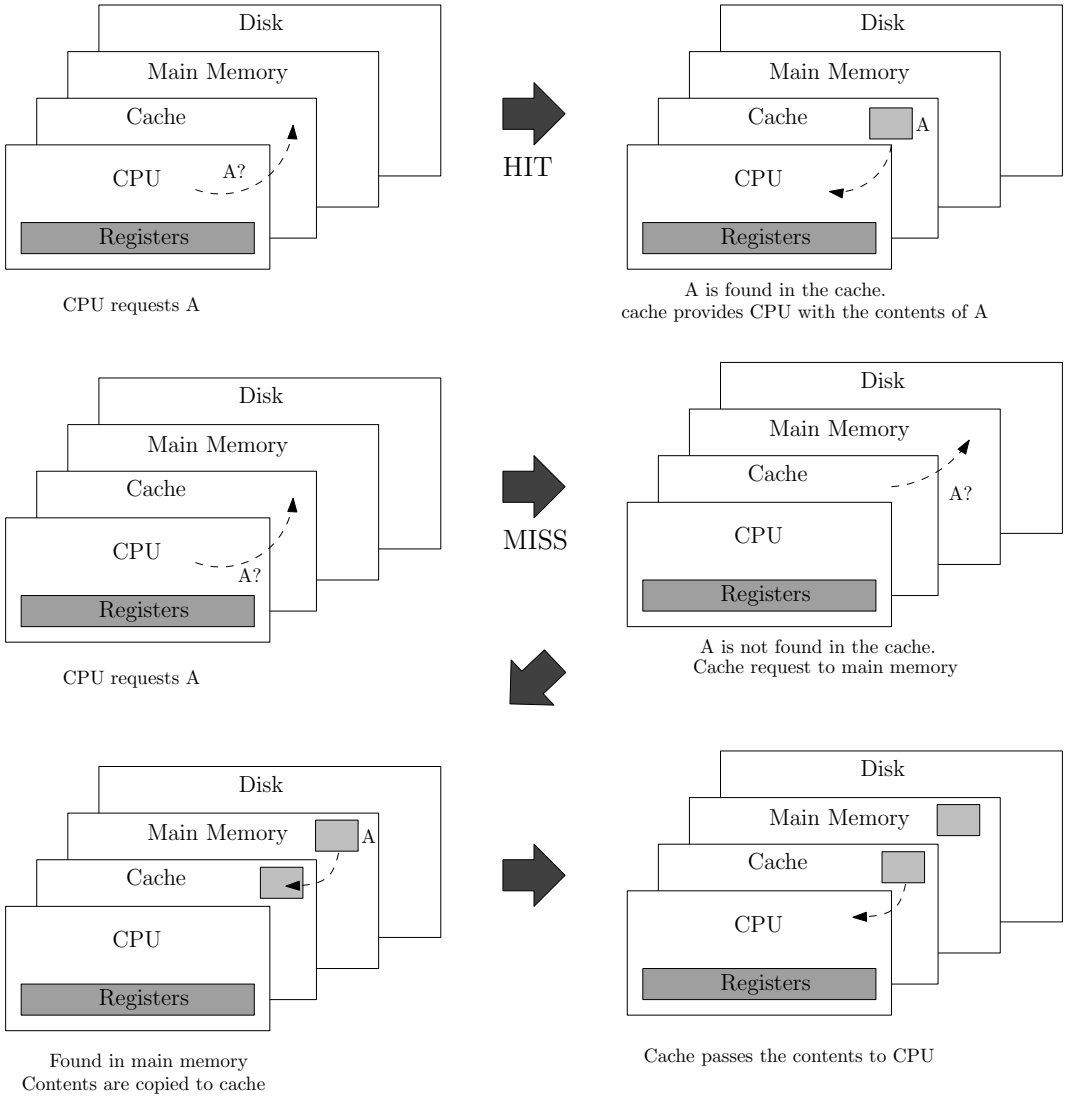

Figure 5.2: Memory access, resulting in a hit or a miss.

1. the content is available in the cache, and,

2. if yes, where it can be found in the cache.

The latter will depend on how the cache is organized. In this subsection, we assume the simplest possible organization, called **direct-mapped** cache. In direct-mapped cache, the content of a location in main memory can be stored at one and only one, specific location in the cache, i.e., it is "mapped" to exactly one location in the cache. Figure 5.3 shows how 32 sequential memory locations are cyclically mapped into only 8 cache locations.

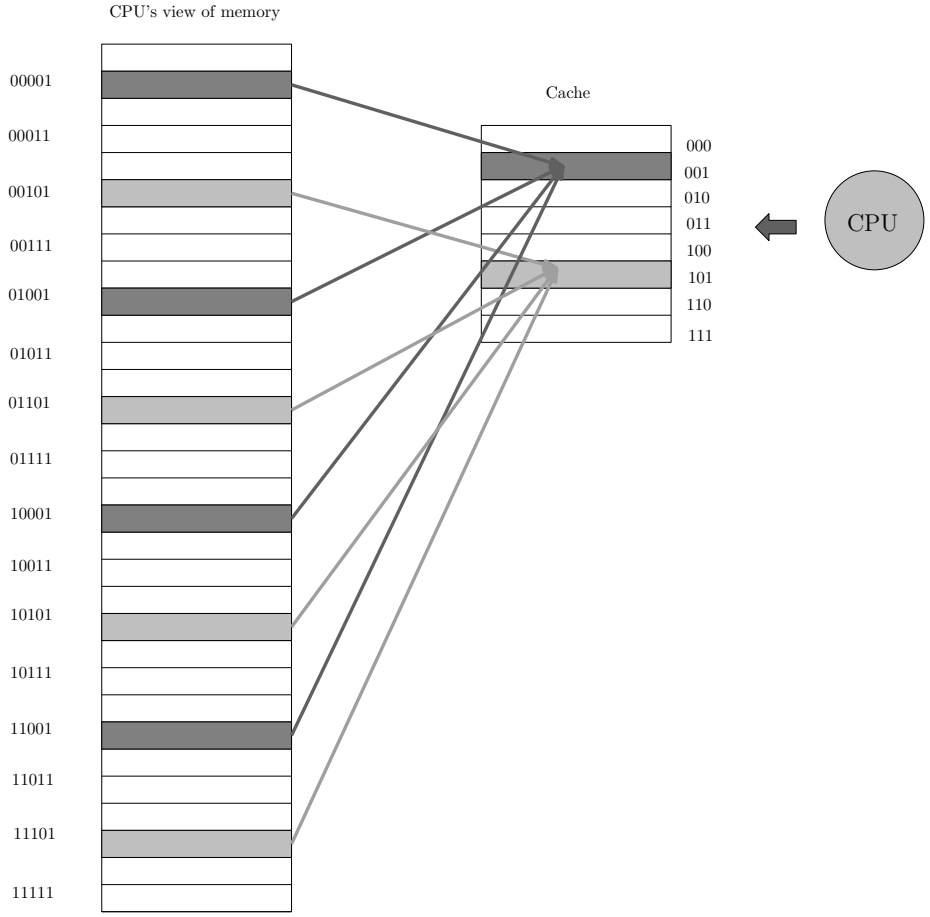

Figure 5.3: Direct-mapped cache.

Such cyclical mapping ensures that every block in main memory (a one-word block, in this subsection) could be stored at just one single location in the cache, indexed by:

(Block address in main memory) MOD (# blocks in the cache),

which uniquely depends on the address of the block in main memory. It is clear from Figure 5.3 that this is a many-to-one mapping. This answers the second question above: if the CPU requests the contents of a specific memory location, there is only one block in the cache, indexed by the result of the previous equation, that could possibly contain that

information. In fact, this equation can be implemented in a very simple way if the number of blocks in the cache is a power of two, 2^x, since

(Block address in main memory) MOD 2^x = x lower-order bits of the block address,

because the remainder of dividing by 2^x in binary representation is given by the x lower-order bits. This allows to index the cache by using the x least-significant bits of the block address, which is easy to implement.

For example, for a cache with 2^3 one-word blocks and a memory of size 2^5 words, as depicted in Figure 5.3, a CPU request for the content of the memory location with word address 01011_2 requires to access the cache block indexed by (01011_2) MOD (2^3) = 011_2, i.e., the 3 least-significant bits of the word address 01011_2. Next, the first question above needs an answer as, indeed, the cache block with index 011_2 could contain the content of memory location 11011_2 as well as 00011_2, 01011_2 or 10011_2 (as all have the same 3 least-significant bits).

Since each block in the cache can contain the contents of different memory locations that have the same x least-significant address bits, every block in the cache is augmented with a **tag** field. The tag bits allow to uniquely identify which memory content is stored in a given block of the cache. This answers the first question above, and helps to determine a hit versus a miss. For a direct-mapped cache, the tag will contain the "other" address info, i.e., the address info that is not used to index the cache: the most-significant bits of the memory address. For the previous example, the cache block with index 011_2 does contain the content of memory location 11011_2, and we have a hit, if the tag provided for that block in the cache is indeed 11_2.

In addition to the tag field, every block in the cache is provided with a **valid** bit, V, to determine whether its contents are representing a valid entry (V = 1), or whether the block just contains a random 0/1 sequence, which is invalid and to be ignored (V = 0). For example, when starting up the computer, all the valid bits are initialized to 0.

Example 5.2.1

Let's consider an example for the setup shown in Figure 5.3. At power-up, every cache line is invalid.

Index	V	Tag	Data (block = 32 bits)
000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
110	0		
111	0		

Let's consider the following sequence of memory references: 10110_2, 11010_2, 10110_2, 10000_2, 10010_2. For the first memory access, at 10110_2, the 3 LSB, to index the cache, are 110. The

corresponding block in the cache is invalid (V = 0), so we have a cache miss. The block containing the requested word is copied into the cache from the next level below in the memory hierarchy (i.e., the main memory), the tag bits are set to 10 (the most-significant bits of the word address 10110_2, that are not used to index the cache) and the valid bit is set (as the cache block is now valid), resulting in the following state of the cache.

Index	V	Tag	Data (block = 32 bits)
000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
110	1	10	Mem[10110_2]
111	0		

The next access is at word address 11010_2. The index bits are 010. The corresponding block in the cache is invalid again, so we have a cache miss, copy the appropriate block from main memory, set the tag bits to 11 and the valid bit to 1, resulting in the cache state below.

Index	V	Tag	Data (block = 32 bits)
000	0		
001	0		
010	1	11	Mem[11010_2]
011	0		
100	0		
101	0		
110	1	10	Mem[10110_2]
111	0		

10110_2 is accessed next. The index bits are 110. The corresponding block of the cache is valid (V = 1), with tag bits 10, which match the tag bits of the word address 10110_2. This implies a cache hit, so the cache can provide the CPU promptly with the requested data, Mem[10110_2]. 10000_2 is accessed next, with index 000, which corresponds to an invalid cache block and thus a miss. Copying the right block from main memory into the cache and adjusting tag and valid bit results in the following state of the cache.

Index	V	Tag	Data (block = 32 bits)
000	1	10	Mem[10000_2]
001	0		
010	1	11	Mem[11010_2]
011	0		
100	0		
101	0		
110	1	10	Mem[10110_2]
111	0		

Lastly, 10010_2 is accessed. The block indexed by 010 is valid, however, the tag bits of the word address, 10, don't match the tag of the corresponding cache block, which is 11. This implies the block indexed by 010, in the cache, is storing the memory word at 11010_2 and not the memory word at 10010_2. Therefore, we have a cache miss and replace this block in the cache by a new block, i.e., the contents of 10010_2 in main memory. After updating the tag, the cache has been updated as follows.

Index	V	Tag	Data (block = 32 bits)
000	1	10	Mem[10000_2]
001	0		
010	1	10	Mem[10010_2]
011	0		
100	0		
101	0		
110	1	10	Mem[10110_2]
111	0		

Using 32-bit byte addressing in MIPS R2000

We can now use this direct-mapped cache organization for the MIPS R2000 architecture we designed before, which uses 32-bit byte addressing. Assume we want to build a cache that contains 64 Kbyte of data. That will require space for 2^{16} bytes of data, i.e., 2^{14} words. Since each line in the cache contains a one-word block of data, this cache requires 2^{14} lines, to be indexed by 14 index bits (the 14 LSB of the word address). Ignoring the 2 least significant bits of the 32-bit byte address (the so-called "byte offset", to specify specific bytes in each word, is ignored because MIPS R2000 usually reads and writes words) leaves the 16 MSB of the address for the tag (32 - 14 index bits - 2 byte offset bits = 16 tag bits), as shown in Figure 5.4.

The total size of the cache is given by $(1 + 16 + 32) \times 2^{14} = 49 \times 16 \times 2^{10} = 784$ Kbits. So, building a cache to contain 64 Kbyte = 512 Kbits of data, requires 784 Kbits of memory. The 16 MSB of the address matching the tag, with the valid bit set, creates a hit.

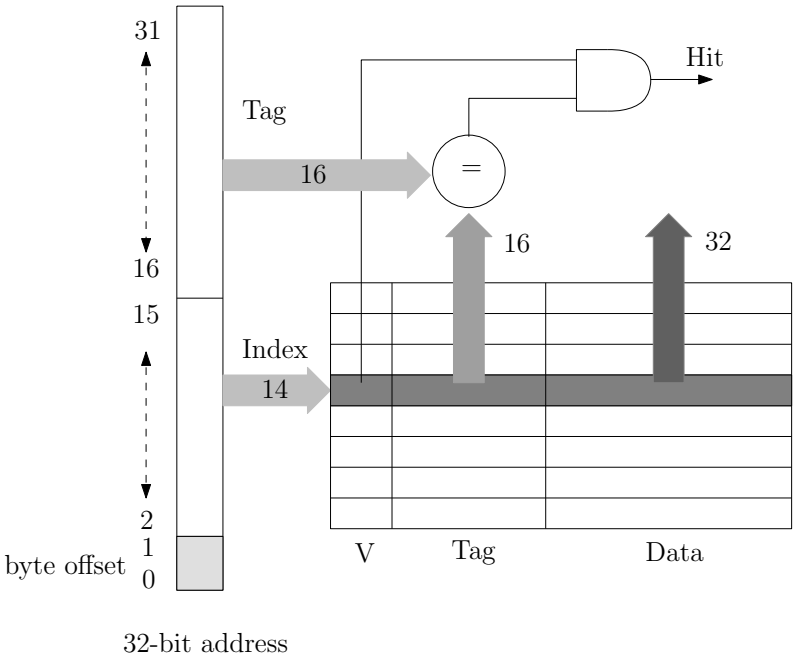

Figure 5.4: Direct-mapped cache with one-word blocks, providing 64 KBytes of data.

Handling cache misses

In case of a cache miss, time is spent on fetching the appropriate word from the main memory, copying it into the data part of the cache, writing the upper bits of the memory address in the tag part of the cache, and setting the valid bit. This is organized by the control unit in the CPU and will, clearly, take longer than one CPU clock cycle (as the CPU clock period is engineered for fast, local memory access, i.e., cache hits). While waiting for a cache miss being handled, the CPU "stalls", i.e., it freezes it current state and spends clock cycles not executing or changing anything.

Handling writes

Writing data is trickier than reading. If data is written into the cache only, main memory and cache become inconsistent and main memory no longer contains the most updated content.

To force consistency between the cache and the main memory, a **write-through** scheme can be used, writing to both the cache and the main memory, whenever a write occurs. Thus, when a write miss occurs, the appropriate block is fetched from main memory, copied into the cache and the write is executed, by overwriting both the cache block as well as the location in main memory. This can be time consuming (it accesses the main memory twice) and since the CPU stalls until the writing process is completed, the CPI can get affected quite badly.

Example 5.2.2

Assume that a main memory write takes 10 extra clock cycles, that 10% of the instructions are writes and that the initial CPI (without writes) is 1. If the CPU stalls for each write

until the main memory write is completed, the effective CPI is given by

$$0.9 \times 1 + 0.1 \times (1 + 10) = 0.9 + 1.1 = 2.$$

Although write-through is a simple write strategy, it can deteriorate the performance significantly. To avoid the latter, we can use a **write buffer**, which stores the write data while it's waiting to be written into main memory. Instead of writing to the main memory directly, the cache writes to the write buffer and allows the CPU to continue execution while data is written from the write buffer to main memory. A write buffer is a simple FIFO which can contain a small number of words (e.g., 4). This is shown in Figure 5.5.

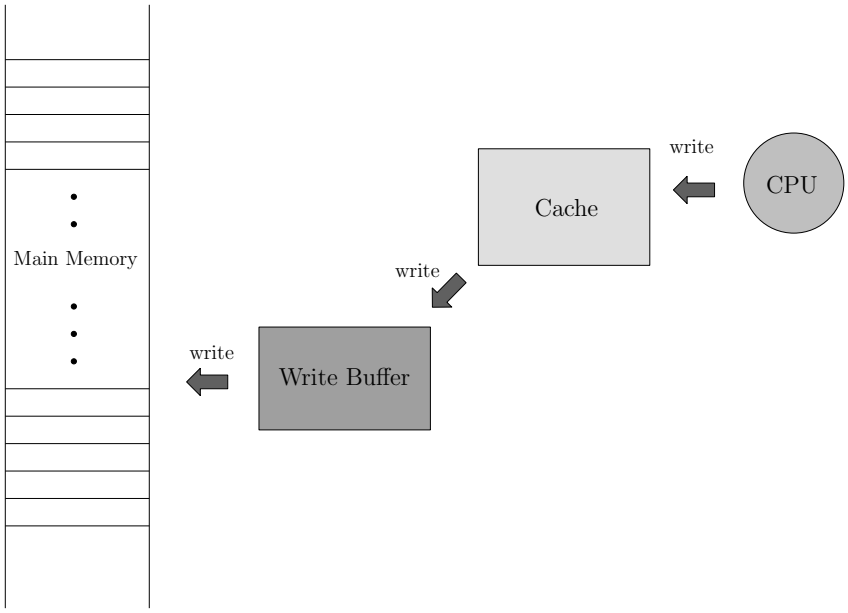

Figure 5.5: Cache writes data to the write buffer, which then writes it to the main memory.

When a write to main memory completes, the corresponding entry in the write buffer is freed up. If, however, the write buffer is full when the CPU reaches a new write instruction, the CPU stalls until there is an empty entry in the write buffer. The key assumption to make a write buffer effective is that the rate at which the processor does writes is less than the rate at which the data can be written into the main memory. By allowing to buffer multiple words, the write buffer can accommodate sporadic write bursts without loss of efficiency.

On the other hand, a **write-back** scheme could be used for writes. In a write-back scheme, a new value only gets written to the corresponding block in the cache and not in the main memory. The modified block is written back from the cache to the lower level in the memory hierarchy only when it is being replaced in the upper level. This scheme can improve the performance if the CPU issues writes at a higher rate than the rate at which the data can be written into the main memory. However, it is more complex to implement. To keep track of whether a cache block has been modified by a write (and, therefore, needs to be written back when replaced), every line in the cache is provided with an additional bit, the **dirty bit**. A block only gets written back into main memory, upon replacement, if the dirty bit is set.

5.2.2 Direct-mapped cache with 4-word blocks

In the previous subsection, we considered the simplest possible cache structure, where each block contains one word. In this subsection, we want the cache to take better advantage of spatial locality, by increasing the block size: by having a cache block larger than one word, multiple words, adjacent to the one the CPU requested, will be fetched from main memory when a cache miss occurs. Because of spatial locality, it is likely that the adjacent words will be requested soon as well, which would increase the cache's hit rate.

For a cache with a block size of 4 words (16 bytes) and the same total number of data bits as the cache in the previous subsection (64 KByte), we obtain the layout shown in Figure 5.6. A cache miss will fetch four words from the main memory (including the word requested by the CPU).

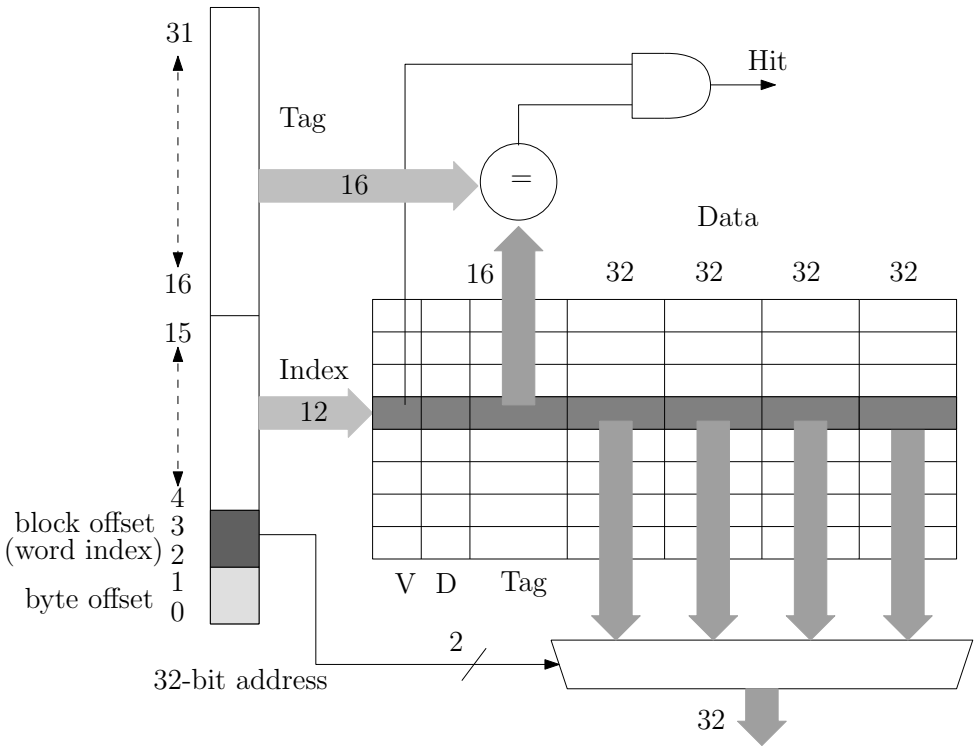

Figure 5.6: Direct-mapped cache with 4-word blocks, providing 64 KByte of data.

Comparing the direct-mapped cache with 4-word blocks, in Figure 5.6, to the direct-mapped cache with one-word blocks, in Figure 5.4, we notice the following differences:

- The presence of **block offset** bits, also called **word index** bits. A variation in these bits will specify a different word, however, as these are the least significant bits in the word address, they specify a different but adjacent word, in one and the same block, corresponding to one single cache index. Thus, in case of a cache miss, the cache will copy the same block from main memory into the cache, if only the block offset differs. With a block size of 4 words, the block offset (or, word index) consists of 2 bits. As these bits specify which specific word in a block is requested by the CPU, a multiplexer is provided to output the correct word, given the block offset;

- Only 12 index bits remain. Since the size of the cache is still 64 Kbyte, increasing the block size with a factor 4, compared to the previous subsection, reduces the number of lines (blocks) in the cache with that same factor, to 2^{12} blocks, which only requires a 12-bit index (bit 4 to 15 of the address).

The 16 MSB in the address (bit 16 to 31) are still the tag bits and a hit is detected in the same way as before.

The effect of a larger block size on the average access time

As mentioned before, the average memory access time is given by $h \times T_h + (1 - h) \times T_m$. Reducing the miss rate, $1-h$, and the miss penalty, T_m, improve the performance. Increasing the block size:

- Will reduce the miss rate, because of spatial locality, as long as the block size doesn't grow too large. This improves the performance. However, if the block size grows that large that each block becomes a significant portion of the cache, the miss rate may start to increase again. Indeed, for very large block sizes, the number of blocks that can be held in the cache becomes rather small. This creates severe competition between blocks, for space in the cache, such that it is likely for a block to get removed from the cache (to make space for a new block) before many of its words have been accessed. This will deteriorate the miss rate. Moreover, when a block is very large, many words in the block might no longer be "spatially local". Hardware designers need to find an optimal block size, that minimizes the miss rate.

- Will increase the miss penalty (the larger the block, the longer it takes to copy the block from the next level below in the memory hierarchy). This partially compensates the gain in miss rate and adversely affects the performance. To minimize the increase in miss penalty, there are several possible approaches. First, as we will see in the next subsection, the interface with main memory can be adjusted, to make it more efficient. Second, one could apply one of the following schemes:

  - **Early restart**: the CPU resumes execution as soon as the requested word has been copied into the cache (which is often before the entire block has been copied); this works well for caching instructions, if the main memory can deliver the next word by the time the next instruction is required.
  - **Fetch requested word first**: first, the requested word (not the whole block) is transferred to the cache and the CPU resumes execution; then, the remainder of the block is transferred (so, this scheme is the same as early restart, if the requested word happens to be the first word in the block).

5.2.3 The interface between cache and main memory

So far, we assumed that the main memory is one word wide and that all main memory accesses are made sequentially, using a connection, called the **bus**, between cache and main memory that is one word (32 bits) wide. This is shown in Figure 5.7.

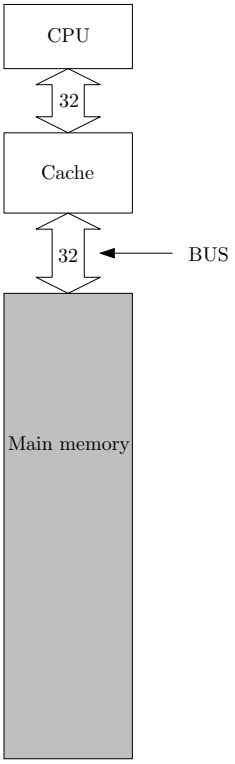

Figure 5.7: One-word wide memory.

Given the physical distance between cache and main memory, the bus is usually clocked at a slower rate than the CPU, to allow electric signals to propagate over the entire length of the bus, within one **memory bus clock cycle** (**MBCC**). Since the bus is clocked at a slower rate, 1 MBCC usually equals several CPU clock cycles. Assume that it takes

- 1 MBCC to send an address from the cache to the main memory;
- 15 MBCCs to access the main memory (D-RAM) once;
- 1 MBCC to send one word of data from the main memory to the cache.

If a cache block consists of 4 words, each of the words in a block needs to be accessed and transferred sequentially, in case of a cache miss. To copy the 4-word block from the main memory to the cache, it takes

$$
\begin{aligned}
&1 \text{ MBCC} \quad \text{(to send the appropriate address to main memory)} \\
&+ 4 \times 15 \text{ MBCCs} \quad \text{(to access four words, sequentially, in main memory)} \\
&+ 4 \times 1 \text{ MBCCs} \quad \text{(to transfer the four words, sequentially, to the cache)} \\
&= 65 \text{ MBCCs}
\end{aligned}
$$

to accommodate a cache miss. During these 65 MBCCs, the CPU stalls, which makes for a very high miss penalty (also, remember that 1 MBCC is significantly longer than 1 CPU clock cycle). Using an "early restart" or "fetch requested word first" scheme is one way to

reduce the miss penalty. In this subsection, we investigate direct adjustments to the interface between cache and main memory, to reduce the 65 MBCCs and improve the miss penalty.

First of all, it is clear that the latency to fetch the first word from memory cannot be improved. To reduce the miss penalty, we therefore focus on *increasing the memory bandwidth*, to fetch the next, adjacent words from main memory faster and reduce the second (4×15 MBCCs) and third (4×1 MBCCs) term in the previous equation.

Interleaved memory

In interleaved memory, the memory bandwidth is increased by widening the memory but not the interconnection bus, as shown in Figure 5.8.

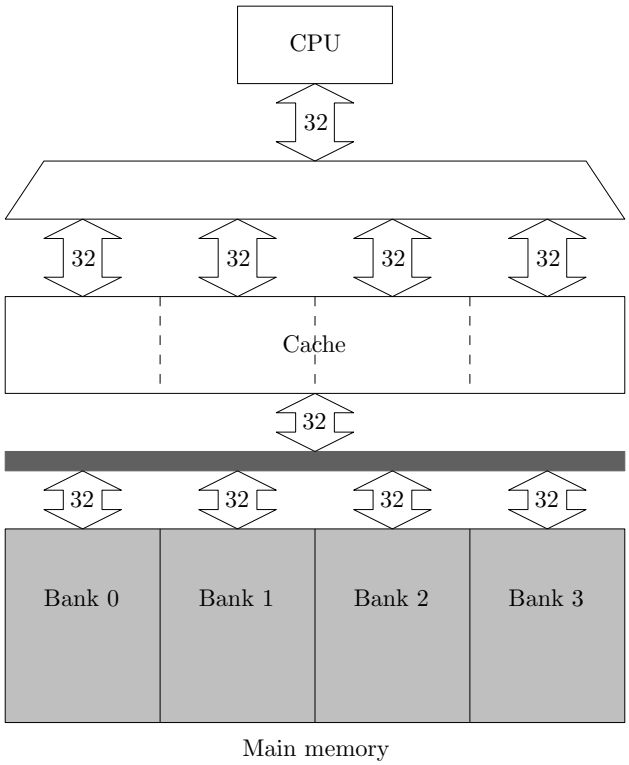

Figure 5.8: Interleaved memory.

Main memory is divided into four one-word wide *banks*, which can be accessed in parallel. The cache spends 1 MBCC to send out the address. Then, the 4 adjacent words of the appropriate block are read, in parallel, from the 4 one-word wide memory banks, which reduces the memory access time from 4×15 MBCCs to 1×15 MBCCs. Finally, the 4 words are sent to the cache, sequentially, over the bus, which takes 4×1 MBCCs. That reduces the total time to copy a block from main memory to $1 + 1 \times 15 + 4 \times 1 = 20$ MBCCs.

4-word wide memory

Further improvement is possible by widening both the memory and the interconnection bus, as shown in Figure 5.9.

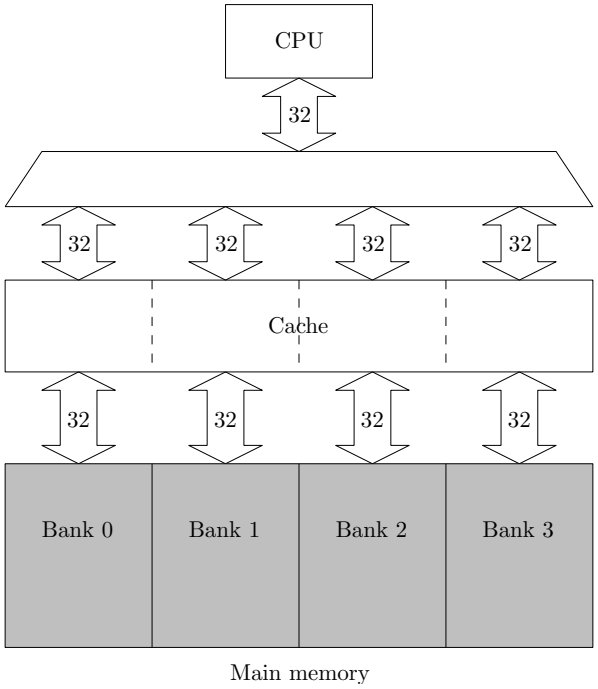

Figure 5.9: 4-word wide memory.

We now have a 4-word wide memory bus. This allows to send four words from main memory to cache, in parallel, in just 1 MBCC. This reduces the total time required to copy a block from main memory further to $1 + 1 \times 15 + 1 \times 1 = 17$ MBCCs.

5.2.4 Fully associative and set associative cache

In **direct-mapped cache**, a block from the next level below, in the memory hierarchy, can only be stored in one specific location in the cache. For such cache organization

- the replacement strategy is easy to implement (a new block can only go in one specific location);
- there is only one tag to compare to, when accessing the cache (i.e., the tag on the cache line indexed by the index bits);
- the hit rate is suboptimal (some lines in the cache might be used only sporadically, while others are used much more frequently; there is no flexibility to place frequently used blocks in sporadically used lines of a direct-mapped cache).

The other extreme would be a cache where a block from the lower level in the hierarchy can be stored in any location of the cache. There are no index bits for such type of cache (since a block can be stored anywhere) and the entire block address provides the tag. This is called **fully associative cache**, which

- requires a more complex replacement strategy (to decide where to place a new block), which makes the hardware slower and more expensive;

- requires to compare against all tags, in parallel (since any line of the cache could contain the requested word);
- allows a more optimal hit rate (by replacing the least frequently used blocks).

A hybrid form of both extremes of cache organization is the so-called **set associative cache**, where a block, copied from a lower level in the memory hierarchy, can be placed in a fixed number of locations in the cache. The group of all possible locations where a block could be placed is called a **set**. So, each block in the main memory maps to a unique set of locations in the cache, given by the index field (sometimes also called the *set index*), and a block can be placed in any element of that set (called a *set entry*). Therefore, all tags within a set must be checked, in parallel, to determine a cache hit (which is better than checking all tags, in fully associative cache). It is clear that direct-mapped cache and fully associative cache are special cases of set associative organization.

For sets of size 2, i.e., two-way set associative cache, the layout is shown in Figure 5.10, assuming blocks contain 4 words and the total amount of data that can be stored is, again, 64 KByte.

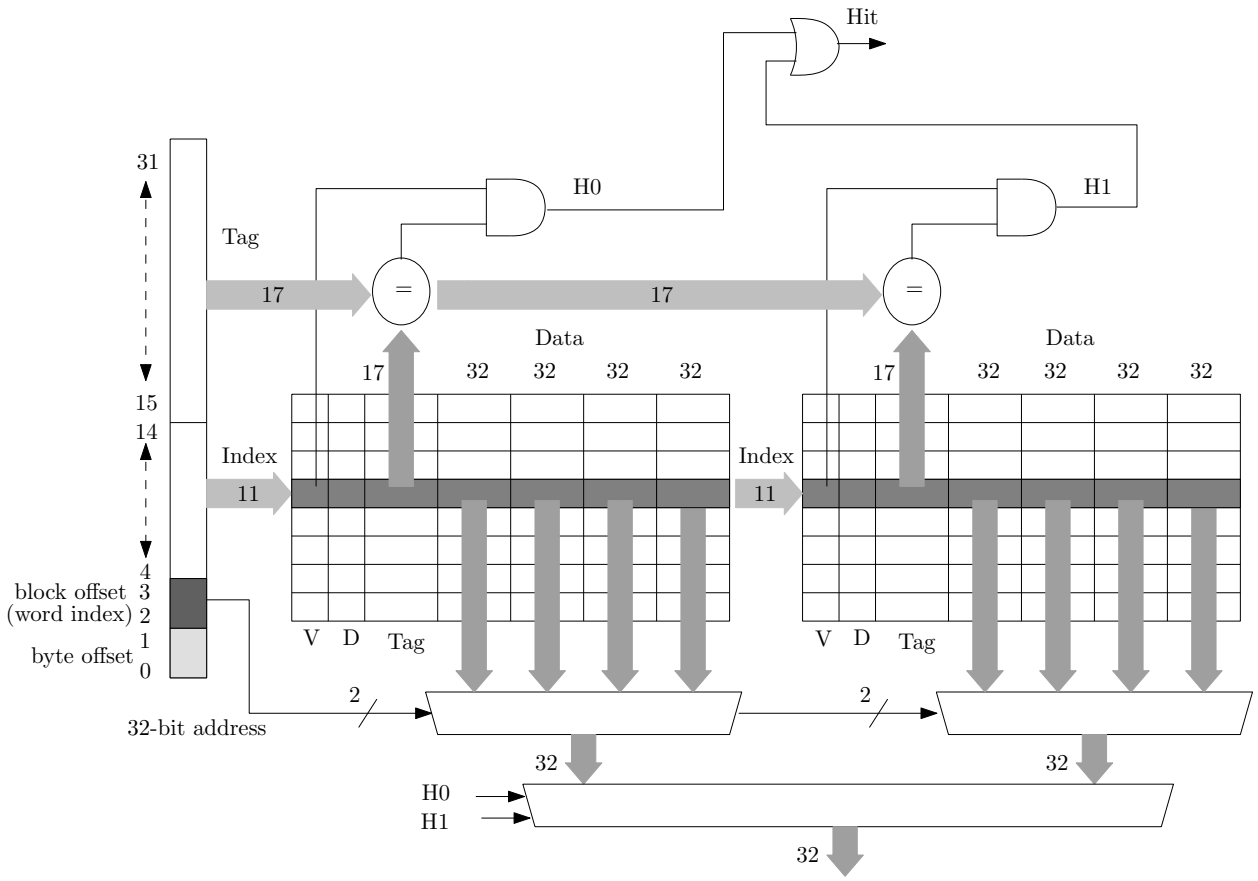

Figure 5.10: 2-way set associative cache with 4-word blocks.

We represent the layout using one line for each set and as many parallel tables as there are blocks per set. Thus, a given block can only get stored in a specific line of the cache and,

within each line, it can be stored either in the left or the right table. The CPU provides tag, index and block offset to both tables. A hit signal is generated from either the left or the right table, or we have a miss. If one of the hit signals is set, we have a hit. The 32-bit multiplexer in the bottom determines which entry in the set is being output, based on which table generated the hit (H0 or H1 set). Storing 64 KByte of data requires 2^{11} sets (lines) in the cache (since there are two 4-word blocks per set), indexed by 11 bits. The remaining 17 bits are tag bits.

In case of a cache miss, a *replacement policy* needs to be followed, to decide which block in the corresponding set will be replaced. This requires additional decision hardware. There are several possible replacement policies:

- **Random** : randomly select which block to replace (cheap and easy in hardware, however, it is possible to replace a block that is often accessed).
- **LRU** (Least Recently Used) : replace the block which was least recently referenced (more complex and expensive hardware, but lower miss rate, assuming that the most recently referenced words are most likely to be referenced again in the near future).
- **FIFO** : replace the block that was replaced least recently (so, blocks are replaced based on the order in which they were copied, rather than accessed).

5.2.5 Cache's impact on performance

In the previous subsections, we described the effectiveness of a cache in terms of the average memory access time (AMAT). Eventually, what we are really interested in is the overall performance of the machine. We illustrate this with an example. Assume we want to compute the effective CPI of a machine with the following specifications:

- instruction cache miss rate is 5%;
- data cache miss rate is 10%;
- frequency of loads and stores is 33%;
- cache miss penalty is 12 clock cycles;
- CPI without cache misses, CPI_{NCM}, is 4.

Moreover, we assume that the CPU must stall in case of cache misses.

To compute the effective, total CPI, we need to know how many clock cycles are spent, averaged per instruction, on cache misses. Once we know that number (let's call it CPI_{CM}), we can simply add it to CPI_{NCM}, to obtain the total, effective CPI, since the CPU will stall on cache misses (so, the cache miss cycles can simply be added). To compute CPI_{CM}, let's first compute the average number of cache misses over I instructions and then divide that by I.

$$\begin{aligned} \#\text{ cache misses over } I \text{ instructions} &= 0.05 \times I + 0.1 \times 0.33 \times I \\ &= 0.083 \times I, \end{aligned}$$

since 5% of all I instructions leads to a cache miss, to load the instruction, and 10% of all data transfer instructions (where 33% of all I instructions are data transfer instructions) leads to a cache miss, to transfer the data. Since the cache miss penalty is 12 clock cycles, the number of CPU stall cycles, spent on cache misses, for I instructions, is given by

$$\begin{aligned}\#\text{ CPU stall cycles over } I \text{ instructions} &= \text{cache miss penalty} \times \#\text{ cache misses} \\ &= 12 \times 0.083I \\ &= I,\end{aligned}$$

or, averaged per instruction,

$$CPI_{CM} = \#\text{ CPU stall cycles over } I \text{ instructions/number of instructions} = I/I = 1.$$

Therefore, the total, effective CPI is $CPI_{NCM} + CPI_{CM} = 4 + 1 = 5$. Keep in mind that, if no cache was available, all memory accesses would require stall cycles (i.e., take a long time) and the effective CPI would be significantly worse.

Next, let's assume that hardware designers have improved the CPU performance with a factor 2, i.e., the CPU clock frequency has been doubled, however, memory speed has not been improved, i.e., memory access time is unchanged. Since we have improved one part of our system, the overall performance should improve. To compare the performance before and after the improvement, we compute

$$\begin{aligned}\frac{\text{Performance of slower system}}{\text{Performance of faster system}} &= \frac{ET_F}{ET_S} \\ &= \frac{CPI_F \cdot I_F \cdot T_F}{CPI_S \cdot I_S \cdot T_S} \\ I_F = I_S \Rightarrow &= \frac{CPI_F \cdot T_F}{CPI_S \cdot T_S} \\ T_S = 2 \cdot T_F \Rightarrow &= \frac{1}{2} \cdot \frac{CPI_F}{CPI_S}\end{aligned}$$

From before, we know $CPI_S = 5$. To compute the total, effective CPI for the faster system, CPI_F, we observe that, since the memory access time has not changed, the cache miss penalty increases from 12 to 24 clock cycles (since a CPU clock cycle is now half as long as before). The number of misses will not change, so the doubling of the miss penalty will double the CPI_{CM} to 2. The overall CPI is then $CPI_{NCM} + CPI_{CM} = 4 + 2 = 6$. Therefore,

$$\begin{aligned}\frac{\text{Performance of slower system}}{\text{Performance of faster system}} &= \frac{1}{2} \cdot \frac{CPI_F}{CPI_S} \\ &= \frac{6}{2 \cdot 5} = \frac{3}{5} = \frac{1}{1.67}.\end{aligned}$$

We conclude that the performance of the overall system has improved by a factor 1.67, which is less than the improvement for the CPU (a factor 2). This is because of Amdahl's law.

5.3 Virtual Memory

5.3.1 Motivation

Just like the cache between main memory and the CPU aids at speeding up main memory access (and creates the illusion that lots of fast memory is available), similarly, the main memory can acts as a "cache" to the next level of memory in the hierarchy, i.e., the disk, using a technique called **virtual memory**. Historically, virtual memory was motivated by two factors:

1. To remove the programming burdens that arise from only having a small, limited amount of main memory available.

   Indeed, programmers often desire a lot more memory than what is available. Virtual memory addresses this by allowing a program to address a large range of virtual memory locations, covering a lot more memory than there is main memory available. The physical main memory will contain a subset of the data in that large, virtual memory and what's not in main memory will be stored on the hard disk. The CPU directly addresses the virtual memory and therefore always issues a virtual memory address.

2. To allow efficient and safe sharing of memory amongst multiple programs, when lots of programs are running concurrently.

   Indeed, at compile time, it is not known which programs will share the main memory with other programs. In fact, even when a program is running, the programs that are sharing the main memory with it usually change dynamically throughout its execution. To prevent one program from altering another program's address space, one compiles each program in its own "virtual" address space, i.e., a separate range of memory locations that is accessible to that program only, as shown in Figure 5.11. A subset of all programs' virtual memories' content will be available in the physical main memory (shared by all programs, including the O.S.), while the remainder is stored on the hard drive. Virtual memory implements the translation of a program's virtual address space to physical memory addresses and enforces protection of a program's address space from other programs.

Based on the same principles of temporal and spatial locality, virtual memory accesses can be made efficient by storing frequently used data in the main memory and other data on the hard disk.

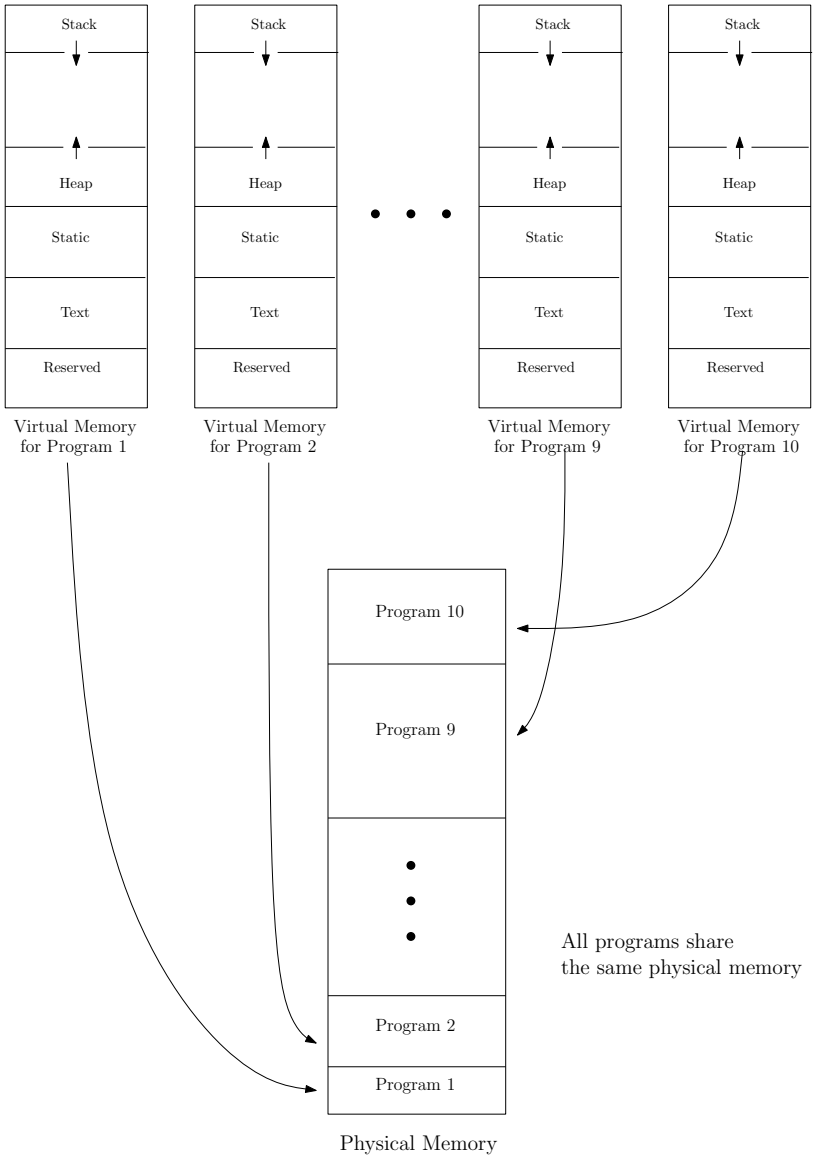

Figure 5.11: Programs share the same physical memory, which contains subsets of each program's virtual memory.

5.3.2 Virtual memory design

To understand virtual memory, we can focus on one single program. The extension to multiple programs is obtained by simply keeping a page table (as explained below) for each individual program.

Virtual memory concepts are very similar to the concepts explained for cache between CPU and main memory. However, the terminology is different since both were historically motivated in a different way: while virtual memory was designed to allow programmers to work with more memory than what was physically available and share the same physical

memory safely and efficiently amongst many programs, cache was merely designed to speed up access to main memory. A virtual memory **page** is analogous to a cache block, and a **page fault** is the equivalent of a cache miss.

From virtual address to physical address

From the CPU's point of view, the only memory it interacts with is the (possibly very large) virtual memory. Therefore, it issues a virtual memory address and then waits to access (read / write) the contents of the corresponding virtual memory location. Physically, that virtual memory location is mapped either to a physical memory location, at a specific physical memory address, or to the hard drive (at a specific hard drive location), as illustrated in Figure 5.12.

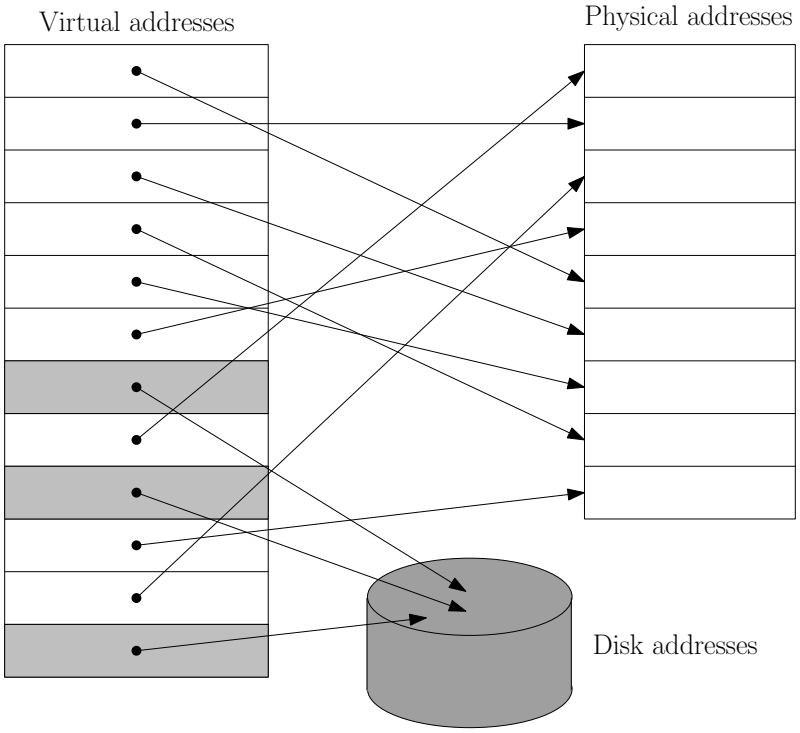

Figure 5.12: A virtual address is mapped to a location in physical memory or a location on the hard disk.

In case the virtual memory location is mapped to a physical memory location, the virtual address needs to be translated to the corresponding physical address, which can then be used to access the physical main memory. This is depicted in Figure 5.13, for 4GB virtual memory (requiring a 32-bit virtual address) which is mapped to 64MB of physical memory (requiring a 26-bit physical address).

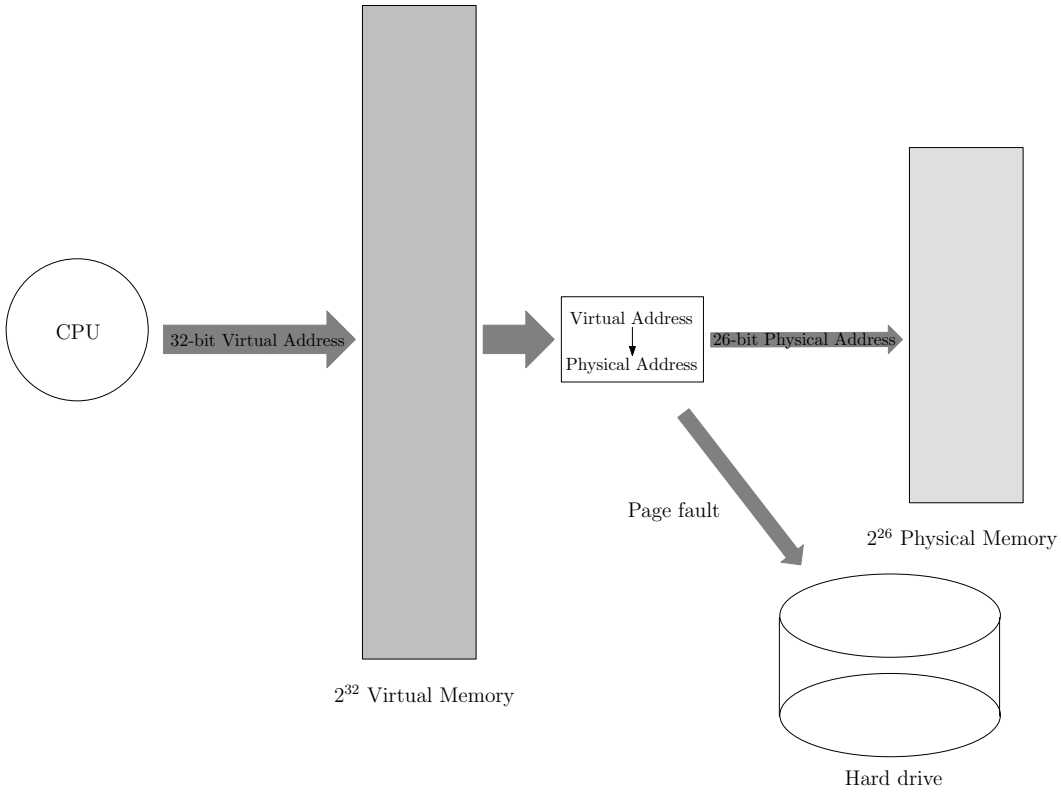

Figure 5.13: Virtual address is translated to physical address.

Just like with cache, a page is the smallest block of information that is transferred between the main memory and the hard drive and is either entirely present or entirely absent in the main memory.

- If a page is present in the physical main memory, accessing any word in the corresponding page in virtual memory will require translating the virtual address of the page, called **virtual page number**, into its corresponding physical address, called **physical page number**. Figure 5.14 shows how a virtual address is composed of a **virtual page number** and a **page offset**. The page offset is similar to the block offset in the case of cache. It specifies a specific memory location (byte) within a page and the number of page offset bits depends on the page size. To translate a virtual address into a physical address, the page offset field is simply copied: only the virtual page number requires translation, into a **physical page number**.

- If a page is not present in main memory, the virtual page number cannot be translated into a physical page number in the main memory: the page needs to be read from the hard disk instead. This is called a **page fault**. In case of a page fault, the required page is copied from the hard disk to the main memory. This will take an "enormous" amount of time (compared to accessing main memory), which is expected to be compensated by many more efficient page accesses (in main memory) due to temporal and spatial locality. Now, the page has a physical page number and the virtual page number can be translated into a physical page number.

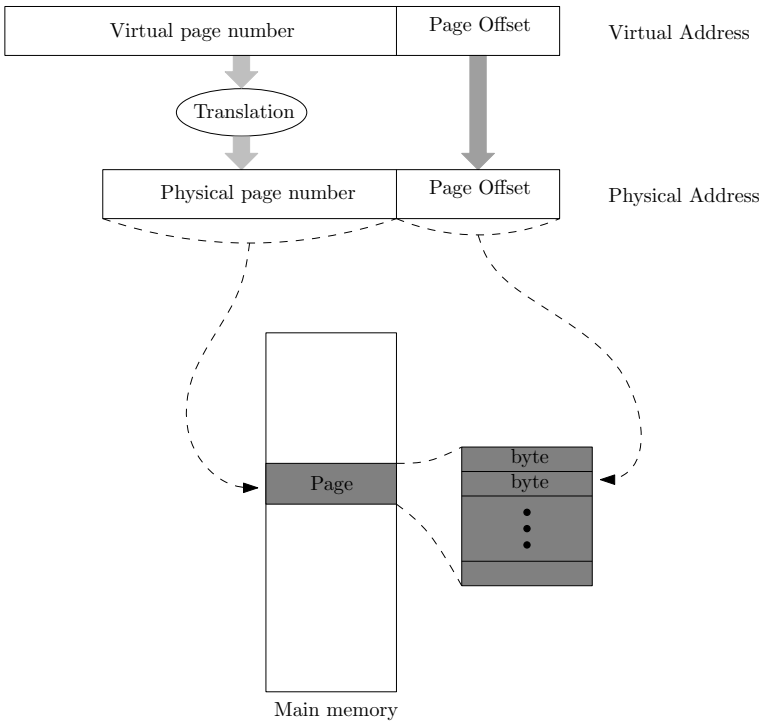

Figure 5.14: Page number and page offset.

Design Decisions

A key factor in making design decisions for virtual memory is the fact that page faults will take thousands of clock cycles to process, since the hard disk is slow. Because of the enormous miss penalty:

1. Pages should be large enough to amortize the miss penalty: typical page sizes range from 4KB to 16KB.

2. Virtual memory is organized to minimize the miss rate as much as possible: the mapping from virtual page numbers to physical page numbers is **fully associative**, i.e., a virtual page can be mapped to any physical page.

3. Page fault handling is done in software (by the OS). The time spent to access the disk is so long that the software overhead is small compared to the disk access time. Using software makes it easier to implement clever algorithms to choose the page to replace.

4. A write-through scheme does not work, since the disk access time is too long. We will use a write-back scheme instead: if a page is in the physical memory, write to the physical memory only; when the page is replaced, copy it back to the disk. A dirty bit is used to keep track of whether a page has been written.

Placing and finding a page: the page table

To implement a fully associative mapping from virtual page numbers (VPN) to physical page numbers (PPN), we use a **page table**, as shown in Figure 5.15. The page table is a

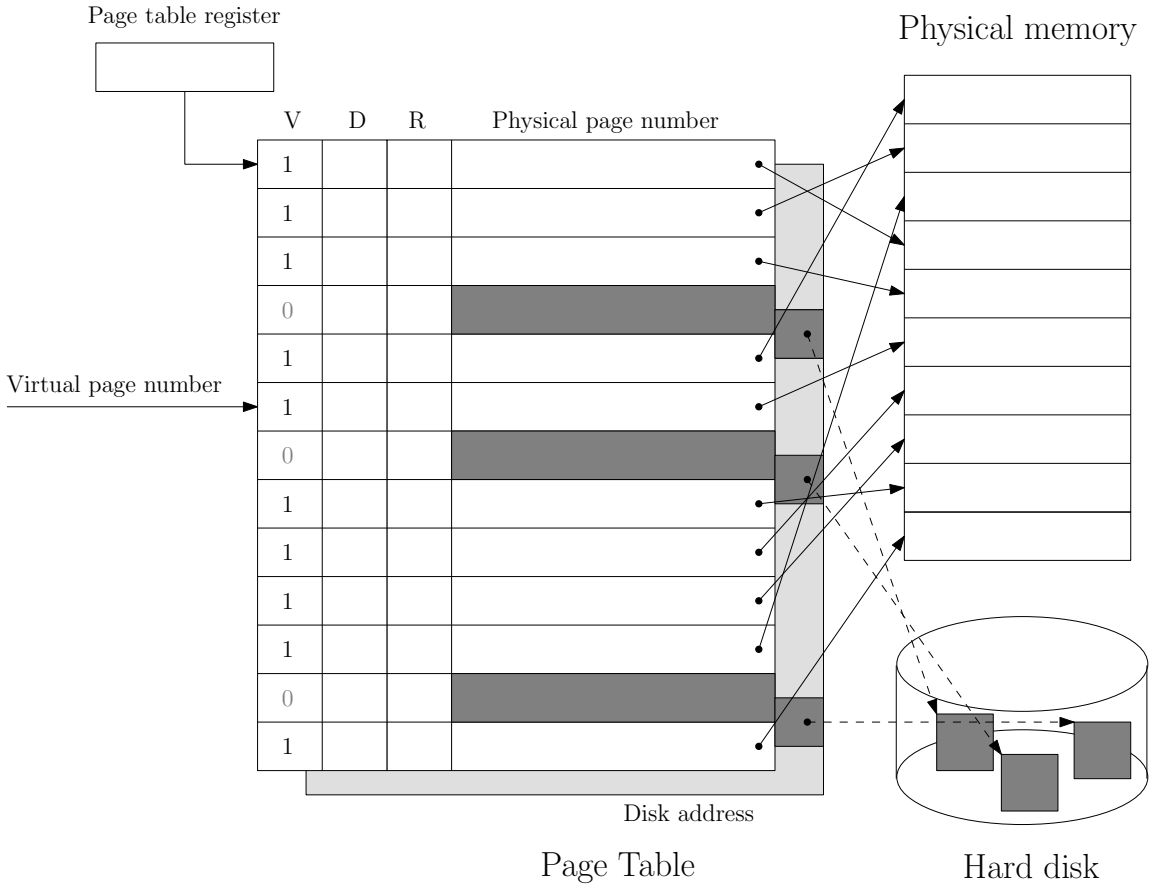

Figure 5.15: The page table.

simple look-up table to translate VPN into PPN: it is indexed by the virtual page number and contains the corresponding physical page number as an entry.

The page table is stored in main memory and, therefore, needs a pointer to it: the **page table register** is used for this purpose and points to the first memory location occupied by the page table. Each program has its own page table (and page table register), mapping its own virtual address space to physical addresses in main memory, shared by all programs. Besides the physical page number, a line in the page table also contains:

- A valid bit V: if V is 1, the page is available in main memory, if it is 0, the page is not available in main memory (and a page fault will occur if it is being accessed in virtual memory).

- A dirty bit D: set to 1 if the page has been written in main memory, so the OS writes the page back to the hard disk before replacing it by another page, in main memory; if D is 0, the page needs not be copied to the disk, when replaced.

- A reference bit R: this bit is used by the OS, to help estimate the least-recently-used (LRU) pages. Implementing a completely accurate LRU scheme is too expensive, so most systems approximate it, e.g., by using a reference bit. By clearing all reference

bits, periodically, and then setting individual bits whenever a page is accessed, the OS can determine which pages have been used recently and which ones haven't. It can then (e.g., randomly) select a page amongst the ones that have not been referenced recently (i.e., the ones with reference bit equal to 0), when looking for a page to replace.

As we just explained, accessing a virtual page for which the valid bit is off, results in a page fault. When a page fault occurs, the OS takes over, finds the page in the next level of the memory hierarchy (usually the hard disk), decides which page in the main memory to replace, then replaces the old page (after a write-back to the hard disk, if the dirty bit is set) with the new page and updates the page table. The virtual address alone doesn't tell the OS where the page is on the hard disk. In order for the OS to know where the virtual page is stored on the hard disk it creates a table that keeps track of the location on the disk of each page in the virtual address space (also depicted in Figure 5.15).

To set up all the previous, the OS goes through several steps when a program (or, process) is being launched.

1. It allocates space on the hard disk, for all pages of that process. This is called the **swap space**, which contains the entire virtual address space of the process.

2. It creates the page table.

3. It creates a data structure that records where each virtual page is stored on the disk (this may be part of the page table, or it may be an auxiliary structure that is indexed in the same way as the page table).

4. It creates a data structure that tracks the usage of the physical pages (how often they're accessed, by which processes, by which virtual page, etc.) This will help the OS to decide which page to replace when a page fault occurs. Usually, an LRU scheme is used and a simple and efficient approximation is offered by introducing a reference bit, as discussed above.

Finally, there is one caveat that needs to be addressed: the page table, as defined above, could be huge. For example, when using 32-bit virtual memory addresses and 4KB pages, there are 2^{32} virtual memory addresses and 2^{12} addresses per page. Thus, there are $2^{32}/2^{12} = 2^{20}$ page table entries. Assuming that every page table entry requires 4 bytes leads to a page table size of 4MB. So, every process would require 4MB of memory for its page table. With lots of processes, the entire physical memory could get filled with page tables only!

To resolve this issue, the size of the page table for a given process is limited, using a limit register, which is adjusted when the process effectively requires more memory. As the process demands more memory, the virtual address space grows, downwards for the stack and upwards for the heap. Therefore, we in fact keep two page tables and two limit registers (one for the stack segment and another one for program and data). If and when the virtual memory address exceeds the limit, the size of the page able is increased and the value of the limit register updated. That way, only processes that demand lots of memory will have large page tables.

5.3.3 Translation lookaside buffer

Since the page table is stored in main memory, each main memory request takes two accesses:

- first, the page table needs to be accessed, in main memory, to look up the physical memory address (since the CPU always issues virtual memory addresses), and,
- second, main memory needs to be accessed to load / store the appropriate data item at the corresponding physical address.

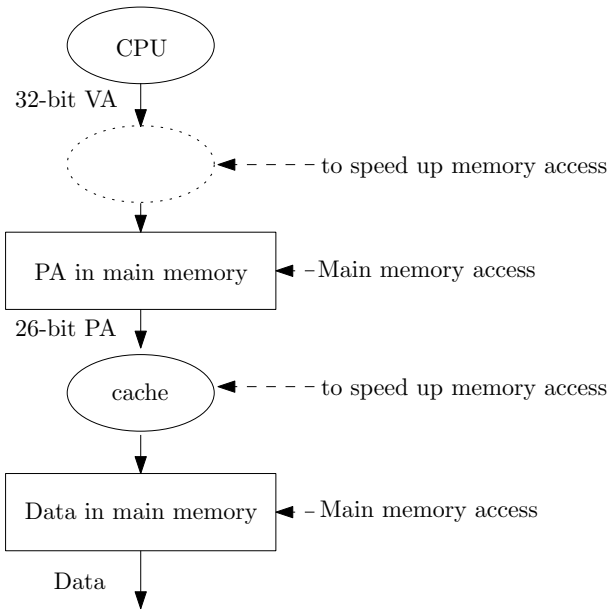

Figure 5.16: Data transfer instructions require two memory accesses: the CPU requests data access using a virtual address (VA), which is translated to the corresponding physical address (PA), by accessing the page table, in main memory. Next, main memory is accessed at that physical address, to load or store the required data item.

From the previous section, we know we can use a cache to speed up the second memory access. To speed up accessing the page table, in main memory, we can use a similar approach: use a cache for address translation, that contains a subset of all virtual to physical page number mappings that are stored in the page table. This cache is called the **translation lookaside buffer** or **TLB** (Figure 5.17). It can be organized as direct-mapped cache, or use any other type of organization. Based on spatial and temporal locality, it is likely that most translations can be derived from the TLB, without requiring access to the page table in main memory.

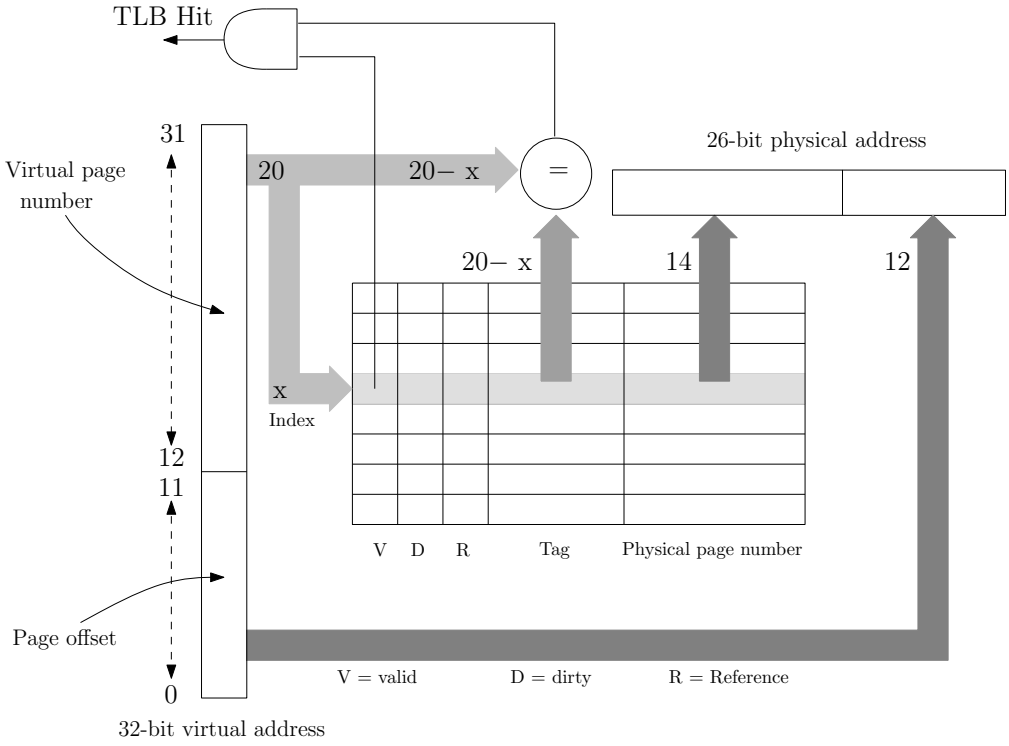

Figure 5.17: Structure of the TLB, for 4KB pages, 32-bit virtual addresses (VA) and 26-bit physical addresses (PA). The 12-bit page offset (because of 4KB pages) can simply be copied from the VA to the PA. To look up the PPN, in the TLB, the x least-significant bits of the VPN are used to index the TLB (where x depends on the size of the TLB) and the remaining VPN bits are used as tag bits, to check for a TLB hit or a TLB miss.

The TLB is read-only cache as far as the CPU is concerned: the CPU will never write in the page table. Nevertheless, the TLB has dirty bits. This is because the TLB also contains a copy of the page table's corresponding dirty bits: that way, no page table access is required to update these bits, when we have a TLB hit on a write instruction. Similarly, reference bits from the page table are also copied and updated in the TLB. Before a TLB entry is replaced, its dirty and reference bits are written back to the page table, to update the page table. The TLB sets its own valid bits, however, because of the process that leads to setting the TLB's valid bits, the TLB's valid bits will be consistent with the page table's valid bits.

5.3.4 Integrating the TLB with virtual memory and the cache

When the CPU provides a virtual memory address, to access memory content, the first component that is accessed is the TLB, to look up the physical page number corresponding to the address' virtual page number.

- In case of a TLB hit, the corresponding physical page number is read out from the TLB and the physical memory address is formed by concatenating it with the page offset. The reference bit in the TLB is set. The dirty bit is set if the instruction issuing the memory address is a write instruction. The physical address is then used to access

the cache, which will result in a cache hit or miss, to access the relevant data. When the physical memory address, obtained from the TLB, is used to access the cache, the boundaries of the relevant fields, to access the cache, do not need to coincide with the separation between page number and page offset. This is shown in Figure 5.20.

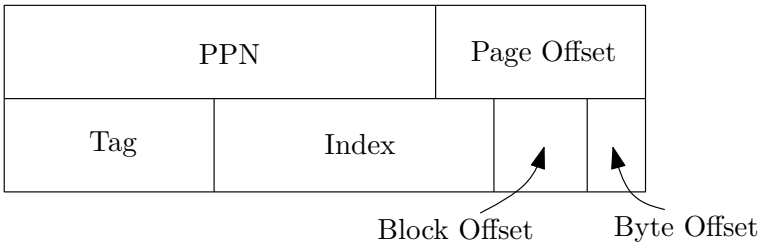

Figure 5.18: Physical address structure.

- In case of a TLB miss, the page table, in main memory, needs to be accessed.
  - If the page table shows the page is valid (i.e., it resides in the main memory), the physical page number, the dirty bit and the reference bit are copied from the page table into the TLB, after writing back (to the page table) the dirty bit and the reference bit of the TLB block that is being replaced (the physical page number does not need to be written back). The TLB then retries the address translation, with success. This provides the physical memory address that can then be used to access the cache.
  - If the page table shows the page is invalid, a page fault occurs. The OS then takes control (the page fault causes an exception) and fetches the missing page in the next level of the memory hierarchy (usually the hard disk). It decides which page in the main memory to replace, and before replacing the old page with the new page, it writes back the old page to the hard disk, if the dirty bit is set. Finally, it updates the page table.

Figure 5.19 shows a flow chart of all the previous steps, while Figure 5.20 gives an overview of the entire memory hierarchy: small and fast memory modules on the left hand side, which interact with slower and larger memory modules on the right hand side, to create an illusion of lots of fast memory, by taking advantage of spatial and temporal locality.

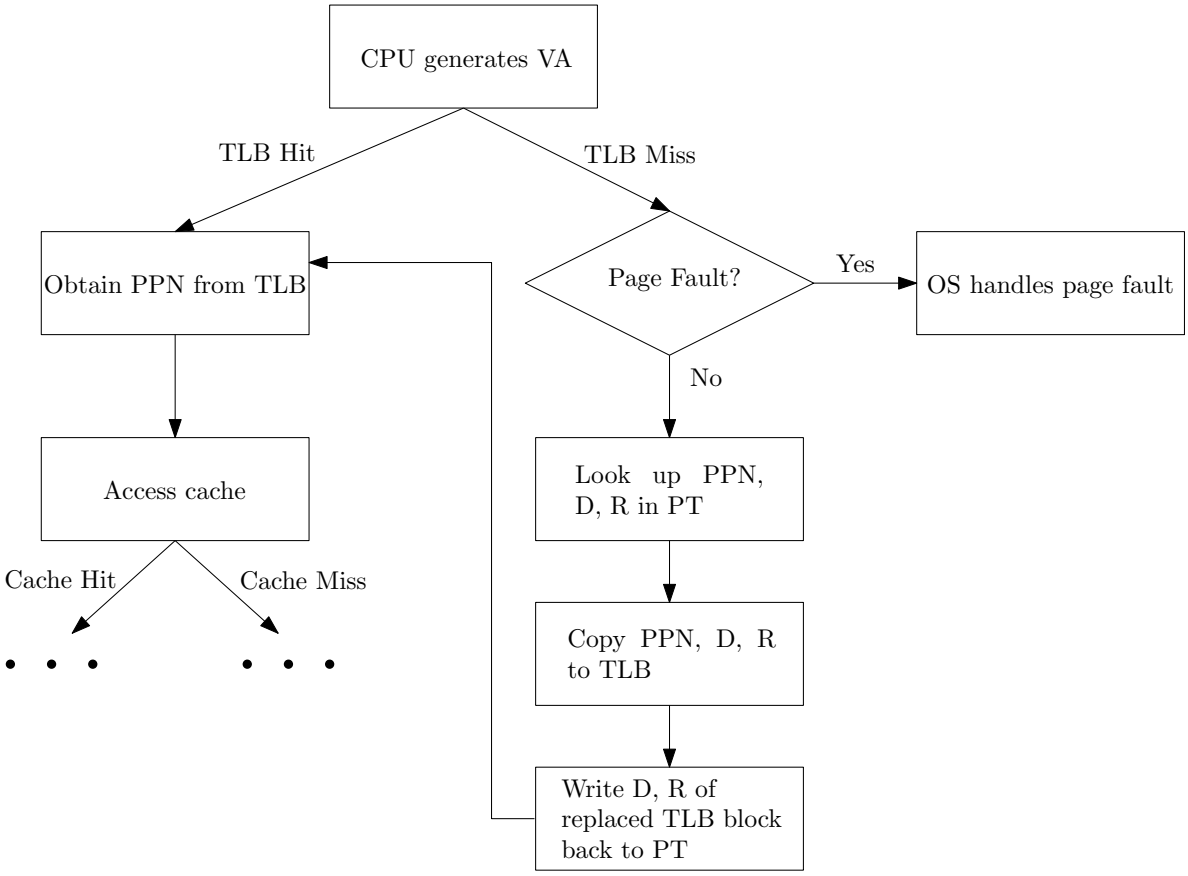

Figure 5.19: Integrating the TLB with virtual memory and cache.

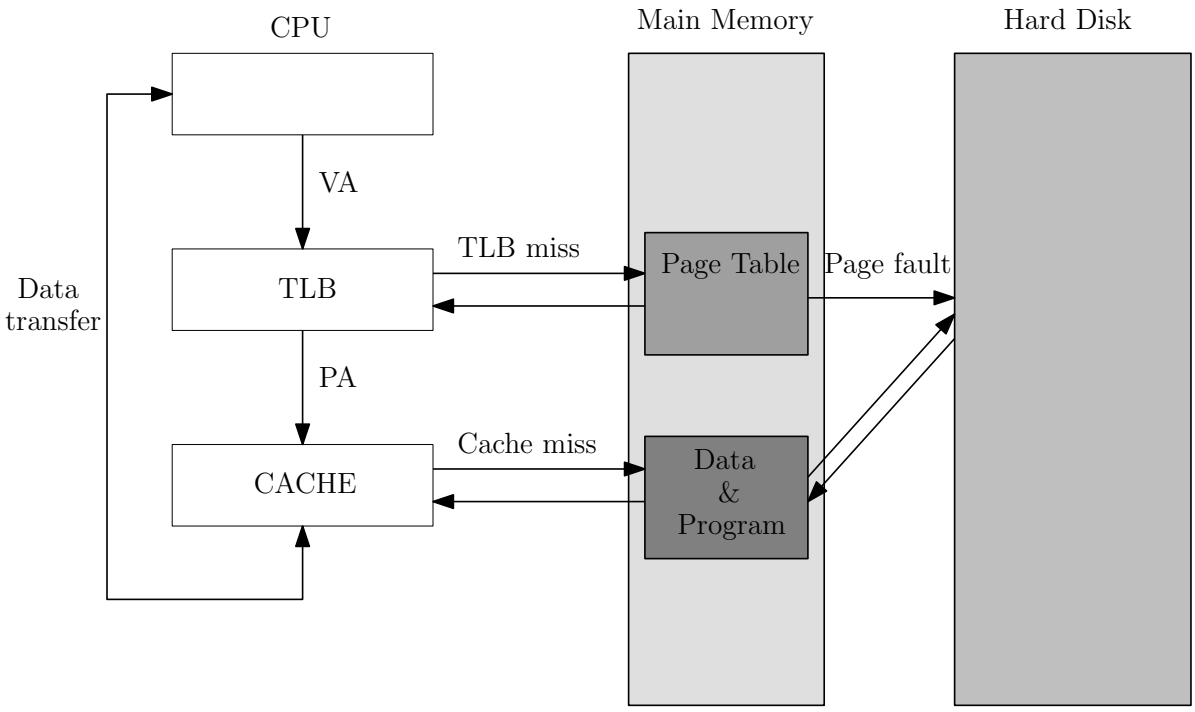

Figure 5.20: Overview of the entire memory hierarchy.