Exercises 5
Memory Hierarchy

Computer Organization and Components / Datorteknik och komponenter (IS1500), 9 hp
Computer Hardware Engineering / Datorteknik, grundkurs (IS1200), 7.5 hp

KTH Royal Institute of Technology
Wednesday 13th July, 2016

Suggested Solutions

Memory Types and Concepts

1. *SRAMs (Static Random Access Memories)* are implemented as integrated circuits, and are often placed on the same chip as the processor.

   DRAM (Dynamic Random Access Memories) are cheaper than SRAMs, but need periodic refresh signals because each memory cell uses a capacitor. Both SRAMs and DRAMs are volatile memories: if the power is turned off, the data is lost.

   Flash memories are non volatile memories that can be reprogrammed many times, but may wear out after extensive usage. This kind of memories are used in USB flash memories and in *Solid State Drivers (SSD)*.

   A *magnetic disk* is the traditional component in a hard drive. It is cheaper than SSD, but slower. Physically, it is a collection of platters that spin at a very high rate.

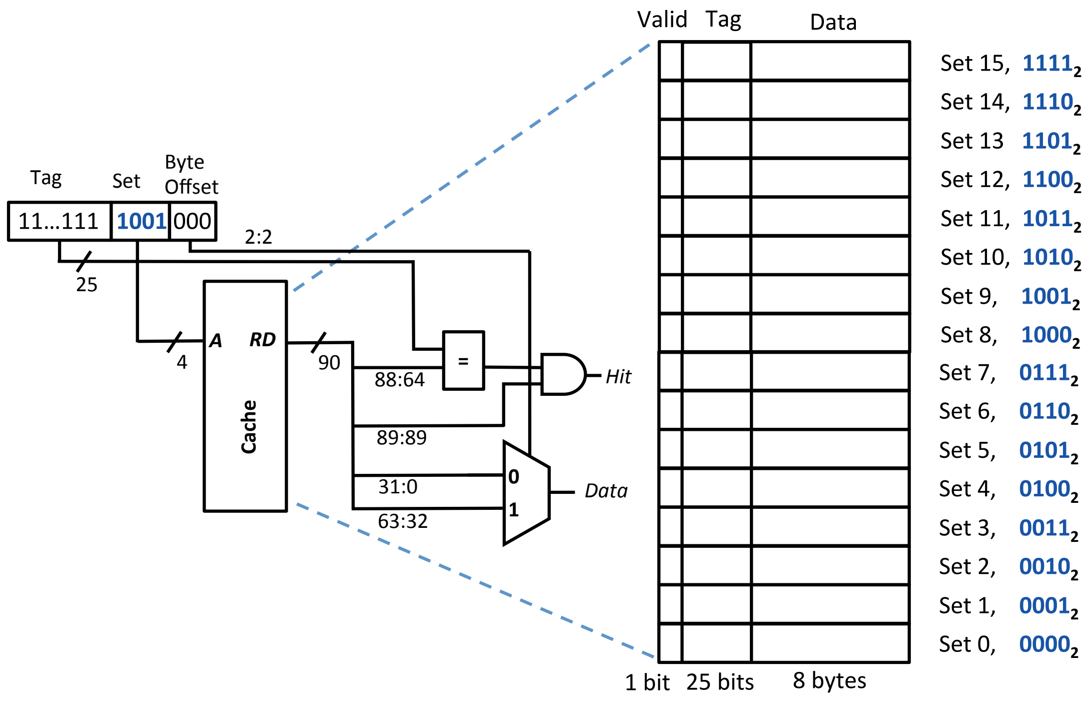
2. *Temporal locality* means that the processor is likely to access recently accessed addresses again. For instance, in a short program loop, the processor will use the same machine instruction in each iteration of the loop.

   Spatial locality means that the processor is likely to access addresses close to each other. For instance, if an instruction is loaded from the memory, the next instruction is located directly after in memory (unless a branch is not taken).

3. The *main memory* of a modern computer is usually physically realized as DRAM memories. These memories are, however, too slow. Therefore, *cache memories* are used to speed up memory accesses. A cache memory can dramatically improve the performance by taking advantage of temporal and spatial locality. If not all the data can fit into the main memory, *virtual memories* can be used to save certain memory regions (called pages) to a slower memory (usually SSD or magnetic disks). Virtual memory is also very important for spatial isolation, that is, to avoid that different processes can access or manipulate the same memory address.

Direct Mapped Cache

4. (a) A conceptual hardware implementation can look as follows:



(b) The address bits can be divided into these three fields.

The *byte offset* determines which byte within the cache line (cache block) that is referenced. In this example, the byte offset field is 3 bits because the block size is $2^3 = 8$ bytes.

The *set* field (also called the *index*) points to the row that should be accessed in the cache. In this case, there are 16 sets (16 rows), which means that the set field size is 4 bits ($2^4 = 16$)

The *tag* field of the address is used for checking if the correct cache line is actually located in the cache. The length of the tag field is the rest of the address. In this case the size of the tag field is $32 - 4 - 3 = 25$ bits.

(c) The step-by-step description is left as an exercise.

i. The cache is never really empty, it will always contain some data when it is powered on. However, if all valid bits are set to 0 when the cache is powered on, the cache is logically "empty" even if there are some data in the cache lines. The meaning of a valid bit is to indicate if the corresponding cache block is valid or not. When a cache block is loaded into the cache, the valid bit is automatically enabled.

ii. The row is selected by directly using the *set* bits of the address.

iii. The word *tag* is used for both describing the *tag field* of the address, and the *tag* that is stored together with each cache block. If a read occurs, the valid bit is 1, and these two tags are equal, then we know that the block for the requested address is in the cache.

iv. The data portion of the cache is where the actual data from the memory is stored. In this example, 8 bytes of data are stored in each block.

v. The program has both temporal and spatial locality. Temporal locality: The loop is executed 10 times, which means that the same instructions will be executed 10 times. Spatial locality: When for instance the first instruction (`addi`) is loaded into the cache, the following instruction in the memory is also loaded into the cache. In this case, the cache block size is 8 bytes, which means that two instructions are loaded into the cache at each memory access.

(d) Becasue each block is 8 bytes and a byte consists of 8 bits, the block size is 64 bits. In each set, there is 1 valid bit, 25 bits for representing the tag (see exercise 4b), and 64 bits for one block. Hence, since there are 16 sets, the total number of bits in this cache are: $16(1 + 25 + 64) = 1440$ bits.

(e) The cache miss rate is computed as the number of cache misses divided by the total number of memory accesses. The program starts on a block aligned address, which means that after executing the first 6 instructions, only 3 cache misses have occurred due to spatial locality; the second instruction load for each cache block becomes a cache hit. Since the whole program fits into the cache, all other memory accesses results in cache hits (due to temporal locality). As a consequence, the miss rate is $\frac{3}{2+4*10} = \frac{3}{42}$.
The hit rate can be computed directly: $1 - \frac{3}{42} = \frac{39}{42}$.

5. (a) The capacity of the cache is 4096 and the block size is 16 bytes. Hence, the number of rows are $4096/16 = 256$. The arrays are declared as `float` elements, which means that each element is 32-bit. When the first element of `v1` is loaded, the cache will at the same time load elements 0-3, because the cache block size is 16 bytes (loads 4 words each time).

The stored data starts at address `0x10008000`, which means that the first element `v1` will be loaded at set 0. Moreover, the size of the 3 arrays together is $3 \cdot 8 \cdot 4 = 96$ bytes, so all arrays can exist in the cache at the same time.

To compute the hit rate, let us start with the first iteration. When the first element in `v1` is loaded via pointer `a`, elements 0-3 of `v1` are also loaded into the cache. Hence, we have so far one memory access and one cache miss, since the cache was initially empty. When we load the first element from `v2`, we follow the same procedure since the cache block is not in the cache. Finally, also when writing to `v3`, we need to first load the block from `v3` into the cache, before we can write to the cache (regardless if it has a *write-through* or *write-back policy*). In summary, for iteration 1, we have made 3 memory accesses, which resulted in 3 cache misses and 0 cache hits.

For the second, third, and fourth iteration, we only get cache hits. However, in iteration 5, we again get cache misses since we need to load new cache blocks. In the last 3 iterations, we again get cache hits. As a consequence, the hit rate can be calculated as follows:

$$\frac{0+3+3+3+0+3+3+3}{3 \cdot 8} = \frac{18}{24} = 0.75 \qquad (1)$$

That is, the hit rate is 75%. The program only uses spatial locality. There is no temporal locality because each individual element is only accessed once. There is spatial locality because four consecutive floating point numbers are read in at once in the cache and then used in the next iteration.

(b) Note that the only difference is that we will now only use vector `v1` as input. This means that we will not get a cache miss when pointer `b` accesses `v1`. Hence, the hit rate can be calculated directly as follows:

$$\frac{1+3+3+3+1+3+3+3}{3\cdot 8} = \frac{20}{24} \approx 0.833 \tag{2}$$

The cache hit rate is approximately 83.3%. There is spatial locality in the same way as argued in (a), but in this case, there is also temporal locality because the elements in `v1` will be accessed twice.

(c) Compared to (a), the difference in this exercise is that we will get cache conflicts. Arrays `v1`, `v2`, and `v3` will all be loaded in the same place in the cache. Recall that the cache block is 16 bytes, which means that the block offset field takes 4 bits of the address. Recall also that there were 256 rows in the cache, so the set field will take 8 bits. This means that the 3 least significant nibbles of the address are used for addressing all the data that can fit into the cache. In this exercise, the fourth nibble of the address states the memory address of the array. Hence, we can directly see that we will get cache conflicts, and an access to one array will evict the cache block for another array.

In summary, the cache hit rate can be calculated as follows:

$$\frac{0+0+0+0+0+0+0+0}{3\cdot 8} = \frac{0}{24} = 0 \tag{3}$$

The cache hit rate is 0%. The program still have spatial locality, but it does not result in better performance due to the cache conflicts.

N-way Set Associative Cache

6. (a) The total number of bytes stored in one way is $8 \cdot 256 = 2048$ bytes. Hence, the associativity of the cache is $8192/2048 = 4$.

(b) For each of the two ways, the following should be included:

- The valid bit should be one 1 bit per row.
- The number of sets are 256, which means that the set field takes 8 bits. The block size is 8 bytes, which means that the byte offset field is 3 bits. Hence, the tag size should be $32 - 8 - 3 = 21$ bits.
- The data column size should be 64 bits because the block size is 8 bytes.

See Lecture 11 for an example layout picture of a two-way set associative cache.

The capacity of the cache is $256 \cdot 8 \cdot 2 = 4096$ bytes.

(c) There is still a potential cache conflict because the 3 arrays are still located in the same sets in the cache.

In the first case, using the cache in 6(a), we have a 4-way associative cache. This means that all three arrays can exist in the cache at the same time, but in different ways. Hence, the cache hit is computed as follows:

$$\frac{0+3+0+3+0+3+0+3}{3\cdot 8} = \frac{12}{24} = 0.5 \tag{4}$$

That is, the cache hit rate is 50%.

However, in case of 6(b), because N = 2 and we use LRU, the oldest cache block will always be replaced. Two of the arrays can exist in the cache at the same time, but the oldest one will always be replaced before we can get a cache hit. Hence, the conflicts still exist, and the cache hit rate is 0%.

Other Concepts

7. These two policies exemplify two different ways of how data is written back to the main memory from the cache, when a write operation occurs. In the *write-through policy*, the data is written back to the main memory simultaneously as the cache is updated. In the *write-back policy*, the data is not written back to the main memory directly on a write operation. Instead, the write is delayed until the cache block is evicted.

8. The concept of multi-level caches means that there is a hierarchy of caches. For instance, there is one fast L1 cache close to the processor, and then a slower L2 cache between the L1 cache and the main memory. The reason for having several caches is that smaller caches are usually faster, whereas larger caches may result in lower miss rate.

9. (a) The page offset is 12 bits because $2^{12} = 4096$.

   (b) There are $18 - 12 = 6$ bits to represent the different physical pages. Hence, the are $2^6 = 64$ physical pages that can be stored in the physical memory.

   (c) There are $32 - 12 = 20$ bits to represent the different virtual pages. Hence, the are $2^{20} = 1048576$ virtual pages.

   (d) The mapping between virtual pages and physical pages is stored in the *page table*. The page table is stored in the physical memory and is updated by the *operating system (OS)*.

   (e) The translation between virtual page numbers and physical page numbers needs to be fast. This is done in hardware by the *Memory Management Unit (MMU)*.