**T**he **A**merican **U**niversity in **C**airo

The School of Sciences and Engineering

CSCE 231/2303 – Assembly Language Programming

Dr. Mohamed Shalan

Project 2: Memory Hierarchy Simulation

Due Date: 20/07/2017

Submitted by:

Ahmed Ghazy ….. 900153471

Ahmed Ibrahim … 900153478

Amr Gouhar ……. 900153482

George Youssif … 900150531

**Inrtoduction**

The cache is a very high speed,expensive piece of memory, which is used to speed up the memory retrival process due to it’s higher cost. And when the CPU needs to access a word in the main memory the CPU checks the cache, if the word is there the CPU copies the desired word, if not, the cache access the main memory and copies a block of data including the desired data.

Cache memory types are direct mapped, fully associative and set associative. The difference between direct mapped and fully associative cache is that the direct mapped cache has an index which make it easier and less expensive for the hardware to compare; however in fully associative cache it doesn’t have index and has to compare the cache tags for all the enteries. Figure(1) below shows the memory hierarchy pyramid which is mainly organized depending on the response time as the top represents faster, smaller and expensive storage devices while the bottom represents slower ,bigger and cheaper storage devices.

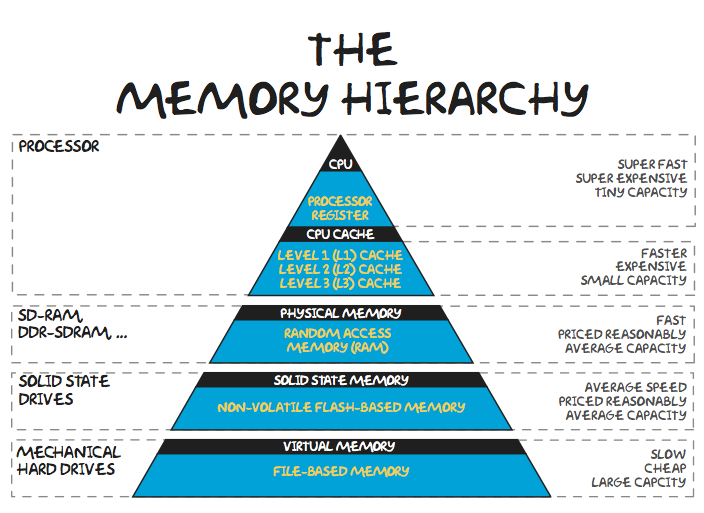


Figure 1: Memory Hierarchy

**Implementation:**

The memory hierarchy was implemented in C++ and integrated with a previously-developped RISC-V 32I disassembler and simulator. A struct was used to represent a line or a block in the cache and for each level of cache; i.e level 1 instructions, level 1 data and level 2 data and instructions, an array of this struct was used to simulate them. The cache simulator has four main functions, two functions to read from or write to level 1 cache and another two were used for the same purpose for the level 2 cache. The definition of the struct and the prototypes of the functions are shown below with explanation for each one.

**The Struct Definition:**

struct line

{

bool valid, dirty;

int tag;

void initialize\_line()

{

valid = false;

dirty = false;

}

};

In our implementation, we did not care about the data because it’s already in the memory, we only cared about the validity bit, the dirty bit and the tag. This is because our goal was to keep track of misses which has to do with the validity bit and the tag, and the write-backs which has to do with the dirty bit. So, the struct that simulates a line in the cache had only the information needed to keep track of the counts.

**The Prototypes of the functions:**

1. **char read\_L1(bool data, int address);**

This function gets an address and a boolean variable to know whether the CPU wants to read from level 1 data or level 1 instructions. Based on the address and the level 1 cache type, which is specified by the user, the tag is calculated, and for the direct-mapped cache, the index is also calculated. For the direct-mapped cache, it checks the validity bit first of the index, if it’s invalid, there is a complusory miss. If valid, it checks the tag in that index to see whether it matches the calculated tag or not, and if they don’t match, then there is a conflict miss. If the tags match, then this is a hit. For the fully associative, there is a loop that stops at one of three cases: 1- the loop counter reaches the cache size, then there is a capacity miss. 2- it finds an invalid line, then there is a compulsory miss. 3- it finds a valid line with a tag equal to the calculated tag, then it is a hit. At the end, in both cases, if the data is not found, then the miss count is incremented by 1, and if it’s a capacity miss, the replacement policy is used, random replacement. And if there is a conflict miss, we check the dirty bit first to determine whether we are supposed to increment the write-back counter and write the dirty data to level 2 or not. Finally, we read from level 2 in all cases of misses and set the tag to the calculated tag, set the valid bit to true and set the dirty bit to false.

1. **void write\_L1(int address);**

In this function, we assumed that we will write data only, so we only need the address to determiner where the data is to be written. We calculate the tag from the address, and the index if it’s a direct-mapped cache. If the line is valid, and the tags don’t match, we check on the dirty bit, if the line is dirty, we write to level 2 cache. If the tags don’t match, we increment the miss count and read that line from level 2. If the line is invalid, we increment the miss count and read level 2. Finally, we set the tag to the calculated tag, the valid bit to true and the dirty bit to true.

1. **void read\_L2(int address);**

Here, we get an address, calculate the tag and the index, since it is direct-mapped. Then, if the line is valid and the tags match, it’s a hit. Else, we check the dirty bit and increment the write-backs if it’s true in a valid line. If there’s a miss, we increment the miss count and set the tag to the calculated tag, the validity bit to true and the dirty bit to false.

1. **void write\_L2(int address);**

This function calculates the tag and the index from the address. If the line is valid, the tags don’t match and the line is dirty, we increment the write-back count and only if the tags don’t match, we increment the miss count. If the line is invalid, we increment the miss count and set the tag to the calculated tag, the validity bit and the dirty bit to true.

**Experiments and Results:**

In order to understand the cache performance and how to achieve the best performance based on the given cost function, we conducted four experiments for each benchmark. In each experiment, we changed the block size of level 1 and level 2 (16 bytes and 32 bytes, as required) caches along with their types (fully-associative and direct-mapped). The reason for this is that in order to measure the performance of the cache, we need to keep track of the miss count and the write-back count for each configuration and measure the cost according to the given function. So, in this section, the results of these experiments are represented using graphs (which were generated using MATLAB, for the code, check Appendix B). These results are discussed and interpreted in the analysis section of the report.

**Benchmark 1:**

The following graph (Figure 2) shows the results obtained for the first benchmark. It shows that the best configuration for it is when the caches are fully-associative and the block size is maximum, in our case 32 bytes, as the cost is minimum, in this case, it’s 316.5 . The cost is maximum, however when the cache is direct-mapped and the block size is minimum, 16 bytes, as the cost is high (around 1360). The reason for this will be shown later, after presenting all the results.

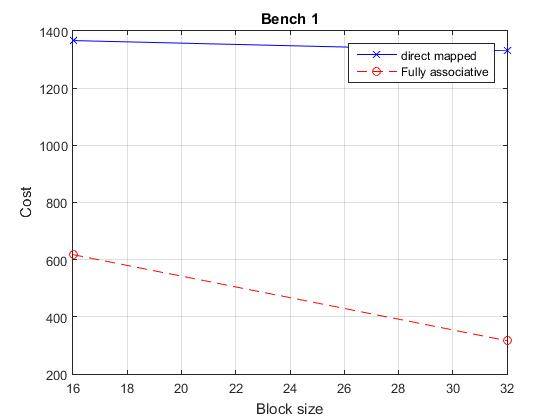


Figure 2: Benchmark 1

**Benchmark 2:**

Similar to the results obtained for the first benchmark, the second one also (Figure 2) has a maximum cost (1279.5) when the cache type is direct-mapped and the block size is minimum. It has a minimum cost when the cache is fully associative and the block size is maximum (320.5).

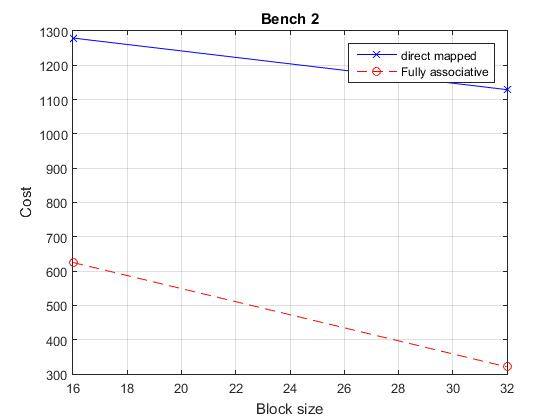


Figure 3: Benchmark 2

**Benchmarks 3 and 5:**

Again, the results are not different from the previous two benchmarks. A minimum cost is obtained for a fully-associative cache with the largest block size and a maximum cost is obtained for a direct-mapped cache with the smallest block size.

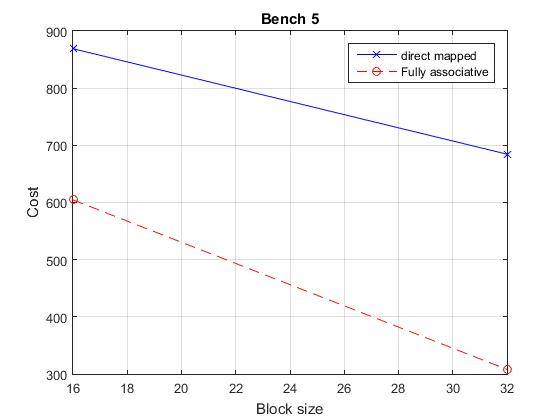
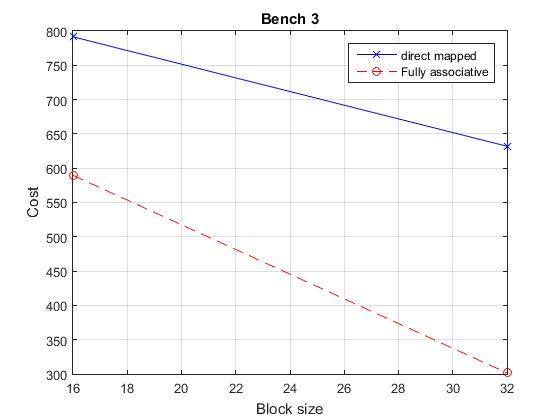


Figure 4: Benchmark 3 Figure 5: Benchmark 5

**Benchmarks 4, 6 and 7:**

For these four benchmarks, the results were more or less the same as before. The cost is maximum whenever the cache is direct-mapped wih the smallest block size (16 bytes) whereas it is minimum if the cache is fully associative with the largest block size (32 bytes).

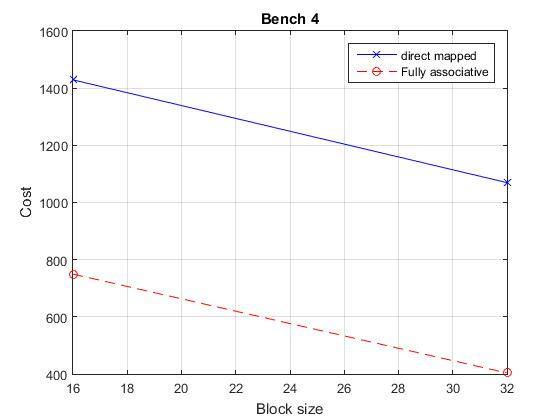


Figure 5: Benchmark 4

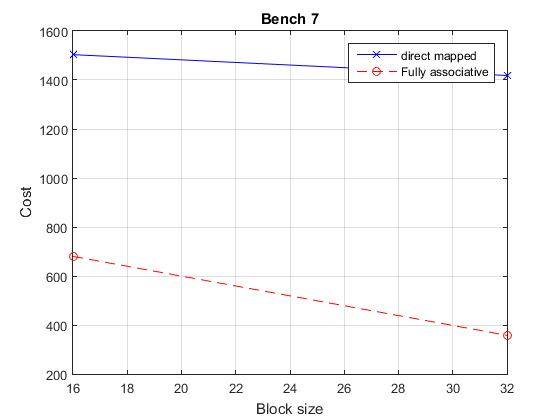
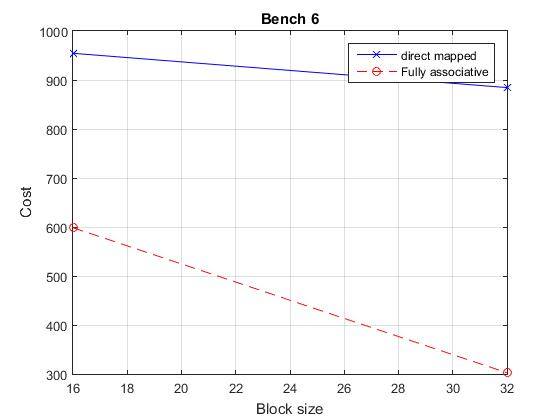


Figure 7: Benchmark 6 Figure 8: Benchmark 7

**Analysis and Conclusion:**

From the results section, the cost is clearly inversly proportional to the block size of the cache regardless of its type as the maximum cost for each type is obtained whenever the block size is minumum in most of the cases. For the given benchmarks, this seems logical. This is because most of the instructions in the benchmarks used in the experiments are load (word, half-word, byte) from memory and strore (word, half-word, byte) in memory. And since the memory is byte-addressbale, the larger the block size, the more bytes will be brought to a single block. This is due to spatial locality since it seems that the benchmarks access the memory sequentially, so the larger the block size, the better our use of spatial locality is.

For benchmarks 1, 6 and 7, the cost for the direct mapped is almost constant because these benchmarks access locations in the memory that are more likely to conflict. So, they didn’t utilize spatial locality and; therefore, increasing the block size won’t affect. For the fully associative, the cost decreased by a factor of 2 because even though spatial locality is not utilized, conflicts don’t occur, so we make use of temporal locality, so whatever was used will remain in the cache most probably.

For benchmarks 3 and 5, the block size was effective in the direct-mapped cache. This is because these benchmarks access the memory sequentially, so the block size will help make the best use of spatial locality and therefore, the cost decreases notably.

As for the reason why the fully associative cache have generally a cost less than the direct-mapped, if we calculate the number of lines in the cache by dividing the cache size over the line size, we will get a large number. This means that it is less likely for the fully associative cache to have capacity misses. Furthermore, in case of direct-mapped cache, if we have a large number of lines, then this means we have a larger number of bits allocated for the index and less number of bits allocated for the tag, so, whenever two addresses have the same index, the probability of having different tags is low (since the number of bits for the tag is small) which will cause a conflict miss. So, our benchmarks, along with the sequential access part, they have a part in which they somehow access the memory randomly, so there is a probabilty that many addresses get mapped to the same index causing conflict misses. In the fully associative, since we have a large number of lines, whenever we have new data, this data will be inserted in a new line and since we use a random replacement policy whenever the cache is full (a very low probability), which is proven to be very efficient, there will be a less number of misses. This is again because programs exhibit locality in the sense that it will most probably use the data in the cache. To summarize, the fully associative cache is better than the direct-mapped cache because the probability for the fully associative cache to be completely filled is low and in case it is filled, the random replacement policy is shown to be efficient since it replaces an existing line with a new line randomly which means that this existing line will somehow not be used by the program in the future. The direct-mapped cache, however, suffers from conflict misses since there is a high probability that two addresses get mapped to the same location in the cache.

To complement this analysis, consider the following numerical example that shows yet another reason why the fully associative cache performs better by utilizing temporal locality more. A 64 KB byte-addressible main memory consists of 65,536 possible addresses. A level-one 16-byte-line-sized 1 KB consists of 64 lines, each can store the data from any memory address in case of the fully associative cache. However, in the case of direct mapped cache, each line is responsible for 4096/64 = 64 different memory locations which produces a higher number of conflicts if it happens to be the case that these 64 different memory locations per line need to be accessed frequently (temporal locality). In other words, the fully associative cache utilizes temporal locality since recently used locations will most likely be always there (unless replaced randomly), whereas the direct mapped cache can significantly hinder the performance of programs that need to access recent addresses that happen to be conflicting all the time.

In conclusion, it is now clear why the fully associative cache with large block size is better than a direct-mapped cache with the same block size. Also, it is shown that a fully associative cache is generally better than the direct-mapped cache.

**Appendix A: Command Line Interface**

The compiled binaries (using, for example, the standard g++ rv32iSim.cpp -std=c++11) provide a command line interface allowing a number of options and cache configurations according to the following format:

rv32iSimExecutable /path/to/benchmark.bin [-#] [-assoc] [-d] [-no\_r\_dump] [-m\_dump]

[-#]: With # replaced with a suitable number, this option represents the L1 and L1 block size. (default: 16)

[-assoc]: Forces the L1 caches to be fully associative. (default: direct mapped)

[-d]: Turns disassembly mode on: prints the disassembled code before simulating its behavior. (default: off)

[-no\_r\_dump]: Disables printing the register dump at the end of the simulation (default: on)

[-m\_dump]: Displays a dump of all memory elements at the end of the simulation. (default: off)

**Appendix B: Test Cases Generation**

A simple bash script was written to run the program with all relevant combinations of command line arguments to simulate the behavior of 16/32-byte-line-sized fully associative/direct mapped caches. It iterates over files in the same directory whose name is “benchX.bin” where X is a number and then extracts the cost from the output (stdout) produced by the simulator and then lists them.

for filename in ./bench\*.bin; do

echo $filename :

echo 'a' | ./a.out $filename | tail -n 1

echo 'a' | ./a.out $filename -32 | tail -n 1

echo 'a' | ./a.out $filename –assoc | tail -n 1

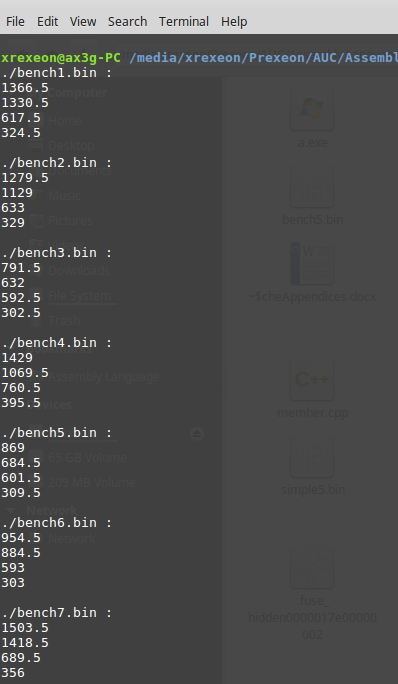
echo 'a' | ./a.out $filename -assoc -32 | tail -n 1

echo

done

benchmark2.sh:

Then the output containing the results would look, for example, as follows:



**Appendix C: Graph Generation**

The data obtained was then graphed automatically using a simple MATLAB script.

graph.m (a snippet):

x = [16 32];

Fully\_associative = [569 287];

direct\_mapped = [1421 952.5];

plot(x,direct\_mapped,'blue-x',x,Fully\_associative, 'red--+');

grid

title('Bench 5');

xlabel('Block size');

ylabel('Cost');

legend('direct mapped','Fully associative');

The output produced can be checked at the results section.