# CS201
# REPORT: Cache lab

1551020 - Vo Tran Thanh Luong

December 6, 2016

## Contents

# 1 Part A

Part A

Our mission is to output the number of miss, hit and eviction. First we need to build up four struct to store everything related to a cache.

---

## Cache parameters

| Parameter | Description |
|---|---|
| S = $2^s$ | Number of sets |
| E | Number of lines per set |
| B = $2^b$ | Block size (bytes) |
| m = $log_2$(M) | Number of physical (main memory) address bits |
| s = $log_2$(S) | Number of set index bits |
| b = $log_2$(B) | Number of block offset bits |
| t = m – s - b | Number of tag bits |
| C = B x E x S | Cache size |

6

Figure 1:

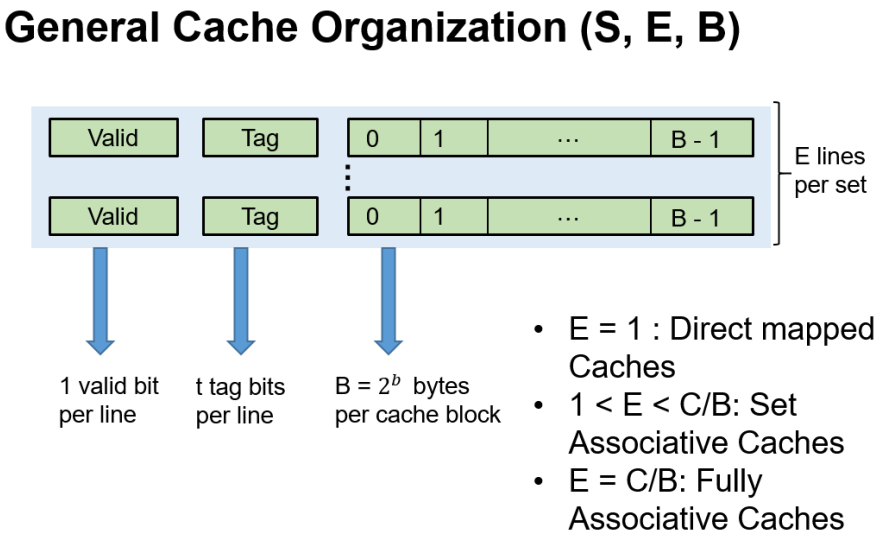


Figure 2:

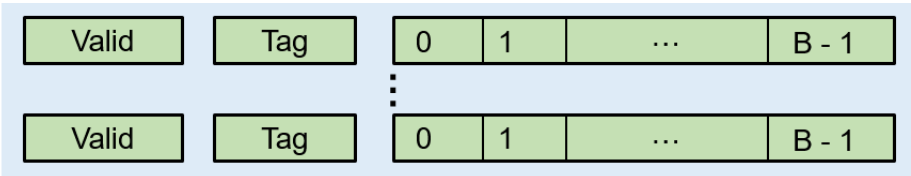


Figure 3:

The first struct is cacheParameter, in which we will store all the basic information about a cache : Number of sets, Number of lines per set, Block size, Number of set index bits, Number of block offset bits. Plus, data about times that a cached is hit, missed or evicted is also stored here.

The second struct is setLine. This represents all data in a set line in a cache. As we all knew from the slides, a set line includes a valid bit, a tag bit, and the block. For the tag bit, we declare it with the custom-defined type memoryAddress. Because this is 64 bit, it would be very annoying to keep repeating the data type declaration Also, because we will surely deal with eviction, it is necessary to maintain a variable called last_used to properly evict the right block later.

The third struct cacheSet actually consists of a pointer to the second struct, because there are many lines of sets in a cache.

The fourth struct cache actually consists of a pointer to the third struct, obviously because there are many sets in a cache.

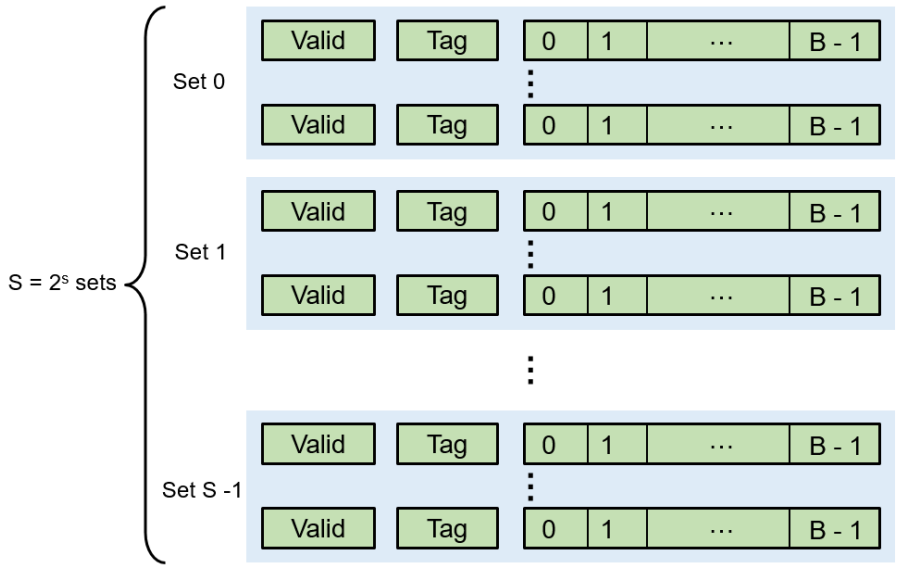


Figure 4:

- Your simulator must work correctly for arbitrary $s$, $E$, and $b$. This means that you will need to allocate storage for your simulator's data structures using the `malloc` function. Type "man malloc" for information about this function.

Figure 5:

```
I 0400d7d4,8
 M 0421c7f0,4
 L 04f6b868,8
 S 7ff0005c8,8
```

Each line denotes one or two memory accesses. The format of each line is

`[space]operation address,size`

Figure 6:

After we defined all the struct needed, we will begin by building a cache. For this, we will build a function that takes in 3 parameters : number of sets, number of lines and size of the block. With these parameters, as we just loop through every line in every set and put the default value 0 inside every slot. Then we return the newly built cache.

Because C doesnt automatically collect the garbage for us, we also need to build a clear function due to the use of pointers in cache and cacheSet structs. We need to loop again, through all lines of all sets of a cache and free everything.

When we put data into the cache, first thing we need to check is whether the line that is matched is empty or not. To serve our purpose, a function to detect the empty line needs to be constructed. This function's parameters are required to supply all the general cache parameter and one exact cache set it is checking for empty line. It shall check on only one cache set to see whether the lines inside are available or not. If all lines are unavailable, we will return -1, if there is some lines available, we will return the index of that line.

No line available will put us in a situation of eviction. However, choosing a line for eviction is not easy so we need another function to handle that. This function detect the right line to force eviction. Basically, we need to passed in the same parameters as the detect empty line function. However, that is not enough because if the next time we met eviction case, we would forget how the usage frequency of the line was. That is dangerous because we cannot decide which one is the latest-used one. Therefore, we need an array that stored 2 values : one is the line that is latest-used line and another is the oldest-used (least-recently-used) line. And notice that the pdf file requires us to "This means that you will need to allocate storage for your simulator's data structures using the malloc function." Thanks to stackoverflow i know how to make a dynamic array in C. (http://stackoverflow.com/questions/12675919/dynamic-array-in-c-is-my-understanding-of-malloc-realloc-correct)If the cache is full, overwrite should be done to the least-recently-used line. This function should return the index of the least-recently-used line.

One of the most important part of our program is the function that actually run to check if the cache is hit/missed/evicted. Here I call that function testCacheFunction and pass in the cache, the cache parameter as well as the address for it to finish its job. In this function, first we need to find the tag size ( using equation tagsize = m-s-b). After that, we need to extract the input tag bit from the address. Moreover, we need to store it in the unsigned long long because the shifting may cause negative value to appear. Plus, we also need to extract the index of the set so that we can find the exact place to put the bits.

Now, simply loop through the line in the sets and check the valid tag. If the valid tag is different from 0 and the input tag matches that line tag, then it is safe for us to raise the hit because we did "cache hit". Also, we need to raise the latest used variable and update the exampleSet.lines because we are working mainly on exampleSet. If we found an empty line then simply reset the checkFullCache to 0 to know that the cache was not full.

When the loop is done, we knew about the "cache hit". But how about "cache miss". It is very simple. We just confirm if the default hit value that we initiate with the struct cacheParameter is equal to the hit value that we got after the loop. Equality means no "cache hit" was done. No "cache hit" means "cache miss" and we need to write the data into the cache ourself. Here comes the momment that we check if the cache is full or not. If it is full then eviction should be done and for eviction to be applied properly, we need to used detectEvictLine that we wrote above. When the cache is full, we overwrite to the line that is least-recently touched. If the cache is not full, we will write our

data to another empty line and remember to raise the valid bit as well as the latestUsed variable. Keeping track of all these things is very important because our traces got a lot of cache accesses to be checked, not one.

- We recommend that you use the `getopt` function to parse your command line arguments. You'll need the following header files:

```
#include <getopt.h>
#include <stdlib.h>
#include <unistd.h>
```

  See "`man 3 getopt`" for details.

Figure 7:

In our main program, we should figure out a way to take in parameters from command line as well as take in cache accessed from the trace files. Reading the pdf file, they suggest we use getopt.h . After google searching, I apply the same syntax found here (https://www.gnu.org/software/libc/manual/html_node/Example-of-Getopt.html) and here ( https://www.gnu.org/software/libc/manual/html_node/Using-Getopt.html ) and convert string to integet (http://www.cplusplus.com/reference/cstdlib/atoi/) was able to take out s E b t g from command line. For reading file I follow instruction and example from here (https://www.tutorialspoint.com/cprogramming/c_file_io.htm) . Of course, we also have to take in command in tracefile because each case needs to be handled differently, especially M case needs double time.

Some additional function was used during the process of finishing this like bzero (http://pubs.opengroup.org/onlinepubs/009695399/functions/bzero.html) to initiate the struct example Parameter ( it's like a constructor for a class in C++ but I cannot find a way to play construtor in C ).

## 2 Part B

WIP