Name: Devansh Gupta
Roll No.: 2019160
Course: Computer Organization

This is a Cache Simulator designed for the end semester assignment for CSE112 Computer Organization.
Follows a compiler based approach which means that there is a text file with a certain number of instructions written in a particular format and the cache is filled according to those instructions and at the end, a report of these instructions is generated in the output file to show cache hit or miss and the final status of the cache.

Description and assumptions taken:-
1> Max of 32 bit addressing but is bounded to the entered main memory size.
2> All representations of addresses are in hexadecimal at the output.
3> Can simulate a fully associative cache, a direct mapped cache, and a n way set associative cache where the user can enter the n.
4> It is a write through cache.
5> Designed only for the memory management of 32 bit/4 byte integers only.
6> Cache parameters should be powers of 2.
7> An integer is assumed to take all four bytes in the memory in a way that actually in the four positions of the array corresponding to that address, the same integer is stored to avoid discrepancy(Last two bits are ignored to calculate the address of the integer in memory).
8>The cache is assumed to be completely empty at beginning i.e. every new addresses entered for the first number of cache lines or n(as in n way associative cache) addresses will go to a miss depending on the type of cache used.
9>All cache parameters will be the powers of 2 and cache size and main memory size would not be less than four bytes i.e. 2^2 bytes.
10>Assuming that the user will input data that is less than MAX_INT32 else the c++ compiler will give the overflow error instead of this one.
11>Assumed a main memory for a deeper understanding but did not display it.

Navigating through the folder:-

- .vscode : Contains the settings.json file for cpp to facilitate the import of header files in the vscode environment.
- Inputs: Contains the input.txt file which is the file used to take the input series of instructions
- Outputs: Contains the files given out by the program after the execution of the code with the input.txt file as input
- Tests: Contains some tested inputs and outputs for each type of cache which ran on this program to test it and corrected till a valid output came out.
- DevanshGupta_2019160_FinalAssignment.cpp: Contains the main program which is to be executed in order to obtain the outputs in the Outputs folder

- Documentation_FinalAssignment.pdf: A guide on how to write the inputs and what to expect as outputs and what precautions to be taken while writing the read/write instructions

Function of the program:-
It takes in the inputs through the Inputs folder in this directory from the inputs.txt file. It is a sequence of read and write instructions which simulate the read and the write operation in the cache and the main memory. The instructions of writing the script will be given later. After the script has been written in a certain format, the user is simply to compile and run the program cacheSim.cpp which creates two files(if the files are not already present) in the Outputs folder which are the following:-
1> report.txt: It contains the hit/miss report of each instruction with respect to the cache and its expected output in a tabular format.
2> cache.txt: It prints out the tag array and the present data array(including each of its elements in the block stored in the data array corresponding to the given tag). These arrays are printed separately

Instructions of use and to make the input.txt script:-
1> The first line of the input contains the parameters on which we are to build a cache, and those
parameters are:-

- Type of Cache: Each of the number denotes a type of cache:-
    1: Fully Associative Cache
    2: Direct Mapped Cache
    3: N Way Set Associative Cache
- Main memory size: This is to specify the size of the main memory in bytes to be simulated so that a cache can be simulated on it and also sets a limit on the address used. It must be a power of 2.
- Cache Size: This is to specify the size of the cache in bytes to be simulated.It must be a power of 2.
- Block Size: This is the size of a block which is the smallest unit of exchange between cache and main memory.It must be a power of 2.
- No. of Indices in a set(n): Only valid for n way associative cache.

2>All the cache parameters must be entered in one line. Eg. 1 65536 64 8 (It is basically an associative cache), 2 65536 64 8 (It is basically a direct mapping cache), 3 65536 64 8 4 (It is basically an n way set associative cache with the one extra parameter which is n). The parameters are to be entered in the same order as mentioned above. Please try not to enter any number other than 1, 2 or 3 in the first argument and try not to exceed the main memory limit.

3>From the second line onwards there are instructions of two formats:-

- read "address": This is the read instruction which reads the data from a particular address in memory but first it checks through the cache and then takes out data from

main memory and insteadof "address" a valid address must be entered in the hexadecimal form starting with "0x".

- write "address" "data": This is the write instruction which writes the data to a particular address in memory and since it is write through it writes in the main memory and the cache and if the block is not present in the cache then the block is inserted into the cache and instead of "address" a valid address must be entered in the hexadecimal form starting with "0x" and instead of data, an integer (max 32 bits) is to be entered to write that data in the memory.

4>At the very end of the script, an instruction "abort" is to be written which denotes the end of the sequence of instructions and ends the script. This instruction is especially important to execute at the end or else the program keeps waiting for input. Once abort it is written, instructions below it will be ignored.

5>Try to enter all the addresses in hexadecimal format starting with "0x" and in proper format else the cache will store or get values from garbage addresses.

6>The addresses entered must be less than the main memory size when being entered or else an error
ERROR at Line x:Invalid Address>>Aborting::Note:Further instructions will not be executed will be shown and the program will stop and give no report.
Here x denotes the line number at which the error is found.

7>The cache parameters entered must powers of 2, the main memory size must be greater than or equal
to the cache size which must be greater than or equal to the block size or else an error
ERROR at Line 1:Invalid Parameters>>Aborting::Note:Further instructions will not be executed
will be shown and the program will stop and give no report.

8>The instructions entered must be of proper format as mentioned above when being entered or else an error
ERROR at Line x:Invalid Instruction>>Aborting::Note:Further instructions will not be executed will be shown and the program will stop and give no report.
Here x denotes the line number at which the error is found.

9>Make sure that you make the input.txt script ready and in correct format before the program is compiled or run, else it will give no significant result i.e. first make the file and then run the program.

An example script is:-
1 65536 64 8
write 0xFFFF 123

read 0xFFFF
write 0x1234 22
write 0x4444 23
read 0x1234
read 0xFFFF
read 0x4444
abort

More elaborate test cases and their outputs have been given in the Tests directory for each type of cache. The Input folder has the input text file and the output folders has the results.

Ways in which this program can be used or experimented with:-
1>Can be used to determine the final state of the cache after all the instructions have been executed:-
1>In that case just write the instructions in the file and compile and run the cacheSim.cpp program after that.
2>Can also be used to see the state of the cache after every instruction in the following way:-

- Have one window which has input.txt open in the Inputs directory
- Have another window/terminal open to create a running environment for the cacheSim.cpp program
- Have a third window named cache.txt in the Outputs directory
- Write the cache parameters at the top of the file and abort at the bottom
- Now simply keep writing one instruction and then compile and run the cpp program after you have written that instruction and view the cache.txt after the program is run. It will show the resultant change in the status of the cache after the instruction was added so that analysis can be done.

Implementation of cache:-
1>The cache was implemented in the exact way that was described in the prescribed video by Dr. Smruti Ranjan Sarangi of Chapter 10:Memory System and Design.

2>First when the address was provided, it was parsed according to the cache parameters given in order to separate tag, index, and offset from the 32 bit address and since these parameters were powers of 2, and positions of parsing were known, therefore it was a simple problem of string segmentation.
There was a different parser for each type of cache namely:-

- vector<string> parse_addresss_associative(string address): Returns 2 parses
- vector<string> parse_address_direct(string address): Returns 3 parses
- vector<string> parse_address_n_way(string address): Returns 3 parses

3>These parsers were useful as they directly gave the index needed to access the main memory which was a 2 dimensional string indexed unordered map of the framework
main_memory[block_address][offset]

4>The different caches had the same underlying principle where there was a data array and a tag array where:-

tag array: vector<pair<string,int>> where int stored its priority when there was time of cache replacement(for which I used pseudo Least Recently Used replacement scheme)

data array: vector<unordered_map[offset]> where each unordered map represented a block

5> As mentioned before I simulated a 3 bit counter to facilitate the pseudo LRU scheme used to block replacement which was used in the fully associative and set associative cache implementation.

6>Now, to check if the cache was filled or not for the initial stages of working, set trackers were enabled which showed the current number of elements in the cache and after the number of cache lines in the cache or the set were reached, then the replacement scheme started to kick in.

7>There was no initialization required in the establishment of a fully associative cache but an initialization were needed in the set associative cache and the direct mapping cache as those followed an index based mapping. These functions initialize the data and tag array with empty data so that the indexing can be done correctly. The initialization functions were:-

Direct Mapping: main_cache.initialize_data_and_tag_array()

Set Associative Mapping:

1>main_cache.initialize_set_tracker()

2>main_cache.initialize_data_and_tag_array()

Note: Set associative mapping requires a tracker for each set and hence a vector needs to be initialized in order to track all of the sets at the same time and the indexing from the address gets us the right set tracker

8> Encoders in a fully and set associative caches were simulated using simple counters which started from the top to the current position of the hit and then the encoding was inserted as an index of the data array to extract data.

9> There were various base converters which were used in order to convert data from binary to decimal to hexadecimal and all the possible permutations and combinations. The base change functions are given below:-

- convert_bin_to_int(string s)
- convert_int_to_bin(ll int num)
- convert_bin_to_hex(string binary)
- convert_hex_to_bin(string hex_string)

These conversions were carried out using the prebuilt hashmaps for string to integer conversion and string array for integer to string conversion in hexadecimal cases.

10> Output formatting functions and preprocessors directives were used in order to construct a table in the output files such as:-

- #define printrow_status(i,v) cout<<"|"<<setw(7)<<i<<"|"<<setw(32)<<v[0]<<"|"<<setw(14)<<v[1]<<"|"<<setw(15)<<v[2]<<"|"<<endl
- #define partition_status cout<<"+"<<"-------"<<"+"<<"--------------------------------"<<"+"<<"--------------"<<"+"<<"----------"<<"+"<<endl
- #define partition_cache_tag cout<<"+"<<"--------------------"<<"+"<<endl
- #define print_tag(s) cout<<"|"<<setw(20)<<s<<"|"<<endl
- #define print_offset(v) cout<<"|"<<setw(27)<<v[0]<<"|"<<setw(24)<<v[1]<<"|"<<endl
- #define partition_block cout<<"+"<<"============================"<<"+"<<"========================"<<"+"<<endl
- #define empty " "
- fillstr(string s)

11> Parameter checker functions were also used to raise errors such as:-

- check_params(ll int a, ll int b, ll int c, ll int d)
- check_pow_two(ll int a)

12> File reading was done by directing stdin and stdout to different files at appropriate times in the code:-

- #define fin freopen("Inputs/input.txt","r",stdin)
- #define fcache freopen("Outputs/cache.txt","w",stdout)
- #define frep freopen("Outputs/report.txt","w",stdout)
- #define fdebug freopen("debug.txt","w",stdout)

Now these lines mean that the code will only take input from the input.txt file in the Inputs directory and will only give the outputs in the Outputs directory. Remember before starting this program, the input.txt file must be ready. The output files will be created automatically if they don't exist. The debug.txt was used for the debugging operations in the code during its development but its use has been removed from the main code now.