Cache Project

Student 1: 2171052 허채린

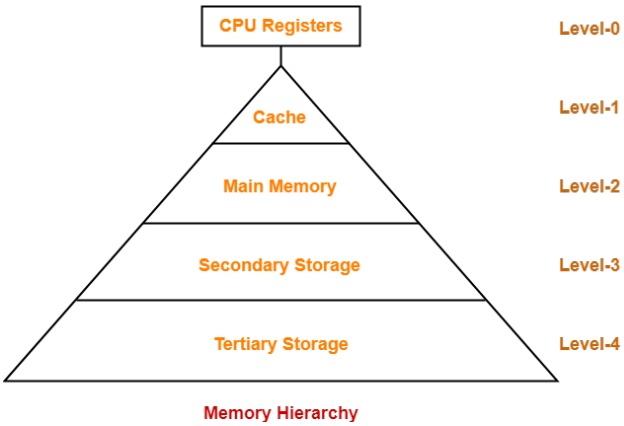
Student 2: 2176123 Jimin Nam

Index

1. Introduction and Background Information

This project conducts to understand the role of cache in a high-level language.

The cache consists of SRAM and is located at the top of the memory hierarchy. The reason why caches are designed is to use memory efficiently according to the principle of locality. The principle of locality can be exploited by implementing a computer's memory as a memory hierarchy. The memory hierarchy consists of several tiers of memory of different speeds and sizes. Cache is expensive and small compared to the main memory and the disk. In addition, the cache is closer to the CPU than the main memory and disk, so that the access time is faster. The unit in which the data is stored in the cache is called a block. Cache stores memory corresponding to a portion of main memory and when the data requested by the processor is in a block of cache, it is called a hit. Conversely, when the data requested by the processor is not found in a certain block in the cache, it is called a miss. When a miss event occurs, data is fetched from main memory. Usually, the hit ratio is used to evaluate the performance of the memory hierarchy. In this case, the hit ratio is composed of (number of hits)/(number of hits + number of misses). Furthermore, a miss event may occur due to the limited space of the cache. In this case, the least recently used data is pushed out. (Direct-mapped cache has its own place, so it is not necessary to do so in direct-mapped cache.)



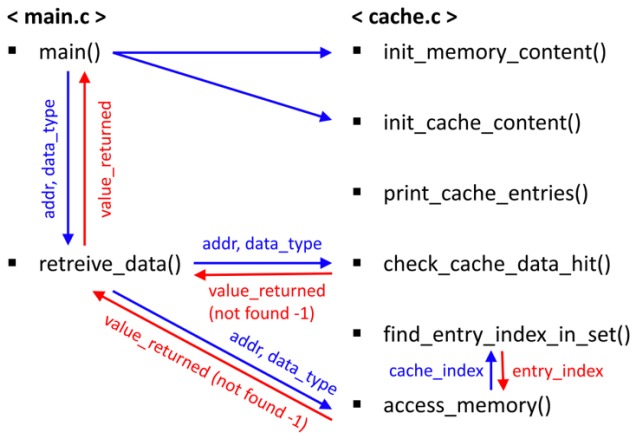
[pic1. The Picture of Memory Hierarchy]

To explain this project, we implement a simulator of n-way set associative cache by using C language. The cache size is set to 32 bytes and one block is define as 8 bytes. Therefore, the number of blocks is 4, so we can implement direct mapped cache, 2-way set associative cache, and fully associative cache(4-way set). This cache simulator writes the data returned to the output file when given an input file. At this time, in the input file, the byte address and data type are given. If there is data corresponding to the cache according to the value given in the input file, a hit event occurs and the data is fetched. Otherwise, the main memory is accessed to receive the data.

2. Overall Flow

Before we start, we cannot access to the actual memory and cache because we implement the cache simulator in a high-level language. So, set the arbitrary data to memory. Since the cache must be empty until the access, the corresponding setting is performed through init_cache_content(). After completing the initial setting, retrieve_data() is called from the main function to retrieve data. Function, retrieve_data() gives the value returned to the main function, which is based on the address and type from the input file.

Since, we need to access the cache call the check_cache_data_hit() function in the main function. In this function, if the data corresponding to the address exists in the cache, the data will be returned, otherwise -1 is returned. f -1 is returned, it means that a miss event has occurred, so the corresponding data must be retrieved from retrieve_data() through access_memory(). At this time, since it is recently accessed data, it must be stored in the cache. It means that the data must be overwritten to the cache. So, the entry index should be found, and the index is returned through the find_entry_index_in_set() function. Finally, print the data obtained through cache or memory access to a file according to the given format.



[pic2. Overall Flow]

3. Description of Given Functions and Variables

First, we will explain the variables defined in the code. In cache_impl.h, lots of variables are defined. To begin with, WORD_SIZE_BYTE indicates that how many bytes are in 1 word. DEFAULT_CACHE_SIZE_BYTE is a variable that determines how many bytes the cache size will be in this project. As described above, 32 bytes is set as the cache size. The DEFAULT_CACHE_BLOCK_SIZE_BYTE variable indicates how many bytes the size of one block is in the cache. In this project, one block size is 8 bytes. Furthermore, the DEFAULT_CACHE_ASSOC variable indicates cache associativity, and since the maximum number of blocks is 4, it is possible to set a total of 3 ways(1, 2, 4). DEFAULT_MEMORY_SIZE_WORD is a variable about how many words to store in the main memory. We specified the size of main memory as 128 words. Furthermore, CACHE_ACCESS_CYCLE is a variable representing the number of cycles required to access the cache, and is defined as 1. Also, MEMORY_ACCESS_CYCLE is a variable representing the cycle required to access memory, and is defined as 100, which reflects the fact that it usually takes longer than cache access time. Finally, CACHE_SET_SIZE is the value obtained by dividing the cache size by the product of the block size and associativity. So, with the variables defined earlier, CACHE_SET_SIZE was defined.

A structure called cache_entry contains variables needed to check the cache status. The 'valid' variable represents valid bit of cache, and the value is 1 if there is data in the cache, or 0 otherwise. The 'tag' variable is a variable that stores the tag of the saved data. 'timestamp' is a variable representing recent access time, and data[DEFAULT_CACHE_BLOCK_SIZE_BYTE] is an array that stores data in char type.

To explain the global variables included in main.c, num_cache_hits is a variable that means the number of hits, and num_cache_misses is a variable that means the number of misses. When a hit or miss event occurs, add one by one to corresponding variable. The variable, num_bytes is the number of accessed bytes. If

the access type is b in the input file, add 1. If the access type is h, add 2. Else if the access type is w, add 4. The num_access_cycle variable is a variable representing the number of clock cycles. The global_timestamp variable is the number of data access trials.

The functions given in the code includes initialize memory and cache contents, and a print_cache_entries(). First, to explain the memory initialization function in detail, a random hexadecimal number is put into an array of char type called sample_upward and sample_downward. At this time, after the shift operation is performed between the value stored in sample_upward and sample_downward, an or operation is performed between these values to generate a random memory value. To create this well, set i, j, and gap as variables. The cache content initially contains no data. However, the valid bit, tag, and timestamp of cache_entry_t must be initialized. So, since there is no data in each cache_entry type, the valid bit is set to 0. Also, since there is no data, there is no tag, so it is initially set to -1. Furthermore, since there is no access trial, all timestamps are set to 0. The print_cache_entries() function is a function for debugging to see which part is wrong by looking at the cache entries when situations such as values do not match when data is retrieved later through the retrieve_data() function. So, in this function, set number of cache, valid bit of each cache entry, tag number, timestamp value and data are to be printed.

4. Explanation of retrieve_data() and Main Function

First of all, to explain the retrieve_data() function, it is a function that receives an address and data access type as arguments, and accesses the cache or main memory to bring the data. We set a variable called value_returned because the function should return the data obtained through cache or main memory access. That is, this function returns value_returned. Also, the initial value of value_returned is set to -1, because if there is no data in the cache or memory, -1 is specified to be returned. This is because if there is data in the cache or main memory, the value of value_returned will be changed to the corresponding data, not -1. While executing retrieve_data(), the number of hit events and miss events can be counted. Prior to calculation, we did not consider the case where the data is neither in cache nor in memory. If the return value is -1 through check_cache_data_hit(), it means that there is no data at the given address in the cache. If there was data at the given address in cache, it has a different data value than (-1). Also, even if the returned data for the corresponding address value is -1, when expressed in hexadecimal, the expression method of the first two digits is different, so this is also distinguished. In this situation, access to memory is required, so use the access_memory function to access memory and bring data. In conclusion, we made a conditional statement through this situation. If value_returned is still -1 even after check_cache_data_hit(), it means that a cache miss event has occurred, so add num_cache_misses one by one. Also, in this situation, access to memory is required, so use the access_memory function to access memory and bring data. Conversely, no cache miss event has occurred means that a cache hit event has occurred, so add num_cache_hits one by one. Finally, value_returned, which is data obtained from cache or memory, is returned.

Looking at the main function, we create a pointer called ifp ofp to read and write the file. After that, the memory and cache are initialized. Open the file with

ifp and ofp respectively, open the input file in read mode, and open the output file in write mode. Writes formal parts to the output file to conform to the provided output format. Since the address and data type given in the input file must also be written in the output file, using a char array to store temporarily. When passing from address to data type in input file, since there is a blank or NULL character, use a conditional statement to save the first part as address and the second part as data type to write to the output file. To fit the provided format, print the address and data type using tabs and alignment, and calculate num_bytes according to the data type. In case of 'b', it means a byte, so 1 is added to num_bytes, and in case of 'h', it is half word, that is, 2 bytes, so 2 is added to num_bytes. In the case of 'w', since a word is 4 bytes, 4 is added to num_bytes. Function retrieve_data is called using the address value and access_type received from the input file. The returned value is stored in variable val, and write it to the output file. The num_cache_cyles variable is also calculated. In the case of a cache hit event, only cache access is made, but in the case of a cache miss event, memory access is performed after cache access, so it must be calculated based on adding CACHE_ACCESS_CYCLE + MEMORY_ACCESS_CYCLE. So we have to add num_cache_hits multiplied by CACHE_ACCESS_CYLE and num_cache_misses multiplied by (CACHE_ACCESS_CYLE+MEMORY_ACCESS_CYLE). In order to fit the provided format, lines were printed, and output statements according to cache associativity were implemented as conditional statements. The hit ratio should also be written in the output file. The hit ratio is the ratio of hit events to the total number of accesses. In this case, num_cache_hits should be divided by (num_cache_hits + num_cache_misses) and displays up to two decimal points. The bandwidth should also be printed to the output file, which is the number of bytes divided by the number of access cycles, and is also display up to two decimal points. Close the input file and output file, and executes

5. Explanation of check_cache_data_hit()

The check_cache_data_hit function receives addr and type as arguments. First, input an argument, addr in to the buffer. Convert it to int type for easy manipulation inside the function. So, we wrote a for loop and added an address variable to convert into int type`. During the loop, if a null character is reached, the input to the buffer is stopped. Also, whenever input is received, '0' is subtracted because char type is converted to int. Considering the relationship with the retrieve_data function, if there is no data corresponding to the address of the argument, -1 should be returned. So, in the check_cache_data_hit function, the variable to be returned is set as val, and the initial value should be set as -1. Next, find the set number and the tag of the cache based on the given address. Set of the address is the remainder of dividing the (address/cache block size) by the cache set size. Tag of the address is the quotient of dividing the (address/cache block size) by the cache set size. Since the number of entries is different depending on the associativity of the cache, the number of repetitions of the loop statement that checks whether the tag is the same varies. So, if the tag is the same during repeating the loop, check how many bytes to fetch through the type. A variable, size stores how much data to fetch at one time. Since 'b' is a byte, set the size as 1, 'h' is a halfword, so it is 2 bytes, and the size is 2. If neither of these applies, it is 'w', and it is one word and is composed of 4 bytes, so it can be said that the size is 4. Next, the current conditional statement is the situation when a hit event occurs in the cache, so data must be stored in the return value. Since we decided to return a variable called val, we need to save the data corresponding to val. Before starting, val was set to -1 earlier, but this is not an appropriate initial value to store the corresponding data, so change it to 0. Since the cache store two words(=8byte), accessed data start at 'address modulo 8' in cache and we store this value into the variable startbyte. When fetching data, since the preceding data is fetched from startbyte, when writing a value to val, startbyte+size-1 is a larger digit than startbyte. Therefore, the loop statement is executed in reverse order from startbyte+size-1 to startbyte. Also, the reason for subtracting 1 is that

when considering the index of an array, it starts from 0, so size-1 is used, and the situation is same in this loop. If the fetched data is greater than or equal to 0, we should shift two digits without thinking signed or not, and then add the newest fetched data. (When a byte to be fetched from the cache is a positive number, when we express the byte that fetched in the form of 0xabcdefgh, it has been confirmed(by sample_upward array and sample_downward array) that 'a' to 'f' does not enter the position of bit7 corresponding to h. Therefore, if the data fetched is a positive then we do not have to do 'a'+10 to change char type into int type.) Since the data is a hexadecimal number, two digits must be shifted left, so multiply by the power of 16 then add the newest fetched data which located in v. If the newly fetched data is a negative number, it is displayed as a to f in hexadecimal, but it cannot be recognized as an integer. In order to proceed this process more systematically, the v(newest fetched) stored using a buffer. Since we are fetching one byte of data at a time, we only need to check the 6th and 7th bits of the buffer. So, the method we thought of as a solution for 6^{th} bit is to subtract the value corresponding to the ASCII code of 'a', when 6^{th} bit of buffer is in between 'a'~'f', because a to f appear as 0 to 5 when this value(ASCII code of 'a') is subtracted. After that, add the number 10 again and process it so that it can be recognized as integer. Conversely, if the 6th bit is not between a and f, just subtract '0'. And since it is the 6th bit, multiply by 16.(shift left once and add with the 7^{th} bit) The same operation is performed for the 7th bit, and at the end, the original val is shifted left 2 and v is added. (shift left 2 means multiply power of 16) When this one process is finished, increase the timestamp by one. Finally, if a cache hit event has occurred, the data obtained through the above process will be stored in val and returned. If a cache miss event has occurred, -1 was set in consideration of returning val earlier, so val can be returned even if a miss event has occurred.

6. Explanation of access_memory()

In the retrieve_data function, the access_memory function is executed if a miss event occurs in the cache. The access_memory function has a part repeated with the check_cache_data_hit function. The work of moving addr, an argument received in char, to address, which is an int type, and the work of moving data to val are repeated. In these parts, the principle is the same as in the previous function, so additional explanation will be omitted.

The byte address is transferred to access_memory() as a parameter. The Block Address, which is 'Byte Address/cache block size(8byte)' is needful to access the memory. However, mem_array[] is integer array, which can only store 1 word(4 bytes) while Block address is based on block size, 2words. The index of mem_array[] for accessed data is 'Block Address*Block size / word size.' The data to write on the cache is 2word, 1 word data from calculated index and right after 1word.
The set is (byte address/DEFAULT_CACHE_BLOCK_SIZE_BYTE)%(CACHE_SET_SIZE) and accessed data will be stored at this calculated set in cache_array. By invoking function find_entry_index_in_set(), find the index of entry to overwrite data. Tag is (byte address/DEFAULT_CACHE_BLOCK_SIZE_BYTE)/(CACHE_SET_SIZE) and used to distinguish data.

Before writing data in cache, make valid bit zero to express this cache entry is used. Then store data, tag, timestamp in cache_array.
Throughout those processes, data which was 'miss' is written in cache. Now, return data according to byte address and data type. The timestamp increases of one whenever data in cache is accessed.

7. Explanation of find_entry_index_in_set()

Function find_entry_index_set() returns the index of entry where the data will be overwritten in cache.

First, if there is empty cache space, data can be written without erasing previous data. It can reduce the possibility of 'Miss'. For each entry in set, if the valid bit is zero, which means empty, return that entry's index.

However, when all of entry is full, the data which has not been accessed for the longest time should be erased. Since the timestamp show the time when the data is accessed, the entry of which timestamp is lowest should be selected.

8. Comparison Between the Result of the Function with the Actual Calculated Result

Example input data: 2-way Set Associative Cache

334 b

490 h

329 w

489 h

338 h

279 b

148 w

339 b

Set	Entry	Valid	Tag	Data	Time
0	0	0	-1		0
0	1	0	-1		0
1	0	0	-1		0
1	1	0	-1		0

1) 334 b

Byte Address: 334

Block Address: 334/8=41=101001

Set=1

Tag=10100_2=20_{10}=0x14

Check (1,0~1) => Valid 0=> Miss=> Copy

Mem index=Block Adress*Blocksize/World Size=41*8/4=82

Set	Entry	Valid	Tag	Data	Time
0	0	0	-1		0
0	1	0	-1		0
1	0	1	0x14	Mem[82-83]=0x76dc8923 65cb9a34	0
1	1	0	-1		0

2) 490 h

Byte Address: 490

Block Address: 490/8=61=111101

Set=1

Tag=11110_2=30_{10}=0x1e

Check (1,0)=> Different Tag

Check (1,1) => Valid 0=> Miss=> Copy

Mem index=Block Adress*Blocksize/World Size=61*8/4=122

Set	Entry	Valid	Tag	Data	Time
0	0	0	-1		0
0	1	0	-1		0
1	0	1	0x14	Mem[82-83]=0x76dc8923 65cb9a34	0
1	1	1	0x1e	Mem[122-123]=0xdc5423ab cb4334bc	1

3) 329 w

Byte Address: 329

Block Address: 329/8=41=101001

Set=1

Tag=10100_2=20_{10}=0x14

Check (1,0)=> Same Tag=> Hit

Set	Entry	Valid	Tag	Data	Time
0	0	0	-1		0
0	1	0	-1		0
1	0	1	0x14	Mem[82-83]=0x76dc8923 65cb9a34	2
1	1	1	0x1e	Mem[122-123]=0xdc5423ab cb4334bc	1

4) 489 h

Byte Address: 489

Block Address: 489/8=61=101001

Set=1

Tag=11110_2=30_{10}=0x1e

Check (1,0)=> Different Tag

Check (1,1) =>Same Tag=> Hit

Set	Entry	Valid	Tag	Data	Time
0	0	0	-1		0
0	1	0	-1		0
1	0	1	0x14	Mem[82-83]=0x76dc8923 65cb9a34	2
1	1	1	0x1e	Mem[122-123]=0xdc5423ab cb4334bc	3

5) 338 h

Byte Address: 338

Block Address: 338/8=42=101010

Set=0

Tag=10101_2=21_{10}=0x15

Check (0,0~1) => Valid 0=> Miss=> Copy

Mem index=Block Adress*Blocksize/World Size=42*8/4=84

Set	Entry	Valid	Tag	Data	Time
0	0	1	0x15	Mem[84-85]=0x54baab45 43a9bc56	4
0	1	0	-1		0
1	0	1	0x14	Mem[82-83]=0x76dc8923 65cb9a34	2
1	1	1	0x1e	Mem[122-123]=0xdc5423ab cb4334bc	3

6) 279 b

Byte Address: 279

Block Address: 279/8=34=100010

Set=0

Tag=10001_2=17_{10}=0x11

Check (0,0)=> Different Tag

Check (0,1) => Valid 0=> Miss=> Copy

Mem index=Block Adress*Blocksize/World Size=34*8/4=68

Set	Entry	Valid	Tag	Data	Time
0	0	1	0x15	Mem[84-85]=0x54baab45 43a9bc56	4
0	1	1	0x11	Mem[68-69]=0x65ba9a45 54a9ab56	5
1	0	1	0x14	Mem[82-83]=0x76dc8923 65cb9a34	2
1	1	1	0x1e	Mem[122-123]=0xdc5423ab cb4334bc	3

7) 148 w

Byte Address: 148

Block Address: 148/8=18=010010

Set=0

Tag=01001_2=9_{10}=0x09

Check (0,0)=> Different Tag

Check (0,1) => Different Tag => Miss=> Copy

Mem index=Block Adress*Blocksize/World Size=18*8/4=36

Timestamp (0,0):4 < (0,1):5 => Use Entry 0 in set 0

Set	Entry	Valid	Tag	Data	Time
0	0	1	0x09	Mem[36-37]=87ba7845 76a98956	6
0	1	1	0x11	Mem[68-69]=0x65ba9a45 54a9ab56	5
1	0	1	0x14	Mem[82-83]=0x76dc8923 65cb9a34	2
1	1	1	0x1e	Mem[122-123]=0xdc5423ab cb4334bc	3

8) 339 b

Byte Address: 339

Block Address: 339/8=42=101010

Set=0

Tag=10101_2=21_{10}=0x15

Check (0,0)=> Different Tag

Check (0,1) => Different Tag => Miss=> Copy

Mem index=Block Adress*Blocksize/World Size=42*8/4=84

Timestamp (0,0):6 > (0,1):5 => Use Entry 1 in set 0

Set	Entry	Valid	Tag	Data	Time
0	0	1	0x09	Mem[36-37]=87ba7845 76a98956	6
0	1	1	0x15	Mem[84-85]=0x54baab45 43a9bc56	7
1	0	1	0x14	Mem[82-83]=0x76dc8923 65cb9a34	2
1	1	1	0x1e	Mem[122-123]=0xdc5423ab cb4334bc	3

There are two Hits and six Miss and final state of cache is

Set	Entry	Valid	Tag	Data	Time
0	0	1	0x09	Mem[36-37]=87ba7845 76a98956	6
0	1	1	0x15	Mem[84-85]=0x54baab45 43a9bc56	7
1	0	1	0x14	Mem[82-83]=0x76dc8923 65cb9a34	2
1	1	1	0x1e	Mem[122-123]=0xdc5423ab cb4334bc	3

This result is same with the counted number of cache hit by project and the return value of print_cache_entries().

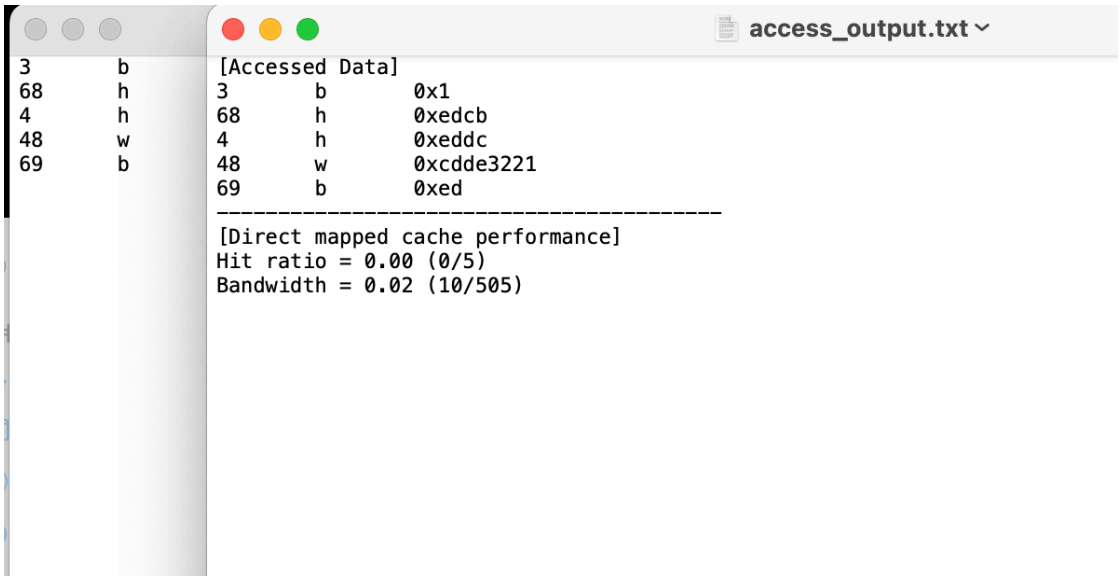
The Cache Project Program also work just like actual calculation. It gets Block address by Byte address and calculates set and tag. Then by comparing tag, divide data into 'Hit' and 'Miss'. If it's 'Miss', access to memory and update the cache.

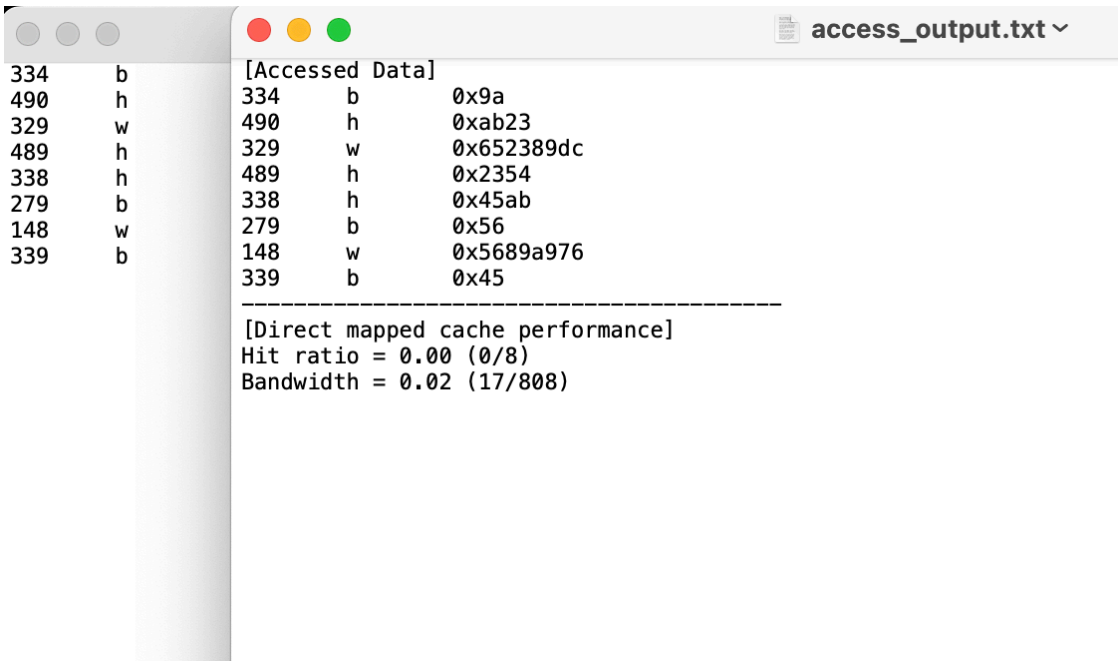
9. Conclusion and the Code

In conclusion, we made a cache simulator. Although direct real memory or real cache access was not possible from a high-level point of view, data was created arbitrarily, and accordingly, when the address and access type of the data were given, the corresponding data was returned. When a computer wants to obtain data at a corresponding address, it first accesses the cache to check if there is data corresponding the address, and if there is no data, it goes down to main memory and fetches the data. Also, when accessing the main memory and fetching data, the surrounding data is stored in the cache.

Also, we added an output window to make sure our output works well.

[1-way set associative cache]

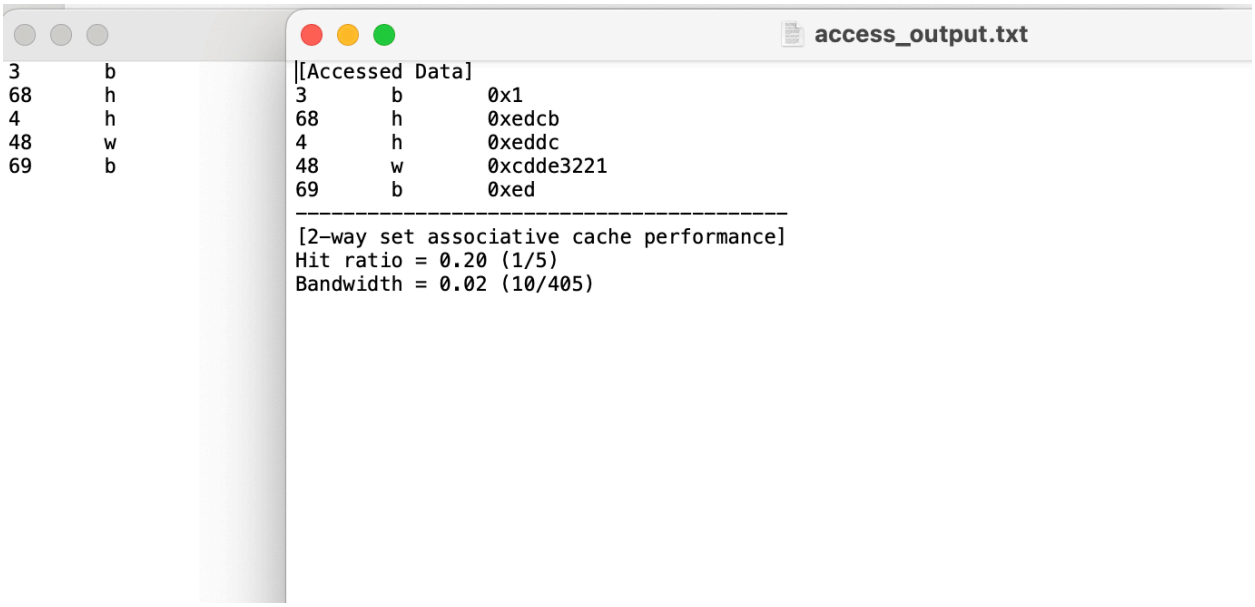




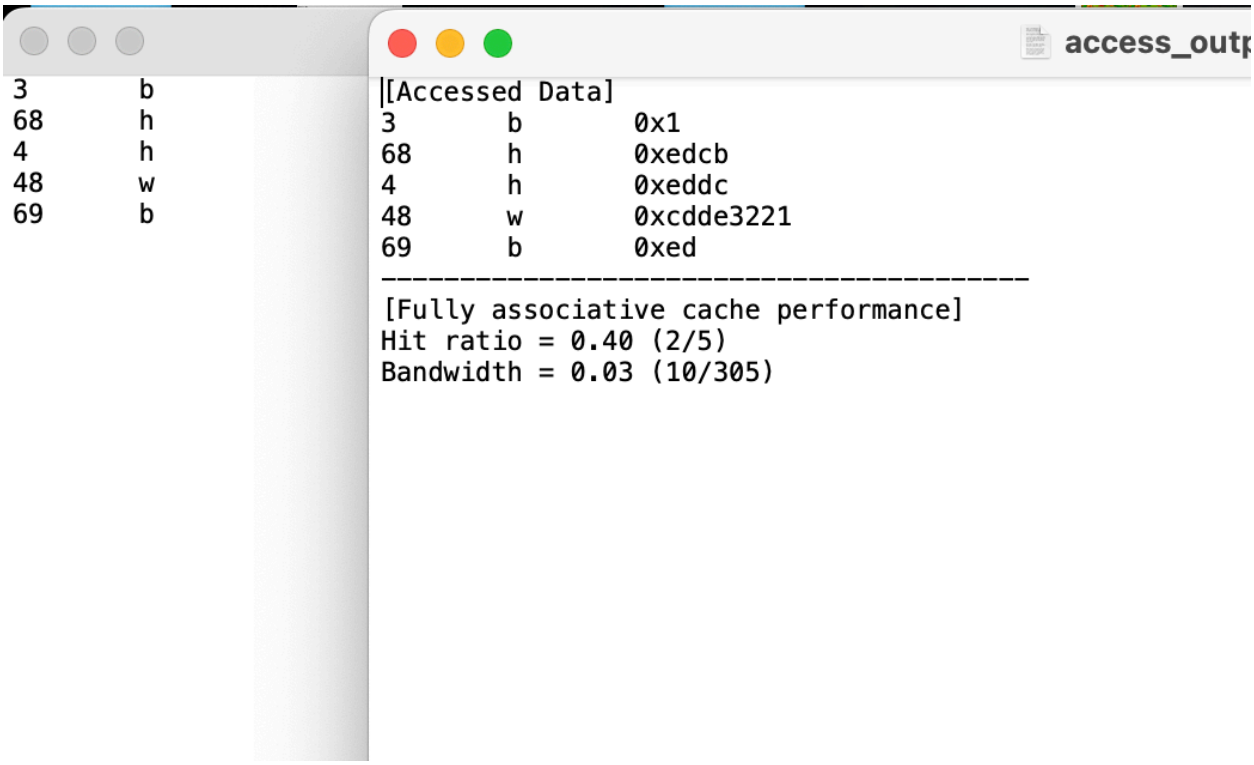
[2-way set associative cache]

access_output.txt

```
[Accessed Data]
334     b       0x9a
490     h       0xab23
329     w       0x652389dc
489     h       0x2354
338     h       0x45ab
279     b       0x56
148     w       0x5689a976
339     b       0x45
----------------------------------------
[2-way set associative cache performance]
Hit ratio = 0.25 (2/8)
Bandwidth = 0.03 (17/608)
```



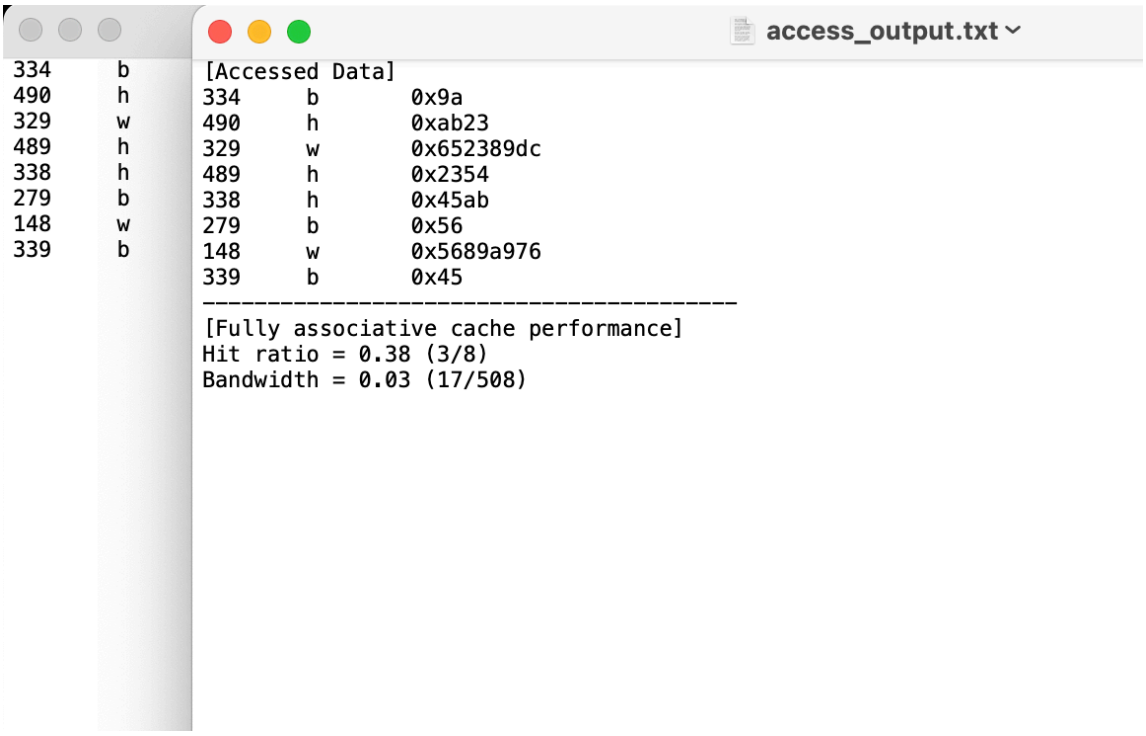
[Fully associative cache performance]



```
3       b
68      h
4       h
48      w
69      b
```

access_outp

```
[Accessed Data]
3       b       0x1
68      h       0xedcb
4       h       0xeddc
48      w       0xcdde3221
69      b       0xed
----------------------------------------
[Fully associative cache performance]
Hit ratio = 0.40 (2/5)
Bandwidth = 0.03 (10/305)
```



```
334     b
490     h
329     w
489     h
338     h
279     b
148     w
339     b
```

access_output.txt

```
[Accessed Data]
334     b       0x9a
490     h       0xab23
329     w       0x652389dc
489     h       0x2354
338     h       0x45ab
279     b       0x56
148     w       0x5689a976
339     b       0x45
----------------------------------------
[Fully associative cache performance]
Hit ratio = 0.38 (3/8)
Bandwidth = 0.03 (17/508)
```

[code: cache_imple.h]

*Some of the comments and codes may be cut off due to the screen size, so please check the commented file.

```c
/*
 * cache_impl.h
 *
 * 20493-01 Computer Architecture
 * Term Project on Implementation of Cache Mechanism
 *
 * Skeleton Code Prepared by Prof. HyungJune Lee
 * Nov 16, 2022
 *
 */

/* DO NOT CHANGE THE FOLLOWING DEFINITIONS EXCEPT 'DEFAULT_CACHE_ASSOC */

#ifndef _CACHE_IMPL_H_
#define _CACHE_IMPL_H_

#define WORD_SIZE_BYTE                  4           //1 word = 4 bytes
#define DEFAULT_CACHE_SIZE_BYTE         32          //cache size = 32 bytes
#define DEFAULT_CACHE_BLOCK_SIZE_BYTE   8           //block size = 8 bytes
#define DEFAULT_CACHE_ASSOC             4           /* This can be changed to 1(for direct mapped cache) or 4(for fully assoc cache) */
#define DEFAULT_MEMORY_SIZE_WORD        128         //main memory size is 128 words = 128 * 4 bytes
#define CACHE_ACCESS_CYCLE              1           //cache access event takes one clock cycle
#define MEMORY_ACCESS_CYCLE             100         //cache miss event takes a hundred clock cycles
#define CACHE_SET_SIZE                  ((DEFAULT_CACHE_SIZE_BYTE)/(DEFAULT_CACHE_BLOCK_SIZE_BYTE*DEFAULT_CACHE_ASSOC))    //cache set siz

/* Function Prototypes */
void init_memory_content();                         //Function to initialize memory value
void init_cache_content();                          //Function to initialize cache content
void print_cache_entries();                         //Function to print all cache entries for debugging
int check_cache_data_hit(void* addr, char type);    //Function that checks whether a cache hit event occurs
int access_memory(void *addr, char type);           //Functions that access data


/* Cache Entry Structure */
typedef struct cache_entry {                        //a structure to check the cache more systematically
    int valid;                                      //variable for checking a valid bit
    int tag;                                        //variable for checking a tag of the cache
    int timestamp;                                  //variable for most recent access time
    char data[DEFAULT_CACHE_BLOCK_SIZE_BYTE];       //data from memory
} cache_entry_t;                                    //define type as cache_entry_t


#endif

```

[code: cache.c]

```c
/*
 * cache.c
 *
 * 20493-01 Computer Architecture
 * Term Project on Implementation of Cache Mechanism
 *
 * Skeleton Code Prepared by Prof. HyungJune Lee
 * Nov 16, 2022
 *
 */


#include <stdio.h>
#include <string.h>
#include "cache_impl.h"

extern int num_cache_hits;                                      //# of hits
extern int num_cache_misses;                                    //# of misses

extern int num_bytes;                                           //# of bytes
extern int num_access_cycles;                                   //# of access cycles

extern int global_timestamp;                                    //most recent access time

cache_entry_t cache_array[CACHE_SET_SIZE][DEFAULT_CACHE_ASSOC];     //data in cache with size
int memory_array[DEFAULT_MEMORY_SIZE_WORD];                         //data in memory with size of 128 words


/* DO NOT CHANGE THE FOLLOWING FUNCTION */
void init_memory_content() {                                    //function for initializing the memory content
    unsigned char sample_upward[16] = {0x001, 0x012, 0x023, 0x034, 0x045, 0x056, 0x067, 0x078, 0x089, 0x09a, 0x0ab, 0x0bc, 0x0cd, 0x0de, 0x
    unsigned char sample_downward[16] = {0x0fe, 0x0ed, 0x0dc, 0x0cb, 0x0ba, 0x0a9, 0x098, 0x087, 0x076, 0x065, 0x054, 0x043, 0x032, 0x021,
    int index, i=0, j=1, gap = 1;                               //set up a variable to be used in a loop statement

    /*loop to construct actual values in memory*/
    for (index=0; index < DEFAULT_MEMORY_SIZE_WORD; index++) {      //loop for memory setup
        memory_array[index] = (sample_upward[i] << 24) | (sample_upward[j] << 16) | (sample_downward[i] << 8) | (sample_downward[j]);   //s
        if (++i >= 16)                                          //add i for cycle
            i = 0;                                              //cycle
        if (++j >= 16)                                          //add j for cycle
            j = 0;                                              //cycle

        if (i == 0 && j == i+gap)                               //difference of i and j == gap
            j = i + (++gap);                                    //increases 1 gap and new j for each value

        printf("mem[%d] = %#x\n", index, memory_array[index]);      //print to check the value in memory
    }
}

/* DO NOT CHANGE THE FOLLOWING FUNCTION */
void init_cache_content() {                                     //function for initializing the cache contents
    int i, j;                                                   //variable for loop

    for (i=0; i<CACHE_SET_SIZE; i++) {                          //loop statement for cache initializing
        for (j=0; j < DEFAULT_CACHE_ASSOC; j++) {               //loop statement for cache initializing
            cache_entry_t *pEntry = &cache_array[i][j];         //Set a pointer of type cache_entry_t to point the cache array.
            pEntry->valid = 0;                                  //Since there is nothing in initial, set the valid bit as 0.
            pEntry->tag = -1;                                   //Likewise, since there is no tag, set it to -1.
            pEntry->timestamp = 0;                              //Since there was no access trial, so set the timestampe as 0.
        }
    }
}
```

```
63
64  /* DO NOT CHANGE THE FOLLOWING FUNCTION */
65  /* This function is a utility function to print all the cache entries. It will be useful for your debugging */
66  void print_cache_entries() {                                    //function for printing cache entries
67      int i, j, k;                                                //set variable to use in the loop statement
68
69      for (i=0; i<CACHE_SET_SIZE; i++) {                          //for each set
70          printf("[Set %d] ", i);                                 //print the set number
71          for (j=0; j <DEFAULT_CACHE_ASSOC; j++) {                //for each entry in  a set
72              cache_entry_t *pEntry = &cache_array[i][j];         //set a pointer of type cache_entry_t to point the cache array to print
73              printf("V: %d Tag: %#x Time: %d Data: ", pEntry->valid, pEntry->tag, pEntry->timestamp);    //print valid bit, tag, and timesta
74              for (k=0; k<DEFAULT_CACHE_BLOCK_SIZE_BYTE; k++) {   //for each block in a entry
75                  printf("%#x(%d) ", pEntry->data[k], k);         //print data
76              }
77              printf("\t");                                       //print tab
78          }
79          printf("\n");                                           //print the line
80      }
81  }
83   int check_cache_data_hit(void *addr, char type) {              //a function that check whether the cache hit event happens
84
85      /* Fill out here */
86      /*Calculate set in the beginning.
87      And then caculate a tag and check whether the value of the tags are same.
88      If tags are same, hit event happens and set the return value.
89      Else if tag values are different, miss event happnes and set the return value as -1.*/
90
91      char buf[100];                                              //set a string buffer
92      sprintf(buf, "%s", (char*)addr);                            //store the address vaule in the buffer
93
94
95      int address=0;                                              //an arbitrary variable to convert the address value in the buffer into
96      for(int i=0;i<10;i++){                                      //
97          if(buf[i]=='\0')break;                                  //if '\0' appears in the buffer value, exit the loop
98          address=address*10+buf[i]-'0';                          //put the buffer value into the address, meanwhile multiply 10 to the o
99      }                                                           //since the buffer is a char type, '0' is subtracted
100
101
102     int val=-1;                                                 //val variable is the value to be returned by the check_cache_data_hit
103                                                                 //if the hit event did not occur, -1 should be returned, so the initial
104
105     int set=(int)(address/DEFAULT_CACHE_BLOCK_SIZE_BYTE)%(CACHE_SET_SIZE) ; //calculate the set
106                                                                 //set of the address is the remainder of dividing the (address/
107     int tag=(int)(address/DEFAULT_CACHE_BLOCK_SIZE_BYTE)/(CACHE_SET_SIZE) ; //calculate the tag
108                                                                 //tag of the address is the quotient of dividing the (address/ca
109     for(int i=0;i<DEFAULT_CACHE_ASSOC;i++){                     //the number of times the loop is repeated depends on the cache associa
110         if(cache_array[set][i].tag == tag){                     //when tags are same(hit event occurs)
111             int size;                                           //a variable named size is used to distinguish bytes according to acces
112             if(type=='b')size=1;                                //if the type is byte, the access type is one byte so the size is 1
113             else if (type=='h')size=2;                          //if the type is halfword, the access type is two bytes so the size is
114             else{ size=4; }                                     //else means that the wordk is the access type, so the size should be 4
115             int startbyte=address%DEFAULT_CACHE_BLOCK_SIZE_BYTE;//Since the cache store two words(=8byte), accessed data start at 'addr
116             val=0;                                              //set val zero
117             for(int j=startbyte+size-1;j>=startbyte;j--){        //access data from 'startbyte' to 'startbyte+size-1'(reverse). subtract
118                 int v=cache_array[set][i].data[j];              //get data from cache to varibale v
119                 if(v>=0)val=val*256+v;                          //if the data is larger or equal to 0
120                                                                 //since data is a hexadecimal number, two digits must be shifted left,
121                 else{                                           //else if the data is less than 0 (if first bit is '1', v is considered
122                     char outputData[10];                        //set another string buffer
123                                                                 //if v<0, v is 8 bit data. necessary data is in 6th, 7th bit and others
124                     sprintf(outputData, "%x", cache_array[set][i].data[j]); //store the data in the outputData buffer
125                                                                 //beacause data is stored as 'char', hexadecimal number A~F cannot reco
126                     if(outputData[6]>='a'&&outputData[6]<='f')v=outputData[6]-'a'+10;   //if data is 'a~f', subtract ascii value of a.
127                                                                 //then 'a~f' become '0~5'. add 10 because 'a' means
128                     else{ v=outputData[6]-'0'; }                //else means data is '0~9' in char. make it '0~9' in integer by subtrac
129                     v*=16;                                      //since data is a hexadecimal number, so multiply by the power of 16
130                     if(outputData[7]>='a'&&outputData[7]<='f')v=v+outputData[7]-'a'+10; //change heximal number 'a~f' to Integer '10~15'
131                     else{ v=v+outputData[7]-'0'; }              //change character '0~9' to integer '0~9'
132                     val=val*256+v;                              //since data is a hexadecimal number, two digits must be shifted left,
133                 }
134             }
135             cache_array[set][i].timestamp=global_timestamp++;   //increase the timestamp to indicate the recent access time
136             break;                                              //exit the loop
137         }
138     }
139     return val;                                                 //return the value (if cache hit events happen, return the data and if
140
141
142
143     /* Return the data */
144     //return 0;
145 }
```

```
int find_entry_index_in_set(int cache_index) {
    int entry_index=-1;                                         //value to check there is empty space and to return proper index in cac

    /* Check if there exists any empty cache space by checking 'valid' */
    for(int i=0;i<DEFAULT_CACHE_ASSOC;i++){                     //for each entry in set
        if(cache_array[cache_index][i].valid==0){               //if valid bit is 0. 'valid==0' means that pointing space is empty
            entry_index=i; break;                               //store index of empty space and break the loop
        }
    }
    if(entry_index==-1){                                        //if there is no empty space, 'entry_index' is still '-1'
                                                                //find entry with lowset timestamp because timestamp continuously incre
            int min=100;                                        //variable to find minimum. initialized by big number to compare
            for(int i=0;i<DEFAULT_CACHE_ASSOC;i++){             //for each entry in set
                if(cache_array[cache_index][i].timestamp<min){  //if timestamp of entry is lower than min,
                    entry_index=i;                              //set entry_index to i
                    min=cache_array[cache_index][i].timestamp;  //change min to minimun timestamp so far
                }
            }
    }

    /* Otherwise, search over all entries to find the least recently used entry by checking 'timestamp' */

    return entry_index;                                         //return proper index to access_memory()
}
```

```
int access_memory(void *addr, char type) {

    /* Fetch the data from the main memory and copy them to the cache */
    /* void *addr: addr is byte address, whereas your main memory address is word address due to 'int memory_array[]' */

    /* You need to invoke find_entry_index_in_set() for copying to the cache */

    char buf[100];                                              //set a string buffer
    sprintf(buf, "%s", (char*)addr);                            //store string of which adress is addr(parameter)
                                                                //byte address in decimal number is store in 'buf' by character.
    int address=0;                                              //variable to store adress
    for(int i=0;i<10;i++){                                      //add i for cycle
        if(buf[i]=='\0')break;                                  //break if string 'buf' has null(if string is done)
        address=address*10+buf[i]-'0';                          //since address is a decimal number, multiply by 10. then add acces
                                                                //since buf[i] is character type, make integer type by subtracting
    }
                                                                //now, Byte Adress is stored in 'address', integer variable

    int byteAddress=((int)(address/DEFAULT_CACHE_BLOCK_SIZE_BYTE))*DEFAULT_CACHE_BLOCK_SIZE_BYTE/WORD_SIZE_BYTE;
                                                                //calculate byte adress

    unsigned int memoryData=memory_array[byteAddress];          //access memory by calculated byte adress
                                                                //one word is stored in memoryData
    char data[DEFAULT_CACHE_BLOCK_SIZE_BYTE];                   //string for data that has to store in cache
                                                                //char holds 1 byte. 1 byte holds two hexadecimal number because on

    for(int i=0;i<4;i++){                                       //store memorry_array[byteAdress] in data[0]~data[3]
                                                                //since Little endian is used, LSB is stored first, at the lowest a
        data[i]=memoryData%(256);                               //store LSB in 'data[i]'
                                                                //since char store 2 hexadecimal number, use 16*16=256
        memoryData/=256;                                        //erase least two bits
    }

    memoryData=memory_array[byteAddress+1];                     //since cache store 2 words(8 byte) access memory of which adress i
                                                                //one word is stored in memoryData
    for(int i=4;i<8;i++){                                       //store memorry_array[byteAdress] in data[4]~data[8]
        data[i]=memoryData%256;                                 //store 2 least bits in 'data[i]'
        memoryData/=256;                                        //erase least two bits
    }
```

```
211
212     int set=(int)(address/DEFAULT_CACHE_BLOCK_SIZE_BYTE)%(CACHE_SET_SIZE) ;   //calculate set
213     int tag=(int)(address/DEFAULT_CACHE_BLOCK_SIZE_BYTE)/(CACHE_SET_SIZE) ;   //calculate tag
214
215     int index=find_entry_index_in_set(set);                          //invoke find_entry_index_in_set() for copying to the cache
216     cache_array[set][index].valid=1;                                 //set valid bit 1 to indicate this array is used
217     cache_array[set][index].tag=tag;                                 //write caculated 'tag value to tag of cache
218     cache_array[set][index].timestamp=global_timestamp++;            //write timestamp and then add 1 to indicate the recent access time
219     for(int i=0;i<8;i++){
220         cache_array[set][index].data[i]=data[i];                     //write 8bits of data in cache
221     }
222
223
224     int size;                                                        //a variable named size is used to distinguish bytes according to a
225     if(type=='b')size=1;                                             //if the type is byte, the access type is one byte so the size is 1
226     else if (type=='h')size=2;                                       //if the type is halfword, the access type is two bytes so the size
227     else{ size=4; }                                                  //else means that the wordk is the access type, so the size should
228     int startbyte=address%DEFAULT_CACHE_BLOCK_SIZE_BYTE;             //Since the cache store two words(=8byte), accessed data start at '
229     unsigned int val=0;                                              //set val zero
230     for(int j=startbyte+size-1;j>=startbyte;j--){                    //access data from 'startbyte' to 'startbyte+size-1'(reverse). subt
231         int v=cache_array[set][index].data[j];                       //get data from cache to varibale v
232         if(v>=0)val=val*256+v;                                       //if the data is larger or equal to 0
233                                                                      //since data is a hexadecimal number, two digits must be shifted le
234         else{                                                        //else if the data is less than 0 (if first bit is '1', v is consid
235             char outputData[10];                                     //set another string buffer
236                                                                      //if v<0, v is 8 bit data. necessary data is in 6th, 7th bit and ot
237             sprintf(outputData, "%x", cache_array[set][index].data[j]); //store the data in the outputData buffer
238                                                                      //beacause data is stored as 'char', hexadecimal number A~F cannot
239             if(outputData[6]>='a'&&outputData[6]<='f')v=outputData[6]-'a'+10;   //if data is 'a~f', subtract ascii value of a.
240                                                                          //then 'a~f' become '0~5', add 10 beacuse 'a' means '10' in
241             else{ v=outputData[6]-'0'; }                             //else means data is '0~9' in char. make it '0~9' in integer by sub
242             v*=16;                                                   //since data is a hexadecimal number, so multiply by the power of 1
243             if(outputData[7]>='a'&&outputData[7]<='f')v=v+outputData[7]-'a'+10;     //change heximal number 'a~f' to Integer '10~15'
244             else{ v=v+outputData[7]-'0'; }                           //change character '0~9' to integer '0~9'
245             val=val*256+v;                                           //since data is a hexadecimal number, two digits must be shifted le
246         }
247     }
248     /* Return the accessed data with a suitable type */
249
250     return val;                                                      //return the value
251 }
252
```

[code: main.c]

```c
/*
 * main.c
 *
 * 20493-01 Computer Architecture
 * Term Project on Implementation of Cache Mechanism
 *
 * Skeleton Code Prepared by Prof. HyungJune Lee
 * Nov 16, 2022
 *
 */

#include <stdio.h>
#include "cache_impl.h"

int num_cache_hits = 0;         //# of hits(use to get the value of hit ration)
int num_cache_misses = 0;       //# of misses(use to get the value of hit ration)

int num_bytes = 0;              //# of accessed bytes(use to get the value of bandwidth)
int num_access_cycles = 0;      //# of clock cycle(use to get the value of bandwidth)

int global_timestamp = 0;       //# of data access trials

//this function is for extracting datas
int retrieve_data(void *addr, char data_type) {
    /*the initial value should be -1, because if the cache hit happens,
    then the variable will be change as the value of data by check_cache_data_hit function*/
    int value_returned = -1; /* accessed data */

    /* Invoke check_cache_data_hit() */
    value_returned=check_cache_data_hit(addr,data_type);    //if cache_hit --> return the value of data, cache miss --> return -1

    /* In case of the cache miss event, access the main memory by invoking access_memory() */
    /*If cache miss event happens, value_returned will be -1 at first.
    Through the conditional statement, when a cache miss event occurs, value_returned is received through memory access.
    Furthermore, since a cache miss event has occurred, add count of num_cache_misses.*/
    if(value_returned ==-1){                            //when cache miss event occurs
        value_returned = access_memory(addr,data_type); //the memory is accessed through the function access_memory
        num_cache_misses++;                             //since, this conditional statement is for cache miss event, count the num_cache_m
    }
    else{                                               //this is because the only number of cases in which value_returned may not be -1 i
        num_cache_hits++;                               //count the num_cache_hits by adding one by one to count the number of cache hit e
    }
    return value_returned;                              //the value returned through cache or memory access is returned to the retrieve_da
}

//main function
int main(void) {                                    //start of main function
    FILE *ifp = NULL, *ofp = NULL;                  //declare the necessary pointers to prepare for opening and closing the file.
    unsigned long int access_addr;                  /* byte address (located at 1st column) in "access_input.txt" */
    char access_type;                               /* 'b'(byte), 'h'(halfword), or 'w'(word) (located at 2nd column) in "access_input.
    int accessed_data;                              /* This is the data that you want to retrieve first from cache, and then from memor

    init_memory_content();                          //initialize the memory through the previously created function
    init_cache_content();                           //initialize the cache throught the previously created function

    ifp = fopen("access_input.txt", "r");           //open the input file to be read, using r (read) mode
    if (ifp == NULL) {                              //conditional statement for when the pointer does not point to any file because the
        printf("Can't open input file\n");          //print to notify that the program cannot open the input file
        return -1;                                  //return -1 to indicate a non-normal termination
    }
    ofp = fopen("access_output.txt", "w");          //open the output file to write, using w (write) mode
    if (ofp == NULL) {                              //conditional statement for when the pointer does not point to any file because the
        printf("Can't open output file\n");         //print to notify that the program cannot open the output file
        fclose(ifp);                                //close the input file
        return -1;                                  //return -1 to indicate a non-normal termination
    }

    /* Fill out here by invoking retrieve_data() */

    fprintf(ofp,"%s\n","[Accessed Data]");          //print into file to fit the provided output format
```

```
71
72      /*Separate the part representing the address and the part representing the access_type through a conditional statement and write it to
73      To do this, we used the adr_str buffer and access_type variables.*/
74      char str[50];                                   //declare a string buffer, named str
75      while (fgets(str, sizeof(str), ifp) != NULL ) {     //read a string from the input file, and the save the string in str until the entir
76          char adrstr[10];                            //declare a string buffer for storing addresses
77          int i=0;                                    //a variable to be used in the loop statement is set into 0 in advance
78          for(;i<15;i++){                             //arbitrary number 15, since it breaks when it encounters a blank or NULL anyway, i
79              if(str[i]=='\0'||str[i]==' '||str[i]=='\t'){adrstr[i]='\0';break;}    //if the character entered into the string buffer is '\0'
80              else{                                   //If a general character that does not satisfy the condition of the above,
81                  adrstr[i]=str[i];                   //put the character of the string buffer into the address string buffer
82              }
83          }
84          for(;i<15;i++){                             //arbitrary number 15, since it breaks when it encounters a blank or NULL anyway, i
85              if(str[i]=='b'||str[i]=='h'||str[i]=='w'){  //if next character is 'b' or 'h' or 'w'
86                  access_type=str[i];                 //set the value in the string buffer as access_type(since the necessary part of the
87                  break;                              //since the access type is one char, we can break immediately after receiving the i
88              }
89          }
90          fprintf(ofp,"%-4s\t%c\t",adrstr,access_type);   //file print to fit the provided output format and print address and the access typ
91          if(access_type=='b')num_bytes+=1;           //if access_type is byte, add 1 in num_bytes since, a byte is one byte
92          else if(access_type=='h')num_bytes+=2;      //if access_type is halfword, add 2 in num_bytes since, a halfword is two bytes
93          else {num_bytes+=4;}                        //else means that the access_type is word, add 4, since a word is fout bytes
94          int val=retrieve_data(adrstr,access_type);  //the value obtained through memory or cache access is returned through the retriev
95          fprintf(ofp,"0x%x\n",val);                  //write a data obtained through memory or cache access to the output file, accordin
96      }
97
98      /*In the calculation method of num_access_cycle, data obtained through cache access is obtained only through cache access cycle, and da
99      The number of num_cache_misses must be multiplied by (memory access cycle + cache access cycle), and num_cache_hits should be multiplie
100     num_access_cycles=num_cache_misses*(CACHE_ACCESS_CYCLE+MEMORY_ACCESS_CYCLE)+num_cache_hits*CACHE_ACCESS_CYCLE;
101
102     fprintf(ofp,"%s\n","----------------------------------------");    //print into file to fit the provided output format
103     if(DEFAULT_CACHE_ASSOC ==1)fprintf(ofp,"%s","[Direct mapped cache performance]\n"); //If cache associativity is 1, the sentence is prin
104     else if(DEFAULT_CACHE_ASSOC ==2)fprintf(ofp,"%s","[2-way set associative cache performance]\n");    //If cache associativity is 2, the
105     else {fprintf(ofp,"%s","[Fully associative cache performance]\n");} //else case is for fully associative cache, so print the sentence a
106
107     //The hit ratio must divide num_cache_hits by (num_cache_hits + num_cache_misses), and we set the result to be printed until two decima
108     fprintf(ofp,"Hit ratio = %.2f (%d/%d)\n",(double)((double)num_cache_hits/(double)(num_cache_hits+num_cache_misses)),num_cache_hits,num_
109     //For bandwidth, num_bytes should be divided by num_access_cycle, and this value is also printed until two decimal point.
110     fprintf(ofp,"Bandwidth = %.2f (%d/%d)\n",(double)((double)num_bytes/(double)num_access_cycles),num_bytes,num_access_cycles);
111
112
113
114     fclose(ifp);                                    //close the input file
115     fclose(ofp);                                    //close the output file
116
117     print_cache_entries();···                       //print the final cache entries by invoking print_cache_entries()
118     return 0;                                       //return 1 to indicate a normal termination
119 }
```

10. References

- pic1 : https://www.gatevidyalay.com/memory-hierarchy-memory-hierarchy-diagram/

- pic2&other materials : Lecture Note, Term Project: Cache Implementation by Yeawon You.