Computer Architecture

Term project

due date : 2017-12-11

Team Mebers : 1415081 Hyunah Oh

1402081 Yujeong Lee

1. Code Analysis and Explanation

main.c

1. [main()] get order from file - the byte address and size of the data
2. [main()] calls retrieve_data()
3. [retrieve_data()] check if cache hit by calling check_cache_data_hit() and count cache access
4. [retrieve_data()]
   1) if cache hit get data from cache
   2) if cache miss call access_memory() get data from memory
5. [main()] write data to file

cache.c

check_cache_data_hit()

1. count global timestamp
2. find if there is same block address in cache, and it is valid
3.
   1) if cache hit, return data
   2) if cache miss, return '-1' <- Actually 'null' is better, and if you use 'null', you should cast return value to 'int'

access_memory()

1. find cache entry index by calling find_entry_index_in_set()
2. get data from memory, and set it in cache
3. change valid bit, tag, and timestamp to global timestamp

find_entry_index_in_set()

1. find cache entry where the valid bit is '0'
2.
   1) if there is, return cache entry index
   2) if there isn't, find Least Recently accessed cache entry by checking min(timestamp), and return cache entry index.

** additional comment : Of the following, yellow marked contents can be changed.

2. Program Code

main.c

```
/*
 * main.c
 *
 * 20493-02 Computer Architecture
 * Term Project on Implentation of Cache Mechanism
 *
 *
 */
#ifdef _MSC_VER
#define _CRT_SECURE_NO_WARNINGS
#endif // _MSC_VER

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

int num_cache_hits = 0;
int num_cache_misses = 0;

int num_bytes = 0;
int num_access_cycles = 0;

int global_timestamp = 0;

int retrieve_data(void *addr, char data_type) {  //get data
	int value_returned = -1;
		int i = 0;
	value_returned = check_cache_data_hit(addr, data_type);	//if(cache hit) {return data} else {return null}
		num_access_cycles += CACHE_ACCESS_CYCLE;
		//cache cycle++!
		if(-1 != (void*)value_returned) {
		//cache hit!
```

```
                num_cache_hits++;
        } else {
                        //cache miss! access_memory()
                num_cache_misses++;
                value_returned = access_memory(addr, data_type);                //get data from memory
                num_access_cycles += MEMORY_ACCESS_CYCLE;
                //mem cycle++!
        }

    return value_returned;
}

int main(void) {
    FILE *ifp = NULL, *ofp = NULL;
    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.txt" */
    int accessed_data;
                /* This is the data that you want to retrieve first from cache, and then from memory */

    init_memory_content();
    init_cache_content();

    print_cache_entries();
    ifp = fopen("access_input.txt", "r");          //input file name
    if (ifp == NULL) {
        printf("Can't open input file¥n");
        return -1;
    }
    ofp = fopen("input_output1.txt", "w");       //output file name
    if (ofp == NULL) {
        printf("Can't open output file¥n");
        fclose(ifp);
        return -1;
    }
```

```
	if (ifp != NULL) {                                  //ifp is file read pointer
		fprintf(ofp, "[Accessed Data] ¥n");
		while(!feof(ifp)) {                         //feof returns if ifp is pointing FILE END or not
			fscanf( ifp, "%d", &access_addr);
			while(1) {                                  //throw ' ' or '¥t'
				fscanf( ifp, "%c", &access_type);
				if (' ' != access_type && '¥t' != access_type) break;
			}
			fscanf( ifp, "¥n");                         //read by line
			accessed_data = retrieve_data((void*)access_addr, access_type); //get data
			fprintf(ofp, "%d %c ¥t 0x%x ¥n", access_addr, access_type, accessed_data);
                                                                //printf to file
		}
	}


	fprintf(ofp, "----------------------------------------¥n");
	switch (DEFAULT_CACHE_ASSOC)
	{
	case 1:
		fprintf(ofp, "[Direct mapped cache performance] ¥n");
		break;
	case 2:
		fprintf(ofp, "[2-way set accociative cache performance] ¥n");
		break;
	case 4:
		fprintf(ofp, "[Fully associative cache performance] ¥n");
		break;
	default:
		break;
	}
	fprintf(ofp, "Hit ratio = %.2f (%d/%d)¥n", (float)num_cache_hits/(num_cache_hits +
num_cache_misses), num_cache_hits, num_cache_hits + num_cache_misses);
	fprintf(ofp, "Bandwidth = %.2f (%d/%d)¥n", (float)num_bytes / num_access_cycles, num_bytes,
num_access_cycles);
```

```
    fclose(ifp);
    fclose(ofp);

    print_cache_entries();
    return 0;
}
```

---

cache.c

```
/*
 * cache.c
 *
 * 20493-02 Computer Architecture
 * Term Project on Implentation of Cache Mechanism
 *
 *
 */


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

extern int num_cache_hits;
extern int num_cache_misses;

extern int num_bytes;
extern int num_access_cycles;

extern int global_timestamp;

cache_entry_t cache_array[CACHE_SET_SIZE][DEFAULT_CACHE_ASSOC];
int memory_array[DEFAULT_MEMORY_SIZE_WORD];
```

```
/* DO NOT CHANGE THE FOLLOWING FUNCTION */
void init_memory_content() {
	unsigned char sample_upward[16] = {0x001, 0x012, 0x023, 0x034, 0x045, 0x056, 0x067, 0x078, 0x089,
0x09a, 0x0ab, 0x0bc, 0x0cd, 0x0de, 0x0ef};
	unsigned char sample_downward[16] = {0x0fe, 0x0ed, 0x0dc, 0x0cb, 0x0ba, 0x0a9, 0x098, 0x087, 0x076,
0x065, 0x054, 0x043, 0x032, 0x021, 0x010};
	int index, i=0, j=1, gap = 1;

	for (index=0; index < DEFAULT_MEMORY_SIZE_WORD; index++) {
		memory_array[index] = (sample_upward[i] << 24) | (sample_upward[j] << 16) | (sample_downward[i]
<< 8) | (sample_downward[j]);
		if (++i >= 16)
			i = 0;
		if (++j >= 16)
			j = 0;

		if (i == 0 && j == i+gap)
			j = i + (++gap);

		printf("mem[%d] = %#x¥n", index, memory_array[index]);
	}
}

/* DO NOT CHANGE THE FOLLOWING FUNCTION */
void init_cache_content() {
	int i, j;

	for (i=0; i<CACHE_SET_SIZE; i++) {
		for (j=0; j < DEFAULT_CACHE_ASSOC; j++) {
			cache_entry_t *pEntry = &cache_array[i][j];
			pEntry->valid = 0;
			pEntry->tag = -1;
			pEntry->timestamp = 0;
		}
	}
```

```
}

/* DO NOT CHANGE THE FOLLOWING FUNCTION */
/* This function is a utility function to print all the cache entries. It will be useful for your debugging */
void print_cache_entries() {
	int i, j, k;

	for (i=0; i<CACHE_SET_SIZE; i++) {
		printf("[Set %d] ¥n", i);
		for (j=0; j <DEFAULT_CACHE_ASSOC; j++) {
			cache_entry_t *pEntry = &cache_array[i][j];
			printf("V: %d ¥tTag: %#x ¥tTime: %d ¥tData: ", pEntry->valid, pEntry->tag, pEntry->timestamp);
			for (k=0; k<DEFAULT_CACHE_BLOCK_SIZE_BYTE; k++) {
				printf("%#x(%d) ", pEntry->data[k], k);
			}
			printf("¥t");
		}
		printf("¥n");
	}
}

int check_cache_data_hit(void *addr, char type) {			//address = [tag][index][offset]
		unsigned long int block_addr;
		int tag, index, offset;
		int i, data=0;
		int j;//
		cache_entry_t *cache_p;
		char *byte;
		int *word;
		short *half;

		block_addr = (int) addr / DEFAULT_CACHE_BLOCK_SIZE_BYTE;	//[tag][index][------]
		offset = (int) addr % DEFAULT_CACHE_BLOCK_SIZE_BYTE;		//[---][-----][offset]
		tag = block_addr / CACHE_SET_SIZE;				//[tag][-----][------]
		index = block_addr % CACHE_SET_SIZE;				//[---][index][------]
```

```
        global_timestamp++;

        for(i = 0; i < DEFAULT_CACHE_ASSOC; i++) {
                if(tag == cache_array[index][i].tag && 1 == cache_array[index][i].valid) {      //hit check
                cache_array[index][i].timestamp = global_timestamp;     //update last modified time
                        cache_p = &cache_array[index][i];
                        switch (type) {
                        //get data as much as you want!
                                case 'b' :
                                byte = &cache_p->data[offset];
                                data = *(byte);
                                num_bytes += 1;
                                break;
                        case 'h' :
                                half = (short*)&cache_p->data[offset];
                                data = *(half);
                                num_bytes += 2;
                                break;
                        case 'w' :
                                word = (int*)&cache_p->data[offset];
                                data = *(word);
                                num_bytes += 4;
                                break;
                        default :
                                printf("error!!! ¥n");
                                break;
                        }//switch
                        return data;
                }//if
        }//hit

    return -1; //miss
}

int find_entry_index_in_set(int cache_index) {
    int entry_index=0, i;
```

```
        long int min_value = 999999, min_index=0;

        for (i = 0 ; i < DEFAULT_CACHE_ASSOC ; i++) {          //first fill out the cache which valid bit is '0'
                if ( cache_array[cache_index][i].valid == 0) {
                        entry_index = i;
                        return entry_index;
                }
        }

        for (i = 0; i < DEFAULT_CACHE_ASSOC; i++) {
        //if all valid bit equals '1', find least recently modified cache by checking min(timestamp)
                if (cache_array[cache_index][i].timestamp < min_value) {
                        min_value = cache_array[cache_index][i].timestamp;
                        min_index = i;
                //find minumum timestamp and return the index as entry_index
                }
        }

        entry_index = min_index;

    return entry_index;
}

int access_memory(void *addr, char type) {

        int mem_index, entry_index, i;
    int data = -1;
        char *mem_p;
        int *wp;
        short *hw;
        unsigned long int block_addr;
        int tag, index, offset;
        int gl;
        block_addr = (int) addr / DEFAULT_CACHE_BLOCK_SIZE_BYTE;
        offset = (int)addr % DEFAULT_CACHE_BLOCK_SIZE_BYTE;
        tag = block_addr / CACHE_SET_SIZE;
```

```
index = block_addr % CACHE_SET_SIZE;

                    //mem_array is word size, 'mem_index' value will additionally needed!
mem_index = block_addr * 2;
                    //[ block_addr ] << 1
entry_index = find_entry_index_in_set(index);              //return cache entry index

mem_p = (char*)memory_array + mem_index*4;

for (i = 0; i < DEFAULT_CACHE_BLOCK_SIZE_BYTE; i++) {          // set data to cache
          cache_array[index][entry_index].data[i] = mem_p[i];
}

cache_array[index][entry_index].valid = 1;                  // update cache status
cache_array[index][entry_index].tag = tag;
cache_array[index][entry_index].timestamp = global_timestamp;
mem_p += offset;

switch (type) {                                    //get data as much as you want!
          case 'b' :
                    data = *(mem_p);
                    num_bytes += 1;
                    break;
          case 'h' :
                    hw = (short*)mem_p;
                    data = *(hw);
                    num_bytes += 2;
                    break;
          case 'w' :
                    wp = (int*)mem_p;
                    data = *(wp);
                    num_bytes += 4;
                    break;
          default :
                    printf("error!!! ¥n");
                    break;
```

```
        }

    return data;
}
```

cache_impl.h

```
/*
 * cache_impl.h
 *
 * 20493-02 Computer Architecture
 * Term Project on Implentation of Cache Mechanism
 *
*/

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

#ifndef _CACHE_IMPL_H_
#define _CACHE_IMPL_H_
#define WORD_SIZE_BYTE                  4
#define DEFAULT_CACHE_SIZE_BYTE         32
#define DEFAULT_CACHE_BLOCK_SIZE_BYTE   8
#define DEFAULT_CACHE_ASSOC             1    /* This can be changed to 1(for direct mapped cache)
or 4(for fully assoc cache) */
#define DEFAULT_MEMORY_SIZE_WORD        128
#define CACHE_ACCESS_CYCLE              1
#define MEMORY_ACCESS_CYCLE             100
#define CACHE_SET_SIZE
((DEFAULT_CACHE_SIZE_BYTE)/(DEFAULT_CACHE_BLOCK_SIZE_BYTE*DEFAULT_CACHE_ASSOC))

/* Function Prototypes */
void init_memory_content();
void init_cache_content();
void print_cache_entries();
int check_cache_data_hit(void* addr, char type);
int access_memory(void *addr, char type);
```

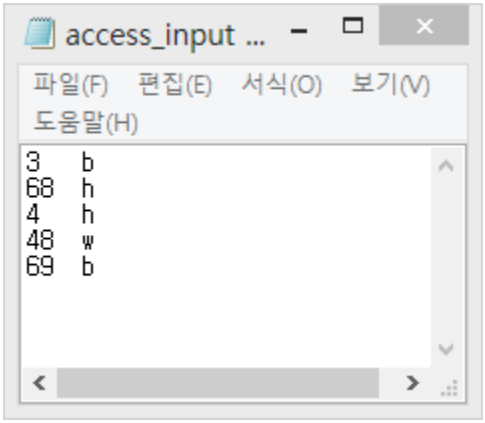
```c
/* Cache Entry Structure */
typedef struct cache_entry {
	int valid;
	int tag;
	int timestamp;
	char data[DEFAULT_CACHE_BLOCK_SIZE_BYTE];
} cache_entry_t;


#endif
```

3. Results

CASE 1 : fopen("access_input.txt", "r");

For input1 : "access_input.txt"



(1) "input_output1.txt" : directed mapped cache

DEFAULT_CACHE_ASSOC 1

fopen("input_output1.txt", "w");

input_output1.txt ×

```
[Accessed Data]
3 b       0x1
68 h      0xffffedcb
4 h       0xffffeddc
48 w      0xcdde3221
69 b      0xffffffed
-----------------------------------------
[Direct mapped cache performance]
Hit ratio = 0.00 (0/5)
Bandwidth = 0.02 (10/505)
```

(2) Output : 2-way set associative cache

DEFAULT_CACHE_ASSOC 2

fopen("input_output2.txt", "w");

input_output2.txt ×

```
[Accessed Data]
3 b       0x1
68 h      0xffffedcb
4 h       0xffffeddc
48 w      0xcdde3221
69 b      0xffffffed
-----------------------------------------
[2-way set accociative cache performance]
Hit ratio = 0.20 (1/5)
Bandwidth = 0.02 (10/405)
```

(3) Output : fully associative cache

DEFAULT_CACHE_ASSOC 4

fopen("input_output4.txt", "w");

input_output3.txt ×

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

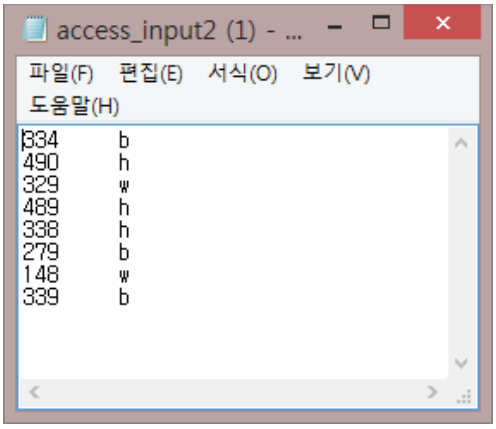
CASE 2 : fopen("access_input2.txt", "r");

For input2 : "access_input2.txt"

(1) "input2_output1.txt" : directed mapped cache

DEFAULT_CACHE_ASSOC 1

fopen("input2_output1.txt", "w");



input2_output1.txt

```
[Accessed Data]
334 b    0xffffff9a
490 h    0xffffab23
329 w    0x652389dc
489 h    0x2354
338 h    0x45ab
279 b    0x56
148 w    0x5689a976
339 b    0x45
----------------------------------------
[Direct mapped cache performance]
Hit ratio = 0.00 (0/8)
Bandwidth = 0.02 (17/808)
```

(2) Output : 2-way set associative cache

DEFAULT_CACHE_ASSOC 2

fopen("input2_output2.txt", "w");

input2_output2.txt

```
[Accessed Data]
334 b    0xffffff9a
490 h    0xffffab23
329 w    0x652389dc
489 h    0x2354
338 h    0x45ab
279 b    0x56
148 w    0x5689a976
339 b    0x45
----------------------------------------
[2-way set accociative cache performance]
Hit ratio = 0.25 (2/8)
Bandwidth = 0.03 (17/608)
```

(3) Output : fully associative cache

DEFAULT_CACHE_ASSOC 4

fopen("input2_output4.txt", "w");

input2_output3.txt

```
[Accessed Data]
334 b    0xffffff9a
490 h    0xffffab23
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)
```