Name: Nguyen Huynh Dang Khoa – Nguyen Thi Minh Hien
Class: 16ES

FINAL PROJECT ECE485

I. External specification:

- **Design 2 L1 cache: Instruction cache, data cache.**

1. Specification:

a. Instruction cache:

- 2-way associative.
- 16K sets.
- 64-byte line.

b. Data cache:

- 4-way associative.
- 16K sets.
- 64-byte line.

- Both caches use write allocate, write back except for the first write to line.
- Use LRU replacement policy, backed by a share L2 cache. The order of caches is inclusive.

2. Output:

a. Mode:

- Mode 1: Displays statistic and response to 9s in the trace file.
- Mode 2: Displays statistic and communicate messages.

b. Log activity:

- Number of cache reads.
- Number of cache writes.
- Number of cache hits.
- Number of cache misses.
- Cache hit ratio.

3. Software specification:

- Implemented on C.
- Cache, sets, lines are data structure.
- Total 2.06MB for instruction cache (1LRU+D+V+12Tag+64B, 2 ways) and 4.124MB for data cache (1LRU+D+V+12Tag+64B, 4 ways).
- Dynamic programming for memory allocation is needed.

II. System design:

1. System block diagram:

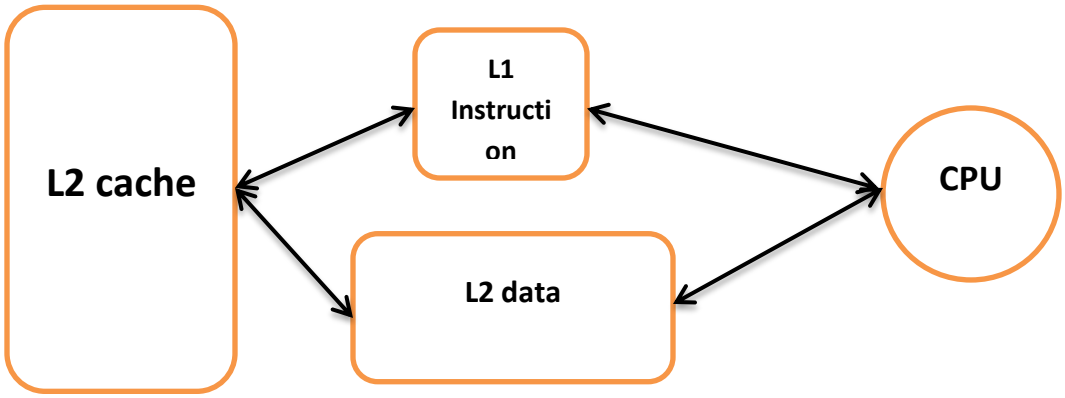


Figure 1: System main block diagram.

- **Cache structure:** Index from 0 ->16K -1

Index 0	Set 0
Index 1	Set 1
Index 2	Set 2
…	
Index 16537	Set 16537

- **Set structure:**
  + Instruction cache's set:

Line 0	Line 1

  + Data cache's set:

Line 0	Line 1	Line 2	Line 3

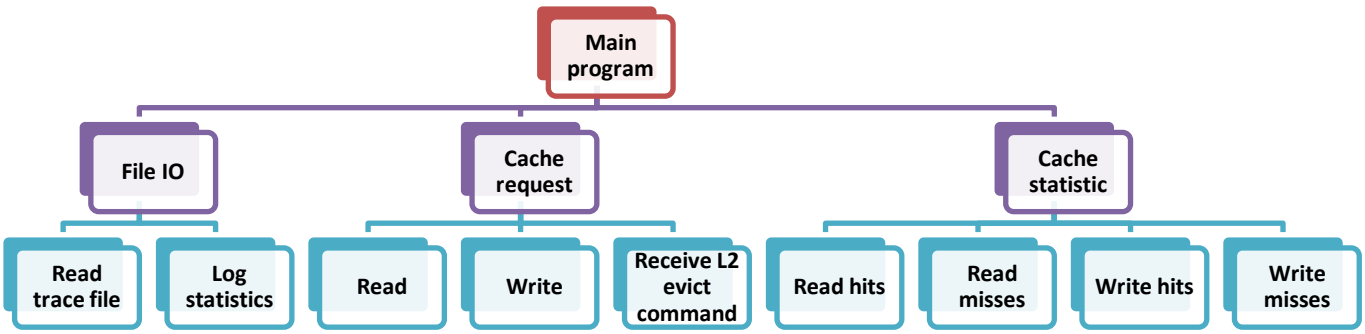
- **Line (block) structure:**
  + Instruction cache's line:

LRU bits(1 bit)	Dirty bit(D_BIT)	Valid bit(V_BIT)	Tag(12 bits)	Data(64 bytes)

  + Data cache's line:

LRU bits(2 bit)	Dirty bit(D_BIT)	Valid bit(V_BIT)	Tag(12 bits)	Data(64 bytes)

2. Functional decomposition:



3. Design:

a. Data structure:

- Cache struct: Consists of LRU, D, V, sets and additional variables for data processing.

```
typedef struct cache_struct {
    int bytes_num_bits;
    int sets_num_bits;
    int tags_num_bits;
    int ways_assoc;
    int LRU_num_bits;
    // int line_size;

    uint16_t D_BIT;
    uint16_t V_BIT;
    uint16_t LRU_line_mask;
    uint32_t tag_mask;
    uint32_t set_mask;
    uint32_t bytes_mask;
    set_t* sets;
}cache_t;
```

- Set and line struct:

```
typedef struct line_struct {
    uint16_t tag_array;
    uint8_t* data;
}line_t;

typedef struct set_struct {
    line_t* lines;
}set_t;
```

b. Function prototypes:

- File IO: #include <stdio.h>

```
fprintf (FILE *__restrict __stream, const char *__restrict __fmt, ...)
{
  return __fprintf_chk (__stream, __USE_FORTIFY_LEVEL - 1, __fmt,
                        __va_arg_pack ());
}
```

```
/* Read formatted input from STREAM.

   This function is a possible cancellation point and therefore not
   marked with __THROW.  */
extern int fscanf (FILE *__restrict __stream,
                   const char *__restrict __format, ...) __wur;
```

```
/* Return the EOF indicator for STREAM.  */
extern int feof (FILE *__stream) __THROW __wur;
```

- Cache request:

```
//Receive all request to cache L1:
int cache_request(int command, uint32_t address,
                  cache_stat_t* instr_stat,
                  cache_stat_t* data_stat);
```

```
//Return cache read hit/miss:
int cache_L1_read(cache_t* cache, uint32_t address, uint8_t*data);
//Return cache write hit/miss:
int cache_L1_write(cache_t* cache, uint32_t address, uint8_t data);
//Receive evict command from L2:
int cache_L2_evict(cache_t* cache, uint32_t address);
//Clear cache:
int cache_L1_clear(cache_t* cache);
```

- Cache statistic:

```
//Log activity:
cache_stat_t* cache_stat_create(char* cache_name, FILE* log_fp, int mode);
int cache_stat_init(cache_stat_t* stat,char* cache_name, FILE* log_fp, int mode);
int cache_stat_update(cache_stat_t*stat, return_t update, uint32_t address);
int cache_log(cache_stat_t *stat);
int clear_stat(cache_stat_t *stat);
```

c. Additional feature data structures and functions:

- command: indicate which command is used when call *cache_request();*

```
typedef enum command_enum {
    READ_DATA=0,
    WRITE_DATA,
    INSTRUCTION_FETCH,
    EVICT,
    CLEAR_CACHE=8,
    PRINT_CONTENT
}command_t;
```

- return status: indicate the status of the request when call *cache_request();*

```
typedef enum return_enum {
    READ_HIT=0,
    READ_MISS,
    WRITE_HIT,
    WRITE_MISS,
    WRITE_L2,
    READ_L2,
    READ_L2_OWN,
    EVICT_L2_OK,
    EVICT_L2_ERROR
}return_t;
```

4. **System workflow:**
   a. **Main program:**

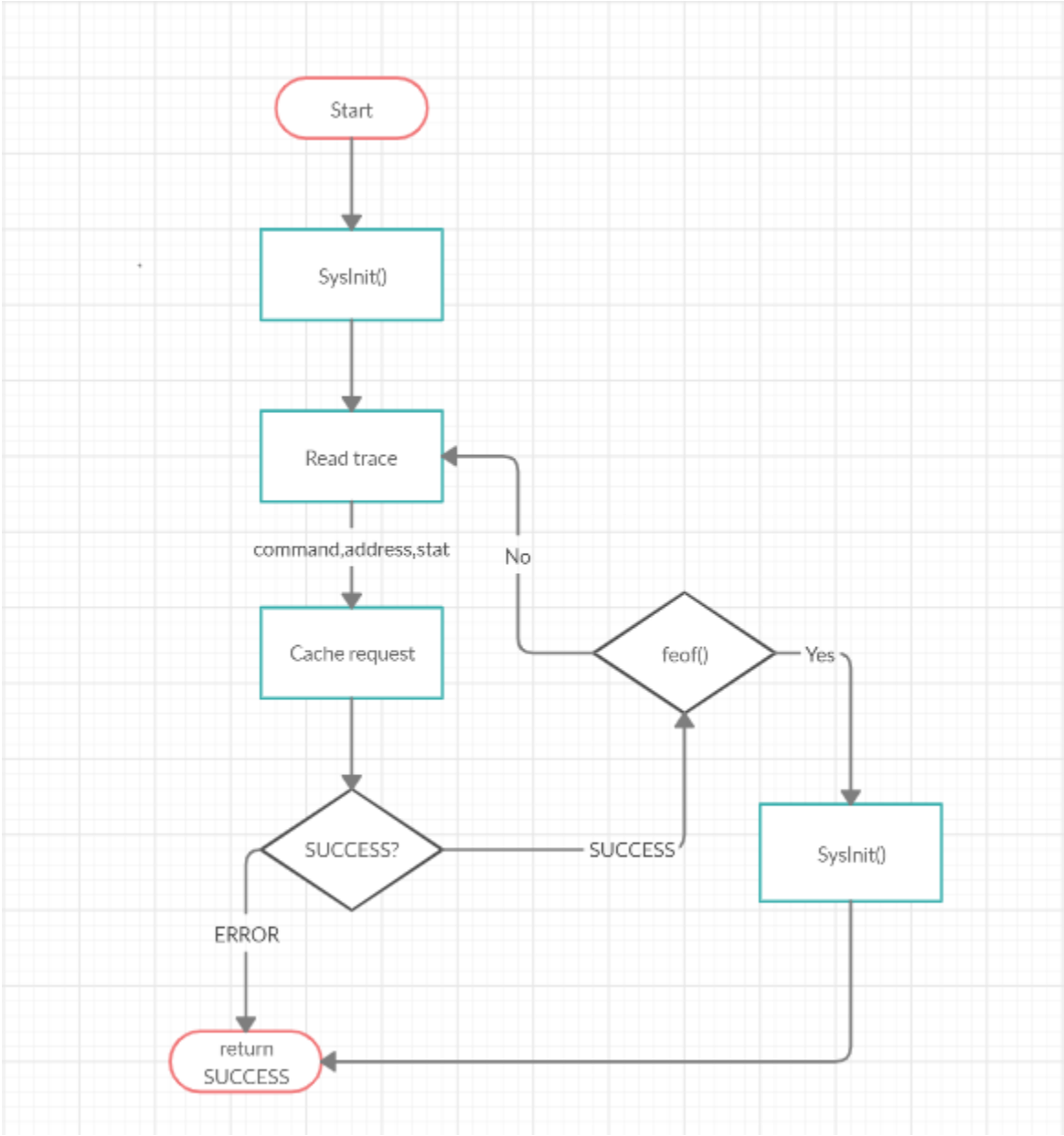


Figure 2: Flow chart of main program.

b. Cache request: Pseudo code for cache_request()

```
if(READ_DATA)
{
  update = cache_L1_read(data_cache, address,...);
  cache_stat_update(data_cache_stat, update,...);
}
else if(WRITE_DATA)
{
  update = cache_L1_write(data_cache, address, ...);
  cache_stat_update(data_cache_stat, update,...);
}
else if(INSTRUCTION_FETCH)
{
  update = cache_L1_read(instruction_cache, address, ...);
  cache_stat_update(instruction_cache_stat, update,...);
}
else if(EVICT)
{
  //to find which cache receives the command
  cache_to_invalidate = get_invalidate_cache(address);
  cache_L2_evict(cache_to_invalidate);
}
else if(CLEAR_CACHE)
{
  cache_L1_clear();
  clear_stat();
}
else if(PRINT_CONTENT)
{
  log to file
}
```

Figure 3: Pseudocode of cache request routine.

c. Cache L1 read: Pseudocode for cache_L1_read()

```
cache_L1_read(cache, address){
  tag, set, bytes_offset = extract address
  update = {}
  if(cache is NULL)
    return ERROR;

  if(set is NULL)
  {
    update add [READ_MISS];
    creat set; //malloc()
    read line from L2, update add [READ_L2];
    deliver the byte to CPU;
    return update;
  }
```

Figure 4: Cache read if set is NULL.

```
15    else//there are already some lines in set:
16    {
17      if(there is required line) //line_tag == addr_tag
18      {
19        update add [READ_HIT];
20        update LRU bits
21        deliver byte to CPU;
22        return update;
23      }
24      else{//miss
25        update add [READ_MISS];
26        if(set is not full)//count_valid < ways_assoc
27        {
28          update LRU bits;
29          read line from L2, update add [READ_L2];
30          place this line to the 1st available space.
31          return update;
32        }
33        else{//set is full:
34          index = calLRU();// get the LRU replacement index;
35          update LRU bits;
36          read line from L2, update add [READ_L2];
37          if  (line[index] not dirty)
38          {
39            place this line to [index];
40          }
41          else{//dirty
42            evict line[index];
43            place this line to [index];
44          }
45          return update;
46        }
47      }
48    }
49  }
```

Figure 5: Cache read if set has already some lines.

d. Cache write: Pseudocode for cache_L1_write()

```
cache_L1_write(cache, address){
  tag, set, bytes_offset = extract address
  update = {}
  if(cache is NULL)
    return ERROR;

  if(set is NULL)
  {
    update add [WRITE_MISS];
    creat set; //malloc()
    read line from L2, update add [READ_L2_OWN];
    write the byte to [bytes_offset];
    return update;
  }
```

Figure 6: Cache write if set is NULL.

```
  else//there are already some lines in set:
  {
    if(there is required line) //line_tag == addr_tag
    {
      update add [WRITE_HIT];
      update LRU bits
      write the byte to [bytes_offset];
      return update;
    }
    else{//miss
      update add [WRITE_MISS];
      if(set is not full)//count_valid < ways_assoc
      {
        update LRU bits;
        read line from L2, update add [READ_L2_OWN];
        place this line to the 1st available space.
        write the byte to [bytes_offset];
        return update;
      }
      else{//set is full:
        index = calLRU();// get the LRU replacement index;
        update LRU bits;
        read line from L2, update add [READ_L2_OWN];
        if  (line[index] not dirty)
        {
          place this line to [index];
        }
        else{//dirty
          evict line[index];
          place this line to [index];
        }
        write the bytes to [bytes_offset];
        return update;
      }
    }
```

Figure 7: cache write when set has already some lines.

e. Cache evict: Pseudocode for cache_L2_evict()

```
cache_L2_evict()
{
  change valid bit to 0
  only change valid line.
}
```

f. Update LRU routine:

```
update_line_LRU(accessed_lru)
{
  if(get a line to invalid place)
  {
    all LRU of valid lines += 1;
  }
  else if(access cache)//any write, read
  {
    +1 to (LRU of valid lines that LESS than accessed_lru)
  }
  else if(evict command)
  {
    -1 to (LRU of valid lines that LARGER than accessed_lru)
  }
}
```

Figure 8: Update LRU routine has 3 rules to update.

5. Testing:

a. Test Read command:

Project Name:	**Cache L1**	**Test Designed by:**	Nguyen Huynh Dang Khoa	Nguyen Thi Minh Hien
Module Name:	**Cache_L1_read**	**Test Designed date:**	30-05-2020	30-05-2020
Release Version:		**Test Executed by:**	Nguyen Huynh Dang Khoa	Nguyen Thi Minh Hien
		Test Execution date:	30-05-2020	30-05-2020

C	command	address	Expected Result	Actual Result	Status (Pass/Fail)	Notes
1	Read instruction	0x408ed4	Read miss	Read miss	Pass	The set is first NULL, then created.
2	Read instruction	0x408edf	Read hit	Read hit	Pass	
3	Read instruction	0x408eda	Read hit	Read hit	Pass	
4	Read instruction	0x108ed4	Read miss	Read miss	Pass	
5	Read instruction	0x408ed3	Read hit	Read hit	Pass	
6	Read Instruction	0x408ed4	Read hit	Read hit	Pass	
7	Read instruction	0x308edf	Read miss	Read miss	Pass	
8	Read instruction	0x308ed4	Read hit	Read hit	Pass	
9	Read instruction	0x408ed4	Read hit	Read hit	Pass	

```
[LOG] Mode: 1
------------------------------
> Cache: Instruction, log: 0
> #reads        : 9
> #writes       : 0
> Read hits     : 6
> Read misses   : 3
> Write hits    : 0
> Write misses  : 0
> Hit rate: 66.7%
------------------------------
```

Figure 9: Test result 1.

b. Test both read and write:

Project Name:	Cache L1	Test Designed by:	Nguyen Huynh Dang Khoa	Nguyen Thi Minh Hien
Module Name:	Cache_L1_read, Cache_L1_write	Test Designed date:	30-05-2020	1-06-2020

Release Version:		**Test Executed by:**	Nguyen Huynh Dang Khoa	Nguyen Thi Minh Hien
		Test Execution date:	30-05-2020	1-06-2020

C	command	address	Expected Result	Actual Result	Status (Pass/Fail)	Notes
1	Read data	0x10019d94	Read miss	Read miss	Pass	The set is first NULL, then created.
2	Write data	0x10019d99	Write hit	Write hit	Pass	
3	Read data	0x20019d88	Write miss	Write miss	Pass	No need to replace because data cache is 4way
4	Read data	0x10019d94	Read hit	Read hit	Pass	
5	Read data	0x40019d94	Read miss	Read miss	Pass	No replacement
6	Write data	0x10019d94	Write hit	Write hit	Pass	
7	Read data	0x30019d94	Read miss	Read miss	Pass	Now set is full
8	Read data	0x70019d94	Read miss	Read miss	Pass	LRU replacement

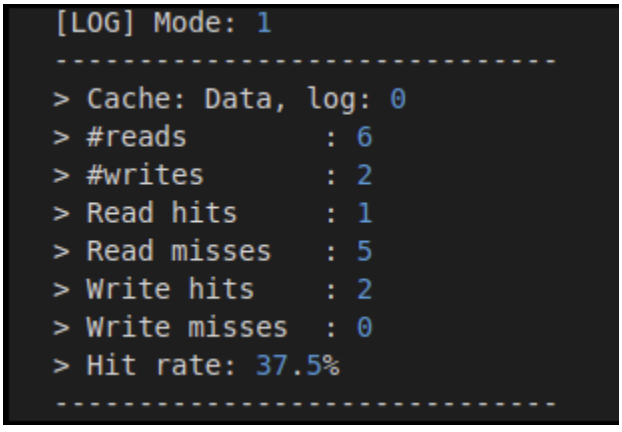


Figure 10: Test result 2.

c. Test evict, read, write, LRU replacement:

Project Name:	**Cache L1**	**Test Designed by:**	Nguyen Huynh Dang Khoa	Nguyen Thi Minh Hien
Module Name:	**Cache_L1_read, Cache_L1_write, Cache_L2_evict**	**Test Designed date:**	1-06-2020	1-06-2020
Release Version:		**Test Executed by:**	Nguyen Huynh Dang Khoa	Nguyen Thi Minh Hien
		Test Execution date:	1-06-2020	1-06-2020

C	command	address	Expected Result	Actual Result	Status (Pass/Fail)	Notes
1	Read data	0x10019d94	Read miss	Read miss	Pass	The set is first NULL, then created.
2	Read data	0x20019d88	Write miss	Write miss	Pass	No need to replace because data cache is 4way
3	Read data	0x40019d94	Read miss	Read miss	Pass	
4	Read data	0x30019d94	Read miss	Read miss	Pass	Set is full now
5	Write data	0x70019d94	Write miss	Write miss	Pass	LRU replacement at index=0
6	Evict L2	0x10019d94	No affect	No affect	Pass	Because tag=0x100 was replaced before
7	Evict L2	0x40019d94	Evict index=2	Evict index=2	Pass	
8	Read data	0x10019d94	Read miss	Read miss		Place in index=2

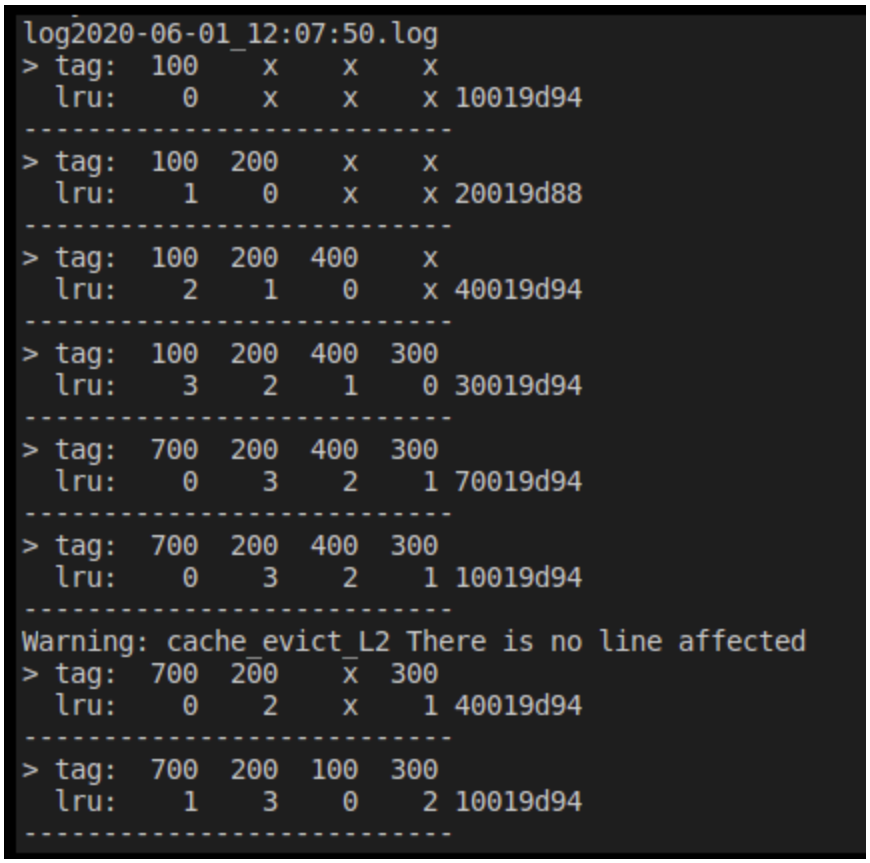


Figure 11: Log result of test 3.

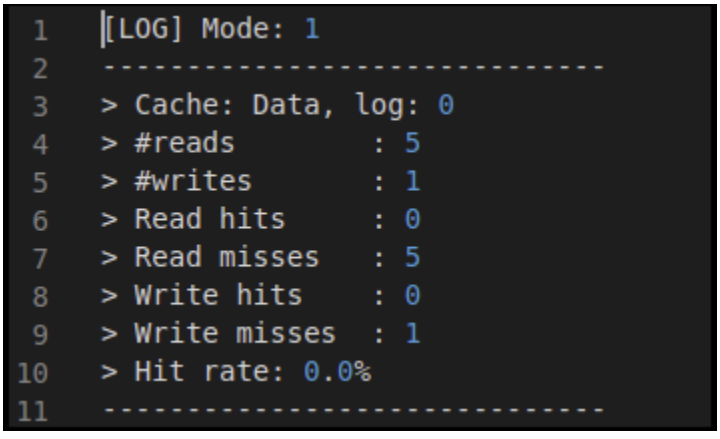


Figure 12: Result of test 3.

6. Maintainability:

a. System re-design ability:

- This design has flexible implementation:
  - User can change specification of the cache and build their own system. These include: address size (32bit default), address segments, number of sets, associativity, line size.

```
//The rest is instruction memory:
#define INSTR_BASE_ADDR 0x0
#define INSTR_END_ADDR  0xffffff
#define INSTRUCTION_CACHE                0
#define INSTRUCTION_CACHE_ASSOC_WAYS     2
#define INSTRUCTION_CACHE_NUM_SETS       16*K
#define INSTRUCTION_CACHE_LINE_SIZE      64

//data memory from 0-> 3/4 * 2^32 -1
#define DATA_BASE_ADDR  0x1000000
#define DATA_END_ADDR   0xffffffff
#define DATA_CACHE                       1
#define DATA_CACHE_ASSOC_WAYS            4
#define DATA_CACHE_NUM_SETS              16*K
#define DATA_CACHE_LINE_SIZE             64
```

Figure 13: re-configure these configurations relate to user's specification.

b. Reliability:

- Single point failure: Only stop immediately when internal errors occur (segmentation fault, …).
- The return of modules is status, makes them easy to debug.
- Dynamic programming makes your program can maintain reliability even the cache is big. The performance of memory usage will be better.

c. Version control system: Git

- This project use Git as version control system to maintain the most robust code as the final prototype.
- Any develop activities will not affect to the "*master"* code.
- Test, log version is also available aside the "*master"* code.