CacheImplementation_main()

2021년 11월 28일 일요일 오후 6:25

global variables

```
/* hit ratio = (num_cache_hits / (num_cache_hits + num_cache_misses)) */
int num_cache_hits = 0;    // # of hits
int num_cache_misses = 0; // # of misses

/* bandwidth = (num_bytes / num_acess_cycles) */
int num_bytes = 0;         // # of accessed bytes
int num_access_cycles = 0;// # of clock cycles

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

main()

```
FILE *ifp = NULL, *ofp = NULL; // file pointers
unsigned long int access_addr; // byte address (located at 1st column)
char access_type;              // 'b'(byte), 'h'(halfword), or 'w'(word)
                               // (located at 2nd column)
int accessed_data;             // data to retrive

/* initialize memory and cache
   by invoking init_memory_content() and init_cache_content() */
init_memory_content();
init_cache_content();

/* open input file as reading mode */
ifp = fopen("access_input.txt", "r");
if (ifp == NULL) {
   printf("Can't open input file\n");
   return -1;
}
/* open output file as writing mode */
ofp = fopen("access_output.txt", "w");
if (ofp == NULL) {
   printf("Can't open output file\n");
   fclose(ifp);
   return -1;
}
```

반복 : 모든 line을 읽을 때까지

ifp ⇒ read line (string line)

line ⇒ (tot 4 space 초과금)

line[0] ⇒ Byte Address line[1] ⇒ Data Type

address → void pointer type.

↳ address로 전달.

- b ⇒ 1byte
- h ⇒ 2byte
- w ⇒ 4byte

retrieve_data(*addr, char type)

↳ return: value_returned (cache or mem data)

print file ⇒ hit ratio, bandwidth, data.

print file ∋ hit ratio, bandwidth, data.
global timestamp ++

If retrieve_data == -1, error.
no such data exist in memory nor cache.

```
/* close files */
fclose(ifp);
fclose(ofp);

/* print the final cache entries by invoking print_cache_entries() */
print_cache_entries();
return 0;
```

CacheImplementation_retrieve_data()

2021년 11월 28일 일요일 오후 6:53

```
int retrieve_data(void *addr, char data_type) {
   int value_returned = -1; /* accessed data */
```

switch data_type :
X case 'w' : length 4 default : length 1.
case 'h' = length 2

int num_bytes += length.

```
void func(void *x, char type) {
   if (type == 'b')
      printf("%#x\n", (char *)x);
   else if (type == 'h')
      printf("%#x\n", (short *)x);
   else (type == 'w')
      printf("%#x\n", (int *)x);
```

value_ret = check_cache_data_hit(addr, data_type)
if (value_ret == -1)
{ miss ++
~~access_cycle += 101~~ → inside access_mem
value_ret = access_memory(addr, data_type)

else
{ hit ++
~~access_cycle ++~~ ⇒ inside check_cache_data_hit

return value_return; → if no data in cache nor memory, return (-1)

CacheImplementation_check_cache_data_hit()

2021년 11월 28일 일요일 오후 7:31

```
/* This function is to return the data in cache */
int check_cache_data_hit(void *addr, char type) {
access_cycle ++ → 헤더파일 상수 이용
# of blocks = cache size / Block size
block address = *addr / Block size
#of set = # of blocks / Default_cache_Assoc. → 헤더파일 상수 이용
tag = block address / # of set
set = block address % # of set
for(i = 0 ; i < Default_cache_Assoc ; i++)   // i → entry
{ if (cache [set] [i] → valid)
  { if (cache[set][i] → tag == tag)
    { → timestamp update
    Byte offset = *addr % Block size
    switch (type) :
      { case 'w' : 4 length
        case 'h' = 2
        case 'b' = 1
    for (j = byte offset ; j < length ; j++)   // byte offset +
    {result = (result << 8)   → hex 2자리 shift
     | (cache[set][i] → data[j])   or로 이어붙이기 연산.
    return result.
  }
}
return -1
```

```
void func(void *x, char type) {
  if (type == 'b')
    printf("%#x\n", (char *)x);
  else if (type == 'h')
    printf("%#x\n", (short *)x);
  else (type == 'w')
    printf("%#x\n", (int *)x);
```

↳ 이걸로 리턴 값 인식되면 ★

(char / short / int *)

*(cache[set][i] → data[byte offset])

```
===== addr 68 type h =====
CACHE >> block_addr = 8, byte_offset = 4, cache_index = 0, tag = 2
=> Miss!
MEMORY >> word index = 16
```

) output 출력 형식.

CacheImplementation_access_memory()

2021년 11월 28일 일요일 오후 8:44

```
/* This function is to return the data in main memory */
int access_memory(void *addr, char type) {
```

access_cycle ~~+= 100~~ → 헤더파일 상속.

~~# of blocks = cache size / Block size~~

Block address = *addr / Block size

~~#of set = # of blocks / Default_cache_Assoc.~~ → 헤더파일 상속.

tag = block address / # of set

set = block address % # of set

entry index = find_entry_index_in_set (set)

copy address = block address * block size / word size

> 예외처리 코드 여기에 (mem 주소 넘어가면 -1 return)

캐시 선택

cache [set][entry index] → valid = 1

cache [set][entry index] → tag = tag

cache [set][entry index] → time = global time stamp.

cache [set][entry index] → Data = Mem [copy address ~ copy address + (

↳ mem 배열구조에 따라 copy 방법 생각하기

→ timestamp update

Byte offset = *addr % Block size

switch (type):

{ case 'w': 4 length

case 'h' = 2

case 'b' = 1

for (j = byte offset; j < length; j++)

{result = (result << 8) → hex 2자리 shift

|(cache[set][i] → data[j]) → 이어붙이기

return result.

byte offset +

mem → cache 카피 후 cache에서 data 찾기 (앞 코드와 동일)

CacheImplementation_find_entry_index_in_set()

2021년 11월 28일 일요일 오후 9:18

```
/* This function is to find the entry index in set for copying to cache */
int find_entry_index_in_set(int cache_index) {
    int entry_index;
```

↳ set #

```
for (i = 0; i < Default_cache_assoc; i++)
{ if(!cache[cache_index][i] -> valid)
  { return i
```

entry 마다 valid 체크, 0이면 리턴.

```
if(Default_cache_assoc == 1)
{ return 0 -> 1way면 0번 index return
else
{ int entry min_time index = 0
  for (i = 0; i < Default_cache_assoc; i++)
  { if (cache[cache_index][i] -> time <
        cache [cache_index][min_time index] -> time)
    { min_time index = i
      entry
```

entry

find minimum timestamp

timestamp 가 제일 작은것 ⇒ 가장 오래전에 사용한것.

```
return entry_index;
```