- 姓名：傅锦龙
- 班级：1621301
- 学号：162130117
- 报告阶段：lab3
- 完成日期：2023.6.12
- 本次实验，我完成了所有内容。

目录

init_cache （20分）

- cache结构设计
    - 思路

1. 首先定义cache行的结构体，包含tag ,data, valid_bit , dirty_bit
2. 定义整个cache的结构体，除了cache的实体外还包含组索引的长度，对应的掩码以及总行数
3. 还定义一些宏用来代表一些数据的掩码，如地址掩码，块索引掩码，块内偏移的掩码
    - 代码

```
#define addr_mask MEM_SIZE - 1
#define block_mask 0x1FF
#define offset_mask 0x3C

typedef struct {
    bool dirty_bit;
    uint8_t data[BLOCK_SIZE];
    uint32_t tag;
    bool valid_bit;
} CacheLine;

typedef struct {
    CacheLine *cache_line;
    uint32_t index_mask;
    uint32_t index_len;
    uint32_t line_num;
} CacheSet;

CacheSet cache;
```

- cache初始化

- 思路

函数接受两个参数 `total_size_width` 和 `associativity_width`，分别表示缓存的总大小和关联度的宽度。函数首先根据总大小和块宽度计算出缓存行数，然后根据总大小、块宽度和关联度宽度计算出索引长度，最后根据索引长度计算出索引掩码。接着，函数为缓存行分配内存，并将每个缓存行的有效位和脏位初始化为 false。

- 代码

```
void init_cache(int total_size_width, int associativity_width)
{
    cache.line_num = exp2(total_size_width - BLOCK_WIDTH);
    cache.index_len = total_size_width - BLOCK_WIDTH - associativity_width;
    cache.index_mask = exp2(cache.index_len) - 1;

    cache.cache_line = (CacheLine *)malloc(sizeof(CacheLine) * cache.line_num);
    for (int i = 0; i < cache.line_num; i++) {
        cache.cache_line[i].valid_bit = false;
        cache.cache_line[i].dirty_bit = false;
    }
}
```

cache_read (30分)

- 思路

`cache_read` 函数接受一个地址 `addr`，首先根据地址计算出索引、标记和偏移量，然后在缓存中查找是否存在该地址对应的数据。如果存在，则返回该数据；否则，如果有空闲的缓存行，则将数据从内存中读入该行，并返回该数据；否则，调用 `wirte_back` 函数将缓存行中的数据写回内存，并返回该行的索引。
`wirte_back` 函数接受一个索引 `index`、块号 `block` 和标记 `tag`，首先根据索引和块号计算出缓存行的索引，然后检查该行的脏位是否为 1，如果是，则将该行的数据写回内存。接着，从内存中读入块数据，并将标记和有效位写入该行，最后返回该行的索引。

- 代码

```
int wirte_back(uint32_t index, uint32_t block, uint32_t tag)
{
    int choose = rand() % 4 + index * 4;
    if (cache.cache_line[choose].dirty_bit == 1) {
        uint32_t addr = (cache.cache_line[choose].tag << cache.index_len) |
index;
        mem_write(addr, cache.cache_line[choose].data);
    }
    mem_read(block, cache.cache_line[choose].data);
    cache.cache_line[choose].tag = tag;
    cache.cache_line[choose].valid_bit = 1;
    return choose;
}

uint32_t cache_read(uintptr_t addr)
{
    try_increase(1);
    addr &= addr_mask;
    uint32_t index = (addr >> BLOCK_WIDTH) & cache.index_mask;
    uint32_t tag = addr >> (BLOCK_WIDTH + cache.index_len);
```

```
    uint32_t offset = addr & offset_mask;
    uint32_t block = (addr >> BLOCK_WIDTH) & block_mask;

    uint32_t index_start = index * 4;
    for (int i = index_start; i < index_start + 4; i++) {
        if (cache.cache_line[i].valid_bit == 1 && cache.cache_line[i].tag ==
tag) {
            hit_increase(1);
            return *(uint32_t *)(cache.cache_line[i].data + offset);
        }
    }

    for (int i = index_start; i < index_start + 4; i++) {
        if (cache.cache_line[i].valid_bit == 0) {
            cache.cache_line[i].valid_bit = 1;
            cache.cache_line[i].tag = tag;
            mem_read(block, cache.cache_line[i].data);
            return *(uint32_t *)(cache.cache_line[i].data + offset);
        }
    }

    int choose = wirte_back(index, block, tag);
    return *(uint32_t *)(cache.cache_line[choose].data + offset);
}
```

cache_write (30分)

- 思路

`cache_write` 函数接受一个地址 `addr`、一个数据 `data` 和一个写掩码 `wmask`，首先根据地址计算出索引、标记和偏移量，然后在缓存中查找是否存在该地址对应的数据。如果存在，则将数据写入该行，并将该行的脏位设置为 1；否则，如果有空闲的缓存行，则将数据从内存中读入该行，并将该行的标记、有效位和脏位设置为 1，然后将数据写入该行；否则，调用 `wirte_back` 函数将缓存行中的数据写回内存，并返回该行的索引。写入数据时，只写入写掩码指定的比特位。

- 代码

```
void cache_write(uintptr_t addr, uint32_t data, uint32_t wmask)
{
    try_increase(1);
    addr &= addr_mask;
    uint32_t index = (addr >> BLOCK_WIDTH) & cache.index_mask;
    uint32_t tag = addr >> (BLOCK_WIDTH + cache.index_len);
    uint32_t offset = addr & offset_mask;
    uint32_t block = (addr >> BLOCK_WIDTH) & block_mask;

    uint32_t index_start = index * 4;
    for (int i = index_start; i < index_start + 4; i++) {
        if (cache.cache_line[i].valid_bit == 1 && cache.cache_line[i].tag ==
tag) {
            hit_increase(1);
            cache.cache_line[i].dirty_bit = 1;
            uint32_t *data_cache = (uint32_t *)(cache.cache_line[i].data +
offset);
            *data_cache = (*data_cache & ~wmask) | (data & wmask);
            return;
```

```
        }
    }

    for (int i = index_start; i < index_start + 4; i++) {
        if (cache.cache_line[i].valid_bit == 0) {
            cache.cache_line[i].valid_bit = 1;
            cache.cache_line[i].tag = tag;
            mem_read(block, cache.cache_line[i].data);
            cache.cache_line[i].dirty_bit = 1;
            uint32_t *data_cache = (uint32_t *)(cache.cache_line[i].data +
offset);
            *data_cache = (*data_cache & ~wmask) | (data & wmask);
            return;
        }
    }

    int choose = wirte_back(index, block, tag);
    uint32_t *data_cache = (uint32_t *)(cache.cache_line[choose].data + offset);
    *data_cache = (*data_cache & ~wmask) | (data & wmask);
    cache.cache_line[choose].dirty_bit = 1;
}
```

最终结果截图（20分）

```
fujinlong@ubuntu:/mnt/hgfs/course/Lab3/cachesim-stu$ make
gcc -Wall -Werror -O2 -ggdb -o a.out main.c cpu.c cache.c mem.c
fujinlong@ubuntu:/mnt/hgfs/course/Lab3/cachesim-stu$ ./a.out
random seed = 1686566268
cached cycle = 16495150
uncached cycle = 16506384
cycle ratio = 99.93 %
total access = 1000000
cache hit = 499875
hit rate = 49.99 %
Random test pass!
fujinlong@ubuntu:/mnt/hgfs/course/Lab3/cachesim-stu$ make
gcc -Wall -Werror -O2 -ggdb -o a.out main.c cpu.c cache.c mem.c
fujinlong@ubuntu:/mnt/hgfs/course/Lab3/cachesim-stu$ ./a.out
random seed = 1686571262
cached cycle = 16488878
uncached cycle = 16505510
cycle ratio = 99.90 %
total access = 1000000
cache hit = 500062
hit rate = 50.01 %
Random test pass!
fujinlong@ubuntu:/mnt/hgfs/course/Lab3/cachesim-stu$
```

备注

助教真帅