南京航空航天大学《计算机组成原理II课程实验》报告

- 姓名：费振环
- 班级：1617301
- 学号：091701327
- 报告阶段：Lab3
- 完成日期：2019.6.29
- 本次实验，我完成了所有内容。

目录

开发环境

- 操作系统：macOS Mojave 10.14.5
- 编辑器：Visual Studio Code

思考题

1. 数据对齐和存储层次结构
   答：访问未对齐的内存时，处理器要访问两次（先读高位，再读低位），而访问对齐的内存，处理器只要访问一次。采用对齐是为了提高处理器读取数据的效率。

实验内容

cache结构

由于本题只是对cache的一个抽象模拟，所以cache用一个二维结构数组表示最恰当。经过计算，cache一共有2^8行，每2^2行一组，一共2^6组。

```
typedef struct
{
  bool is_valid;    // 有效位
  bool is_dirty;    // 脏位
  uint32_t tag;   // 标记
  uint32_t block_num; // 主存块号
  uint8_t block[64];  // 块
} cache_line;
cache_line cache[64][4]; // 根据计算，cache有2^6组，每组2^2行，一共2^8行
```

cache的抽象结构如下图

```
| 有效位(8位) | 脏位(8位) | 标记(32位) | 主存块号(32位) | 块(64位) |
```

这么一看，似乎cache结构浪费了非常多的空间，但是为了运算的方便（避免位操作）我觉得浪费一些空间是值得的。经过计算，主存的地址划分如下图

```
| 标记(3位) | cache组号(6位) | 块内地址(6位) |
```

init_cache()函数

初始化cache的函数是三个函数中最简单的了，没有需要多解释的

```
void init_cache(int total_size_width, int associativity_width)
{
  int group_num = ((1 << total_size_width) / (1 << BLOCK_WIDTH)) / (1
<< associativity_width);
  // 初始化cache数组
  for (int g = 0; g < group_num; g++)
  {
    for (int r = 0; r < (1 << associativity_width); r++)
    {
      cache[g][r].is_valid = 0;
      cache[g][r].is_dirty = 0;
      cache[g][r].tag = 0;
      cache[g][r].block_num = 0;
      memset(cache[g][r].block, 0, sizeof(cache[g][r].block));
    }
  }
}
```

cache_read()函数

根据main.c中random_trace()函数的提示，最终需要比较ret和ret_uncache是否相等来判断访问cache是否正确。读入的参数是64位无符号long型变量addr，当然，真正有效的只有低15位。具体怎样映射课上已经说得非常清楚了，但是本题有一个地方给我们设置了障碍，就是在mem_uncache_read()函数中的uint32_t *p = (void *)mem_diff + (addr & ~0x3);这一行代码，这行代码的表面意思是取addr的高13位（一共也就15位有效），但是深层含义是保证不会溢出，因为在讲义写的很清楚“从cache中读出addr地址处的4字节数据”，如果没有这行代码，可能发生溢出错误。那么在取块内地址的时候就要略微做一些修改（取(addr & 0x3f) & 0x3c）。最终返回的*p是一个无符号32位整形指针，它指向块内指定地址向后32位。具体的注释见代码

```
uint32_t cache_read(uintptr_t addr)
{
  try_increase(1);
  uint32_t *p = NULL;

  uint32_t group_num = (addr >> 6) % 64;
  bool is_hit = false;

  for (int i = 0; i < 4; i++)
  {
    if (cache[group_num][i].tag == (addr >> 12) && cache[group_num]
[i].is_valid == 1)
    {
      // 命中
      hit_increase(1);
      is_hit = true;
      p = (void *)cache[group_num][i].block + ((addr & 0x3f) & 0x3c);
      break;
    }
  }
  if (is_hit == false) // 未命中
  {
    uint8_t buf[64];
    bool is_full = true; // 假设cache的该组中没有空闲行
    for (int i = 0; i < 4; i++)
    {
      if (cache[group_num][i].is_valid == 0) // 找到空闲行
      {
        is_full = false;
        cache[group_num][i].is_valid = 1;
        cache[group_num][i].tag = addr >> 12;
        cache[group_num][i].block_num = (addr >> 6);
        mem_read(addr >> 6, buf);
        memcpy(cache[group_num][i].block, buf, BLOCK_SIZE);
        p = (void *)buf + ((addr & 0x3f) & 0x3c);
        break;
      }
    }
```

```
    if (is_full == true) // 没有空闲行
    {
      int ran_row = rand() % 4;
      if (cache[group_num][ran_row].is_dirty == 0) // 脏位为0，不需要把
cache内容写回到主存
      {
        cache[group_num][ran_row].is_valid = 1;
        cache[group_num][ran_row].tag = addr >> 12;
        cache[group_num][ran_row].block_num = (addr >> 6);
        mem_read(addr >> 6, buf);
        memcpy(cache[group_num][ran_row].block, buf, BLOCK_SIZE);
        p = (void *)buf + ((addr & 0x3f) & 0x3c);
      }
      else // 脏位为1，先把cache内容写回主存再进行替换
      {
        memcpy(buf, cache[group_num][ran_row].block, BLOCK_SIZE);
        mem_write(cache[group_num][ran_row].block_num, buf);

        cache[group_num][ran_row].is_valid = 1;
        cache[group_num][ran_row].is_dirty = 0;
        cache[group_num][ran_row].tag = addr >> 12;
        cache[group_num][ran_row].block_num = (addr >> 6);
        mem_read(addr >> 6, buf);
        memcpy(cache[group_num][ran_row].block, buf, BLOCK_SIZE);
        p = (void *)buf + ((addr & 0x3f) & 0x3c);
      }
    }
  }

  return *p;
}
```

cache_write()函数

能够完成cache_read()函数的话，cache_write()函数就不是非常难了，其中有很多代码有相似的地方，只是稍微复杂了一些些。cpu的写命令在cache一共会有3种情况：(1)命中，那么直接修改cache而不把修改后的内容写回主存，等到被替换是一次性写回（回写法）；(2)不命中，cache中有空闲行，先更新主存块内容，再写到cache中（写分配法）；(3)不命中，cache中没有空闲行，使用随机替换，先检查脏位，如果脏位为1，先把cache写回主存，如果脏位为0，直接修改cache。同样的，怎么修改cache要参考mem_uncache_write()函数的代码。具体的注释见代码

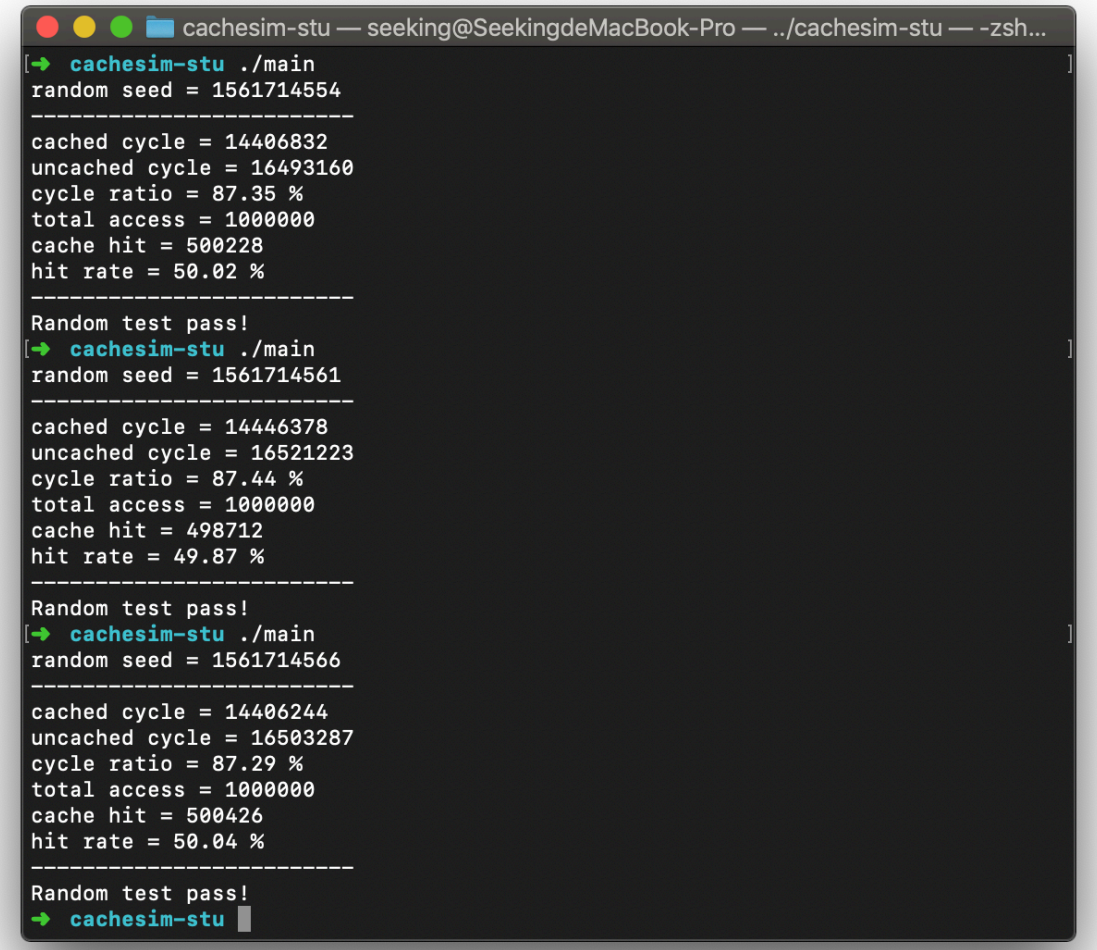
```
void cache_write(uintptr_t addr, uint32_t data, uint32_t wmask)
{
  try_increase(1);
  uint32_t group_num = (addr >> 6) % 64;
  bool is_hit = false;
```

```
  for (int i = 0; i < 4; i++)
  {
    if (cache[group_num][i].tag == (addr >> 12) && cache[group_num]
[i].is_valid == 1)
    {
      // 命中
      hit_increase(1);
      is_hit = true;
      // 直接修改cache
      uint32_t *p = (void *)cache[group_num][i].block + ((addr & 0x3f)
& 0x3c);
      *p = (*p & ~wmask) | (data & wmask);
      cache[group_num][i].is_dirty = 1; // 脏位变为1
      break;
    }
  }
  if (is_hit == false) // 没有命中
  {
    uint8_t buf[64];
    bool is_full = true; // 假设cache的该组中没有空闲行
    for (int i = 0; i < 4; i++)
    {
      if (cache[group_num][i].is_valid == 0) // 找到空闲行，先更新主存内
容，再更新cache内容
      {
        is_full = false;
        // 先更新主存内容
        mem_read(addr >> 6, buf);
        uint32_t *p = (void *)buf + ((addr & 0x3f) & 0x3c);
        *p = (*p & ~wmask) | (data & wmask);
        mem_write(addr >> 6, buf);
        // 再更新cache
        cache[group_num][i].is_valid = 1;
        cache[group_num][i].is_dirty = 0;
        cache[group_num][i].tag = addr >> 12;
        cache[group_num][i].block_num = addr >> 6;
        memcpy(cache[group_num][i].block, buf, 64);
        break;
      }
    }
    if (is_full == true)
    {
      int ran_row = rand() % 4;
      if (cache[group_num][ran_row].is_dirty == 0) // 脏位为0，不需要把
cache内容写回到内存
      {
        cache[group_num][ran_row].is_valid = 1;
        cache[group_num][ran_row].tag = addr >> 12;
        cache[group_num][ran_row].block_num = (addr >> 6);
```

```
            // 先更新主存内容
            mem_read(addr >> 6, buf);
            uint32_t *p = (void *)buf + ((addr & 0x3f) & 0x3c);
            *p = (*p & ~wmask) | (data & wmask);
            mem_write(addr >> 6, buf);
            // 再更新cache
            memcpy(cache[group_num][ran_row].block, buf, 64);
        }
        else // 脏位为1，需要先把cache内容写回到内存
        {
            // 先把cache内容写回到内存
            memcpy(buf, cache[group_num][ran_row].block, BLOCK_SIZE);
            mem_write(cache[group_num][ran_row].block_num, buf);

            cache[group_num][ran_row].is_valid = 1;
            cache[group_num][ran_row].is_dirty = 0;
            cache[group_num][ran_row].tag = addr >> 12;
            cache[group_num][ran_row].block_num = (addr >> 6);
            // 先更新主存内容
            mem_read(addr >> 6, buf);
            uint32_t *p = (void *)buf + ((addr & 0x3f) & 0x3c);
            *p = (*p & ~wmask) | (data & wmask);
            mem_write(addr >> 6, buf);
            // 再更新cache
            memcpy(cache[group_num][ran_row].block, buf, 64);
        }
    }
  }
}
```

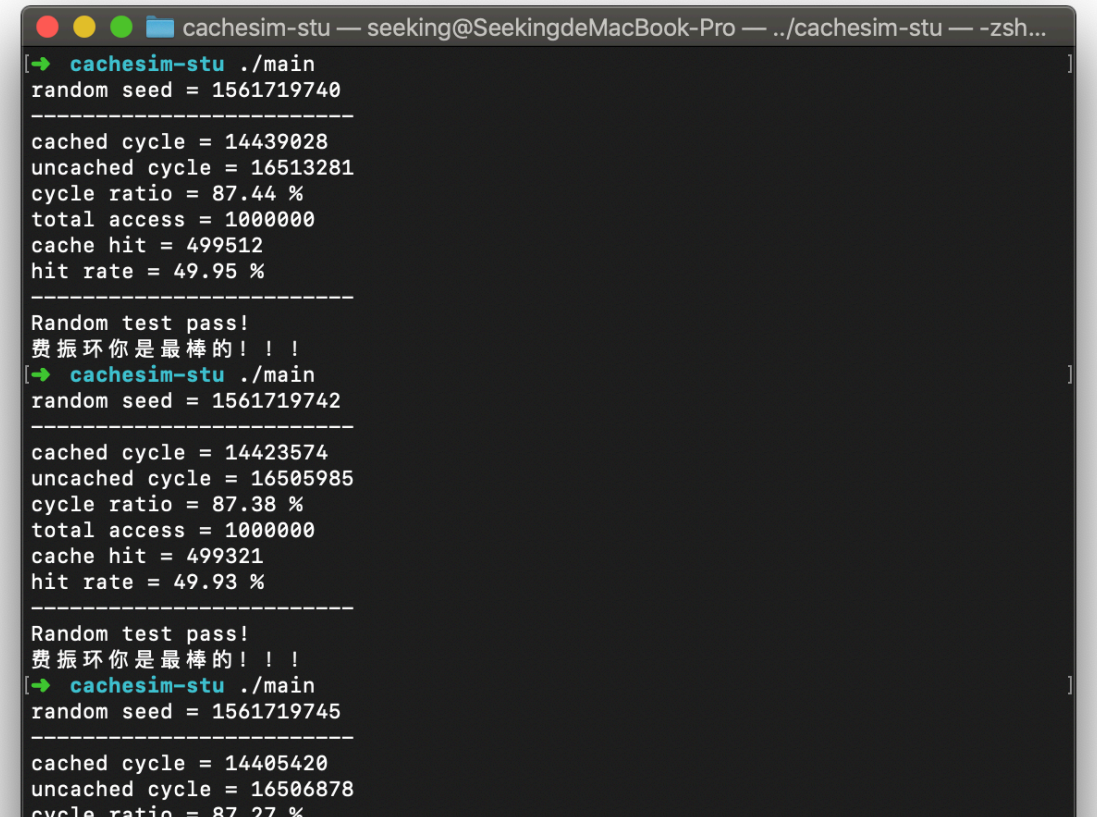
实验截图


cachesim-stu — seeking@SeekingdeMacBook-Pro — ../cachesim-stu — -zsh...
➜ cachesim-stu ./main
random seed = 1561714554
----
cached cycle = 14406832
uncached cycle = 16493160
cycle ratio = 87.35 %
total access = 1000000
cache hit = 500228
hit rate = 50.02 %

Random test pass!
➜ cachesim-stu ./main
random seed = 1561714561
----
cached cycle = 14446378
uncached cycle = 16521223
cycle ratio = 87.44 %
total access = 1000000
cache hit = 498712
hit rate = 49.87 %

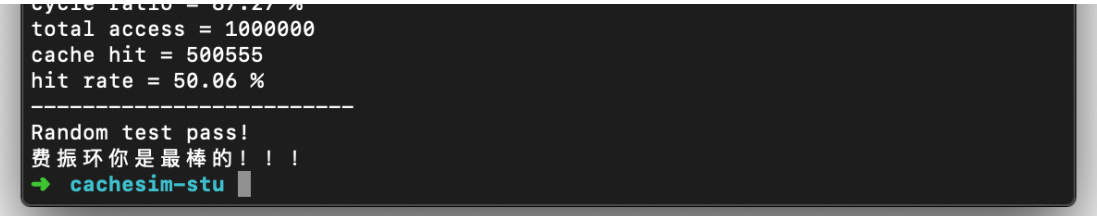
Random test pass!
➜ cachesim-stu ./main
random seed = 1561714566
----
cached cycle = 14406244
uncached cycle = 16503287
cycle ratio = 87.29 %
total access = 1000000
cache hit = 500426
hit rate = 50.04 %

Random test pass!
➜ cachesim-stu


cachesim-stu — seeking@SeekingdeMacBook-Pro — ../cachesim-stu — -zsh...
➜ cachesim-stu ./main
random seed = 1561719740
----
cached cycle = 14439028
uncached cycle = 16513281
cycle ratio = 87.44 %
total access = 1000000
cache hit = 499512
hit rate = 49.95 %

Random test pass!
费振环你是最棒的！！！
➜ cachesim-stu ./main
random seed = 1561719742
----
cached cycle = 14423574
uncached cycle = 16505985
cycle ratio = 87.38 %
total access = 1000000
cache hit = 499321
hit rate = 49.93 %

Random test pass!
费振环你是最棒的！！！
➜ cachesim-stu ./main
random seed = 1561719745
----
cached cycle = 14405420
uncached cycle = 16506878



可以发现命中率稳定在50%左右，好像不高，可能是代码优化不够完善吧?

遇到的问题及解决办法

1. 框架代码中有部分宏定义（如`#define addr_offset_bit(addr) (((addr)&0x3) * 8)`）看不懂
   解决方法：经过仔细研究，其实这些代码无关紧要，看不看的懂对我写代码没有影响。这些宏定义的作用应该是保证我的结果不会发生溢出。

实验心得

说实话，Lab3的时间有点紧了。我花了大约两天时间看懂了框架代码，然后就是连续接近十个小时的代码编写代码加测试才终于完成了这个任务。这是本学期三个实验中代码量最大的一个实验，当然，也是难度最大的实验（包括网上能够参考的资料非常少也是导致此实验较难的一个重要原因）。但是，我还是把它给攻克下来了，虽然消耗了我大量精力和时间。测试通过的那一瞬间我感觉非常轻松也非常有成就感，这才是真正完全靠自己的理解和能力完成的实验啊！接下来要学的东西还非常多，我要再接再厉，加油！！

其他备注

不知道在助教大大的电脑能不能通过测试，有点慌...