Part A

本部分旨在设计一个模拟简化版cache的程序，即不用关心Block中的内容，只需要关注是否命中。

因此我设计了一个block结构体：

```
typedef struct {
    int valid;
    int tag;
    int time_stamp;
} cache_block;
```

来表示一个block，其中valid是该block是否被使用，time_stamp是命中的时间戳，tag即cache的tag。

由于替换时采取LRU策略，我使用时间戳来表示该block距离上次被访问的时间。具体而言，时间戳最大的block会被替换。一个block在被替换或刚被使用或命中时block被设置为0。在一次访问后，所有block的时间戳都要增加。

cache_block组成了一个S*E的二维数组cache。该数组根据接受的S和E而开辟。在每次访问中，我们先对地址进行处理得到对应的tag和s。然后对cache[s]进行遍历。

- 先搜寻有无valid为1且tag相符的，有则命中数增加并返回。无则继续遍历。
- 再搜索有无空余的block（即valid为0），有则不命中数增加并修改该block为当前地址，然后返回。无则继续遍历。
- 最后搜索时间戳最大的，依据LRU被替换。此时不命中数和替换数增加，被查找到的block被替换。

以上便是大致设计

Part B

本题s = 5, E = 1, b = 5。即cache有32行，每行存32/4 = 8个int类数字。

- **32 X 32**
    - 由于一行能放8个int，拿我们就一次定义8个本地变量来存取A和B中的数组，这样能充分利用了cache一次取8个int的特性，使得cache中基本只有冷命中带来的miss，代码如下。
    - ```
      for (int i = 0; i < 32; i += 8)
          for(int j = 0; j < 32; j += 8)
              for(int k = i; k < (i + 8); ++k)
              {
                  a1 = A[k][j];
                  a2 = A[k][j+1];
                  a3 = A[k][j+2];
                  a4 = A[k][j+3];
                  a5 = A[k][j+4];
                  a6 = A[k][j+5];
                  a7 = A[k][j+6];
                  a8 = A[k][j+7];
                  B[j][k] = a1;
                  B[j+1][k] = a2;
                  B[j+2][k] = a3;
                  B[j+3][k] = a4;
      ```
      ```

```
                B[j+4][k] = a5;
                B[j+5][k] = a6;
                B[j+6][k] = a7;
                B[j+7][k] = a8;
            }
```

    - 如上操作能使所有的取A操作只有冷命中带来的miss，即128次。而所有的存B操作也只有冷命中和对角线上元素被覆盖造成的miss，即128+32次，总miss不超过300次。

- **64 X 64**
    - 由于此时每4行就能填满一个cache，故原本的直接取8个一取的策略并不能很好地利用已取到cache里的内容。
    - 仿照32X32，将64X64的矩阵分割成4个32X32的小部分，记为左上，右上，左下，右下。通过以下顺序来赋值
        - 先把A的左上右上转置地赋给B的左上右上
        - 把B的右上之间赋给B的左下，把A的左下转置地赋给B的右上
        - 把A的左下转置地赋给B的左下
    - 上述操作中左上和右下的操作其实和32X32一致。右上和左下的交换则避开了cache的多次覆盖，最大限度地减小了多次加载带来的miss
- 61X67
    - 这题没啥规律，书上有一道类似的题目使用了一种“扰动”的方法。具体而言就是每次取的数目是8的倍数加上1，2。所以我们尝试一下一次取多少miss最少。经过尝试后，一次取一个17X17的方阵时miss最少。