Homework #2. Cache Simulator

이번 과제에서는 캐시 시뮬레이터를 구현합니다. 시뮬레이터는 다음과 같은 특성을 갖고 있습니다.

- **I-cache, D-cache 즉, instruction cache 와 data cache 로 구분됨.**
- **캐시 크기: 1024, 2048, 4096, 8192, 16384 bytes**
- **블록 크기: 16, 64 bytes**
- **연관성(Associativities): 1-way (Direct-mapped), 2-way, 4-way, 8-way**
- **교체 알고리즘 (Replacement Policies): LRU (Least Recently Used)**
- **쓰기 및 할당 정책 (Write and Allocate Policy): Write Back & Allocate for D-cache**

캐시 시뮬레이터는 캐시 크기, 블록 크기, 연관성 및 교체 정책의 각 조합에 대한 **캐시 미스** (Cache Miss)를 추적해야 합니다. 해당 캐시 시뮬레이터의 집합(set)의 개수는 연관성, 블록 크기 및 총 캐시 크기를 기반으로 결정됩니다.
예를 들어, 연관성이 2-way, 블록 크기는 16Byte, 총 캐시 크기는 8KB 이라면 set 의 개수는 256 입니다. ($256\ sets \times 2 - way \times 16Byte = 8KB$)

프로그램에 주어지는 입력은 수행 모드 (mode)와 주소 값들입니다. 여기서 모드는 주어진 주소에 대한 읽기, 쓰기와 같은 작업입니다.
여러분들이 구현해야 할 캐시 시뮬레이터의 동작은 다음과 같습니다.

1. 주소 값을 받으면, 해당 주소가 캐시에 존재하는 지 (즉, Cache Hit) 확인해야 합니다. 주소가 캐시 내에 없으면 캐시 미스 (Cache Miss)의 값을 증가 시킵니다.
2. 교체정책 (즉, LRU)에 따라 캐시를 업데이트 해야 합니다. 교체 작업으로 인해 더티 블록 (Dirty Block)이 제거되는 경우 메모리 쓰기 횟수도 증가시켜야 합니다. 여기서 주소의 크기는 32bit (4Byte) 라고 가정합니다.

테스트로 주어진 입력은 다음과 같습니다. 해당 파일은 캐시 동작과 주소 값을 추출한 트레이스 파일입니다. 주소는 16 진수 형태로 표현되고, 첫 숫자는 캐시 동작 모드를 나타냅니다. 0 은 데이터 읽기, 1 은 데이터 쓰기, 2 는 명령어 가져오기 (Fetch) 입니다.

아래 트레이스파일 예제에서는 "0x403664" 주소에 있는 명령어를 가져오는 것이고, "7ffd8ce0" 주소에 있는 데이터를 읽기, "0x404bc8" 메모리 주소에 데이터 쓰기를 수행합니다.

```
2 403664
0 7ffd8ce0
1 404bc8
```

1. 코드 구현

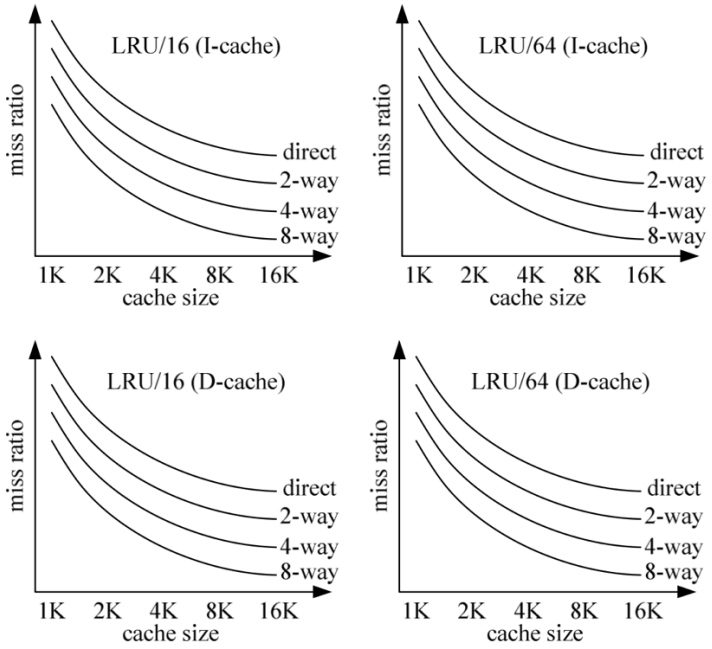
1-1. 아래 스켈레톤 코드와 주석 내용을 참고하여 캐시 시뮬레이터를 완성하세요. 코드는 **구름에 제출**하시면 됩니다.

1-2. E-class 에서 과제파일을 다운로드 받은 후,
코드를 "참고. 동작방법" 을 참고하여 실행 시킨 후, "2.보고서 제출"을 참고하여 보고서를 작성하면 됩니다.

2. 보고서 제출

보고서는 다음과 같은 내용이 들어가면 됩니다.
단, 길게 작성하지 않아도 됩니다. 핵심 내용만 서술하면 됩니다. 코드는 예시와 함께 설명하면 됩니다.

2.1. 본인이 작성한 캐시 시뮬레이터 코드 구현 방식에 대해 서술하시오.
2.2. trace1.txt 와 trace2.txt 에 대해 각각 출력 결과를 표로 나타내시오.
(표 형식은 자유입니다.)
2.3., 1-way (direct-mapped), 2-way, 4-way, 8-way 에 대해 "캐시 사이즈"에 따른 "Miss rate"를 블록 크기 (16, 64 Byte)에 대해 그래프로 나타내고 결과에 대한 이유를 간략하게 작성하시오.
- 즉, X 축은 캐시 크기, Y 축은 Miss rate
- 총 아래와 같이 4 개의 그래프를 표현하면 됩니다.



참고. 동작 방법

```
gcc hw2.c -o cache
./cache < trace1.txt &> trace1-out.txt
```

참고. 힌트

1. 스켈레톤 코드로 주어진 함수 외에도 추가로 구현해야 하는 함수들이 존재합니다. (예: LRU 구현)

2. LRU 구현을 위해 캐시에 해당 메모리 접근하는 기록을 어떻게 남기면 좋을 지 고민해보시기 바랍니다.

<스켈레톤 코드 설명>

```
// 제약조건
// 1. input 으로 주어지는 파일의 한줄은 100 자를 넘지 않음.
// 2. input 으로 주어지는 파일의 길이는 100000 줄을 넘지 않음. 즉, address trace 길이는
100000 줄을 넘지 않음.
#define MAX_ROW     100000
#define MAX_COL     2
#define MAX_INPUT   100
```

해당 과제의 제약조건은 다음과 같습니다.

```
// 구현해야하는 함수
void solution(int cache_size, int block_size, int assoc);
void read_op(int addr, int cache_size, int block_size, int assoc);
void write_op(int addr, int cache_size, int block_size, int assoc);
void fetch_inst(int addr, int cache_size, int block_size, int assoc);
```

해당 과제에서 필수적으로 구현해야 하는 함수입니다.

solution 함수의 경우, 각각 캐시 크기, 블록 크기, 그리고 연관 수를 매개변수로 가집니다.

read_op, write_op, fetch_inst 함수는 각각 메모리 주소, 캐시 크기, 블록 크기, 연관수를 매개변수로 가집니다.

```
// 전역변수
// 문제를 풀기 위한 힌트로써 제공된 것이며, 마음대로 변환 가능합니다.
enum COLS {
   MODE,
   ADDR
};

int i_total, i_miss;          /* instructino cache 총 접근 횟수, miss 횟수*/
int d_total, d_miss, d_write;   /* data cache 접근 횟수 및 miss 횟수, memory write 횟수
*/
int trace[MAX_ROW][MAX_COL] = {{0,0},};
int trace_length = 0;
```

Instruction cache 와 data cache 총 접근 횟수, miss 횟수, 그리고 data cache 에 대해서는 memory write 횟수도 저장합니다.

입력으로 주어진 메모리 트레이스는 이차원 배열 trace 에 저장합니다. 첫번째 열은 메모리 주소 동작 Mode (읽기, 쓰기, 인스트럭션 가져오기)를 저장하고 두번째 열은 메모리 주소 값을 저장합니다.

trace_length 는 입력 trace 갯수입니다.

```
int main(){
```

```
    // DO NOT MODIFY -- START --  //
    // cache size
    int cache[5] = {1024, 2048, 4096, 8192, 16384};
    // block size
    int block[2] = {16, 64};
    // associatvity e.g., 1-way, 2-way, ... , 8-way
    int associative[4] = {1, 2, 4, 8};
    int i=0,j=0,k=0;

    /* 입력 받아오기 */
    char input[MAX_INPUT];
    while (fgets(input, sizeof(input), stdin)) {
        if(sscanf(input, "%d %x\n", &trace[trace_length][MODE],
&trace[trace_length][ADDR]) != 2) {
            fprintf(stderr, "error!\n");
        }
        trace_length++;
    }


    /* 캐시 시뮬레이션 */
    printf("cache size || block size || associative || d-miss rate || i-miss rate
|| mem write\n");
    for(i=0; i<5; i++){
        for(j=0; j<2; j++){
            for(k=0; k<4; k++){
                solution(cache[i], block[j], associative[k]);
            }
        }
    }
    // DO NOT MODIFY -- END --  //
    return 0;
}
```

main 함수는 입력으로 주어지는 trace 파일을 2 차원 배열에 저장하는 함수이고,
전체 캐시 시뮬레이션을 수행하는 main 함수입니다.
// DO NOT MODIFY -- START -- // 와 // DO NOT MODIFY -- END -- //
사이는 절대 수정하지 마세요.

```c
void solution(int cache_size, int block_size, int assoc) {

   // DO NOT MODIFY -- START --  //
   int mode, addr;
   double i_miss_rate, d_miss_rate;    /* miss rate을 저장하는 변수 */

   int index = 0;
   while(index != trace_length) {
      mode = trace[index][MODE];
      addr = trace[index][ADDR];

      switch(mode) {
         case 0 :
            read_op(addr, cache_size, block_size, assoc);
            d_total++;
            break;
         case 1 :
            write_op(addr, cache_size, block_size, assoc);
            d_total++;
            break;
         case 2 :
            fetch_inst(addr, cache_size, block_size, assoc);
            i_total++;
            break;
      }
      index++;
   }
   // DO NOT MODIFY -- END --  //

   // hint. data cache miss rate 와 intruction cache miss rate를 계산하시오.
   // ? 에는 알맞는 변수를 넣으면 됩니다.
   /*
      i_miss_rate = ? / ?
      d_miss_rate = ? / ?
   */

   // DO NOT MODIFY -- START --  //
   printf("%8d\t%8d\t%8d\t%.4lf\t%.4lf\t%8d\n", cache_size, block_size, assoc,
d_miss_rate, i_miss_rate, d_write);
   // DO NOT MODIFY -- END --  //
}
```

입력이 저장된 trace 2 차원 배열을 하나씩 읽으면서 캐시 동작을 처리하면 됩니다.

`// DO NOT MODIFY -- START --  //` 와 `// DO NOT MODIFY -- END --  //` 사이는 절대 수정하지 마세요.