Machine Organization and Assembly Language Programming

Problem Set #9

Due: Thursday, March 11

In this assignment you will write in C or C++ the skeleton of a trace-driven simulator for assessing the performance of a cache.

Trace-driven simulation is a widely used technique to assess the performance of the components of the memory hierarchy. In broad terms, the simulator works as follows:

Input

- A trace, i.e., a string of memory references.
- One or more cache descriptions (I-cache, D-cache, cache hierarchy) with size, associativity, block size, and replacement algorithm.
- A write policy.
- Access times of the various components of the memory hierarchy.

Output

- A set of statistics, e.g. hit ratio, or more precisely read hits, write hits, average memory access time, etc.
- Cycles spent waiting at the various levels of the memory hierarchy, etc.

Algorithm

- Process each memory reference in turn. Decompose the address into (tag, index, displacement) components.
- Check if the memory reference hits in the appropriate cache (note that you don't bring data into the simulated cache, you only simulate the presence or absence of particular blocks).
- Take appropriate actions. In case of a hit, record statistics, maybe turn on some valid/dirty bits, etc. In case of a miss, bring in the missing block, replace an old one, record statistics, etc.

Your assignment is to write a D-cache simulator in C or C++. It will support a variety of cache sizes, block sizes, organizations, and write policies, and must conform to the following specifications.

Some code will be given to you to assist in parsing the arguments to the program and in reading trace files. Your program should probably look something like this:

```
#include "trace.h"

main( int argc, char** argv )
{
    CacheInfo ci;
    TraceCode tc;
    unsigned int address;
    int count;
    int read_hits, read_misses;
    int write_hits, write_misses;
    int bytes_written_back;

    parse_args( &ci, argc, argv );

    while ( ( tc = get_next_reference( &address, &count ) ) != TraceEOF )
    {
        switch( tc )
        {
            case TraceRead:
                /* process a read reference */
                break;

            case TraceWrite:
                /* process a write reference */
                break;

            default:
                /* an error in the trace file, print a message and abort. */
                break;
        }
    }

    show_statistics( read_hits, read_misses, write_hits, write_misses,
                     bytes_written_back );
}
```

`parse_args`, `get_next_reference`, and `show_statistics` will be provided in the `trace.c` and `trace.h` files available on the course web "Software" page.

- `parse_args` takes a pointer to a CacheInfo structure, as well as the `argc` and `argv` arguments passed to main. It will parse the program arguments and fill in the fields of the CacheInfo structure, defined like this:

```
typedef struct
{
    int cachesize;
    enum { DirectMapped, TwoWayLRU } organization;
    int blocksize;
    enum { WriteThrough, WriteBack } policy;
} CacheInfo;
```

  `cachesize` is the total size of the cache data in bytes. It will be a power of 2 between 4 and 1024, inclusive. `blocksize` is the sizes, in bytes, of a single cache block. It will be a power of 2 between 4 and `cachesize`, inclusive.

  If `policy` is set to `WriteThrough`, you should simulate writing both the cache and the memory on a write hit, and only the memory on a write miss (a *write-around* policy). If `policy` is `WriteBack`, write hits should write only the cache, while write misses should allocate a block in the cache for writing to (a *write-allocate* policy).

  If the cache is 2-way, then each time you place a block in the cache you have 2 possibilities for where to put it. You should decide by looking at the two blocks already in those locations in the cache, checking things in this order:

  1. Replace a block that is invalid, if there is one. Otherwise,
  2. (write-back policy only) replace a block that is not dirty, if there is one. Otherwise,
  3. replace the block that was used least recently. A block is "used" when it when either a read or write reference touches it.

- `get_next_reference` fetches the next reference from the input trace file. It places the address referenced in the first pointer passed to it, and the count of references read so far in the second (this could be used to determine which reference is least recently used). The function returns `TraceRead` if it is a read reference, `TraceWrite` if it is a write reference, and `TraceEOF` if there are no more references in the trace. Any other return code indicates a syntax error in the trace file.

- `show_statistics` takes some statistics you must collect and prints them out. You should pass it

  – the number of reads which hit in the cache

  – the number of reads which missed in the cache

- the number of writes which hit in the cache
- the number of writes which missed in the cache
- the total number of bytes written to memory. Assume that all write instructions are word-sized, and that a write-through policy only sends the word written to memory. Your simulator should have only a single dirty bit per cache block.

Running the program

Your program will be invoked with the following arguments:

progname cachesize organization blocksize policy tracefile

where *cachesize* is a number in bytes, *organization* is either "dm" or "2way", *blocksize* is a number in bytes, *policy* is either "wt" or "wb", and *tracefile* is the name of a file containing a memory trace. The `parse_args` function will parse these arguments into the CacheInfo structure for you. If an illegal combination of arguments is given, `parse_args` will not return – you do not need to handle error cases.

A sample command line might be:

```
% ./sim 256 dm 4 wt trace.txt
```

Your program should then simulate a 256 byte, direct-mapped, write-through cache with 4 byte cache blocks, running through the memory references in "trace.txt".

Trace files

Sample trace files will be provided to you on the course web page, but you will need to also produce your own for testing purposes. Trace files have a simple, one line per reference format:

code	address
⋮	⋮
code	address
0	0

where each "address" is a 32-bit address **in hexadecimal** and each "code" is 1 to indicate a read and 2 to indicate a write. The trace is terminated by a line with code and address both zero.