**Cache Simulator**

You are to write a cache memory simulator. Your cache simulator must be able to:

1. Handle any cache data size.
2. Handle any block blocks sizes which is a power of 2.
3. Be able to simulate direct mapped, or N–way set–associative caches (N is a power of two)
4. Be able to simulate either write back or copy–back caches.
5. Be able to handle either an allocate or non–allocate policy on write miss.
6. Implement Least–recently–used block replacement .
7. Accept a parameter for memory access time (in clocks) and compute average access time, hit/miss ratios for both read and write operations.
8. Be able to specify a victim buffer of K blocks (victim buffer is fully associative)

You can write in any computer language. You cannot work in groups. A sample of what your cache program should output is shown below:

% cache –s 1 –b 4 –W –c –a 1 –m 3 < sample.din

*size (kbytes) = 1*
*# of lines = 256*
*bytes/line = 4*
*associativity = 1*
*write policy = Copyback*
*write allocate = Yes*
*mem penalty = 3*

---

*accesses = 1000 [reads = 704 ( 70.40%) writes = 296 ( 29.60%)]*
*hits = 765 ( 76.50%) [reads = 559 ( 55.90%) writes = 206 ( 20.60%)]*
*misses = 235 ( 23.50%) [reads = 145 ( 14.50%) writes = 90 ( 9.00%)]*
*access time = 1.5270*
Program options are:
*cache {options}:*
*–s size cache size in K bytes (power of two)*
*–b size cache size in bytes (power of 2), assume each access is word*
*–a num cache associativity, 1 is direct mapped*
*–c copyback, normally write through*
*–W write allocate, normally not write allocate*
*–m num memory access time in clock cycles*
*-v number of victim buffer blocks*
*Cache access time is assumed to be 1 clock cycle*

The associated ZIP file with this project contains a 'cache' binary you can use for sample runs; this is my solution to the project. There some cache trace files ('*.din file extension') that you can use for test purposes. The format of the trace data files is show below:.

*2 20d* Dinero input format ”din” is an ASCII file with
*2 211* one LABEL and one ADDRESS per line. The rest of
*0 1fc780* the line is ignored so that it can be used for
*1 7fffccb0* comments.
*2 213*
*2 217* LABEL = 0 read data
*0 1fc77c* 1 write data
*1 7fffccac* 2 instruction fetch
*2 219* 3 escape record (treated as unknown access type)
*2 21d* 4 escape record (causes cache flush)
*0 1fc778*
*1 7fffcca8* 0 <= ADDRESS <= ffffffff where the hexadecimal addresses
*2 21f* are NOT preceded by ”0x.”
*2 223*

The important thing to note about this data input file is that each line specifies a memory access via address information and whether the access was read or write. It DOES NOT contain the data for that memory access. The data bytes actually associated with the access are not needed when determining hit/miss information for the cache; we just need to know which LOCATION (address) is being accessed.

When simulating your cache, you will need to keep an integer array *(cache_arra*y) whose size is equal to the number of blocks in the cache. This array will be used to store the tag value currently associated with that block. The size of *cache_array* is the number of blocks in the cache which is computed as:

# of blocks = (cache_size * 1024)/(block_size).

where *cache_size* is the size of the cache in Kbytes. You can think of this linear array being grouped into *line*s, where each line contains ’associativity’ number of blocks. The number of lines is computed as:

#lines = # of blocks / associativity

Note that if associativity = 1, then #lines = # of blocks and we have a direct mapped cache. Each entry in *cache_array* will contain the *tag* value for that block. Given an address, we need to compute the *index* (or line number) and the tag value which corresponds to that address and check this tag against the tag values stored in *cache_array* at that line number. The index and tag value can be computed from the address via:

index = (address / (block_size)) modulo #lines
tag = address / (block_size * # lines)

*cache_array* entries

Let number of Blocks = N

Let number of Lines = L

Direct Mapped (associativity =1)

# of lines = # of blocks

| Line # | |
|---|---|
| 0 | 0: tag value |
| 1 | 1: tag value |
| 2 | 2: tag value |
| 3 | 3: tag value |
| | |
| | |
| | |
| | |
| L-3 | N-3: tag value |
| L-2 | N-2: tag value |
| L-1 | N-1: tag value |

For 4-way set associative, # of lines = # of blocks/4

| Line # | | | | |
|---|---|---|---|---|
| 0 | 0: tag value | 1: tag value | 2: tag value | 3: tag value |
| 1 | 4: tag value | 5: tag value | 6: tag value | 7: tag value |
| 2 | 8: tag value | 9: tag value | 10: tag value | 11: tag value |
| 3 | | | | |
| | | | | |
| | | | | |
| | | | | |
| L-2 | | | | |
| L-1 | N-4: tag value | N-3: tag value | N-2: tag value | N-1: tag value |

Initially, each cache array value should initialized to a '–1' value indicating that this cache block is empty. To check for a hit given an address you should:

1. compute the *tag* corresponding to that address
2. compute the *index* (line number) corresponding to that address
3. Starting at array value = index * associativity, check the next 'associativity' entries to see if the tag stored in the array is equal to the tag computed from the address. If a match is found this indicates a hit, else it is a miss.

If a cache miss is found and associativity > 1, then you must choose a particular block to replace. If one of the blocks is 'empty' (tag = –1), then this block should be chosen. If all blocks in a line have valid tags, then you must choose a block using Least–Recently–Used replacement. The easiest way to do this is to keep a separate integer array ('lru_count') whose size is equal to the number of blocks in the cache. Initially, these values should be '0'. When checking all of the blocks in a line for a hit, if the block tag value does NOT match the address tag value then the associated 'lru_count' entry should be incremented by one (this is a miss). If the block tag value DOES match the address tag value, then the associated 'lru_count' entry should be set back to zero (a hit, a zero value indicates this block is the most–recently–used block in this cache line). When picking a block in a particular line to replace, choose the block with the highest 'lru_count' value – this will be the 'oldest' or least–recently–used block in that cache line.

You must also keep a separate array which tracks dirty bit values for each block in the cache – this will be used when you are simulating a copyback policy.

The easiest way to compute average access time is to keep two counters – one for total number of accesses and one for total number of clocks. Increment the accesses counter for each address processed. Increment the clock counter by 1 for each word (4 bytes) read/written to the cache; increment the clock counter by the memory penalty for each word read/written to main memory. Divide the total number of accesses into the clock counter for average word access time.

Your program output must match the output of my 'cache' program fairly closely; it must match the 'hit/miss' information exactly for the direct mapped case (associativity = 1) and for associativity > 1 when using LRU replacment. You do not have to match my access time information exactly but it should be close.

There is another cache simulator program in the 'project' directory called *dinero*. Read the *dinero. cat* file for information on how to run the *dinero* program. You should be able to match the hit/miss information produced by dinero as well. I am having trouble matching the read miss information produced by dinero when a copyback/write–allocate policy is specified – I will be interested to see what you are able to do. The following are equivalent runs for my 'cache' program and 'dinero':

```
% cache –b 16 –s 16 –a 8 –c –W < tex.din
```

```
% dinero –b16 –u16K –a8 –wc –Aw < tex.din
```

Both programs simulate a 16 Kbyte cache, block size = 16, 8–way set associative, copyback with write allocate and use the ’tex.din’ file as the input trace file