**CS546 Spring 2018**

**Laboratory Assignment 4**

This is the fourth lab assignment for this course. It is due no later than Tuesday, May 22, 2018 at the start of the lecture period (8:00 PM). You will submit it by uploading your .cpp source file to Canvas. There will be a 20 percent penalty for late assignments.

The naming convention specified for this assignment is to use your initials (three letters please), the word Lab, and the digit 4 for this assignment. Use 'X' if you do not have a middle name or last name. Thus, if W.C.Fields were taking this class and turning in this lab, he would provide a file named WCFLab4.cpp. If Madonna were taking this class, her lab would be MxxLab4.cpp.

Only turn in the program source file, i.e. the C++ source.

Your program should start with the following heading:

**// Lastname, Firstname CS546 Section 3122 Date completed**

**// Fourth Laboratory Assignment – Cache Simulation**

**The Problem**

For this assignment, you will write a C++ program to simulate a Level 1 write-back cache. You do not need to write any part of this assignment in assembler language. You may assume that the memory being simulated uses 32-bit words and byte addressing, just like the Intel x86 family.

**Input**

The input will be a series of parameters and test data within a file. The input will be in character format. The input data will comprise:

1. Line size of the cache.
2. Total size of the cache.
3. Associativity of the cache.
4. Total size of main memory.
5. Address, type of access, and data for simulated memory accesses.
6. Address of data to be displayed.

There are only one instance of items a through d. Items e and f will repeat multiple times. All input data will comprise a character code (A-F) and one data value (or three for item e).

For example,

**E 4004 W 5657**

Means Write the value 5657 to location 4004. Also,

**F 4004**

Means that the contents of the cache and main memory for location 4004 should be displayed.

**Processing**

Read in the four primary variables (items a through d above). Validate them with the following rules:

1. All values must be greater than zero
2. All values must be multiples of a power of 2. Thus, 2, 4, 8, 16, 32, 64, and so forth are valid values; odd numbers such as 1, 13, or 111 are not valid, and even numbers which are not a multiple of a power of 2 are invalid, such as 6, 22, or 46. Thus, 384 is valid (multiple of 2 to the seventh power, or 128), while 386 is not. However, for this assignment, you only need to check that the values are even numbers.
3. Line size (a) may not exceed ten percent of the cache size.
4. Cache size may not exceed ten percent of the total size of main memory.

The program should issue an error message and end if any of these tests fail, or if any of parameters a through d are omitted.

Once parameters a through d have been validated, initialize the simulated memory system. Acquire memory for the cache and for the memory. You also may find it useful to establish a separate array for an associative memory. (These should all be separate variables.) Set all memory to zeroes, minus 1, or some other identifiable value. Then begin a processing loop.

Read the input data and process all memory accesses. For write accesses, store the value in the cache. For read accesses, get the data from the cache if present, or from main memory if not, and move the data into the cache in that case. If a write access references a cache line which already has data, copy that data from the cache to the simulated main memory before storing the incoming value into the simulated cache.

For item f, display the address, value in the simulated cache for that address, and data in simulated main memory for that address.

**Output**

Your program should produce the following lines of output as appropriate:

Output Line 1:

Invalid value *n* for *item*

Replace item with the name (line size, cache size, et cetera) and n with the value from the file.

Output Line 2:

*Item n* too large

Replace *item* with cache size or line size, and *n* with the value from the file.

Output Line 3:

Input address xxxxxxxx invalid

Replace xxxxxxxx with the address field from the memory access line (input type E)

Output Line 4:

Address: xxxxxxxx memory:nnnnnnnn cache:mmmmmmmm

Replace xxxxxxxx with the address field from the memory data display field (input type F), nnnnnnnn with the simulated memory contents for that address, and mmmmmmmm with the simulated cache contents for that address. If the cache line is not valid, display -1 for that value.

**Sample Input File:**

A 16

B 2048

C 2

D 32768

E 0 W 22222222

E 8 W 1234

E 32 W 1233

E 16 W 888

F 0

E 4 W 33333333

F 8

E 1028 W 3231

E 16388 W 5657

F 0

F 4

F 8

F 12

F 16

F 1024

F 1028

F 16384

F 16388

**Sample Output:**

To be provided

**Grading Standards**

Properly accept parameters A through D and set up memory accordingly – 4 points

Display output messages correctly in specified format – 1 point

Properly write to memory and to cache (E) – 2 points

Properly display memory and cache (F) – 2 points

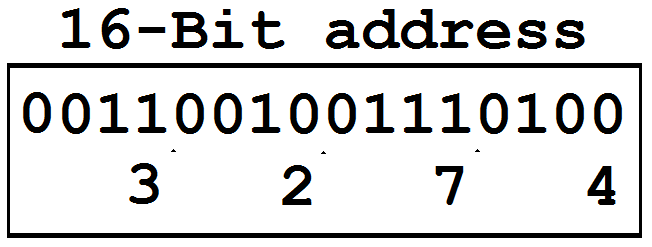
Successful compile of properly named file – 1 point

**Notes On Address Processing**

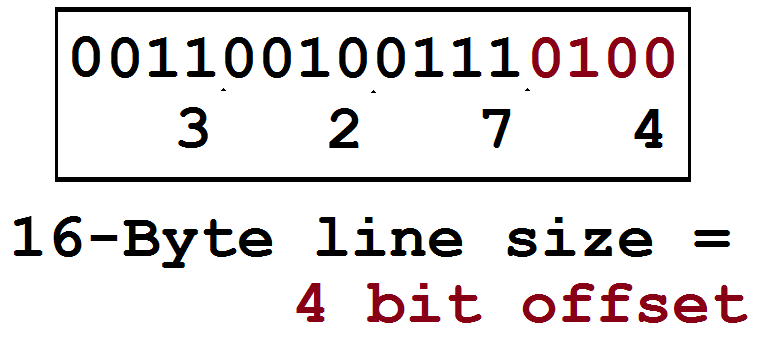
In order to fulfill this assignment, you will have to break the data address down into three pieces:

* The byte offset into the cache line
* The cache line number
* The rest of the address for associativity

Let’s assume a 16-bit address space:

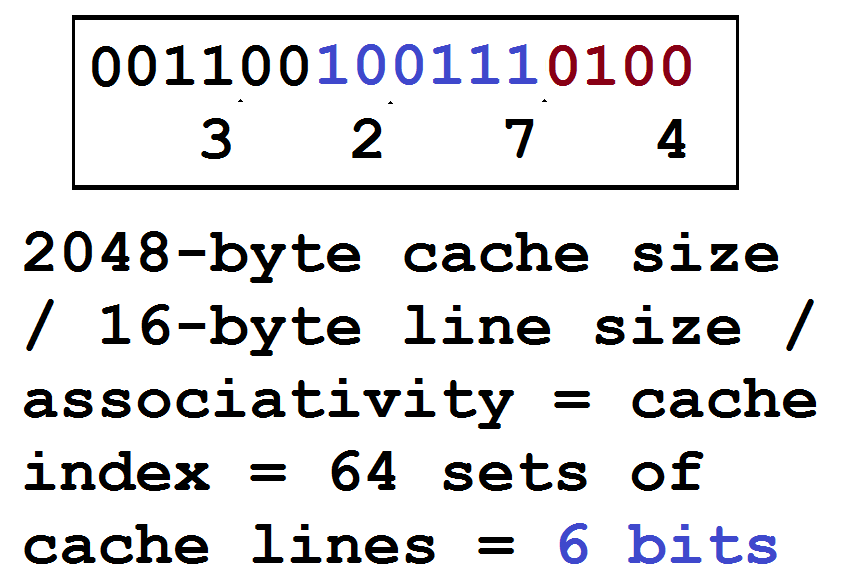


Addresses are in hexadecimal. On this 16-bit address, we will use a 2048-byte cache size, a 16-byte cache line size, and two way associativity. First, the cache line byte offset:



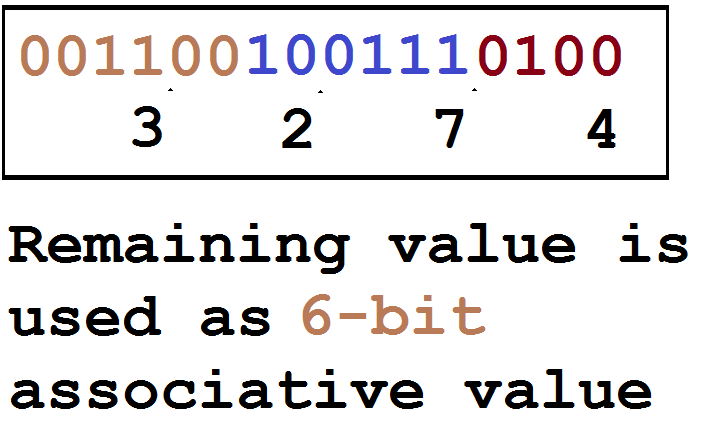
How do we determine that the line size takes up 4 bits of the address? Ask what the power of two is which is equal to the line size. Write a while loop to calculate that power of two.

Next, how do we determine the number of bits for the cache line?



After you do the arithmetic to determine the number of cache line sets, repeat the while loop process to determine the power of two. Note that, in this while loop, you should also calculate how many bits were needed – a simple count of the number of iterations will do that.

Finally, the remaining part of the address will be the associativity value for the cache line:



One does not need to calculate the power of two, but the number of bits will be whatever is left over after the byte offset and cache line are taken away:

16 minus 4 minus 6 equals 6.