Project 1: Analyzing Cache Performance

Project 2: Analyzing Cache performance

What to submit:

You will be submitting a text file containing two C++ functions, **requestMemoryAddress()** and **getPercentageOfHits().** In addition graph your results of your simulation using different block sizes and hit ratio for each address sequences. Provide a brief discussion of those results. Submit your project functions on Canvas. Name your file cache_name where name is substituted by your name.

Introduction

Your project is to simulate a 4K direct mapping cache using C++. The memory of this system is divided into 2, 4, 8, 16, or 32-word blocks, which means that the 4K cache has 4K/2 = 2048 lines, 4K/4 = 1024 lines, 4K/8 = 512 lines, 4K/16 = 256 lines, or 4K/32 = 128 lines. Some significant code is given and described below. You will first need to write two functions, **requestMemoryAddress()** and **getPercentageOfHits().** requestMemoryAddress() will simulate a processor request to the cache while getPercentageOfHits() will return the performance of the cache. You are to run your completed simulator on a set of sample address sequences and plot your results on a graph which you will hand in along with the code.

Class DirectCache

The class Direct Cache will be used to define the cache. First, let's look at its functions.

```
DirectCache(int number_of_word_bits);
bool requestMemoryAddress(unsigned int address);
unsigned int getPercentageOfHits(void);
```

The first function, DirectCache(int number_of_word_bits), is the constructor. Since we are looking to test the cache with different numbers of lines, we need to pass an argument to the constructor to indicate the number of lines. The cache is a fixed size, 4096 words. If we know the block size, i.e., the number of words per block, and since one block is stored per line, we can determine the number of lines by knowing the number of word id bits. This function is already written for you.

The second function, bool requestMemoryAddress(unsigned int address), is used to simulate a memory request from the processor. You will be writing this function. An address is passed to this function as its argument. You will need to determine the block, load it from a memory array, and set the appropriate tag in the cache. Note that the address isn't a pointer! It is an integer between 0 and 65,535 that is supposed to represent the address of the desired data. Assume that the cache is empty to start with, so your very first call to this function will require a figurative "loading of the cache". requestMemoryAddress() will return a bool true if the data

Project 1: Analyzing Cache Performance

was found in the cache or a bool false if the data was not found and the block had to be loaded from memory. In other words, true indicates a "hit" and false indicates a "miss".

The second function, unsigned int getPercentageOfHits(void), is called to see how well the cache performed. You will use this data to compare the different block sizes. The result of getPercentageOfHits() should be an integer from 0 to 100 representing the percentage of successful hits in the cache. The equation should be something like:

$$\text{Percentage of hits} = \frac{\text{number of hits}}{\text{number of memory requests}} \times 100\%$$

Each time a request is made to requestMemoryAddress(), increment number_of_memory_requests, and each time one of those requests results in a "true" returned, increment number_of_hits. These values will be used in getPercentageOfHits().

There are a number of private variables declared in DirectCache too.

```
int *tags;
int *blocks;
int number_of_hits;
int number_of_memory_requests;
int number_of_lines_in_cache;
int number_of_words_in_block;
unsigned int tag_mask;
unsigned int line_mask;
unsigned int word_mask;
```

Since the number of lines varies depending on the declared number of words per block, we need to dynamically declare an array to hold the tags and the blocks for each line. The arrays **tags** uses a line number as its index. The array **blocks** also uses a line number as its index, but it has a second index which identifies the word positions. Both of these arrays are declared in the constructor for DirectCache.

The integers number_of_hits and number_of_memory_requests are to be incremented as needed by requestMemoryAddress(), then used to calculate the output from getPercentageOfHits().

number_of_words_in_block can be calculated from the number of bits that define a word id and is simply included so that you don't have to keep calculating it throughout your code.

The last three integers are also included for your convenience. They are bit masks that you can use to isolate the bits in the address that identify the tag, the line number, and the word id. They are created in the DirectCache constructor. For example, assume we have a 16 bit address

with blocks defined using 3 word id bits and a 256 line cache. The partitioning of our address would look like that shown below.

Tag (5 bits)	Identifying line in cache (8 bits)	Bits identifying word offset in a block (3 bits)

The masks would then be defined like the following.

```
tag_mask =      1111100000000000;
line_mask =     0000011111111000;
word_mask =     0000000000000111;
```

Function powerOfTwo()

A short function int powerOfTwo(int exponent) to return powers of 2 is provided to you in the source code provided. powerOfTwo() returns the value of two raised to the power exponent. It only allows for positive exponents because of the return value. If exponent is less than zero, an error condition of -1 is returned.

main()

A short "incomplete" main() is also provided. It declares and initializes memory as an array of char based on the number of bits in the address. It also prompts the user through the console for the number of word id bits so that you don't have to recompile in order to test different cache configurations. It then creates the instance of the cache.

Sample Address Sequences

In order to test your cache and take measurements of its performance, you are given some sample sequences of address requests. These will be given to you in an array called memory_request_sequence[]. There will be more than one in an effort to simulate loops, sequential code, and other types of program constructs.