EE 352 Final Project Report

Anthony Wittemann - 1704835940

Description of Motivation for the Design

We decided to make the cache a 1KB size because we thought it would be best to keep things simple. We knew we had to create a set associative cache so we used a 256 byte 1D array to simulate a 64x4 2D cache array. There was no particular reason for making the cache 4-way set associative rather than 2-way set associative. For much of the design, we were motivated to keep things simple, because MIPS can get confusing very quickly. An example of how we kept things simple was that we broke down tasks into many smaller subroutines so there was never a large block of code that we'd have to go through to debug. This allowed us to identify faults in the code more quickly.

Detailed Description of Approach and Procedures

The main part of program takes place within a while loop which runs for approximately 10,000 simulated CPU cycles. Within that while loop, 3 function calls are made: 1 to generate a random memory address, another to map that memory address to a set in cache (since we're simulated a set associative cache architecture), and the last one to check if the value we're searching for is in the row/set in cache we've mapped to.

For the actual execution of the program, we started by simulating the information retrieval calls a CPU would make to the cache by randomly generating an address of

memory. To do this, we randomly generated a hexadecimal value because we thought it we had to use hexadecimal values for memory addresses.

In the final stages of the program, we calculate and display the results. To calculate the hit rate and the average memory access latency, we had to convert the variables we were using to single precision floating point numbers.

Throughout the program, we commented in normal/pseudocode language so we always knew what was going on. We also used all the $t temporary variables for the important things that needed to have global scope, like the hit count, and used the $s temporary variables for local scope tasks, like counters.

Results

Total Hit Rate (The percentage of memory ops
(i.e. lines in the trace file) that were hits): 0.9326161

Total Runtime (total processor cycles assuming
that the last memory access was the last instruction of the program): 100002

Average Memory Access Latency
(The average number of cycles needed to complete a memory access): 1.3422818

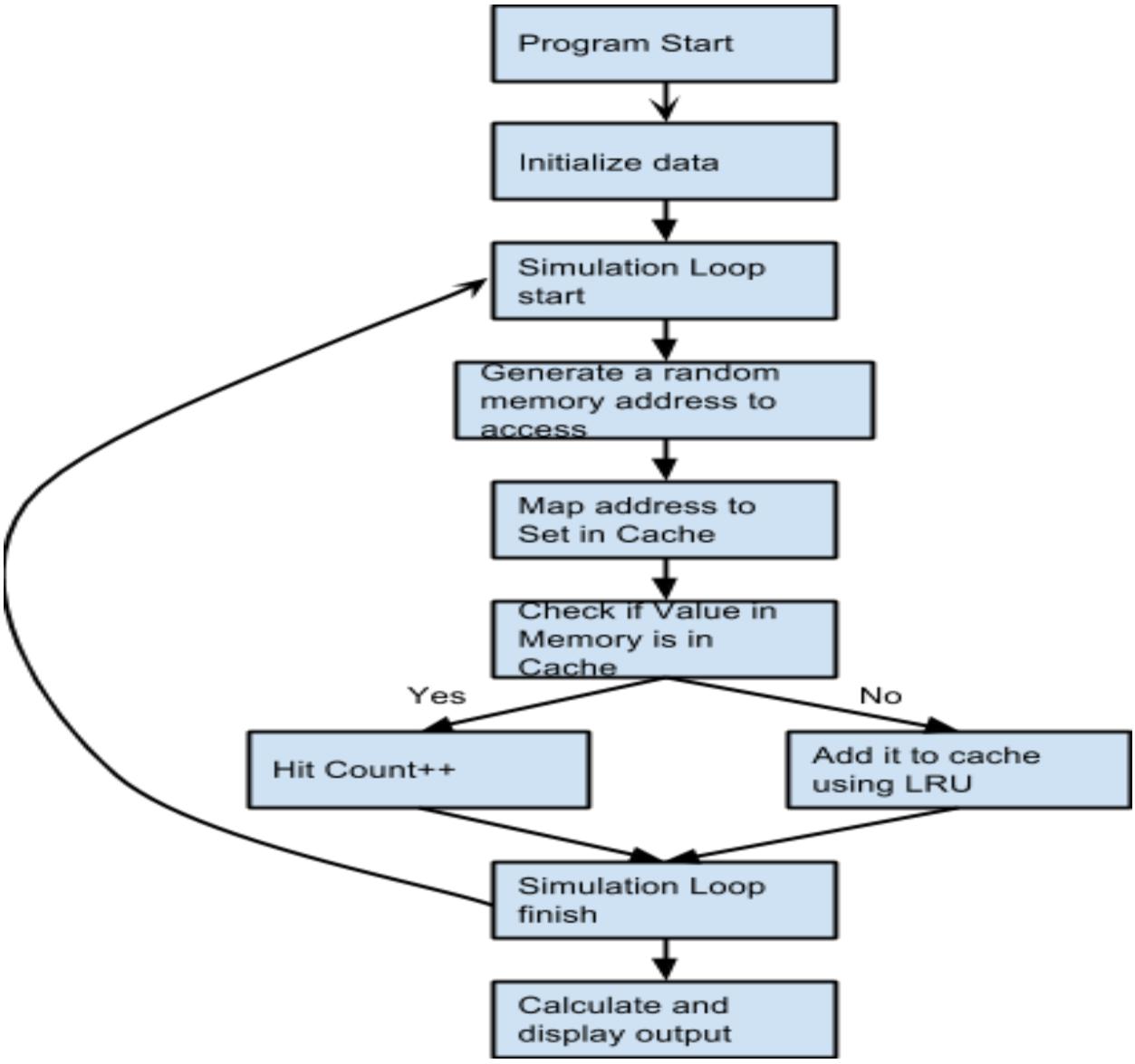
Discussion of Results

The hit rate was calculated by dividing the number of hits by number of simulated CPU cycles the program ran for. (Note: the hit rate is displayed as a decimal, not a percentage.) Our hit rate should be fairly high given the fact that we implemented a 1KB 4-way set associative cache and simulated memory addresses ranging from 0 to 2^10-1 and ran the simulation for around 100000 CPU cycles.

The total runtime also makes sense given the fact that the miss penalty was 10 CPU cycles. What the above runtime indicates is that there was a miss on the 999992nd CPU cycle. The simulation loop ends after at least 100000 CPU cycles.

The average memory access latency is calculated by dividing the total number of CPU cycles by the total number of memory accesses. The above result makes sense, since the miss penalty is 10 CPU cycles and a miss occurs less than 10% of the time, the average memory access latency should be slightly higher than 1.

Program Flow-chart



Description of Each Function

generateMemAddress:

Input: $t6 (empty variable which the memory address will be stored in)

Output: $t6 (random Mem address)

Function: Generates a random 8 bit hexadecimal memory address that the CPU needs to access in addition to printing out address for trace file. Also has two sub-functions explained below.

randomNumber:

Input: $a0 (seed value of 0), $a1 (upper bound of 2^8-1)

Output: $t6 (random memory address), $a0 (same value as $t6)

Function: Generates a random 8 bit hexadecimal memory address that the CPU needs to access. Also has two sub-functions explained below.

printAfterGeneratingNumber:

Input: $a0 (random memory address)

Output: $a0 printed to the console

Function: prints out the memory address we are accessing for the trace file

mapToSetInCache:

Input: $t6 (memory address)

Output: $t5 (set in cache to store word in)

Function: takes a memory address modulo 64 and maps it to a set in cache

loopingOverRow:

Input: $s1 (column counter), $t1 (number of columns), $t2 (header value we're searching for in cache)

Output: replaceCacheHit (hit branch) or replaceCache (miss branch)

Function: looks for a hit in a set in cache by iterating over the 4 columns

replaceCacheHit:

Input: $t8 (number of hits), data (storage for 64x4 cache matrix as 1D array), $s4 (value in the current cache location), $t2 (header value we're searching for in cache)

Output: $t8 (number of hits), data (storage for 64x4 cache matrix as 1D array)

Function: replace from the back of the column counter and report a hit

replaceCache:

Input: $t7 (miss count), data (storage for 64x4 cache matrix as 1D array), $s4 (value in the current cache location), $t2 (header value we're searching for in cache)
Output: $t7 (miss count), data (storage for 64x4 cache matrix as 1D array)
Function: replace from the back of the column counter and incur a miss penalty

displayResults:
Input: totalHitRateMsg, totalRuntimeMsg, avgMemAccessLatencyMsg, $t4 (number of memory calls), $t8 (number of hits), $t7 (total runtime)
Output: hit rate, total runtime, average memory access latency to console
Function: calculates and prints to the console the total hit rate (hits/CPU cycle), total runtime (CPU cycles), and the average memory access latency (CPU cycles)

Relationship of each Function

The displayResults function is dependent on all the previous functions to perform the correct computations in order for the results that are output to be legitimate.

The replaceCache and the replaceCacheHit functions have similar functionality because they both utilize the LRU cache replacement scheme but differ in the fact that in the hit function, the hit counter is incremented, whereas in the other function a miss penalty is incurred.

The loopingOverRow function is a precursor to the functions that deal with cache replacement. It decides whether there is a hit or not and thus determines which of the following two functions it will follow.

All the functions are dependent on the generateMemAddress function to perform as expected, because all of the other functions, either directly or indirectly, use the memory address that it generates.

The mapToSetInCache function is dependent on the generateMemAddress function, and many of the other functions are dependent on it to correctly map the memory address that was generated to a set in cache.

The printAfterGeneratingNumber function is dependent on the previous randomNumber function, but none of the data used by any of the other functions depend on it.

The randomNumber function is a subroutine of the generateMemAddress, and like that function it also is used by all the other functions in the program.

Function of what has been Learned in Class and Design

Using my knowledge of how set associativity worked, I designed a cache such that every memory address that was generated mapped to a set in cache. To implement this, I created a 64x4 2D array to simulate a 4 way set associative 1KB cache.

From there, the exact location within the set in cache was determined using the LRU (least recently used) concept. I used the LRU concept in my design by always having the 0th column within each set to be the location where the new item to be loaded into cache was placed. I shifted the other items that were previously in cache over by 1 column, so the item that was in the 0th column was placed in the 1st column, the item that was previously in the 1st column was shifted into the 2nd column and so on until the item that was in the last column in the set was removed from cache.

Project Relation to Material Discussed in Class

This project incorporates the MIPS assembly programming language we learned in lab with the memory and cache concepts we learned in class. Using MIPS, I constructed a program that simulated a 4-way set associative cache. This project also related to what was covered in class because we incorporated the concepts of LRU into how items were replaced in cache, hits and misses in information retrieval, hit rate, and cache latency. We used the formulas in the book for the above stats in our simulation program. The data types we used, such as single precision floating point numbers also came into play when we were calculating decimal averages or ratios.