Jason Gomes 164000677
Computer Architecture
Professor Santosh Nagarakatte
4/20/17

Design

The design of my cache simulation took me a long time to figure out. I ended up using two different structs to create the cache object. I created a Line struct, which held the tag, index, and offset bits in separate variables. In my implementation, I decided against a valid bit, as my code either has a NULL spot in the cache if a block has not been loaded in yet, or a block with a line. There is never a time where a value gets removed without being immediately replaced, which to me meant there wasn't a need for a valid bit. Next, I created a Queue struct which held a Line, a pointer to a previous queue, and a pointer to the next queue. Essentially, I implemented a linked list and called it a queue, but have a dequeue and enqueue instead of a basic insert and delete. Finally, my cache is just an array of queues, and it has a size that equals the number of sets. To summarize, the cache is an array of queues, and each queue contains the Line struct, and pointers to more lines.

I needed multiple methods in order to properly implement this design. The first method **checkInput** checked the command line input, and makes sure the cache size, associativity, block size, and input file are all properly formatted. Once the input is correct, we send this information into the **fillCache** method. In fillCache, the text of the input file is tokenized. If the file has a W for write, then the memAWrite and memBWrite variables are incremented. For both options, the tokenized word (the address) is passed into the **parseMemory** method. This is looped until the input file is NULL.

In parseMemory, the hex address is passed into the **binaryConversion** method. In binaryConversion, the hex address is translated into binary, and then the method **extendAddress** checks to see if the address is 48 bits, and if it's not, it adds a number of leading zeroes to the string. After this, both methods send this new binary address to the **search** method. Inside search, the address is parsed into a Line struct via the **getBitInfo** method. Then, the index determines what set to search through, and both caches are searched for the address. If it is found, then cacheHit is incremented. If not, then cacheMiss and memRead variables are incremented, and either cacheA or cacheB are updated in the **updateCacheA** or **updateCacheB** method.

The updateCache method has three base cases. The first is assuming there's nothing in the queue at that given spot in the array (AKA the set). The second is assuming the queue has at least one input, but is not full. In this case, you loop until you find an empty block, and then you enqueue the Line struct. Finally, if the queue is full, you have to enqueue the new line and dequeue the line that was the first in of the remaining lines.

```
//Method Declarations
void printResults(); //Prints final tally of all global variables
void checkInput(char ** input); //Makes sure input format is correct
void mallocError(); //Exits program if malloc returns NULL
bool PowerOfTwo(int x); //Checks if input is proper power of two
void fillCache(Cache * tableA, Cache * tableB); //Interprets input and sends it to the read or write method
void parseMemory(char * val, Cache * tableA, Cache * tableB); //Helper method that passes address into binaryConversion and search
void updateCache(Cache * table, Line * temp, char * number); //Update given queue in given cache
char * binaryConversion(char * hexVal); //Converts hexadecimal location to binary
char * extendAddress(char * binary); //Extends the binary conversion to 48 bits
Line * getBitInfo(char * binary, bool ifAthen0); //Returns tag, index, and offset bits for each cache in line struct
void search(char * data, Cache * tableA, Cache * tableB); //Searches through the cache for given address
int btoi(char * address); //In search, translates index into an integer
int max(int a, int b); //Returns max for strncmp uses
```

Which cache produced more hits?

From my perspective, cache A produced more hits. This is just because the index bits being at the beginning of cache B meant that if you had to extend the address, the index bits would always turn out to be zero. Extending the address would add leading zeroes, and the first few bits in cache B would turn out to be the index bits. Also, if the address was already 48 bits and did not have to be extended, most of the addresses would start with the same character. If they all started with the same character, that means the index would always be the same number, and all values would be stored in the same set. Alternatively, in cache A, the index bits being almost at the end would produce very different numbers, and would therefore be all kept in different sets. The main difference between these two implementations is that cache A stores more lines, and therefore has a higher likelihood of having the memory stored in the cache.