In this lab, you will use C (or C++) to write a program that will evaluate the design space of single-level cache and TLB. Given a particular memory organization (size of address space, physical memory size, and size of TLB or cache memory), the program will read provided input files, each containing a memory trace (a sequence of memory addresses) and determine the performance of each cache/TLB configuration. The program should keep track of the total number of hits and misses generated by the TLB & cache over the entire trace. The hit rate should be plotted for different cache and TLB configurations (as detailed below). The trace files are in zip format from the Web page:

http://www.ece.gatech.edu/~leehs/ECE3055/homework/trace3055.zip
http://www.ece.gatech.edu/~leehs/ECE3055/homework/trace3055-48bit.zip

There are a bit more than 1.1 million memory references in the 3 trace files, each line stores one memory access including both instruction fetch and data access. (TA might use different traces to test your program.) These were collected from three selected benchmark, 2 from SPEC CPU2000 Integer (176.gcc and 181.mcf) and one from SPEC CPU2000 floating-point (177.mesa), all based on Alpha binaries. The address is the second field on the line. The first field gives the type of the memory reference (2 for instruction fetch, 1 for store, and 0 for load). The third field gives the machine instruction encoding for a fetch and is always 0 for loads and stores. (Note that, even though Alpha is a 64-bit architecture, the traces were processed to contain only 32-bit addresses.) In this assignment, we assume a unified instructions and data cache (or TLB) is used, i.e. both instruction and data are kept inside the same memory structure.

To help you get started on this assignment, a simple program that reads the trace file, extracts the address, and performs some simple bit-wise operations similar to what is needed to extract bit fields from the address to determine the cache address is available at the URL below. You may not need to use all of the fields in the trace data file. Note that the TA might use additional traces during the sign-off to perform more verification of your implementation.

http://www.ece.gatech.edu/~leehs/ECE3055/homework/read_trace.c

(50%) Part 1: Writeback Cache. In this part, you will write a single cache simulation program to simulate all scenarios for (1) and (2) below by specifying the cache parameters: the number of sets and the line size as part of your main() arguments. Another advice to your program is that you better perform bound-check to ensure the parameters are in power of 2, also, print out the number of index bits and tag bits so your TA can easily identify the root cause of your program if it does not work properly.

(1) Given a fixed cache line size of 32 bytes, plot the total cache miss rates versus cache memory size (for all three traces) for a single direct-mapped cache (without any virtual memory system). In calculating the cache miss rates, your numerator is the total number of all misses and the denominator is the total number of references including reads and writes for instructions and data. Data points should be generated for the number of cache sets starting from 128 to 128K (i.e. 131,072) in power of two. Then plot the number of dirty writebacks for all cache configurations in another chart. Note that our cache is a unified cache so that all the instruction and data references look up the same cache. Assume write-allocate policy is applied, namely, if a write miss occurs, the missed line (or block) will be brought into the cache and evict the conflict line before being written.
(2) Given a fixed total (direct-mapped) cache size budget of 64KB. Now, vary the number of the cache sets from 128 to 8192 in power of two and generate the same plot of miss rates and dirty writeback plots of (1). (Note that the cache line size will vary too due to the fixed total size.)

(50%) Part 2 - Virtual Memory: The virtual address space is fixed to 48 bits and all virtual addresses are byte addresses. Plot the total numbers of page faults versus physical memory size for a Least Frequently Used (LFU) page replacement policy after running the trace file data. For LFU, you have to maintain counters to count the number of usage for each page. Data points should be generated for physical memory sizes equal to 128KBytes, 256KB, and 512 KB. Please vary the page for 4KB, 16KB, and 64KB. In this part, you must develop two types of page table: (1) a two-level page table; (2) an inverted page table, to track the address translation for the traces. For the two-level page table cases, simply divide the VPN into 2 halves. For example, if the VPN is 36 bits, use 18 bits for the first level and 18 bits for the second level. You do not need to implement hashing in your inverted page table because the time to search this relatively small page table is not a significant concern in your program. In addition to the page faults, please also show the memory size needed for keeping the actual page tables (count both the pointer table and the actual page table) for each simulated combination when using a two-level page table.

Again, you need to sign off with the TA for this assignment before the due day. But you need to prepare and show the TA the miss rate plot with different configurations on the same plot. (You can use MS Excel, gnuplot, OpenOffice, or any other plotting tools.)