EE204 — Computer Architecture
Course Project

Due Date: Monday, May 25th by 8am.

This project is a group activity to be done in groups of two or three. No late submissions will be accepted. Submit your source code on SLATE on the day that it is due.

Submissions that do not comply with the specifications given in this document will not be marked and a zero grade will be assigned.

You are to hand in at the start of class on the due date a performance evaluation report as specified in this document with your names and email address on a cover sheet. Submit your source code on SLATE. **You do not have to submit a printed copy the source code listing of your program.**
You are also to submit a single file *cache.txt* containing your documentation and source code along with a zip file (*cache.zip*) containing an archive of your source program on SLATE.
As a group, you will present your project and results in a presentation during the last week of classes.

Cache Simulation

In this project you will implement a simulation of the Level-1 cache. Your implementation will be able to simulate direct-mapped, set-associative and fully associative unified and split Level-1 caches.

In this project you will implement a trace driven Level-1 data cache simulator and run a set of experiments using your simulator. The simulator is known as a trace-driven simulator because it takes as input a trace of events, in this case memory references. The traces, which are provided for you, were acquired on another machine. Once acquired, they can be used to drive simulation studies. In this project, the memory reference events specified in the trace(s) given to you will be used by your simulator to drive the movement of data into and out of the cache, thus simulating its behavior. Trace-driven simulators are very effective for studying caches. By the end of the semester you will become familiar with the working of caches and how to evaluate their performance. To achieve these goals, you will implement a cache simulator and validate its correctness. You will use your cache simulator to study many different cache organizations and management policies discussed during the lectures.

Build your simulator by incrementally adding functionality, and this is a general software engineering rule of thumb. **The biggest mistake you can make is to try to implement the cache functions all at once. This may happen if you start too close to the due date.** *Begin with the simplest cache model possible, and test it thoroughly before proceeding. Then add a small piece of functionality; test it thoroughly before proceeding further.*

- You can write your program in either Java or C++ but it must run on Linux machines (specifically Ubuntu).
- Your Level-1 Cache Simulator should accept the following command-line options:

  `-s` $< split >$ or $< unified >$.
  `-c` $< capacity >$ with $< capacity >$ in KB: 4, 8, 16, 32, or 64.
  `-b` $< blocksize >$ with $< blocksize >$ in bytes: 4, 8, 16, 32, 64, 128, 256, or 512.
  `-a` $< associativity >$ where $< associativity >$ is integer size of set: 1, 2, 4, 8 or 16.

- The baseline configuration (i.e., the first one that you should test) is: capacity = 8K, blocksize = 16 bytes (i.e., 4 words), set associativity = 4 with one of the following commands (NOTE: the code provided requires no spaces between the letter and number):

  ```
  cache -s -c8 -b16 -a4
  java cache -s -c8 -b16 -a4
  ```

  and for the *unified* configuration:

  ```
  cache -c8 -b16 -a4
  java cache -c8 -b16 -a4
  ```

- The `-s` option specifies a *split* cache. This option indicates that the L1 cache is split equally into L1D (Data Cache) and L1I (Instruction Cache). The `-c` option gives the combined size of the L1 cache, split equally between L1D and L1I. The block size and associativity is the same for both L1D and L1I. If the `-s` option is not given then the cache is $unified$ by default, as before (i.e. instruction reads are also treated as data reads).

- Additionally, your cache should have the following functionality to handle data write hits and misses with optional command-line options:

  - `-wb` write back (if write strategy not specified then write-back should be default for write hits)
  - `-wt` write through
  - `-wa` write allocate (if write strategy not specified then write-allocate should be default for write misses)
  - `-wn` write no-allocate

- You should implement LRU (least-recently-used) replacement policy for fully associative and set-associative caches.

- The input to the cache simulator will be a sequence of memory access traces, one per line, terminated by end of file. In the following format, with a leading 0 for data loads, 1 for data stores and 2 for instruction load. You should ignore anything (possible comments) that follows.

  ```
  0 <address>
  1 <address> <dataword>
  2 <address>
  ```

- Each address will be the address of a 32-bit data word. The address and data are expressed in hexadecimal format.

- To support testing of your data cache simulator, you will implement a simple model of main memory. The capacity of the memory should be 16 MB (4 megawords). At simulator start-up time, initialize the contents of each word to be the word's address. For example, the 32-bit word that starts at byte `0x0000100F` should be initialized to `0x0000100F`.

- The output of your program will consist of a set of statistics gathered during the simulation as well as the contents of the cache and memory at the end of the simulation. The format of the output is described in detail at the end of this document.

- Also, you are provided the initial framework for both C++ and Java versions of the code to get you started. These files will help with input and output, so you can focus on cache simulation.

- Your program will be graded for correctness and design.
  - **Design**: It is expected that you will carefully construct classes that reflect the application. Be sure that your method names and variable names reflect memory hierarchy terms. When done, anyone reading your code should have learned how a cache works. Towards that end, you should define constants such as '*dirty*'.
  - **Correctness**: You should create your own trace files to help you test and debug your cache configurations.
- Here are some more details on how you should implement your simulator:
  - You should initialize all of the cache locations to be *invalid* and initialize all of the tag and data fields to *zero*.
  - All of the parameters on the command line are base 10 numbers (e.g. 32, 64, 128, etc.)
  - The addresses and data in the trace files are expressed as base 16 (hexadecimal) numbers. Note that these numbers might (or might not) have extra zeros in front... e.g. `00de2a` vs. `de2a`.
  - Read your input from *stdin* and write your output to *stdout*. (i.e., read from the terminal and write to the terminal)
  - Use the unix shell's ability to redirect input and output to pull input from disk files and write output to disk files. For example:

    ```
    csh> cache -s -c32 -b16 -a4 < test.trace > test.output
    ```
  - NOTE: There are no spaces between the number and the letter, which the code provided requires. Do not make changes to this specification.
  - Your output should be formatted as described below in *output format*. You will print out the entire contents of the cache, and 1024 words of memory starting at address `0x0040bc8c`.
  - At the end of the trace, write all dirty blocks to the memory, so that the memory contains updated data. However, do not count them in *Number of Dirty Blocks Evicted From the Cache*, since they have not really been evicted from the cache.
- Source code(C++ Files): The following files can be used as a starting point for the project.
  - *Makefile* is the configuration file for building your project. You will probably need to make changes to it as your project evolves. Look in its comments for more details, or in the man page:

    ```
    $ man make
    ```

    Copy the *Makefile* to the directory where your source files are and run `make`. The *Makefile* has some comments that will help you add your source files. into the *Makefile*. The *Makefile* currently compiles the files *main.cc* and *funcs.cc* and produces a final binary *cache*
  - *funcs.cc* contains *parseParams*(), which helps you parse the command line parameters that are passed to your cache simulator. You must add code to support optional command line options described earlier. The command line options are:

    `-s` = split cache
    `-c` = cache capacity
    `-b` = blocksize
    `-a` = associativity

    You need only some small modifications to this code to support more command line options.

    The *parseParams* function reads and parses these options; just call it with the appropriate variables. Remember to check the return value of the function as in *main.cc*.

  - *main.cc* is the top-level entry point for the project. It currently simply illustrates the use of $parseParams()$ to parse the command-line. This file is the main file which illustrates the use of the function $parseParams()$ which is in the file $funcs.cc$
  - $funcs.h$ is the header file for $funcs.cc$
  - *io.cc* is sample code for reading the input trace format. This is simply a sample file and should not form part of your final project.
  - The simulator can now be invoked with the following command line
    `cache -c32 -b32 -a4`

- Source code(Java):
  - *cache.java* is the top level entry point for the project and contains the $parseParams$ routine.
  - *io.java* contains routines for reading in data and writing out final cache statistics and contents. You should move these routines to your own classes and code. This is simply a sample file and should not form part of your final project.

1 Incremental Approach

Build the simulator by incrementally adding functionality. The biggest mistake you can make is to try to implement the cache functions all at once. Instead, build the very simplest cache model possible, and test it thoroughly before proceeding. Then, add a small piece of functionality, and then test that thoroughly before proceeding and so on, until you have finished the project. We recommend the following incremental approach:

1. Build a unified (without the -s flag), fixed block size, direct-mapped cache with a write-back write policy and a write allocate allocation policy.
2. Add variable block size functionality.
3. Add variable associativity functionality.
4. Add split organization functionality.
5. Add write through write policy functionality.
6. Add write no-allocate allocation policy functionality.

- You are not provided the solutions to the small test cases, but the test cases are small enough that you can hand calculate the results. Once your program is complete and passes the simple test cases, you can generate additional trace files for ‘stress testing’.

2 Report and Experiments

After you test your cache simulator, you will use it as a tool to evaluate several cache configurations, to determine which one results in the lowest miss rate for the given benchmark traces. The trace file for the compress benchmark will be provided. You will write a 3-page report summarizing the design of your simulator and your experimental results. Your report and experiments should be structured as follows.

- **Performance Evaluation**

  You should finally have a cache simulator that can be configured (for total size, block size, unified versus split, associativity, write-through versus write-back, and write-allocate versus write-no-allocate). Now you will use your cache simulator to perform studies on the three sample traces that will be provided and plot results in a report. You will measure the performance of a few program traces (to be provided later) for various cache configurations using your simulator as follows:

1. Working Set Characterization

   In this performance evaluation exercise, you will characterize the working set size of the three sample traces given.

   Using your cache simulator, plot the hit rate of the cache as a function of cache size. Start with a cache size of 4 KB, and increase the size (each time by a factor of 2) until the hit rate remains insensitive to cache size. Use a split cache organization so that you can separately characterize the behavior of instruction and data references (i.e., you will have two plots per sample traceone for instructions, and one for data). Factor out the effects of conflict misses by *always* using a fully associative cache. Also, always set the block size to 4 bytes, use a write-back cache, and use the write-allocate policy.

   – Explain what this experiment is doing, and how it works. Also, explain the significance of the features in the hit-rate vs. cache size plots.

   – What is the total instruction working set size and data working set size for each of the three sample traces?

2. Impact of Block Size

   Set your cache simulator for a split I- D-cache organization, each of size 8 KB. Set the associativity to 2, use a write-back cache, and use a write-allocate policy. Plot the hit rate of the cache as a function of block size. Vary the block size between 4 bytes and 4 K-bytes, in powers of 2. Do this for the three traces, and make a separate plot for instruction and data references.

   – Explain why the hit rate vs. block size plot has the shape that it does. In particular, explain the relevance of spatial locality to the shape of this curve.

   – What is the optimal block size (consider instruction and data references separately) for each trace?

   – Is the optimal block size for instruction and data references different? What does this tell you about the nature of instruction references versus data references?

3. Impact of Associativity

   Set your cache simulator for a split I- D-cache organization, each of size 8 KB. Set the block size to 128 bytes, use a write-back cache, and use a write-allocate policy. Plot the hit rate of the cache as a function of associativity. Vary the associativity between 1 and 16, in powers of 2. Do this for the three traces, and make a separate plot for instruction and data references.

   – Explain why the hit rate vs. associativity plot has the shape that it does.

   – Is there a difference between the plots for instruction and data references? What does this tell you about the difference in impact of associativity on instruction versus data references?

4. Memory Bandwidth

   Using your cache simulator, compare the total memory traffic generated by a write-through versus write-back cache. Use a split I- D-cache organization. Simulate a couple of reasonable cache sizes (such as 8 K-bytes and 16 K-bytes), reasonable block sizes (such as 64 and 128 bytes), and reasonable associativities (such as 2 and 4). For now, use a write-no-allocate policy. Run 4 or 5 different simulations total.

   – Which cache has the smaller memory traffic, the write-through cache or the write-back cache? Why?

   – Is there any scenario under which your answer to the question above would flip? Explain.

   – Now use the same parameters, but fix the policy to write-back. Perform the same experiment, but this time compare the write-allocate and write-no-allocate policies. Which cache has the smaller memory traffic, the write-allocate or the write-no-allocate cache? Why?

   – Is there any scenario under which your answer to the question above would flip? Explain.

5. Now, research some of the modern architectural designs. Choose your favorite processor and determine the configuration of one of its caches. (It is suggested you look at some embedded processors.) Then run the given spice trace for this configuration on your simulator and report the results. To get full credit you must give the complete cache configuration of your favorite processor including processor name and model, cache level, associativity, capacity and block size.

- Submit all of your source files, header files, and makefiles (if used) on SLATE. All of these files must be well documented with comments. If your submission is in C++, then the command `make` should build your executable binary. If your submission is in Java, then `javac cache.java` should build your java binary. A compiled binary that executes on linux machines. The binary file must be named *cache* and accept command line arguments in the exact form specified above.

- Hardcopy must be turned in at beginning of class on the due date. This should include a printout of the cache contents and the 1024 words of memory contents for ONLY *cc.trace* (to be provided later) and a document containing a detailed performance analysis as follows.

- Debugging — For those not familiar with `gdb` or do not know the `gdb` commands, there is a visual front end to the `gdb` debugger called `ddd` (Data Display Debugger).
  Just type `ddd` at the command prompt and it should invoke the debugger.
  You can set breakpoints, watch variable values, step through code, and lots more.
  The debugger is quite easy to use and you will find this tool helpful in debugging your program.
  To start debugging your program load the binary file into the debugger by choosing File → Open Program.
  If you have compiled your code with the `-g` option (which is already present in the $Makefile$), the debugger will load your program source code and you are set to go.

- **Format of Output:**
  Key to the following description:
  base of output: (10 = decimal and 16 refers to hex)
  rate should be a real number, with 4 digits of accuracy
  0/1: print either 1 for True or 0 for False

  The sample given below is for the split configuration. For a unified cache your program should print accordingly.

  ```
  ------------------------------------------------------------------------------
  STATISTICS:
  L1I Misses: <Total (10)> <InstructionReads (10)>
  L1I Miss Rate: <for total> <for instructionread>

  L1D Misses: <Total (10)> <DataReads (10)> <DataWrites (10)>
  L1D Miss Rate: <for total> <for dataread> <for datawrites>
  Number of Dirty Blocks Evicted From L1D Cache: <Total (10)>

  L1 DATA CACHE CONTENTS:
  SetNumber Valid Tag Dirty Word0 Word1 ...
  <base 16> 0/1 <16> 0/1 <16> <16>...
  ...

  L1 INSTRUCTION CACHE CONTENTS:
  SetNumber Valid Tag Word0 Word1 ...
  <base 16> 0/1 <16> <16> <16>...
  ...

  MAIN MEMORY: {print 1024 words of memory starting at address address 0x003f7f00.
  Print 8 words of memory per line, all values should be base 16}

  <address of word0> <word0> <word1> <word2> ... <word7>
  <address of word8> <word8> <word9> <word10> ... <word15>
  ...
  <address of word1016> <word1016> <word1017> <word1018> ... <word1023>


  ------------------------------------------------------------------------------
  ```

Honor Policy
This project is a group learning opportunity that will be evaluated based on your ability to think and work as a team, work through a problem in a logical manner and implement a software program on your own. You may however discuss verbally or via email the general nature of the conceptual problem to be solved with your classmates or the course instructor, but you are to complete the actual programming for this assignment without resorting to help from any other person or other resources that are not authorized as part of this course. If in doubt, ask the course instructor. You may NOT use the Internet to search for solutions to the problem.