"The greatest enemy of knowledge is not ignorance, it is the illusion of knowledge." – Stephen Hawking

Learning Objectives

1. Basic Cache Design and Implementation

Work that needs to be handed in (via SVN)

1. `simplecache.cpp`: contains your `find` and `insert` implementations. **This is due by the first deadline.**
2. `cacheblock.cpp`, `cacheconfig.cpp`, `cachesimulator.cpp`, and `utils.cpp`: contain your cache implementation. **This is due by the second deadline.**

Guidelines

- You may want to spend some time drawing out data structures before you begin coding.
- Follow good programming practices: comment your code, label variables and functions according to their purpose, use a debugger, and stick to a single coding style.
- We will be testing your code using the EWS Linux machines, so we will consider what runs on those machines to be the final word on correctness.
- Our test programs will try to break your code. You are encouraged to create your own test scenarios to verify the correctness of your code. One good test is to run your cache with different cache block lengths and sizes, with different levels of associativity, and with different writing policies.
- For the second part, we **strongly recommend** following the steps listed in the Specifics section. Writing small portions and then testing them immediately will make your life a lot easier.

Simple cache [20 points]

For the first deadline, you have to finish `simplecache.cpp`. A header file `simplecache.h` has been provided to you with a simplified cache store and some public functions that you can use. Read the file to understand how the Simple Cache Simulator works.

You have to write the two basic functions a cache should be capable of: `find` and `insert`.

For `find`, you will have to look up the set of blocks that corresponds to the given index, and find the correct block within that: meaning that the **block should be valid**, and its tag should match the given tag. If such a block exists, return the correct byte within the block. If not, return `0xdeadbeef`, the universal ~~slaughterhouse~~ 'bad address' code.

For `insert`, you will again look up the set of blocks corresponding to the given index. Next, look through the set for an invalid block. If there is one, replace it with the block being inserted and return. If not, replace the 0^{th} block of the set (overwriting whatever was in it). Use the provided `replace` function in `simplecache.h`.

The purpose of this exercise is to get you used to how the cache hierarchy and some basic cache properties work: sets, blocks, bytes, indices, tags, etc. Keep in mind that this is a (heavily) simplified version of Part 2, so understanding it will make it easier for you to understand the subtler aspects of caching (as well as help you for the exam and lab).

Compiling and Testing

Compile and run using the following commands:

```
make simplecache
./simplecache
```

The expected output is in the comments in `simplecache_main.cpp`

Cache Simulator [80 points]

For the second deadline, you will be completing a program to simulate a cache for a processor. The simulator will read in a trace of memory instructions (32-bit addresses) that has two kinds of operations: read (`R x` reads the data at location `x`) and add (`A x y z` reads the data at memory locations `y` and `z` (in that order), adds them, and writes the result into location `x`) and determine which are cache hits and which are cache misses. The pattern of writing data to memory will be based on the various paradigms that you have learned in class. We have provided some functions to get you started so that you do not need to worry about the specific implementation of how we store and manage the data (though you are welcome to go through it for your own interest).

A cache simulator is a program that is used by computer architects to estimate the performance of caches of different sizes for a given workload. A key idea of a simulator is that it isn't a cache design itself; it is instead a model. So we design a simulator slightly differently than a cache. For instance, keeping track of flag values and the such are done in ways that are easy to do in software, and are not implemented how they actually are in hardware. Also, while cache indices are tracked implicitly in hardware (they are intrinsic to the circuit design), in the simulator we explicitly store the index of each cache block within the block for simplicity.

Your cache should be able to handle variable total sizes, as well as variable block sizes and degrees of associativity, and also different writing paradigms. The block size has to be a power of two. The cache size does not necessarily have to be a power of two. The number of sets will be a power of two to allow for easy indexing, but the number of blocks per set (the associativity), does not need to be a power of two. For example, 20 cache blocks with 5-way associativity will mean 4 sets.

For associative caches, you should implement a Least Recently Used replacement policy. LRU can be implemented in a variety of ways. For this lab, the simulator keeps a "clock" (the `_use_clock` member of `CacheSimulator`), which is just a counter of accesses. Each block has a `last_used_time` member. When a block is accessed, you increment the clock and update the last used time of the block. For LRU replacement, the block in a set with the oldest (i.e. smallest) last used time is the LRU block.

Specifics

The simulator is split across a few classes. You should take a look at all the header files to see what functionality each class provides, and how you can use that functionality in your implementation.

We've provided both unit tests (in `cacheblock_test.cpp`, `cachesimulator_test.cpp` and `utils_test.cpp`) and full simulator tests (in the `trace_files` directory). As always, you should augment these tests with your own. You can compile and run the unit tests using the following commands:

```
make unit_tests
./unit_tests
```

The details of the simulator tests and how to run them are in the next section. Here's the order we **very strongly** suggest following for completing this lab; the key idea is completing a portion at a time and then testing it immediately (and fixing any bugs).

1. Implement the `CacheConfig` constructor in `cacheconfig.cpp`, which computes and stores the number of bits in each portion of the address for a specific cache configuration. Remember that our addresses are 32 bits. **Test it using the unit tests**; we've provided one test called `CacheConfig.NumBits`.
2. Implement the `extract_block_offset`, `extract_index` and `extract_tag` functions in `utils.cpp`, and **test them using the unit tests**. Some handy functions are available in `utils.h`. We've provided a regular test `Utils.Extract` and an edge case test `Utils.Extract32BitTag`. *Hint:* to get a bitstring of `N` 1s, you can do `(1 << N) - 1`. However, shifting by an amount greater than or equal to the width of your type (in this case, 32 bits) is undefined behavior, so make sure you don't run into that, hence the edge case test.
3. Implement the `get_address` function in `cacheblock.cpp`, which computes the *starting* memory address of a block. **Test it using the unit tests**; we've provided one test called `CacheBlock.GetAddress`. At this point, all of our provided unit tests should pass.
4. Implement the `find_block` function in `cachesimulator.cpp`, which searches for the block containing a particular address in the cache (if such a block exists). The function comment has implementation details.
5. Implement the `bring_block_into_cache` function in `cachesimulator.cpp`, which loads a block from memory into the cache. The function comment has implementation details.
6. Implement the `read_access` function in `cachesimulator.cpp`. The function comment has implementation details, and our LRU implementation is detailed in the previous section.
7. Try out the simulator with the `basic`, `block`, `associativity` and `lru` traces; details are in the next section. Make sure cache reads work correctly for a variety of cache sizes, block sizes and associativities.
8. Implement the `write_access` function in `cachesimulator.cpp`. The function comment has implementation details.
9. Try out the simulator with the `write_back` and `write_allocate` traces to make sure the writing policies are implemented correctly.
10. For final testing, run your simulator against `gcc-trace.verbose` and then `gcc-trace`. Unfortunately, the sheer size of these traces make debugging difficult, but they at least tell you if your simulator is behaving correctly for complicated traces.

Running the simulator

Compile and run the simulator as follows:

```
make cachesim
./cachesim <trace_file> <data_file> <cache_size> <block_size> <associativity>
           <write_back> <write_allocate> [verbose]
```

- `trace_file` is a file containing the loads and adds the simulator will run.
- `data_file` is a file containing the initial values stored in each memory location.
- `cache_size` is an integer representing the cache size in words.
- `block_size` is an integer representing the number of words in each block.
- `associativity` is an integer representing the associativity of the cache.
- `write_back` is a `1` or a `0`, if the cache is a write-back cache or not, respectively.
- `write_allocate` is a `1` or a `0`, if the cache allocates on writes or not, respectively
- `verbose` is an optional argument. If you include `verbose` as the eighth argument, the code will print out the result of each operation

For example:

```
./cachesim trace_files/lru data_files/data 4 1 4 0 0 verbose
```

Provided Traces

We've provided some small traces to facilitate testing and debugging. Each trace should be run against the data file `data_files/data`, and you can use the verbose parameter to see exactly how your cache is behaving. You should test your cache's behavior for a variety of cache sizes, block sizes, associativities and write policies. We recommend creating more small traces to cover other scenarios.

- `basic` consist of 5 accesses to the same address (the first should miss, the rest should hit).
- `block` consists of accesses to sequential addresses; run this trace varying the block size of the cache (the first access to a block should miss, the rest should hit; so for a sufficiently large block size, all accesses but the first should be hits).
- `associativity` accesses two memory locations alternately. Depending on the cache configuration, this could result in all 4 accesses being misses, only the first two being misses, or only the first one being a miss.
- `lru` tests LRU eviction. If you run with a cache size of 4, a block size of 1 and an associativity of 4, the first four accesses should miss, the next one should hit (making 1 the LRU block), the next one should miss, and the last three should hit, indicating the LRU block got evicted.
- `write_back` tests write-back vs. write-through behavior. The number of memory writes will vary depending on that, and you should also check the data values to ensure dirty blocks are written back correctly.
- `write_allocate` tests write-allocate vs. write-around behavior, which will cause the second access to either be a hit or a miss.

We've also provided a large trace (`gcc-trace`) and a prefix of it (`gcc-trace.verbose`) to verify your simulator's behavior. Expected outputs for a variety of different cache configurations are provided in the `test_cases` directory in the files `gcc-trace.output` and `gcc-trace.verbose.output`. These files contain the commands to run each configuration, and you can then use `diff` to compare your output to the expected output.