CS-UY 2214 — Project 2

Ratan Dey,Jeff Epstein

1 Introduction

This project represents a substantive programming exercise. Like all work for this class, it is to be completed individually: any form of collaboration is prohibited, as detailed in the syllabus. This project is considered a take-home exam.

Before even reading this assignment, please read the E20 manual thoroughly. Read the provided E20 assembly language examples.

If you have not completed your E20 simulator, please do so before beginning this assignment.

2 Assignment: Cache simulator

Your task is to write an E20 cache simulator: a program that will monitor the effects that an E20 program's execution has on a memory cache. This project will build on your E20 simulator, by adding a simulated cache subsystem.

Your simulator will be given a *cache configuration*, specifying the number and type of caches. Your simulator will support up to two caches. Therefore, your program should be prepared to accept a cache configuration with either just an L1 cache, or both an L1 and an L2 cache.

Each cache will be defined by three parameters, given as integers: the size of the cache, the cache's associativity, and the cache's blocksize (measured in number of cells):

cache size	The total size of the cache, excluding metadata. The value is expressed as a multiple of the size of a memory cell. This will be a positive integer power of 2 that will be at least the product of the associativity and blocksize.
cache associativity	The number of blocks per cache row. This will be an integer in the set $\{1, 2, 4, 8, 16\}$. An associativity of 1 indicates a direct-mapped cache. If the product of the associativity and the blocksize is equal to the cache size, and the associativity is greater than 1, then the cache is fully associative.
cache blocksize	The number of memory cells per cache block. This will be an integer in the set $\{1, 2, 4, 8, 16, 32, 64\}$.

Your simulator will monitor the effects on the cache or caches while executing an E20 program. The output of your simulator will be a log of all cache hits and cache misses, in chronological order, caused by all executed `lw` and `sw` instructions.

- When an `sw` instruction is executed, the log will be appended with a line indicating so, along with the current value of the program counter, the memory address that was written, and the cache row where the data was cached. The format is as follows:

```
L1 SW    pc:    2   addr:  100  row:   4
```

If there are two caches, then the write to both affected caches should be logged separately, as follows:

```
L1 SW    pc:    2   addr:  100  row:   4
L2 SW    pc:    2   addr:  100  row:   0
```

In the above case, the data was written to both L1 and L2, as dictated by a write-through policy.

- When an `lw` instruction is executed, the log will be appended with a line indicating so, along with whether the memory read was a hit or a miss, the current value of the program counter, the memory address that was written, and the cache row where the data was cached. The format is as follows, in the case of a miss:

```
L1 MISS  pc:   22   addr:  128  row:   4
```

  Alternatively, in the case of a hit, you should indicate that there was a hit:

```
L1 HIT   pc:   22   addr:  128  row:   4
```

  If there are two caches, then we will consult the L2 cache only when the L1 cache misses. In that case, the access to both affected caches should be logged separately, as follows:

```
L1 MISS  pc:   22   addr:  128  row:   4
L2 HIT   pc:   22   addr:  128  row:   0
```

  In the above case, the read was a miss on L1, but hit on L2. Another situation to consider is when the read misses on both caches.

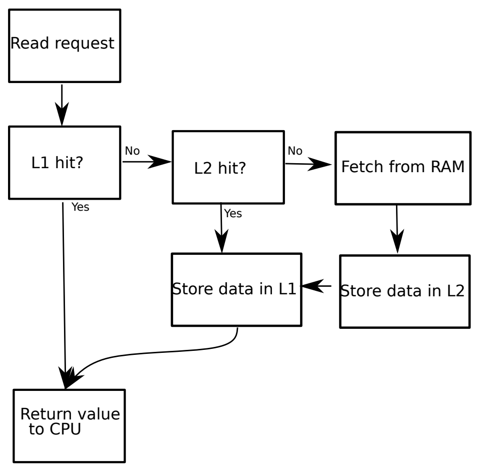
Instructions that do not access memory (such as `addi`, `j`, etc) will not be logged.

For the purposes of this project, we assume that instruction fetches bypass the caches, and therefore will not be logged. In a real computer, reading instructions from memory, just like reading data, would be cached.

The initial value of all memory cells and all registers is zero. The initial state of all caches is empty.

If both caches are present, the blocksize of L1 will not be larger than the blocksize of L2.

For memory reads, if the desired cell is already in a cache, it will result in a hit. When there are two caches, a memory read will access the L2 cache only when the L1 cache misses. Consider the following diagram describing the logic:



For memory writes, your program will use a *write-through* with *write-allocate* policy in all cases. That is, a memory write will simultaneously write the value to all caches, as well as memory. Effectively, writes are handled by the caches as if they were misses.

For associative caches, your program should use the *least-recently-used* (LRU) replacement policy. This policy is relevant when an associative cache row is full, and a memory access causes a new block to be cached to that row, so we need to evict one of the blocks to make room for it. The block that is to be removed is always the block that has been used least recently. Here, "used" means read (via a cache hit), written (via a cache miss), or written (via a memory write). Therefore, whenever you access a cache, you will need to update bookkeeping to keep track of the order in which blocks have been used.

2.1 Input

The input to your simulator will be the name of an E20 machine language file, given on the command line. By convention, E20 machine language files have an `.bin` suffix. In addition, the command line will specify the cache configuration.

The cache configuration will be specified via a `--cache` command-line parameter, which is to be followed by a comma-separated sequence of either 3 or 6 integers. For example, `--cache 2,1,2` or `--cache 2,1,1,8,4,2`. The interpretation of this parameter is as follows:

- If three integers are given, they specify, in order, the total size, associativity, and blocksize of the L1 cache. There is no L2 cache.
- If six integers are given, they specify, in order, the total size of the L1 cache, the associativity of the L1 cache, the blocksize of the L1 cache, the total size of the L2 cache, the associativity of the L2 cache, and the blocksize of the L2 cache.

Here is an example of an E20 machine language program, in a file named `test.bin`, which was produced by assembling the file `test.s`:

```
ram[0] = 16'b0010000010000110;         // movi $1,foo
ram[1] = 16'b1000010100000000;         // lw $2,0($1)
ram[2] = 16'b1000010110000001;         // lw $3,1($1)
ram[3] = 16'b0000100111000000;         // add $4,$2,$3
ram[4] = 16'b1010001000000110;         // sw $4,foo($0)
ram[5] = 16'b0100000000000101;         // halt
ram[6] = 16'b0000000000001010;         // foo: .fill 10
ram[7] = 16'b0000000000010100;         // .fill 20
```

2.2 Output

Your program should print to stdout a summary of the cache configuration, followed by a line for each cache event (hit, miss, or write). Your program should *not* print out the final state of registers and memory, as was done previously.

Below we give several invocations of the cache simulator from Linux's `bash`. In each case, we are executing the E20 machine code file given above, but with a different cache configuration each time. Text in italics represents a command typed by the user.

Below, we simulate a machine with a single cache. The total size is 2, the associativity is 1, and the blocksize is 1 cell (i.e. 16 bits, since E20 cells are 16 bits). We can therefore conclude that this is a direct-mapped cache with 2 rows.

Examine the format of the output below. Note that we start with a line summarizing the structure of the cache, following by a sequence of log lines. Note that each log line indicates the program counter of the `lw` or `sw` instruction causing the memory access. Each log line also indicates the address being accessed, as well as the cache row. The program counter displayed should correspond to the value of the 16-bit program counter register, expressed as an unsigned decimal number. The address displayed should correspond to the actual 13-bit memory location being read or written, expressed as an unsigned decimal number.

The program reads from two memory locations, 6 and 7. Both miss. The first is cached to row 0, then the second to row 1. Then we do a write, overwriting the cached value in row 0.

```
user@ubuntu:~/e20$ ./simcache.py test.bin --cache 2,1,1
Cache L1 has size 2, associativity 1, blocksize 1, rows 2
L1 MISS  pc:    1   addr:    6  row:   0
L1 MISS  pc:    2   addr:    7  row:   1
L1 SW    pc:    4   addr:    6  row:   0
```

This next example is similar to the previous, except that we've expanded the blocksize to 2. The total size of the cache has not changed, so this cache has only one row.

Now, when the program reads from address 6, the miss causes both address 6 and 7 to be cached. The subsequent read from 7 is therefore a hit.

```
user@ubuntu:~/e20$ ./simcache.py test.bin --cache 2,1,2
Cache L1 has size 2, associativity 1, blocksize 2, rows 1
L1 MISS  pc:    1   addr:    6  row:   0
L1 HIT   pc:    2   addr:    7  row:   0
L1 SW    pc:    4   addr:    6  row:   0
```

In the example below, we simulate a machine with two caches. The L1 cache has a total size of 2, associativity of 1, and a blocksize of 1; thus, it is a direct-mapped cache with two rows. The L2 cache has a total size of 8, associativity of 4, and a blocksize of 2; it is thus a fully-associative cache with four blocks.

When the program reads address 6, it misses on both caches. Because the L2 cache has larger blocks, the miss causes address 7 to be cached in the L2, but not in the L1. Therefore, when the program reads address 7, it misses on L1 but hits on L2.

When the program does a write, it is performed on both caches, as well as memory, because we're using write-through.

```
user@ubuntu:~/e20$ ./simcache.py test.bin --cache 2,1,1,8,4,2
Cache L1 has size 2, associativity 1, blocksize 1, rows 2
Cache L2 has size 8, associativity 4, blocksize 2, rows 1
L1 MISS  pc:    1   addr:    6  row:   0
L2 MISS  pc:    1   addr:    6  row:   0
L1 MISS  pc:    2   addr:    7  row:   1
L2 HIT   pc:    2   addr:    7  row:   0
L1 SW    pc:    4   addr:    6  row:   0
L2 SW    pc:    4   addr:    6  row:   0
```

This final example below is similar to the previous, but now both caches have a blocksize of 2.

The read of address 6 still misses. The read of address 7 hits on L1, and therefore the L2 cache is not consulted.

```
user@ubuntu:~/e20$ ./simcache.py test.bin --cache 2,1,2,8,4,2
Cache L1 has size 2, associativity 1, blocksize 2, rows 1
Cache L2 has size 8, associativity 4, blocksize 2, rows 1
L1 MISS  pc:    1   addr:    6  row:   0
L2 MISS  pc:    1   addr:    6  row:   0
L1 HIT   pc:    2   addr:    7  row:   0
L1 SW    pc:    4   addr:    6  row:   0
L2 SW    pc:    4   addr:    6  row:   0
```

Please make sure you understand the above examples before starting to code. Your simulator should be able to reproduce them exactly. Your simulator should produce output in exactly the format shown above.

Your solution will be checked mechanically, so it is important that your simulator produce output identical to the output above. Please avoid losing points for superficial deviations.

2.3 Testing

Several example machine code files have been provided for you. Each example file includes the expected execution result. You can use these examples to verify the correctness of your simulator. However, you should not rely exclusively on these examples, as they are not sufficient to exercise every aspect of a simulator. You are therefore expected to develop your own test cases.

2.4 Starter code

The bulk of the starter code for this project is your completed E20 simulator. If you have not yet completed that project, please do so before beginning this one.

In addition, I provide some helpful code you can use in the files `simcache-starter.cpp` and `simcache-starter.py`. In particular, this file contains three things that you should integrate into your simulator:

- A revised `main` function that handles parsing of the `--cache` command-line parameter. It will provide you with variables `L1size`, `L1assoc`, `L1blocksize`, `L2size`, `L2assoc`, and `L2blocksize`, which you can use in constructing your cache data structure.
- A `print_cache_config` function that will print a correctly-formatted cache configuration line. As discussed above, when your simulator starts, it should first print out the configuration of all caches, in the following format:

  ```
  Cache L1 has size 2, associativity 2, blocksize 1, rows 1
  Cache L2 has size 32, associativity 1, blocksize 4, rows 8
  ```

  The values for size, associativity, and blocksize are given to your program as command-line parameters. Your program needs to calculate the number of rows.
- A `print_log_entry` function that will print a correctly-formatted log entry. As discussed above, for every cache access, your simulator should print out the details of that access, in the following format:

  ```
  L1 MISS  pc:    6   addr: 1024  row:   0
  L2 MISS  pc:    6   addr: 1024  row:   0
  ```

3 Hints

- In order to run a Python program from the Linux command line, it must first be marked executable. Otherwise, you may get a "permission denied" error message.

  To mark your Python file as executable, use the following command (assuming your file is named `simcache.py`) from `bash`:

  ```
  chmod u+x simcache.py
  ```

  Also make sure that the first line of the file specifies the path to the Python interpreter: it should be `#!/usr/bin/python3`. See the provided starter code. If you get an "exec format error," the problem is usually that that the first line is wrong.

  Alternatively, you can run the program by typing `python3 simcache.py`.

- Your program must access its command-line parameters in order to know the name of the machine code file. In Python, you can use `sys.argv[1]`, although I recommend you use the `argparse` library, as shown in the starter code. In C++, you should use the `argv` parameter to `main`.

- The use of strings to store binary numbers is *strongly discouraged.* In short: numbers are numbers and should be stored as such; converting numbers to strings for the purpose of bit-extraction and bit-twiddling will lead to inflexible, unreadable code, and may introduce unanticipated bugs.

  Please review the hint from the earlier project, in which we advise you of techniques to manipulate bits in a number without resorting to strings.

- Recall that the addition of a cache subsystem may not interfere with the execution of the program. That is, any E20 program that we run on your cache-enhanced simulator should produce exactly the same results (in terms of register values and memory cell values) as the same program running on your simulator without cache. If you find that the results differ, then something is broken. Please test your simulator thoroughly.

- The first thing your simulator will do is build the data structures necessary to implement the cache. Your data structure will probably be a table similar to the tables representing caches that we discussed in class. The table will be indexed by row number, and will need to store tags. In addition, for associative caches, your data structure will need to keep track of the order of use of elements in each set.

  Because all we care about are the hits and misses, we do not actually need to store data from memory: note that the required output of your simulator contains no information about the actual data values loaded or stored, but rather it contains only addresses. Therefore, all you need to implement is the cache tags. The tags will identify the cache contents, which is enough to tell if a cache hit or miss has occurred. You can assume that, on a cache miss, the requested data will be loaded into the cache.

- Note that since this is a simulator, we don't expect any actual performance improvement over your uncached simulator. Instead, the purpose of this project is to build a tool that would allow us to explore potential performance improvements. Such a tool could help design a performant cache subsystem, or help us write programs that take advantage of the cache subsystem.

- Your simulator should never crash for any valid input. This is because your simulator should reproduce the behavior of a hardware component that *cannot* crash.

  In this case, "valid input" means a machine-code program (i.e. a sequence of 16-bit values) such that no invalid instruction is ever executed.

- If you are using C++, remember to initialize all variables. Uninitialized variables may have an arbitrary value that varies between executions or environments, resulting in apparently "random" behavior when the program is run.

4 Rules

Language You should implement this project in Python 3 or in C++.

File names and building If you are using Python 3, you must name your program `simcache.py`. If your solution consists of multiple source files, submit them as well. Assume that your program will be invoked by running `simcache.py` with a filename as its first parameter, followed by the `--cache` flag and configuration, as specified above, using Python 3.6.

If you are using C++, you must name your program's main source file `simcache.cpp`. If your solution consists of multiple source files, submit them as well. Assume that your program will be built by gcc 8.3.x using the command `g++ -Wall -o simcache *.cpp` and then run by the executable `simcache` with a filename as its parameter, followed by the `--cache` flag and configuration, as specified above. If you use C++, your program should compile cleanly (i.e. no errors or warnings) with gcc 8.3.x.

Libraries You are free to make use of all packages of the standard library of your language (that is, all libraries that are installed by default with Python 3 or C++, respectively). Do not use any additional external libraries. Do not use any OS-specific or compiler-specific extensions.

Tools Your program submission will be evaluated by running it under the GNU/Linux operating system, in particular a Debian or Ubuntu distribution. Your grade will therefore reflect the behavior of your project code when executed in such an environment. While you are welcome to develop your project under any operating system you like (such as Windows or Mac OS), you are responsible for any operating system-dependent deviations in program behavior.

Academic integrity You should write this assignment entirely on your own. You are specifically prohibited from submitting code written or inspired by someone else. Code may not be developed collaboratively. You may rely on publicly-accessible documentation of the language and its libraries. Please read the syllabus for detailed rules and examples about academic integrity.

Code quality You should adhere to the conventions of quality code:

- Indentation and spacing should be both consistent and appropriate.
- Names of variables, types, fields, and functions should be descriptive. Local variables may have short names if their use is clear from context.
- All functions should have a documenting comment in the appropriate style describing its purpose, behavior, inputs, and outputs. In addition, where appropriate, code should be commented to describe its purpose and method of operation.
- Your code should be structured to avoid needless redundancy and to enhance maintainability.

In short, your submitted code should reflect professional quality. Your code's quality is taken into account in grading your work.

Submission You are obligated to write a `README` file and submit it with your assignment. The `README` should be a plain text file (not a PDF file and certainly not a Word file) containing the following information:

- Your name and NYU email address.
- The state of your work. Did you complete the assignment? If not, what is missing? Be specific. If your assignment is incomplete or has known bugs, I prefer that students let me know, rather than let me discover these deficiencies on my own.

- Any other resources you may have used in developing your program.
- Justify your design decisions. Why did you write your program the way you did? If you feel that your design has notable strengths or weaknesses, discuss them.

Submit your work on Gradescope. Submit all source files necessary to build and run your project. Do not submit external library code. Do not submit binary executable files.