Project 4: Cache Organization (10%)

ENEE 350: Computer Organization, Fall 2009
Assigned: Thursday, Nov 19; Due: Tuesday, Dec 8

Purpose

This project is intended to help you understand in detail how *caches* work by building a cache model that processes an *address trace*. This project also returns to the topic of performance measurement, so your simulator will also keep track of performance numbers and support different design options. For instance, your program should support caches of different sizes, block sizes, and associativities. Your simulator must maintain statistics during execution to report performance numbers when it finishes processing the address trace.

Problem

You will implement a cache system that emulates both a *Harvard architecture* (separate instruction and data caches, also called a *split cache architecture*) or a *unified architecture* (one big cache that holds both instructions and data). The choice for a particular simulation will be made on the command line. All of your caches should be **write-through**, **write-allocate** (technically, this only applies to data caches, since you cannot explicitly write to an instruction cache). All associative caches will use an **NMRU** replacement policy: to replace a block, keep track of the most recently used block in the set, and choose among the remaining blocks randomly. The cache **size** should be variable, as should the cache's **blocksize** and the cache's **associativity**; these will be variables passed in on the command-line.

You are to keep statistics on the following event types:

- instruction and data cache references
- instruction and data cache misses and miss rates
- total cycles required to execute program (assuming a 1-cycle hit and a 100-cycle miss)

Output these at the end of the simulator's execution, using the provided **printStats** function.

Address Traces

An *address trace* represents the dynamic behavior of a program. Typically, an address trace is a stream of data items, each of which contains the informaiton for one instruction. Here is an example stream:

```
1070016
32832           0       14894476
2129920         1       14894545
1900288
1859360
1403456
1403520         0       14900052
1479936         0       14901872
1408128         0       14903660
2456704         0       14903729
826560          1       14904253
1411392
```

The first column represents the address of the instruction (the program counter), the next column represents whether the instruction is a read (0) or a write (1), and the last column represents the data

address that the instruction reads or writes. If a line does not have an entry in columns 2 and 3, then it is not a load/store instruction, and it therefore only references the instruction cache.

Running and Demonstrating Your Program

Your program should be run using the following command format:

```
cachesim isize iassoc iblock dsize dassoc dblock < input > output
```

where the arguments are as follows:

- **isize** is the size of the instruction cache, in Kbytes: support 1, 4, 8, 16, 32, 64, or 128 Kbytes
- **iassoc** is the associativity of the instruction cache: support 1, 2, 4, 8, or 16-way
- **iblock** is the blocksize of the instruction cache, in bytes: support 8, 16, 32, or 64 bytes
- **dsize** is the size of the data cache, in Kbytes: support 1, 4, 8, 16, 32, 64, or 128 Kbytes
- **dassoc** is the associativity of the data cache: support 1, 2, 4, 8, or 16-way
- **dblock** is the blocksize of the data cache, in bytes: support 8, 16, 32, or 64 bytes

If *dsize* is 0, then you should implement a unified cache architecture; if *dsize* is non-zero, then you should implement a split cache architecture.

The first thing your program will do is build the data structures necessary to implement the cache. Because all we care about are the hit/miss rates, we do not actually need to store any data—note that the address trace above contains no information about the actual data values loaded or stored; it only contains 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.

On every cycle, the cache simulator should read the next line of input from *stdin*, which corresponds to the next instruction in the dynamic execution of the program. The format will be what is illustrated above: the line will contain either one number or three numbers. I will provide you with the code to read a line of input.

Once the address for the instruction is known, you make a request to the instruction cache. If the instruction also access the data cache, you make a request to the data cache. On every access, keep track of whether the access hit the cache or not. At the end of the address trace, you will print out the cache statistics.

Grading and Formatting

We will grade solely on functionality. As with the previous project, there will be an autograder running that will let you know if your cache provides the correct answers.

Submitting Your Program

Just like Project 3: use the *submit* utility to send in your C file.