MIPS Simulator: Data Cache
Project 5 – CS 3339 – Spring 2016

Due: Wednesday, 4/13/16 @ noon
100 points

PROBLEM STATEMENT

In this project, you will further enhance your simulator to model pipeline stalls due to memory latency, and you will simulate a data cache in order to study the performance impact of caching. You should begin with a copy of your Project 4 submission. Additionally, I have provided a class skeleton for a CacheStats class intended to be instantiated inside your existing CPU class. You may add any functions, function parameters, etc. to it that you want.

The simulated data cache should store 1 KiB (1024 Bytes) of data in block sizes of 8 words (32 bytes). It should be 4-way set associative, with a round-robin replacement policy and a write policy of write-back write-allocate. All blocks in the cache are initially invalid. For simplicity, assume the cache has no write buffer: a store must be completely finished before the processor can proceed. Since this is a data cache, only loads and stores access it; instruction fetches should still be assumed to hit in a perfect I-cache with immediate access (i.e., there is never a stall for an instruction fetch).

Note that since you only have to model the hit/miss/timing behavior of the cache, you do not have to actually store any data in your cache model. It is sufficient to simulate the valid, tag, and dirty bits as well as the round-robin replacement policy.

Your cache model will calculate and report the following statistics:

- The total number of accesses, plus the number that were reads vs. writes
- The total number of misses, plus the number caused by reads vs. writes
- The number of writebacks
- The hit ratio

Every time an access is made to the cache model, the cache model should return the number of cycles that the processor must stall in order for that access to complete. It takes 0 cycles to hit in the cache (i.e., data will be returned or written immediately). A read access to the next level of the memory hierarchy (e.g., main memory) has a latency of 30 cycles, and a write access has a latency of 10 cycles. Note that an access resulting in the replacement of a dirty line requires both a main memory write (to write back the dirty block) and a read (to fetch the new block) consecutively. Because the cache has no write buffer, all stores must stall until the write is complete.

Before computing the cache statistics, be sure to drain all the dirty data from the cache, i.e. write it back. Count these as writebacks, but do not count any stalls/latency resulting from them.

Inside your Stats class, add a new function similar to the bubble() and flush() functions. This stall() function should stall the entire pipeline for a specified number of cycles. The Stats class should track the total number of stall cycles that occur during program execution.

Your simulator will report the following statistics at the end of the program:
• The exact number of clock cycles it would take to execute the program on this CPU
• The CPI (cycle count / instruction count)
• The number of bubble cycles injected due to data dependencies (unchanged from Proj. 4)
• The number of flush cycles in the shadows of jumps and taken branches (also unchanged)
• The number of stall cycles due to cache/memory latency
• The data cache statistics reported by the cache model

I have provided the following new files:
• A CacheStats.h class specification file, which you'll need to add member variables to
• A CacheStats.cpp class implementation file, which you should enhance with code to
model the described cache and count accesses, misses, and writebacks
• A new Makefile

In addition to enhancing the CacheStats.h/.cpp skeleton, you will need to modify your existing
Stats.h/Stats.cpp in order to implement memory stalls. You will also need to modify CPU.h
to instantiate a CacheStats object, and CPU.cpp to call both CacheStats and Stats class functions
appropriately to model cache behavior and resulting pipeline stalls. You’ll also need to change
CPU::printFinalStats() to match my expected output format (see below).

ASSIGNMENT SPECIFICS

All provided files are on TRACS under Resources → Project Files → cs3339_project5.tgz.

Begin by copying all of your Project 4 files into a new Project 5 directory. Then extract the
additional files in cs3339_project5.tgz and add them to your project5 directory. You can compile
and run the simulator program identically to previous projects, and test it using the same *.mips
inputs. (Only sssp.mips yields interesting cache behavior; I’ll only test your code with this one).

If you examine CacheStats.h, you’ll notice that I’ve already defined constants for you for all of
the cache configuration options you’ll need (e.g., number of sets, number of ways, block size, read
miss latency, etc.). You should not need to change any of these defines. (They are defined to be
modifiable from the compilation command line, but for this project, I will not change any of them.
Stay tuned for an extra credit assignment that will explore other cache configurations).

The following is the expected result for sssp.mips. Your output must match this format verbatim:

CS 3339 MIPS Simulator
Cache Config: 1024 B (32 bytes/block, 8 sets, 4 ways)
   Latencies: Hit = 0 cycles, Read = 30 cycles, Write = 10 cycles
Running: sssp.mips

7 1

Program finished at pc = 0x400440  (449513 instructions executed)

Cycles: 1396800
CPI: 3.1
Bubbles: 481710
Flushes: 51990
Stalls: 413580
Accesses: 197484
  Reads: 146709
  Writes: 50775
Misses: 12044
  Read misses: 8559
  Write misses: 3485
Writebacks: 5229
Hit Ratio: 93.9%

The CacheStats skeleton already includes code to disable the cache (i.e., all loads result in a read access to the next level of the memory hierarchy, and all stores result in a write access). To explore the performance impact of adding a cache, you can compile your simulator with the cache disabled:

$ make clean; make CACHE_EN=0

Re-run the simulator on sssp.mips. What happens to the CPI? How big is the difference?

Additional Requirements:
- Your code must compile with the given Makefile and run on zeus.cs.txstate.edu
- Your code must be well-commented, sufficient to prove you understand its operation
- Make sure your code doesn’t produce unwanted output such as debugging messages. (You can accomplish this by using the D(x) macro defined in Debug.h)
- Make sure your code’s runtime is not excessive
- Make sure your code is correctly indented and uses a consistent coding style
- Clean up your code before submitting: i.e., make sure there are no unused variables, unreachable code, etc.

SUBMISSION INSTRUCTIONS

Submit all of the code necessary to compile your simulator (all of the .cpp and .h files as well as the Makefile) as a compressed tarball. You can do this using the following Linux command:

$ tar czvf project5.tgz *.cpp *.h Makefile

Do not submit the executables (*.mips files). Any special instructions or comments to the grader, including notes about features you know do not work, should be included in a separate text file (not inside the tarball) named README.txt.

All project files are to be submitted using TRACS. Please follow the submission instructions here:
http://tracsfacts.its.txstate.edu/trainingvideos/submitassignment/submitassignment.htm
Note that files are only submitted if TRACS indicates a successful submission.

You may submit your file(s) as many times as you’d like before the deadline. Only the last submission will be graded. **TRACS will not allow submission after the deadline**, so I strongly recommend that you don’t come down to the final seconds of the assignment window. Late assignments will not be accepted.