2020 01 – CS 3853 Computer Architecture

Group Project: Cache Simulator

Final Project Due: Thu May 7th, 2020 7:30 pm

NO LATE ASSIGNMENTS ACCEPTED

1 Objectives

The goal of this project is to help you understand the internal operations of CPU caches. You are required to simulate a Level 1 cache for a 32-bit CPU. Assume a 32-bit data bus. The cache must be command line configurable to be direct-mapped, 2-way, 4-way, 8-way, or 16-way set associative and implement both round-robin and random replacement policies for performance comparisons. You may implement LRU in lieu of either of the others.

2 General Project Descriptions

2.1 Groups

You may work in groups of 2 to 3(preferred) students. This project requires coding, testing, documenting results and writing the final report. Everyone must contribute to receive credit. Anyone not contributing will either have to finish their own project by the due date or receive a zero. Note: NO WORK OF ANY KIND IS ACCEPTED AFTER May 7th, 2020 at 7:30pm – except the final exam.

2.2 Programming Languages and Reference Systems

You are allowed to use any of the following programming languages: Python, C/C++ and Java.

2.3 Simulator Input and Memory Trace Files

Your simulator will have the following input parameters:

1.	–f <trace file name>	[name of text file with the trace]
2.	–s <cache size in KB>	[1 KB to 8 MB]
3.	–b <block size>	[4 bytes to 64 bytes]
4.	–a <associativity>	[1, 2, 4, 8, 16]
5.	–r <replacement policy>	[RR or RND or LRU] ← Implement any two of these

Sample command lines:

```
Sim.exe –f trace1.txt –s 1024 –b 16 –a 2 –r RR
```

That would read the trace file named "trace1.txt", configure a total cache size of 1 MB with a block size of 16 bytes/block. It would be 2-way set associative and use a replacement policy of Round Robin. We will assume a write-through policy. **Cost: $0.05 / KB**

2.5 Simulator Outputs

Your simulator should output the simulation results to the screen (*standard out*, or *stdout*). The output should have a short header formatted as follows:

Cache Simulator CS 3853 Spring 2020 – Group #nn (where "nn" is your group number)

Demonstrate you have correctly parsed the command line by printing these individually. This should not be a replay of the command line, but rather the numeric interpretation for cache and block size, and associativity.

Trace File: <name of trace file>
Cache Size: <size typed in KB>
Block Size: <size typed in bytes>
Associativity: <direct, 2-way, 4-way, etc.>
R-Policy: <characters typed in>

MILESTONE #1: Input Parameters and Calculated Values

```
Cache Simulator - CS 3853 - Team XX

Trace File: Trace2a.trc

***** Cache Input Parameters *****

Cache Size:                     1024 KB
Block Size:                     16 bytes
Associativity:                  2
Replacement Policy:             Round Robin

***** Cache Calculated Values *****

Total # Blocks:                 65536
Tag Size:                       13 bits
Index Size:                     15 bits
Total # Rows:                   32768
Overhead Size:                  114688 bytes
Implementation Memory Size:     1136.00 KB (1163264 bytes)
Cost:                           $56.80
```

MILESTONE #2: - Simulation Results

```
***** Cache Simulation Results *****

Total Cache Accesses:   282168
Cache Hits:             275383
Cache Misses:           6785
--- Compulsory Misses:     6625
--- Conflict Misses:       160


***** *****  CACHE MISS RATE:  ***** *****

Hit Rate:               97.5954%
CPI:                    4.14 Cycles/Instruction
Unused Cache Space:     920.48 KB / 1024 KB = 89.89 %  Waste: $46.02
Unused Cache Blocks:    58911 / 65536
```

4 Trace Files

I will provide several trace files for testing which will be formatted as shown below. The trace file contains an execution trace from a real program execution. At least one trace file will be very short so that you can manually determine the miss rate.

The trace files provided contain two lines – the instruction fetch line and the data access line. The instruction fetch line contains the length of the instruction (number of bytes read), the 32-bit hexadecimal address, the machine code of the instruction, and the human-readable mnemonic.

The data access line shows addresses for destination memory "dstM" (i.e. the write address) and source memory "srcM" (i.e. the read address) and the data read. ASSUME all data accesses are 4 bytes.

For the instruction line, you need the length and the address. For the data line, the length is given as 4 bytes and you need the dst/src addresses. Additionally, if the src/dst address is zero, then **<u>IGNORE</u>** it – that means there was no data access. SPECIAL NOTE: The destination write actually appears on the next line. If there is a mov [memory address], eax, that destination address will appear on the next line. That doesn't matter. Process each address access independently.

4.0 Sample Trace File Format:

Below is an example of what two blocks in the trace file look like. EIP is the Extended Instruction Pointer – this identifies the memory that is read containing the instruction. The number in parenthesis (highlighted in green) is the length of that instruction. The next number, highlighted in yellow is the numeric address containing the instruction.

The next set of hex digits are the actual machine code for this particular instruction – in other words, this is the data actually read. For our cache simulator, we <u>do not care</u> about the data so it should be ***ignored***.

```
EIP (04): 7c809767 83 60 34 00  and dword [eax+0x34],0x0
dstM: 00000000 --------    srcM: 00000000 --------        ← IGNORE these!!!

EIP (07): 7c80976b 8b 84 88 10 0e 00 00  mov eax,[eax+ecx*4+0xe10]
dstM: 7ffdf034 00000000    srcM: 7ffdfe2c 901e8b00
```

The above has 4 address accesses: 4 bytes read at 0x7c809767, 7 bytes read at 0x7c80976b, 4 bytes written at 0x7ffdf034, and 4 bytes read at 0x7ffdfe2c

Note that when an instruction is executed, the dstM is not yet written, so the dstM shown is actually the destination from the prior instruction. For our purposes, ignore that effect and treat it as an access in the same block in which you read it – this will not affect simulation results.

Here is the data that should be processed by your cache simulator for the two instructions above.

Address: 0x7c809767, length = 4. No data writes/reads occurred. <There is a data write by the "and" instruction BUT it is not seen until the next block.>

Address: 0x7c80976b, length = 7. Data write at 0x7ffdf034, length = 4 and data read at 0x7ffdfe2c, length = 4.

4.1 How to Parse:

The file is very structured with the characters at the same line offset for each line. In C/C++, use fgets to read a line into a char array.

`Line[] = "EIP (04): 7c809767", so line[0] = 'E', line[5,6] = "04", and line[10,17] = "7c809767". You will have to convert the character representation of the address into a hex value.  In C/C++, a sscanf("%x"); can do this.`

4.2 CPI CALCULATION:

We will calculate CPI in the following manner: The data bus is 32 bits wide which means we can access four bytes of data simultaneously. We require clock cycles for instruction fetch/decode, instruction execution, effective address/branch calculation, and memory access. For our simulation, reading memory requires 3 clock cycles while reading the cache requires only 1.

Consider the trace example below with the "AND" instruction. In reality, we have to read the memory at 0x7C809767 to fetch the instruction. Then we must read the memory at [eax+0x34] to get the data, AND it with zero, and then write the result back to memory. Also, the WRITE to memory, from the "AND" instruction is depicted on the next line as "dstM: 0x7ffdf034". We ignore that complexity - process the trace line by line.

So in this example, the "AND" instruction has no data reads or writes, but the "MOV" instruction has both a "dstM:" (destination) and a "srcM:" (source) data access. For the simulation, no need to determine that the "AND" performs a read nor that the dstM: goes with the "AND" instruction. Just read the line and process the addresses.

```
EIP (04): 7c809767 83 60 34 00  and dword [eax+0x34],0x0
dstM: 00000000 --------    srcM: 00000000 --------

EIP (07): 7c80976b 8b 84 88 10 0e 00 00  mov eax,[eax+ecx*4+0xe10]
dstM: 7ffdf034 00000000    srcM: 7ffdfe2c 901e8b00
```

To fetch the "and" instruction, 4 memory bytes must be read. This requires 2 clock cycles.

CPI determination:

A. Fetch 4 bytes at 0x7c809767
   a. cache hit : 1 cycle
   b. cache miss : (3 cycles * number of memory reads to populate cache block)
      i. number of reads == CEILING (block size / 4)
      ii. the 4 is because the data bus is 32 bits (i.e. 4 bytes)
   c. NOTE: Multiple cache rows per address possible, 'a' and 'b' apply for each cache row accessed
   d. +2 cycles to execute "AND" instruction (do NOT count effective address time here)
B. Fetch 7 bytes at 0x7c80976b
   a. cache hit: 1 cycle
   b. cache miss: (3 cycles * number of memory reads to populate cache block)
   c. SAME NOTE:
   d. +2 cycles to execute "MOV" instruction
C. Write 4 bytes at 0x7ffdf034
   a. cache hit : 1 cycle
   b. cache miss : (3 cycles * number of memory reads to populate cache block)
   c. +1 cycle to calculate effective address
D. Read 4 bytes at 0x7ffdfe2c
   a. cache hit : 1 cycle
   b. cache miss : (3 cycles * number of memory reads to populate cache block)
   c. +1 cycle to calculate effective address

NOTE: Each address is processed the same way! For an instruction, add 2 cycles. Remember, each address can result in multiple cache rows accessed. When we read 4 bytes at x9767 we REALLY access x9767, x9768, x9769, and x976A. IF we had an 8-byte block, one cache row would be accessed by x9767, and a second cache row would be accessed by x9768 – x976A.

5 Experiment Guidelines

Once the simulator is complete, you will need to simulate multiple different cache parameters and compare results. You may graph them in Excel or some similar software. Below is the minimum required comparisons, but you may do additional ones as desired.

For each trace file provided, apply each associativity with the following parameters:

Cache Sizes: 8 KB, 64 KB, 256 KB, 1024 KB, Block Sizes: 4 bytes, 16 bytes, 64 bytes, Replacement Policy: RR and RND

The total number of simulation runs will be: #trace files * 4 cache sizes * 3 block sizes * 2 replacement policies.

So 24 simulation runs for each trace file. You may automate the execution of your simulator and/or the collation of results. In the report, document the various simulation runs and any conclusions you can draw from it.

THIS IS AN IMPORTANT PART OF THE PROJECT – WILL REQUIRE SOME TIME

ALSO, if you fail to get a working cache simulator by the due date, I will let you use mine to get results and write up the report.

6 Grading

For the code, I will execute your simulator. It's in your best interest to make this as easy as possible for me. For C/C++ projects in Linux, include a makefile. For windows, MAKE SURE to set the **multi-threaded debug option** in Visual Studio.

I will execute it with several small memory traces to test if it can produce the correct cache miss rates. The memory trace files used in grading have the exact same format as the provided trace files.

The report will be graded based on the quality of implementation, the complexity and thoroughness of the experiments, and the quality of the writing. Individual grade will be negatively affected if a student does not exhibit a fair share of contribution.

7 Submission Schedule

This project is divided into 3 parts with 3 due dates. Each ***due date is very strict*** and a late turn-in results in a ***ZERO*** for that portion.

(30 pts) Milestone #1 – Input parameters, Calculated Values, and parsing the trace file.

DUE: Mon Apr 6th, 11:59pm. Blackboard.

Upload a .zip file with the following name to blackboard:

"2020_01_CS3853_Team_XX_M#1.zip"

TEN points deducted for incorrect name. XX is your team number.

The .zip file should include a copy of your source code and output files from 3 different runs on Trace1.trc using different parameters each time. The output files should have all the header information printed as described in Section 2.5 (**MILESTONE #1: Input Parameters and Calculated Values**) and be named "O#.txt" where # = 1, 2, 3.

Additionally, print the first 20 addresses and lengths formatted like this:

0xhhhhhhhh: (xxxx)

Where the "hhhhhhhh:" is the hexadecimal address and the (xxxx) contains the length of the read in decimal.

(70 pts) Milestone #2 – The Cache Simulator program.
DUE: Mon Apr 27th, 11:59 pm. Blackboard
Upload a .zip file with the following name to blackboard:

"2020_01_CS3853_Team_XX_M#2.zip"

TEN points deducted for incorrect name. XX is your team number.

The zip file should contain your source code and 5 runs for the "A-9_new_1.5.pdf.trc" file showing your results as shown in Section 2.5 (**MILESTONE #2: - Simulation Results**). INCLUDE Milestone #1 Calculations as well. Do NOT output the addresses and lengths for this milestone.

If you want to more easily graph your results, you are welcome to output a second file that has only the results for importation into Excel. If you do that, make sure to include samples of those output files as well.

Comments on M#2 Outputs:

```
***** Cache Simulation Results *****

Total Cache Accesses:   282168      // how many times you checked an address
Cache Hits:             275383      // it was valid and tag matched
Cache Misses:           6785        // it was either not valid or tag didn't match
--- Compulsory Misses:     6625     // it was not valid
--- Conflict Misses:       160      // it was valid, tag did not match

***** *****  CACHE MISS RATE:  ***** *****

Hit Rate:         97.5954%                // (Hits * 100) / Total Accesses
CPI:              4.14 Cycles/Instruction // Number Cycles / Number Instructions

// Unused KB = (Compulsory Misses * BlockSize) / 1024
// The 1024 KB below is the total cache size for this example
// Waste = COST/KB * Unused KB
Unused Space:     920.48 KB / 1024 KB = 89.89 %  Waste: $46.02
Unused Blocks:    58911 / 65536     // Total Blocks - Compulsory Misses / Total Blocks
```

(50 pts) Milestone #3 – Analysis.
DUE: Thu May 7th, 7:30 pm. Blackboard for softcopy. 7:31pm is LATE for the softcopy. ZERO!
Upload a .zip file with the following name to blackboard:

"2020_01_CS3853_Team_XX_M#3.zip"

TEN points deducted for incorrect name. XX is your team number.

The zip file must contain your final source code and the final analysis report.

For the report, **I am expecting a thorough analysis**. Assume you are making the recommendation to management as to which cache to implement on a new chip based on these trace results. Consider the miss rate, CPI, cost, and how much of the cache is unused (number of blocks never populated). There may be a range of recommendations for different costs/performance.

Writing a few paragraphs and tacking on a few graphs at the end will not cut it. You need to run the simulator using multiple configurations to try to hone in on the properties that will balance cost/performance. The advanced team will automate the execution (scripting for instance) and create an output that can be easily put in Excel for analysis (such as a CSV file).

Let me know if you have any questions. I will be posting my program, once I upgrade it, so you can compare results. If the results are close, you probably have some minor bug and that will not cost a great deal of points. If you can prove to me that I have some type of bug/error, you can get some bonus points.