2021 08 – CS 3853 Computer Architecture

Group Project: Cache & Virtual Memory Simulator

Final Project Due: Wed, Dec 1st, 2021 11:59 pm

NO LATE ASSIGNMENTS ACCEPTED

1 Objectives

The goal of this project is to help you understand the internal operations of CPU caches and implementing a simple virtual memory scheme. You are required to simulate a Level 1 cache for a 32-bit CPU and map virtual memory to physical memory. 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 one of the following replacement policies: round-robin or random, for performance comparisons. The amount of physical memory must be configurable from as low as 64 KB to 512 GB. Virtual address space is 4GB (32 bits)

2 General Project Descriptions

2.1 Groups

You may work in groups of 2 to 3 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.

2.2 Programming Languages and Reference Systems

You are allowed to use any of the following programming languages: Python, C/C++ or 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]
    a. You must accept 1, 2, or 3 files as input
    b. Each file will use a "-f " to specify it
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] ← Implement any one of these
6. –p <physical memory in KB> [64 KB to 512 GB]

```
Sim.exe –f trace1.trc –f trace2.trc –s 512 –b 16 –a 2 –r RR –p 1GB
```

This command line would read the trace file named "trace1.trc", configure a total cache size of 512 KB with a block size of 16 bytes/block. It would be 2-way set associative and use a replacement policy of Round Robin. Physical memory available would be set at 1 GB (Note that in my sample simulation, for cache size I only accept numbers, so for 8MB would have to put in "-s 8192"). We will assume a write-through policy. **Cost of cache memory: $0.15 / 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 Fall 2021– Group #nn (where "nn" is your group number)

MILESTONE #1: Input Parameters and Calculated Values **Due: 11:59 pm Thu, Nov 11th 2021**

```
Cache Simulator - CS 3853 - Instructor Version: 2.02

Trace File: Trace2A.trc

***** Cache Input Parameters *****
Cache Size:                     512 KB
Block Size:                     16 bytes
Associativity:                  8
Replacement Policy:             Random
Physical Memory                 ??? MB

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

Total # Blocks:                 32768
Tag Size:                       16 bits
Index Size:                     12 bits
Total # Rows:                   4096
Overhead Size:                  69632 bytes
Implementation Memory Size:     580.00 KB (593920 bytes)
Cost:                           $75.40
```

MILESTONE #2: - Simulation Results **Due: 11:59 pm Tue, Nov 23rd, 2021**

```
***** CACHE SIMULATION RESULTS *****

Total Cache Accesses:   282168      // how many times you checked an address
Cache Hits:             275489      // it was valid and tag matched
Cache Misses:           6679        // it was either not valid or tag didn't match
--- Compulsory Misses:     6656     // it was not valid
--- Conflict Misses:       23       // it was valid, tag did not match

***** *****  CACHE HIT & MISS RATE:  ***** *****

Hit  Rate:              97.6330%    // (Hits * 100) / Total Accesses
Miss Rate:              2.3670%     // 1 - Hit Rate
CPI:                    4.13 Cycles/Instruction // Number Cycles/Number Instructions

// Unused KB = ( (TotalBlocks-Compulsory Misses) * (BlockSize+OverheadSize) ) / 1024
// The 1024 KB below is the total cache size for this example
// Waste = COST/KB * Unused KB

Unused Cache Space:     462.19 KB / 580.00 KB = 79.69% Waste: $60.08
Unused Cache Blocks:    26112 / 32768

***** VIRTUAL MEMORY RESULTS *****  --- any format OK, similar to above

Starting parameter - Physical memory in MB
How much physical memory was used by each trace?
How many compulsory misses/conflict misses (i.e. page faults)
Optional Parameter - allocate ?? MB of physical memory at start for stack/heap/program
Any other metric you can simulate and think would be interesting - extra points possible
```

4 Trace Files

I will provide several trace files **for testing** which will be formatted as shown below. Other files will be used for Milestone #3 simulations. 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 **IGNORE** 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 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 do not care 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 4 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
```

CPI determination:

A. Fetch 4 bytes at 0x7c809767
   a. cache hit : 1 cycle
   b. cache miss : (4 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: (4 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 : (4 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 : (4 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 for simulation, 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 minimum 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 Milestone #2 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 C/C++ on 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.

(40 pts) Milestone #1 – Input parameters, Calculated Values, and parsing the trace file.
DUE: Thu Nov 11th, 11:59pm. Blackboard.
Upload a .zip file with the following name to blackboard:

"2021_08_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 "Team_XX_Sim_n_M#1.txt", where n = 1, 2, 3.

Additionally, for M#1 only, 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.

(80 pts) Milestone #2 – The Cache Simulator program.
DUE: Tue Nov 23rd, 11:59 pm. Blackboard
Upload a .zip file with the following name to blackboard:

"2021_08_CS3853_Team_XX_M#2.zip"

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

The zip file should ONLY contain your source code and the output for 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**) and be named "Team_XX_Sim_n_M#2.txt", n = 1,2,3,4,5. INCLUDE Milestone #1 Calculations as well. Do NOT output the addresses and lengths for this milestone. Do NOT include trace files in .zip file – doing so costs 10 points.

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.

(60 pts) Milestone #3 – Analysis.
DUE: Wed Dec 1st, 11:59 pm. Blackboard --- NOTE: Use ONLY the 3 large Trace Files for Analysis
Upload a .zip file with the following name to blackboard: TEN points deducted for incorrect name.

"2021_08_CS3853_Team_XX_M#3.zip" XX is your team number.

The zip file must contain the final analysis report.

For the report, **I am expecting a detailed 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. Also suggest an amount of physical memory that would be optimal. You need to run the simulator using multiple configurations to try to hone in on the properties that will balance cost/performance.