2025 01 – CS 3853 Computer Architecture

Group Project: Virtual Memory & Cache Simulator

Final Project Due: Thu, May 8th, 2025 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 a simple virtual memory scheme. You are required to simulate Virtual Memory and a Level 1 cache for a 32-bit CPU. Then analyze the results and do a performance comparison.

2. Groups

This is a group project for teams of 4, although a team of 3 is acceptable. 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.

NEW POLICY for 2025 and beyond. Every team member must contribute on EVERY milestone or receive a zero for that milestone.

3. Programming Languages and Reference Systems

You may use any of the following programming languages: Python or C/C++. For any other languages desired, you must get prior approval from the instructor.

4. Project Specifications

Assume a 32-bit data bus and 32-bit virtual address space. The simulation must be command line configurable.

- Cache Size: 8 KB to 8 MB in powers of 2
  - 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192
- Block Size: 8, 16, 32, 64 byte blocks
- Associativity: direct-mapped, 2-way, 4-way, 8-way, or 16-way set associative
- Replacement Policy: round-robin OR random
  - Physical Memory: 1 MB to 4096 MB in powers of 2 < 1, 2, 4, 8, … 4096>
- Virtual address space is set at 4GB (32 bits)
- Able to handle up to 3 trace files – each represents a different process

4.1. Simulator Input and Memory Trace Files

- –s <cache size - KB> [8 to 16384 KB]
- –b <block size> [8 bytes to 64 bytes]
- –a <associativity> [1, 2, 4, 8, 16]
- –r <replacement policy> [RR or RND] ← Implement one of these
- –p <physical memory - MB> [128 MB to 4 GB]
- -u <% phys mem used by OS> [0% to 100%]
- -n <Instr / Time Slice [1 to 0xFFFFFFFF] = max (Enter -1 for max)
- –f <trace file name> [name of text file with the trace]
    - You must accept 1, 2, or 3 trace files as input
    - Each file will use a "-f " to specify it

4.2. Example

Your program should be a command line/console style program. The following is an example run of the program. This is all one line, but shown below as 2 lines for clarity

```
VMCacheSim.exe -s 512 -b 16 -a 4 -r rr -p 1024 -n 100 –u 75
-f Trace1.trc -f Trace2_4Evaluation.trc –f Corruption.trc
```

This command line would read the trace files named “trace1.trc”, “trace2_4Evalaution.trc”, and “Corruption.trc” configure a 512 KB cache with a block size of 16 bytes/block, 4-way set associative with a replacement policy of Round Robin. Physical memory would be set at 1 GB <equals 1024 MB> (Note: In my sample sim I only accept numbers for cache and physical memory size, so for an 8 MB cache, you would need to enter “-s 8192”). The cache write policy is irrelevant so just assume memory is kept in sync with the cache.

4.3. Simulator Outputs

Your simulator should output the simulation results to the screen (*standard out*, or *stdout*). The output must be formatted as follows the specifics are based on the example above. None of the trace files will use addresses greater than 0x7FFFFFFF, so our page table(s) can be 512 KB entries. Do not output what is in red.

Cache Simulator CS 3853 Fall 2025 – Group #XX (where “XX” is your group number)

MILESTONE #1: Input Parameters and Calculated Values

```
Cache Simulator - CS 3853 - Team #XX

Trace File(s):
        Trace1.trc
        Trace2_4Evaluation.trc
        Corruption1.trc

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

Cache Size:                     512 KB
Block Size:                     16 bytes
Associativity:                  4
Replacement Policy:             Round Robin
Physical Memory:                128 MB
Percent Memory Used by System:  95.0%
Instructions / Time Slice:      100

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

Total # Blocks:                 32768
Tag Size:                       10 bits
Index Size:                     13 bits
Total # Rows:                   8192
Overhead Size:                  45056  bytes
Implementation Memory Size:     556.00 KB  (569344 bytes)
Cost:                           $66.72 @ $0.12 per KB

***** Physical Memory Calculated Values *****

Number of Physical Pages:       32768
Number of Pages for System:     31129     ( 0.95 * 32768 = 31129.6 )
Size of Page Table Entry:       16 bits   (1 valid bit, 15 for PhysPage)
Total RAM for Page Table(s):    3145728 bytes  == 512K entries * 3 .trc files * 16 / 8
```

MILESTONE #2: - Virtual Memory Simulation Results

```
***** VIRTUAL MEMORY SIMULATION RESULTS *****

Physical Pages Used By SYSTEM:  31129     // -u % * total physical pages
Pages Available to User:         1639

Virtual Pages Mapped:          545499
        ------------------------------
        Page Table Hits:       544761     // - the virtual page is already mapped in
                                          //   the page table - a hit!
        Pages from Free:          738     // - # times a virtual page is mapped to a
                                          //   physical page not currently in use
        Total Page Faults:        0       // - # times when no physical page is available
                                          //   and must be swapped with one in use

Page Table Usage Per Process:
------------------------------
[0] Trace1.trc:
        Used Page Table Entries: 132   ( 0.03%)
        Page Table Wasted: 1048312 bytes

[1] Trace2_4Evaluation.trc:
        Used Page Table Entries: 303   ( 0.06%)
        Page Table Wasted: 1047970 bytes

[2] Corruption_1_1.5.pdf.trc:
        Used Page Table Entries: 303   ( 0.06%)
        Page Table Wasted: 1047970 bytes
```

MILESTONE #3: - Cache Simulation Results

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

Total Cache Accesses:   600948 (545499 addresses)     // # times cache row hit
--- Instruction Bytes:  1275390
--- SrcDst Bytes:       507376
Cache Hits:             586957       // it was valid and tag matched
Cache Misses:           13991        // it was either not valid or tag didn't match
--- Compulsory Misses:  13909        // it was not valid
--- Conflict Misses:    82           // it was valid, tag did not match


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

Hit  Rate:              97.76626718%      // (Hits * 100) / Total Accesses
Miss Rate:              2.3282%           // 1 - Hit Rate
CPI:                    4.5824 Cycles/Instruction  (418655) // # Cycles/# Instr
Unused Cache Space:     334.40 KB / 556.00 KB = 60.14%  Waste: $40.13/chip
Unused Cache Blocks:    19708 / 32768

// NOTE: A cache access is any time an address maps to a row.
//       reading 7 bytes and hitting two rows is counted as two accesses, not 7.
// 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
```

5. 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. 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 4 bytes (ALWAYS!) 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.

5.1. Sample Trace File Format:

Below is an example of 3 instructions in a trace file. The first line with EIP (Extended Instruction Pointer), identifies the memory that has the instruction. The number in parenthesis (highlighted in green) is the length of that instruction. The next number, highlighted in yellow is the starting address containing the instruction. The next set of hex digits are the actual machine code for this particular instruction (grey) – this is the data actually read. For our simulator, we do not care about the data so it should be ***ignored***.

The second line contains the possible destination (dstM) and source (srcM) addresses. Those addresses are highlighted in cyan. When they are valid, they are followed by 4 hex digits (the data). If the address is not valid, it is followed by 8 space savers: -------- and should be ignored (because it was never accessed by the program). Assume all dstM and srcM accesses are 4 bytes.

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

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

EIP (03): 01798317 8d 41 07  lea eax,[ecx+0x7]
dstM: 00000000 --------    srcM: fffffffe --------
```

The above set of instructions accesses 5 addresses.

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.

Your simulator should read and process the following from the example above:

```
0x7c809767 : 4 bytes
0x7c80976b : 7 bytes
0x7ffdf034 : 4 bytes
0x7ffdfe2c : 4 bytes
0x01798317 : 3 bytes
```

***NOTE: Lengths are in decimal not hex**

5.2. 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.

EIP should always be valid. An address of all zeros will never be valid, but as you can see above, there are a few data items which have a non-zero “address” yet are not valid. This is because it’s not a value used as an address.

5.3. Implementation Hints:

- For this sim, all virtual addresses are 31 bits
  - MSB is kernel memory on Windows so we will not see it
  - SO, the number of Page Table Entries (PTE) is fixed at 512K
- The Page Table, being an array makes it quick to find the physical address given a virtual address: phys = PageTable[virt]
- Handling a Page Fault requires finding a suitable physical page number to map to the Virtual Address
  - Check if any physical pages are FREE
    - If so, use it
  - Determine which process to give up a physical page
    - Could be your own process (1 trace file or others have completed)
  - Find a physical address in that process to snag
    - One way is to do a linear search of the sparse 512KB Page Table
    - Or create a list of physical addresses used by the Page Table
  - Once you have the physical address, unmap the virtual address
  - Invalidate any cache blocks associated with that physical page
    - Since a page is 4K, there are 4K addresses BUT add by block size
      - Assume 16 byte block, physical page number = 0xC3
      - Range: 0xC3000 to 0xC3FFF, counting by 0x10
      - 256 total, invalidate any you find with matching tag
  - Map that page to new process
- When a process ends (trace is done) must free ALL physical pages and invalidate their cache entries
- Check the cache size vs. physical memory size: cache ALWAYS smaller

5.4. CPI CALCULATION: (Milestone #3)

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
```

5.5. 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

Add 100 cycles for every Page Fault.

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.

6. 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.

7. 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.

8. Submission Schedule

(40 pts) Milestone #1 – ALL Input parameters and Calculated Values

DUE: Thu, Apr 3rd, 2025 → 11:59pm.

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

"2025_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, the executable (if C/C++), and output files from 3 different runs using

"A-9_new_trunk.trc". Choose different parameters for each run.

The output files should have all the header information printed as described in Section 4.3 (**MILESTONE #1: Input Parameters and Calculated Values**) and be named "Team_XX_Sim_n_M#1.txt", where n = 1, 2, 3.

(70 pts) Milestone #2 – The Virtual Memory Simulator

DUE: Thu, April 17th, 2025 → 11:59pm.

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

"2025_01_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, the executable (if C/C++), and the output for 3 runs using any combination of files from the "__Traces4Analysis.zip" archive. Your results should be formatted like those in Section 4.3 (**MILESTONE #2: - Virtual Memory Simulation Results**) and be named "Team_XX_Sim_n_M#2.txt", n = 1,2,3. INCLUDE Milestone #1 Calculations as well.

TEN points deducted for including trace files in your .zip file or for zipping up your entire project folder.

(80 pts) Milestone #3 – The Cache Simulator program + Final Report.

DUE: Thu, May 8th, 11:59 pm. --- NOTE: Use ONLY Trace Files from the "__Traces4Analysis.zip" archive

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

"2025_01_CS3853_Team_XX_M#3.zip" XX is your team number.

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

The zip file must contain the final analysis report, all relevant source code, the executable (if C/C++), and 3 output files showing the cache simulation results from 5 different runs using different parameters.

If the executable is on Linux, then include a make file in lieu of the executable. For java include the jar file(s) as needed to run it without compiling it.

Do NOT INCLUDE trace files and DO NOT zip up the entire project folder – 10 to 20 point deduction!

For the report, 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.