MPCS 52018 - Spring 2025
Project 1: Cache Emulator
Due: April 22 @ 6 pm

Logistics

- In this project, you will build a CPU emulator capable of modelling a simplified memory hierarchy. You are free to implement your emulator in the programming language of your choice.
- Include a README with your name, the assignment, and a discussion of any shortcomings. You will receive more partial credit if you are able to clearly identify problems with your code rather than letting us find them ourselves. Your README should also contain explicit instructions for how to compile and run your code. If using a compiled language, you MUST include a makefile. If using an interpreted language, be sure to specify which language version you are using.
- We must be able to run your code on the CS linux cluster.
- The analysis portion of this assignment is significant and may require some research to complete. Make sure to thoroughly test your code and verify that it is producing correct results before beginning the analysis, and leave enough time after finishing your implementation to collect all of the data you will need and to complete all sections of the analysis. Include a single PDF writeup that contains your responses and tables for all questions in the analysis.

Introduction

In the implementation portion of your assignment, you will build an emulator for a CPU with physical memory addressing and first a single-level and then two-level cache. The cache will simulate several block placement and replacement strategies. The emulator will not measure any absolute times (such as execution time or memory stall time); rather, it will measure events such as cache hits and misses. The emulator will not execute programs written in a high-level programming language; rather, it will execute "programs" consisting of less than a dozen pseudo-assembly instructions, written for your simulator.

In the analysis portion of your assignment, you will then use your emulator to provide performance predictions for running simple algorithms with different cache configurations.

Design Considerations

The design of your emulator will be based on the descriptions in Patterson and Hennessey, *Computer Organization and Design*, 5th or 6th Edition, Chapters 5.3 - 5.4. The considerations can be summed up by the following questions:

1. Block placement: Where can a block be placed in cache?
2. Block identification: How is a block found if it is in cache?
3. Block replacement: Which block should be replaced on a miss?
4. Write strategy: What happens on a write?

1 Emulator Implementation

1.1 Capabilities (45 points)

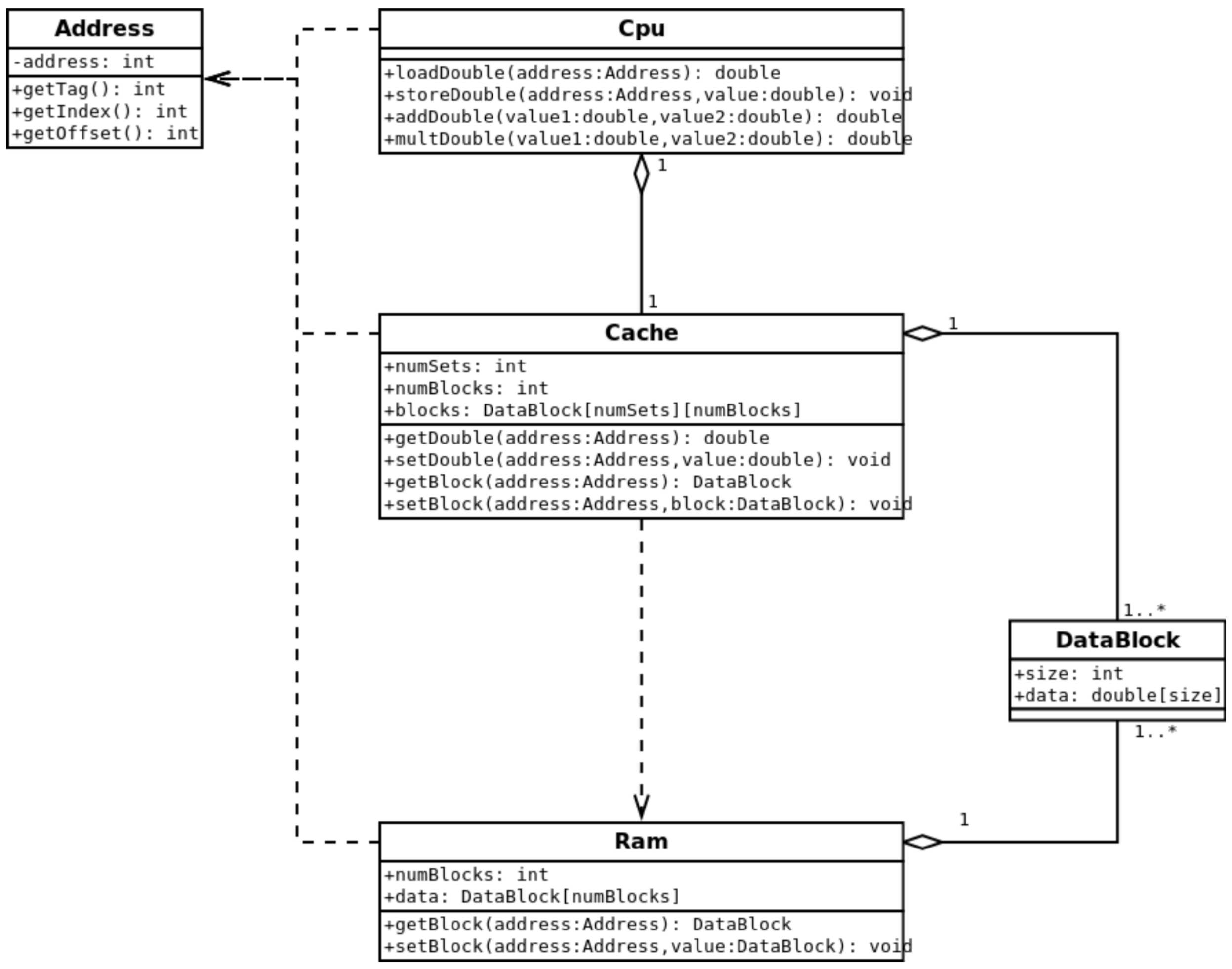
Your emulator must be configurable so it can simulate a variety of architecture designs. The configuration of the hardware architecture and resource sizing will depend on runtime parameters that the user provides. Your emulator should provide the following capabilities:

1. Variable total RAM size
2. Variable total cache size
3. Variable block size
4. Block placement strategy:
   (a) Direct-mapped cache
   (b) Fully associative cache
   (c) n-way associative cache for arbitrary n
5. Block replacement policy:
   (a) Random
   (b) Least Recently Used (LRU)
   (c) First In, First Out (FIFO)
6. Write policy: write-through with write allocate
7. Algorithms:
   (a) Daxpy
   (b) Matrix-Matrix Multiplication
   (c) Matrix-Matrix Multiplication with blocking and variable blocking factor
8. CPU instructions for double precision floating point arithmetic:
   (a) `value3 = addDouble(value1, value2)`
   (b) `value3 = multDouble(value1, value2)`
   (c) `value = loadDouble(address)`
   (d) `storeDouble(address, value)`

1.2 Example Objects and Structures

Your goal is to provide the capabilities above in your emulator so that a user can write and test short "programs" using the CPU instruction interface. There are many ways you could implement the emulation functionality. Figure 1 shows an example model of the objects and data structures that may be helpful to consider for your implementation. You do not need to implement the entire model: you may change the interfaces, add additional objects, or forgo an object-oriented approach completely.

In Figure 1, the `Address` data structure implements a memory address as described in Section 5.3 of Patterson and Hennesy. The `Address` contains the entire address (an integer) as well as methods to access the tag, index, and offset fields; one possible way to retrieve these fields is through bitmasking. The `DataBlock` data structure contains a data block; in this implementation, the data is an array of doubles, but a more refined implementation may contain an array of raw bytes. `DataBlocks` are copied to and from `Cache` and `Ram` via the `getBlock` and `setBlock` methods. The `Cache` organizes its `DataBlocks` into sets, whereas the `Ram` simply contains a flat array of `DataBlocks`. Finally, the `Cpu` provides methods to load and store doubles (not entire data blocks) from `Cache`, as well as floating point operations for doubles that have been loaded from `Cache`. You should assume double values are 8 bytes in size. You can think of the methods in `Cpu` as a psuedo-assembly language. Note that the `Cpu` is not aware of `DataBlocks`, since the latter abstracts them away from the former.



Address
-address: int
+getTag(): int
+getIndex(): int
+getOffset(): int
Cpu
+loadDouble(address:Address): double
+storeDouble(address:Address,value:double): void
+addDouble(value1:double,value2:double): double
+multDouble(value1:double,value2:double): double
1
1
Cache
+numSets: int
+numBlocks: int
+blocks: DataBlock[numSets][numBlocks]
+getDouble(address:Address): double
+setDouble(address:Address,value:double): void
+getBlock(address:Address): DataBlock
+setBlock(address:Address,block:DataBlock): void
1
1..*
DataBlock
+size: int
+data: double[size]
1..*
1
Ram
+numBlocks: int
+data: DataBlock[numBlocks]
+getBlock(address:Address): DataBlock
+setBlock(address:Address,value:DataBlock): void

Figure 1: Example of a object scheme for your CPU emulator

1.3 Using Objects in your Emulator "Program"

Your emulator will run "programs" on the CPU by calling methods of the `Cpu` class; the intent is that the emulator's programs will be written in the pseudo-assembly language of the `Cpu` class. Your emulator should provide at least the instruction commands that were defined in subsection 1.1. For example, an emulator written in Python might execute a daxpy ($\vec{c} = D\vec{a} + \vec{b}$) as in Figure 2.

```
# Initialize a Cpu object with whatever args you need
myCpu = Cpu(**kwargs)

# Assume 8 bytes per Double
sz = 8

# Construct arrays of Addresses for length = 100 problem
n = 100
a = list(range(      0,    n*sz, sz))
b = list(range(  n*sz, 2*n*sz, sz))
c = list(range(2*n*sz, 3*n*sz, sz))

# Initialize some dummy values
for i in range(n):
    myCpu.storeDouble(address=a[i], value=i)
    myCpu.storeDouble(address=b[i], value=2*i)
    myCpu.storeDouble(address=c[i], value=0)

# Put a random 'D' value into a register
register0 = 3

# Run the daxpy. Registers are just local variables.
for in in range(n):
    register1 = myCpu.loadDouble(a[i])
    register2 = myCpu.multDouble(register0, register1)
    register3 = myCpu.loadDouble(b[i])
    register4 = myCpu.addDouble(register2, register3)
    myCpu.storeDouble(c[i], register4)
```

Figure 2: Example of a daxpy implemented on your emulator

1.4 Measuring Events (5 points)

To measure events (such as cache misses and instruction counts), you will need to add logging functionality to your implementation. In this implementation, the logging could be abstracted away in the class methods. For example, the `Cache.getDouble` method would log a cache hit if the desired block is in `Cache.blocks` and log a miss if not. In either case, `Cache.getDouble` would return the desired double to the Cpu.

Your program should monitor the following events, and report the results at the end of the run:

- Instruction Count (adds, mults, loads, stores)
- Read Hits
- Read Misses
- Write Hits
- Write Misses

Additionally, your code should report at the end the read miss rate and the write miss rate (in percentages).

1.5 Interface (5 points)

Your final compiled executable or script should be named `cache-sim` (or `cache-sim.py`, `cache-sim.jl`, etc). It should support the following command line options and defaults, so that the grader can test programs in a uniform manner:

- `-c` : The size of the cache in bytes (default: 65,536)
- `-b` : The size of a data block in bytes (default: 64)
- `-n` : The n-way associativity of the cache. `-n 1` is a direct-mapped cache. (default: 2)
- `-r` : The replacement policy. Can be `random`, `FIFO`, or `LRU`. (default: `LRU`)
- `-a` : The algorithm to simulate. Can be `daxpy` (daxpy product), `mxm` (matrix-matrix multiplication), `mxm_block` (mxm with blocking). (default: `mxm_block`).
- `-d` : The dimension of the algorithmic matrix (or vector) operation. `-d 100` would result in a 100×100 matrix-matrix multiplication. (default: 480)
- `-p` : Enables printing of the resulting "solution" matrix product or daxpy vector after the emulation is complete. Elements should be read in emulation mode (e.g., using your `loadDouble` method), so as to assess if the emulator actually produced the correct solution.
- `-f` : The blocking factor for use when using the blocked matrix multiplication algorithm. (default: 32)

Your code should also have an interface that clearly prints out the configuration for the cache and ram specified, as well as a summary of the result metrics. Figure 3 gives an example of what we are looking for. You are free to add in additional information, and can certainly change the formatting, but please list at least all the information found in Figure 3. Ram size is computed as the minimum memory needed to hold the daxpy or matrix-matrix multiply data structures.

```
$> ./cache-sim -a mxm_blocked -d 400 -n 16 -f 8
INPUTS====================================
Ram Size =                    3840000 bytes
Cache Size =                  65536 bytes
Block Size =                  64 bytes
Total Blocks in Cache =       1024
Associativity =               16
Number of Sets =              64
Replacement Policy =          LRU
Algorithm =                   blocked mxm
MXM Blocking Factor =         8
Matrix or Vector dimension =  400
RESULTS====================================
Instruction count: XXXXXXXXXXX
Read hits:         XXXXXXXXXXX
Read misses:       XXXXXXXXXXX
Read miss rate:    X.XX%
Write hits:        XXXXXXXXXXX
Write misses:      XXXXXXXXXXX
Write miss rate:   X.XX%
```

Figure 3: Example of the output of an emulator run

2 Analysis

In this portion of the assignment, you will answer a series of questions that you'll need your emulator to solve. Include all filled tables and responses in a single PDF write-up.

2.1 Correctness (5 points)

Run your emulator in a wide variety of configurations for the daxpy, matrix-matrix multiply, and blocked matrix-matrix multiply routines. Use a vector length of 9 for the daxpy routine, and a 9×9 matrix size for the matrix multiply routines. For the blocking matrix-matrix multiply, use a blocking factor of 3.

For the daxpy routine, set the multiplier D to 3 and initialize the two source vectors as:

- $\vec{A} = \{0, 1, 2, ..., 8\}$
- $\vec{B} = 2\vec{A} = \{0, 2, 4, ..., 16\}$

For the matrix multiply routines, initialize the two source matrices in the same manner, using row-major indexing. e.g., for a 3×3 matrix this would be:

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} \; B = \begin{bmatrix} 0 & 2 & 4 \\ 6 & 8 & 10 \\ 12 & 14 & 16 \end{bmatrix}$$

In your write-up, write the solutions to the dimension 9 daxpy and matrix-matrix multiply routines. Be sure to make sure your solutions are correct before moving on.

2.2 Associativity (5 points)

Intel Skylake CPUs typically feature 8-way set associative caches. From the standpoint of our blocked matrix-matrix algorithm, is this a reasonable design decision? Use your emulator to support your answer by filling in and submitting the following table. For settings besides cache associativity, use defaults.

Cache Associativity	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
1							
2							
4							
8							
16							
1024 (Fully Associative)							

Table 1: Associativity table

2.3 Memory Block Size (5 points)

Use your emulator to run a study measuring the impact of different block sizes on cache miss rate. Fill in and submit the following table. Explain (in detail) the reasons causing the trend you see in miss rate percent as the block size increases.

Block Size (bytes)	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
8							
16							
32							
64							
128							
256							
512							
1024							

Table 2: Memory Block size table

2.4 Total Cache Size (5 points)

Adding SRAM cache to a chip design is expensive, as it takes up valuable die space that could otherwise be allocated to other resources. Therefore, we wish to minimize the amount of cache while still providing for reasonably good performance. The intel Skylake architecture typically uses a 32,768 byte (32 KB) cache size. Using your default emulator settings, estimate the minimum cache size required to achieve a 0.5% data read miss rate or less when running the blocked matrix-matrix product algorithm. Fill in and submit the following table as part of your analysis. If you find that the Skylake cache size of 32 KB is insufficient to meet this 0.5% data miss rate target, experiment with different settings and propose an architectural change that would allow us to meet this target when using a 32KB cache and report your optimization (hint: what else do we know about the Skylake architecture from subsection 2.2?).

Cache Size (bytes)	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
4096							
8192							
16384							
32768							
65536							
131072							
262144							
524288							

Table 3: Total Cache size table

2.5 Problem Size and Cache Thrashing (10 points)

In this analysis, your task will be to investigate the impact of matrix-matrix multiply problem size on the cache miss rate. Use your emulator to fill in and submit the following 3 tables (each using a different associativity). Using your results and your knowledge of the caching algorithms you implemented in your emulator, provide a detailed explanation for the following questions:

1. Begin by examining your results in Table 4. Given that there are only minor differences in problem sizes one might expect cache performance to be approximately the same for all tests when using the same algorithm. Is this the case for the regular matrix-matrix multiply algorithm? If you notice significant cache performance differences for the various problem sizes, explain why.
2. The blocked matrix-matrix multiply algorithm is designed to improve cache performance compared to the regular method. Is it successful in this goal for the 512×512 matrix problem size in Table 4? Why or why not?
3. Looking at Tables 5 and 6, if you notice an improvement in performance when using a higher associativity and/or fully associative caches, explain why this is the case. Is there a reason(s) hardware designers would not elect to use a fully associative cache?
4. You have a friend that is planning on developing a code that makes heavy use of matrix-matrix multiplication, and who plans to run on a system similar to the one represented in your emulator, with an 8-way set associative cache. A common problem size for them will be 512×512. Suggest at least two different software solutions they can take to help maximize the cache performance of their code.

You are encouraged to use specific numerical examples in your answers so as to clearly illustrate the impact of stride length on cache set indexing for the different problem sizes and cache configurations.

Matrix Dimension	MxM Method	Blocking Factor	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
480	Regular	-							
480	Blocked	32							
488	Regular	-							
488	Blocked	8							
512	Regular	-							
512	Blocked	32							

Table 4: Matrix Multiply Problem size table – Associativity = 2

Matrix Dimension	MxM Method	Blocking Factor	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
480	Regular	-							
480	Blocked	32							
488	Regular	-							
488	Blocked	8							
512	Regular	-							
512	Blocked	32							

Table 5: Matrix Multiply Problem size table – Associativity = 8

Matrix Dimension	MxM Method	Blocking Factor	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
480	Regular	-							
480	Blocked	32							
488	Regular	-							
488	Blocked	8							
512	Regular	-							
512	Blocked	32							

Table 6: Matrix Multiply Problem size table – Fully Associative

2.6 Replacement Policy (5 points)

Which of your replacement policies results in the best performance? Fill in and submit the following table to support your response.

Replacement Policy	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
Random							
FIFO							
LRU							

Table 7: Replacement policy table

3 Second-Level Cache Extension (10 points)

In this section, you will extend your emulator to include a second-level (L2) cache in addition to the L1 cache already implemented. This reflects modern cache hierarchies found in CPUs and enables more accurate modeling of memory performance.

L2 Cache Configuration (Fixed for this assignment)

For consistency across submissions, use the following fixed parameters for the L2 cache:

- L2 cache size = `131072` bytes (128 KB)
- L2 block size = `64` bytes
- L2 associativity = `8`-way
- L2 replacement policy = `LRU`

These are to be used in conjunction with the default L1 cache configuration unless otherwise specified:

- L1 cache size = `32768` bytes (32 KB)
- L1 block size = `64` bytes
- L1 associativity = `8`-way
- L1 replacement policy = `LRU`

Behavior

The L2 cache sits between the L1 cache and RAM. All memory accesses should first check L1, then check L2 if L1 misses, and finally access RAM if both caches miss. The L2 cache is *inclusive* of L1: all L1 blocks must also reside in L2.

Metrics

In addition to the existing metrics, you must also track:

- L2 read and write hits
- L2 read and write misses
- Number of RAM accesses (any access not satisfied by L1 or L2)

Analysis

Run your emulator on a blocked matrix-matrix multiplication of dimension 480 with blocking factor 32. Fill out the following two tables.

Config	**L1 Read Miss %**	**L1 Write Miss %**	**L2 Read Miss %**	**L2 Write Miss %**
L1 Only				
L1 + L2				

Table 8: Miss rates at each cache level

Config	**RAM Accesses**	**Read Hits (L1+L2)**	**Write Hits (L1+L2)**
L1 Only			
L1 + L2			

Table 9: Memory access totals and hits with and without L2