

# **North Carolina State University**

**Department of Electrical and Computer Engineering**

**ECE463/563: Fall 2019**

## **Project1: Cache and Memory Hierarchy Design**

**Design Specifications v1**

**Project 1 Part A Submission Deadline  
September 20, 2019  
11:59PM**

## Ground rules

1. All students must work alone. The project scope is reduced (but still substantial) for ECE 463 students, as detailed in this specification.
2. Sharing of code between students is considered cheating and will receive appropriate action in accordance with University policy. The TAs will scan source code (from current and past semesters) through various tools available to us for detecting cheating. Source code that is flagged by these tools will be dealt with severely.
3. A Wolfware message board is provided for posting questions, discussing and debating issues, and making clarifications. It is an essential aspect of the project and communal learning. Students must not abuse the message board, whether inadvertently or intentionally. Above all, never post actual code on the message (unless permitted by the TAs/instructor). When in doubt about posting something of a potentially sensitive nature, email the TAs and instructor to clear the doubt.
4. You must do all your work in the C/C++ or Java languages. Exceptions must be approved. The C language is fine to use, as that is what many students are trained in. Basic C++ extensions to the C language (e.g., classes instead of structs) are encouraged (but by no means required) because it enables more straightforward code reuse.
5. Use of the Grendel environment is required. This is the platform where the TA will compile and test your simulator. Please test your simulator on Grendel machines before submission.

**CAUTION:** If you develop your simulator on another platform, get it working on that platform, and then try to port it over to Grendel at the last minute, you may encounter major problems. Porting is not as quick and easy as you think unless you are an excellent programmer. Worse, malicious bugs can be hidden until you port the code to a different platform, which is an unpleasant surprise close to the deadline. So, keep this in mind.

# Contents

Project Overview.....	4
1. Part A: The generic CACHE module.....	5
1.1. Configurable Parameters .....	5
1.2. Configurable Policies.....	6
1.2.1 Replacement Policy.....	6
1.2.2 Write Policy .....	6
1.3. Modeling the CACHE.....	6
1.4. Input to the Simulator.....	7
1.5. Raw Measurements .....	8
1.6. Performance Analysis .....	8
1.7. Validation Requirements .....	9
1.7.1 Building and Running the simulator .....	9
1.7.2 Validation .....	10
1.8. Simulation Run time.....	11
1.9. What to submit via Wolfware.....	12
1.10. Grading Policy .....	12

## Project Overview

In this project, you will implement a flexible cache and memory hierarchy simulator and use it to study the performance of memory hierarchies using the SPEC benchmarks.

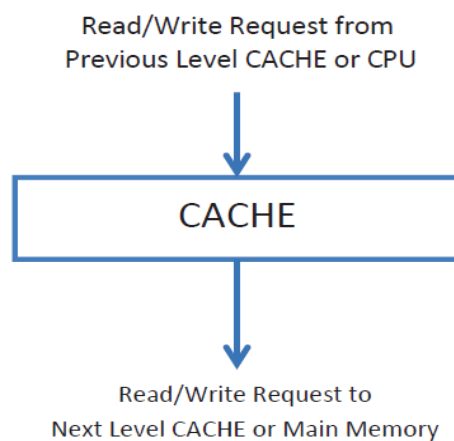
This project is divided into two parts. Both Part A and Part B are to be submitted separately. (Specifications of Part B will be released soon.) In Part A, you will design a generic cache simulator module with some configurable parameters. This cache module can be instantiated (used) as an L1 cache, an L2 cache, or an L3 cache, and so on. Since it can be used at any level of the memory hierarchy, it will be referred to generically as CACHE throughout this specification. In Part B, you will design a flexible two-level memory hierarchy simulator with certain changes made to the CACHE module with the help of additional configurable parameters designed in Part- A.

Both simulators will take an input in a standard format which describes the read/write requests from the processor. Simulator output is also expected to be in a standard format as explained in further sections.

## 1. Part A: The generic CACHE module

Design a generic CACHE module that can be used at any level in a memory hierarchy. This generic CACHE can be configured using different design parameters. It takes read/write requests as input and optionally generates appropriate read/write request for the next level of memory hierarchy.

In Part A, you will design a one level cache memory hierarchy. Hence, all the read/write requests come from the CPU and the next level of memory hierarchy is always the main memory.



### 1.1. Configurable Parameters

CACHE should be configurable in terms of supporting any cache size, associativity, and block size, specified at the beginning of simulation

- BLOCKSIZE: Number of bytes in a block.
- L1\_SIZE: Total bytes of data storage for L1 Cache.
- L1\_ASSOC: Associativity of L1 cache (ASSOC=1 is a direct-mapped cache)
- L2\_SIZE: Total bytes of data storage for L2 Cache ('0' for PartA)
- L2\_ASSOC: Associativity of L2 cache ('0' for PartA)
- L2\_DATA\_BLOCKS: Number of data blocks in a sector ('0' for PartA)
- L2\_ADDR\_TAGS: Number of address tags pointing to a sector ('0' for PartA)
- Trace\_file: Input Trace file (gcc\_trace, etc.)

There are a few assumptions for the above parameters:

- BLOCKSIZE is a power of two.
- The number of sets in a cache is also a power of two.

- ASSOC and SIZE need not be a power of two.
- SUGGESTION: If you want to make a single function for a cache, build this hierarchy using recursive calls or a generic class for cache for the entire project 1 containing all the parameters above, you can assign DATA\_BLOCKS and ADDR\_TAGS to '1' for a normal L1 cache, like in this project. You will find out why when you study the specifications for partB.

As you know, the number of sets is determined by the following equation.

$$\text{Number of Sets} = \frac{\text{SIZE}}{\text{ASSOC} \times \text{BLOCKSIZE}}$$

## 1.2. Configurable Policies

Apart from the configurable parameters, the CACHE can be configured in terms of policies. Corresponding policies will be specified at the beginning of the simulation.

### 1.2.1 Replacement Policy

All students (ECE463 and ECE563) need to implement the LRU (Least Recently Used) replacement policy, as discussed in the class.

### 1.2.2 Write Policy

All students (ECE463 and ECE563) need to implement WBWA (write-back + write-allocate) write policy.

#### 1.2.2.1 Write-Back Write –Allocate (WBWA)

A write updates the corresponding block in CACHE, making the block dirty. It does not update the next level in the memory hierarchy (next level CACHE or the main memory) at that time. If a dirty block is evicted from CACHE, a “Writeback” is performed and the entire block will be sent to the next level in the memory hierarchy. A write that misses in CACHE will cause a new block to be allocated in CACHE. Therefore, both write misses and read misses cause blocks to be allocated in CACHE.

## 1.3. Modeling the CACHE

This simulator can model various instances of CACHE in a memory hierarchy. For Part A, the simulator will model only a single level memory hierarchy. CACHE

receives a read or write request from the higher level (CPU). Only situation where CACHE must interact with the next level below it (main memory) is when the read or write request misses in the CACHE. CACHE always allocates a new block of data when a read or write request is missed, as the WBWA write policy is implemented.

### Allocation of a new block

Think of one of the above scenarios in which CACHE needs to allocate a new block X. The allocation of requested block X is actually a two-step process. The two steps must be performed in the following order.

**1. Make space for the requested block X.** If there is at least one invalid (free) block in the set, then there is already space for the requested block X and no further action is required. (Go to step 2). To be correct during the simulation, make sure you initialize the LRU counter for all the data blocks in an index. So, if there are more than one invalid blocks, place the fetched block in a position depending upon the LRU counter and not in any one of them. On the other hand, if all blocks in the set are valid, then a block must be singled out for eviction, according to the replacement policy ([Section 1.2](#)). For WBWA policy, if this evicted block is dirty, then a write-back of the evicted block must be issued to the next level of the memory hierarchy. For this part of the project, you do not need to do anything in such scenario.

**2. Bring in the requested block X.** Issue a read of the requested block X to the next level of the memory hierarchy and put the requested block X in the appropriate place in the set (determined in step 1).

## 1.4. Input to the Simulator

The simulator reads a trace file which follows following format.

```
r|w <hex address>
r|w <hex address>
...
```

Here  $r$  indicates a “load” and  $w$  indicates a “store” from the processor. The simulator must parse the trace file and issue the corresponding read/write request to the highest level in the memory hierarchy.

### Example trace file

```
R ffe04540
r ffe04544
w 0eff2340
r ffe04548
r ffe04544
w 0eff2340
r ffe04548
...
```

## 1.5. Raw Measurements

This simulator aims at collecting the data to calculate certain statistics for the CACHE. The simulator should be able to compute following raw statistics at the end of simulation for a given configuration of CACHE. In Part A, L1 is the only CACHE in the single level memory hierarchy.

- number of L1 reads
- number of L1 read misses
- number of L1 writes
- number of L1 write misses
- $L1 \text{ miss rate} = MR_{L1} = (b + d) / (a + c)$
- number of write-backs from L1 to memory
- total memory traffic = number of blocks transferred to/from memory

*Note: g should match  $(b + d + f)$  if L1 uses WBWA policy.*

The simulator will print out the CACHE configuration, the statistics and the status of CACHE at the end of the simulation to output console in a specified format.

## 1.6. Performance Analysis

From the simulation results, you can analyze the performance of various single level cache memories.

### Average Access Time (AAT)

It is the average time it takes for a CACHE to service a single read/write request from the processor. For memory hierarchy with only L1 cache, AAT can be computed using following equations.



$$\text{Total access time} = (\text{Reads}_{L1} + \text{Writes}_{L1}) \times \text{HT}_{L1} + (\text{ReadMisses}_{L1} + \text{WriteMisses}_{L1}) \times \text{MissPenalty}_{L1}$$

$$\text{L1 Miss Rate, } MR_{L1} = \frac{\text{ReadMisses}_{L1} + \text{WriteMisses}_{L1}}{\text{Reads}_{L1} + \text{Writes}_{L1}}$$

$$\text{Average Access Time} = \frac{\text{Total Access Time}}{\text{Reads}_{L1} + \text{Writes}_{L1}}$$

$$\text{Average Access Time} = \text{HT}_{L1} + (MR_{L1} \times \text{MissPenalty}_{L1})$$

Note:  $\text{HT}_{L1}$  (L1 hit-time) and the L1 Miss Penalty can be obtained from the course website for this project.

## 1.7. Validation Requirements

Your simulator code will be tested electronically. Trace files will be used as simulator inputs. Trace files are explained in section 1.4. Various configurations of CACHE consisting of cache size, associativity, block size will be tested on the trace files. Sample outputs from the simulator called “Validation Runs” will be posted on the course website.

Each validation run includes

1. The memory hierarchy configuration
2. The final contents of all caches in the memory hierarchy
3. All measurements described in [Section 1.5](#)

### 1.7.1 Building and Running the simulator

You will submit the source code electronically and the TA will compile and run your simulator. As such, you must meet the following strict requirements. Failure to meet these requirements will result in point deductions (see [Section 1.10](#) for Grading Policy).

1. You must be able to compile and run your simulator on Grendel (32 bit, x86) machines. This is required so that the TA can compile and run your simulator. If you are logging into a Grendel machine remotely and do not know whether or not it is Linux (as opposed to SunOS), use the **uname** command to determine the operating system.
2. Along with your source code, you must provide a **Makefile** that automatically compiles the simulator. This Makefile must create a simulator

named **sim\_cache**. The TA should be able to build the simulator using single **make** command. The TA should be able to remove object and executable files using **make clean** command. To make your life easy, an example Makefile will be posted on the course website, which you can copy and modify according to your needs.

3. Your simulator must accept exactly 6 command-line arguments in the following order:

```
sim_cache <BLOCKSIZE> <L1_SIZE> <L1_ASSOC> <L2_SIZE> <L2_ASSOC>
        <L2_DATA_BLOCKS> <L2_ADDR_TAGS> <trace_file>
```

<BLOCKSIZE>	Block size in bytes. Positive Integer, Power of two
<L1_SIZE>	Total CACHE size in bytes. Positive Integer
<L1_ASSOC>	Associativity of Cache.
<L2_SIZE>	'0' for all runs of Project1A
<L2_ASSOC>	'0' for all runs of Project1A
<L2_DATA_BLOCKS>	'0' for all runs of Project1A
<L2_ADDR_TAGS>	'0' for all runs of Project1A
<trace_file>	Character string. Full name of the trace file including any extensions

**Example:** 8KB 4-way set-associative L1 cache with 32B block size, LRU replacement policy and WTNA write policy will be simulated for “gcc\_trace” with following command.

```
$ ./sim_cache 32 8192 4 0 0 0 0 ./trace/gcc_trace
```

4. Your simulator must print outputs to the console (i.e. to the screen). This way, when a TA runs your simulator, he can simply redirect the output of your simulator to a file for validating the results.

### 1.7.2 Validation

Your output must match both numerically and in terms of formatting when compared to the validation runs. The TA will literally **diff** your output with the correct output. You must confirm correctness of your simulator by following these two steps for each validation run.

1. Redirect the console output of your simulator to a temporary file. This can be achieved using “>” operator. For example,

```
$ ./sim_cache 32 8192 4 0 0 0 0 ./trace/gcc_trace.txt > my_output
```

2. Test whether or not your outputs match properly, by running this Linux command. This command must output “nothing” indicating a correct match.

```
$ diff -iw my_output validation_run
```

The `-iw` flags tell **diff** to treat upper-case and lower-case as equivalent and to ignore the amount of white space between words. Therefore, you do not need to worry about the exact number of spaces or tabs as long as there is some white space where the validation runs have white space.

For guidelines on how to get started with designing your simulator, see [Appendix](#).

## 1.8. Simulation Run time

Correctness of your simulator is of paramount importance. That said, making your simulator efficient is also important for a couple of reasons. First, the TA needs to test every student’s simulator. Therefore, we are placing the constraint that your **simulator must finish a single run in 2 minutes or less**. If your simulator takes longer than 2 minutes to finish a single run, please see the TA as they may be able to help you speed up your simulator.

Second, you will be running many experiments. Therefore, you will benefit from implementing a simulator that is reasonably fast.

One simple thing you can do to make your simulator run faster is to compile it with a high optimization level. The example **Makefile** posted on the web page includes the `-O3` optimization flag.

Note that, when you are debugging your simulator in a debugger (such as `gdb`), it is recommended that you compile without `-O3` and with `-g`. Optimization includes register allocation. Often, register-allocated variables are not displayed properly in debuggers, which is why you want to disable optimization when using a debugger. The `-g` flag tells the compiler to include symbols (variable names, etc.) in the compiled binary. The debugger needs this information to recognize variable names, function names, line numbers in the source code, etc. When you

are done debugging, recompile the code with `-O3` and without `-g` to get the most efficient simulator again.

## 1.9. What to submit via Wolfware

You must hand in a single zip file called **<unity-id>\_project1A.zip** that is no more than 1MB in size. Notify the TA beforehand if you have special space requirements. However, a zip file of 1MB should be sufficient for any code.

Your **<unity-id>\_project1A.zip** must contain only the following (any deviation from the following requirements may delay grading your project and may result in point deductions, late penalties, etc.):

1. Source code. You must include the commented source code for the simulator program itself. You may use any number of `.c/cpp/.cc/.h` files.
2. Makefile. See [Section 1.7](#), for strict requirements. If you fail to meet these requirements, it may delay grading your project and may result in point deductions.

Below is an example showing how to create “<unity-id>\_project1A” from a Grendel machine. Suppose you have a bunch of source code files (`*.cc`, `*.h`), the Makefile. Replace `<unity-id>` by your unity ID. Make sure the file is correct, as your grades would be given on the basis of the `unity_id` in this file name. The simulator should build an executable by a single ‘make’ command and run the simulator by the example command shown above. Any warnings present in the build would result in deduction of points. If there are any errors, you would get no marks.

```
$ tar -zcvf <unity-id>_project1A *.cc *.h Makefile
```

As TA will unzip and grade your code electronically using scripts your zip file must have all files inside directly (not in a folder inside the zip). So, keep this in mind if you create zip from GUI program in windows.

## 1.10. Grading Policy

Definition of “project works”:

- ✓ **ALL raw measurements match (Section 1.5)**
- ✓ **The final contents of the cache match**

There will be no credit for a run if it does not work according to the definition above.

Item		Points
Substantial Simulator can be compiled and run		10
L1 works with LRU and WBWA	Validation Runs	10
L1 works with LRU and WBWA	Mystery Runs	10
Total		30

Various deductions (out of 30 points):

**-1 point** for each day late, according to Wolfware timestamp.

Reminder: since it's Part A, whose major purpose is to split a big project into 2 parts and push you to reach a safe milestone by the date, we don't want to be harsh on late policy. In future project, the late policy is likely -1 point for each hour. Please keep in mind.

**TIP:** Submit whatever you have completed by the deadline just to be safe. If, after the deadline, you want to continue working on the project to get more features working and/or finish experiments, you may submit a second version of your project late. If you choose to do this, the TA will grade both the ontime version and the late version (*late penalty applied to the late version*), and take the maximum score of the two. Hopefully you will complete everything by the deadline, however.

**Up to -5 points** for not complying with specific procedures. Follow all procedures very carefully to avoid penalties. Complete the **SUBMISSION CHECKLIST** to make sure you have met all requirements.

**Cheating:** Source code that is flagged by tools available to us will be dealt with according to University Policy. This includes a 0 for the project and other disciplinary actions.

# Appendix

## *Design Guidelines*

Here are some guidelines to get you started with design of your CACHE simulator. In this simulator you need to simulate only the **tags** for each cache block, and you don't need to simulate any kind of data-transfer to and from CACHE. You can use C/C++ languages for your code. In this guide, use of C/C++ languages is assumed.

The guidelines provided here are just for your reference. It's not mandatory to follow these guidelines in your project. As long as your simulator output matches with the validation runs, you are good.

In your program, CACHE can be represented as a data structure implemented with a C **struct** or a C++ **class**. Use of C++ is not necessary but is recommended as it simplifies the development process. As we are designing a generic CACHE which can be used at any level in memory hierarchy, the implementation of CACHE should be independent of the position in memory hierarchy.

For this, the CACHE data structure needs to be a node in a linked list. Each CACHE will have access to its immediate next level in memory hierarchy. This can be implemented as a pointer to the CACHE data structure inside CACHE. For CACHE just above the main memory, this pointer will simply be NULL.

In this way, your simulator will only give read/write requests to the first level CACHE (just below the CPU) using its member functions. This CACHE will forward the requests to next level if needed using the **nextLevel** pointer.

CACHE will keep track of number of reads, writes, misses, write-backs, memory accesses etc. using some counter variables. These counter variables will be incremented during the simulation as needed.

In a simulator run, it will first initialize the CACHE data structures using command line arguments. Then it will start reading the trace file and issue the read/write requests to the first level (L1 CACHE) and L1 CACHE will increment its counters and updates its tags and if needed, it will issue read/write requests to L2 CACHE (if it exists). Similarly L2 CACHE will increment its counters and updates its tags and so on. At the end of simulation when all requests from the trace file are completed, simulator will read the counter values and display raw measurements and contents of each CACHE.

```
class CACHE
{
private:
/* AddCACHE data members
add variables for parameters like
size, block size, associativity,
write and replacement policies
dynamic Array for tag storage,
all counter variables
and other variables needed
*/

//pointer to the next CACHE in memory hierarchy
CACHE *nextLevel;
public:
//CACHE member functions
bool readFromAddress(unsigned int add);
bool writeToAddress(unsigned int add);
//functions to add more
//functionality in your CACHE
};
```