CISC 340 – Project 4
Due: 11:59 pm, Friday, May 15
(15 points)

1. Purpose

The purpose of this project is to teach you about cache design and how a caching processor generates and services address references.
You will write procedural C.
You will use GCC on the departmental linux system (rusty).
You will do this project with a partner.

2. Problem

In this project, you will simulate a CPU cache (unified instruction/data) and integrate the cache into a Project 1-type single-cycle (behavioral) simulator. As the processor simulator executes an assembly-language program, it will access instructions and data. These accesses will be serviced by the cache, which will transfer data to/from memory as needed.

3. Cache Simulator

The central part of this project is to write a simulator that implements a flexible cache simulator. The cache functionality will be used by the processor simulation when the processor accesses addresses. Your cache simulator should be able to simulate a variety of cache configurations. Once integrated into your processor simulator, the program will need to input the following values upon starting:

> Enter the machine code program to simulate:
> Enter the block size of the cache (in words):
> Enter the number of sets in the cache:
> Enter the associativity of the cache:

In addition to the manual entry of parameters at run time, your program should take all of the same information as input from command line arguments using -f, -b, -s, and -a flags, respectively. Command line arguments should be handled with GNU getopt.

Your cache function should simulate a cache with the following characteristics:

1) **WRITE POLICY**: allocate-on-write, write-back
2) **ASSOCIATIVITY**: varies according to the "associativity" parameter. Associativity ranges from 1 (direct-mapped) to the number of blocks in the cache (fully associative).
3) **SIZE**: the total number of words in the cache is:
   block_size_in_words * number_of_sets * associativity
4) **BLOCK SIZE**: varies according to the "block_size_in_words" parameter. All transfers between the cache and memory take place in units of a single block.
5) **PLACEMENT/REPLACEMENT POLICY**: when looking for a block within a set to replace, the best block to replace is an invalid (empty) block. If there are none of these, replace the least-recently used block (LRU).
6) **Halt**: Upon a halt, your cache should write back any dirty entries and then invalidate all entries.

Make sure the units of each parameter are as specified. Note that the smallest data granularity in this project is a word, because this is the data granularity of the UST-3400 architecture.

block_size_in_words, number_of_sets, and associativity should all be powers of two. To simplify your program, you may assume that the maximum number of cache blocks is 256 (this small number allows you to use a 2-dimensional array for the cache data structure).

4. Origin and Servicing of Address References

As the processor executes an assembly-language program, it accesses addresses. The three sources of address references are instruction fetch, lw, and sw. When the program starts up, it will initialize the memory with the machine-code file as in Project 1. These initialization references should NOT be passed to the cache simulator; they should simply set the initial memory state.

Each memory address reference should be passed to the cache simulator. The cache simulator keeps track of what blocks are currently in the cache and what state they are in (e.g. dirty, valid, etc.). To service the address reference, the cache simulator may need to write back a dirty cache block to memory, then it may need to read a block into the cache from memory. After these possible steps, the cache simulator should return the data to the processor (for read accesses) or write the data to the cache (for write accesses). Each of these data transfers will be logged by calling the print_action function (please follow the output format of the example code below):

```
/*
* Log the specifics of each cache action.
*
* address is the starting word address of the range of data being transferred.
* size is the size of the range of data being transferred.
* type specifies the source and destination of the data being transferred.
*
* cache_to_processor: reading data from the cache to the processor
* processor_to_cache: writing data from the processor to the cache
* memory_to_cache: reading data from the memory to the cache
* cache_to_memory: evicting cache data by writing it to the memory
* cache_to_nowhere: evicting cache data by throwing it away
*/
enum action_type {cache_to_processor, processor_to_cache, memory_to_cache, cache_to_memory,
                                        cache_to_nowhere};

void print_action(int address, int size, enum action_type type)
{
        printf("transferring word [%i-%i] ", address, address + size - 1);
        if (type == cache_to_processor) {
                printf("from the cache to the processor\n");
        } else if (type == processor_to_cache) {
                printf("from the processor to the cache\n");
        } else if (type == memory_to_cache) {
                printf("from the memory to the cache\n");
```

```
        } else if (type == cache_to_memory) {
                printf("from the cache to the memory\n");
        } else if (type == cache_to_nowhere) {
                printf("from the cache to nowhere\n");
        }
}
```

5. Test Cases

An integral (and graded) part of writing your cache simulator will be to write a suite of test cases to validate any UST-3400 cache simulator. This is common practice in the real world--software companies maintain a suite of test cases for their programs and use this suite to check the program's correctness after a change. Writing a comprehensive suite of test cases will deepen your understanding of the project specification and your program, and it will help you a lot as you debug your program.

The test cases for this project will be short assembly-language programs that, after being assembled into machine code, serve as input to a simulator. You will submit your suite of test cases together with your simulator, and we will grade your test suite according to how thoroughly it exercises an UST-3400 cache simulator. Each test case may generate at most 200 print_action statements on a correct simulator, and your test suite may contain up to 20 test cases. These limits are much larger than needed for full credit.

Each test case will specify the cache parameters to use when running the test case. These parameters will be communicated via the name of the test case file. Each test case should have a 3-part suffix, where each part identifies a cache parameter and the parts are separated by periods:

testname.<block_size_in_words>.<number_of_sets>.<associativity>

For example, the test case in Section 8 would be named "test.as.4.2.1". The combination of cache parameters should be legal (e.g. they should all be powers of two; number_of_sets x blocks_per_set should not exceed 256).

Writing good test cases for this project will require careful planning. Think about what different types of behavior a cache can exhibit and generate test cases that cause the cache to exhibit each behavior. Think about how to test the various algorithms used in the cache simulator, e.g. LRU, write-backs, read and write hits, read and write misses.

As you write the code for your simulator, keep notes on different conditions you think of, and write test cases to test those conditions.

6. Grading and Formatting

I will grade on the correctness of your cache simulator and the comprehensiveness of your test suite. In particular, I will run your program on various machine-language programs and check the outputs you report for the various cache actions. Since I'll be grading on getting the exact right answers, it behooves you to spend a lot of time writing test assembly-language programs and testing your program. It is the best way to debug your program and is also one of the best ways to learn the concepts in the project.

The student suite of test cases for the simulator will be graded according to how thoroughly they test a cache simulator. I will judge thoroughness of the test suite by how well it exposes potential bugs in a

cache simulator. A test case exposes a buggy simulator by causing it to generate a different answer from a correct simulator. Some suggestions of details to keep an eye on (and ideas to help your professor keep his sanity while grading):

1) **Have your "print_action()" use EXACTLY the format of the example!**
2) Check your program's output on test case programs against the solution simulator provided.

7. Turning in the Project

The project will be turned in via Canvas. The following components (**and structure!**) are necessary for submission:

- code for your simulator
- a Makefile to compile your code
- Sub-Folder of assembler test files (assembly code and machine code) named tests.
- A README file with instructions and explanations of files. Ensure that you are clear about the purpose of each file of your test suite in your explanations
- An overview document (pdf) telling me how your cache simulator works. This document should also communicate any difficulties you had in developing each program, as well as any shortcomings that remain (e.g., did not finish implementing eviction policy).

Place all of this stuff inside a directory named "project4_username1_username2" using whatever your login names are. You can then zip this directory using the linux command "zip -r project4_username1_username2.zip project4_username1_username2". Submit the zip file via Canvas. Only one submission per group!

NOTE: When I unzip your submission I should have a single directory, not a whole bunch of files that I have to clean up.

Hand in a hardcopy of your overview document in class on the due date.

8. Sample Assembly-Language Program and Output

Here is a sample assembly-language program:

```
sw 1 0 6
lw 1 0 23
lw 1 0 30
halt
```

and its corresponding output with the following cache parameters:

block_size_in_words = 4, number_of_sets = 2, and associativity = 1.
Make sure you understand each of the data transfers and their order!

```
transferring word [0-3] from the memory to the cache
transferring word [0-0] from the cache to the processor
transferring word [4-7] from the memory to the cache
transferring word [6-6] from the processor to the cache
transferring word [1-1] from the cache to the processor
transferring word [4-7] from the cache to the memory
```

transferring word [20-23] from the memory to the cache
transferring word [23-23] from the cache to the processor
transferring word [2-2] from the cache to the processor
transferring word [20-23] from the cache to nowhere
transferring word [28-31] from the memory to the cache
transferring word [30-30] from the cache to the processor
transferring word [3-3] from the cache to the processor