CDA3101 Project 3: Cache Simulator (Graduate Version)
Due 8/1

I. Purpose

The purpose of this project is to exercise your understanding of caches of various sizes and configurations, as well as the write policies of write-through and write-back.

II. Basic Description

You will be implementing a cache simulator in C, which will be contained in the single file proj3.c. You program should take as command line arguments the following information:

- b: size of block in bytes.
- s: number of sets in the cache (or total number of sets between split caches).
- n: associativity of cache.

Your submitted proj3.c must compile and run on linprog as in the following example:

```
$ gcc –lm proj3.c
$ ./a.out –b 8 –s 256 –n 2 < test1.trace
```

You should not rely on any special compilation flags (although –lm will be allowed for use of the math module) or other input methods. If your program does not compile and execute as expected with the above commands, you will receive zero points on project 3.

The contents of the .trace files used as input are described below. There is no provided code for this project so you should start by writing C code to parse the command line arguments and trace files.

You will need to simulate two types of L1 caches: unified and split. Unified caches do not distinguish between instruction and data references. Split I vs. D caches separately cache instruction and data references in their own L1 caches. Typically, these caches are configured independently, but for this project they will use the same specifications (as provided in the command line). When simulating the split L1 cache, assume that each sub-cache receives half the number of sets (s/2) so that the total size is the same between the two implementations.

After reading in the command line arguments, you should do all of the following for each scheme (unified vs. split): print back to the user the number of bytes in a block, the number of

sets in the cache, and the associativity of the cache. Additionally, you should print the number of bits required for each partition of a memory reference: tag, index, and offset. Afterward, you should prepare to simulate the behavior of two caches on the sequence of references in the trace file. The first cache has a write-through, no write allocate policy. The second has a write-back, write allocate policy. For every reference, you should determine for each cache whether the reference is a hit or a miss, and whether main memory needs to be accessed. After all references have been processed, print the statistics for each cache.

To summarize, you will need to simulate:
- A write-back, write allocate unified cache
- A write-through, no write allocate unified cache
- A write-back, write allocate split cache (note that instructions are only ever read)
- A write-through, no write allocate split cache (note that instructions are only ever read)

For example, consider the contents of the trace file test.trace:

```
R 0 I
W 1036 D
R 4 I
R 8 I
R 12 I
R 1024 D
R 16 I
R 4116 D
R 20 I
W 4120 D
R 24 I
R 28 I
W 8208 D
R 32 I
R 8228 D
R 36 I
R 40 I
W 8228 D
R 44 I
R 304 D
R 8228 D
```

Let's use this trace file as input to our program and specify that our caches should have 64 sets total with 1 block each and 16 bytes per block.

```
$  gcc -lm proj3.c
$  ./a.out -b 16 -s 64 -n 1 < test.trace
================ Unified ================
Block size: 16
Number of sets: 64
Associativity: 1
Number of offset bits: 4
Number of index bits: 6
Number of tag bits: 22

****************************************
Write-through with No Write Allocate
****************************************
Total number of references: 21
Hits: 7
Misses: 14
Memory References: 14

****************************************
Write-back with Write Allocate
****************************************
Total number of references: 21
Hits: 4
Misses: 17
Memory References: 20

============= Split I vs. D =============
Block size: 16
Number of sets: 32
Associativity: 1
Number of offset bits: 4
Number of index bits: 5
Number of tag bits: 23

****************************************
Instructions:
****************************************
Total number of references: 12
```

```
Hits: 9
Misses: 3
Memory References: 3

****************************************
Data: Write-through with No Write Allocate
****************************************
Total number of references: 9
Hits: 3
Misses: 6
Memory References: 8

****************************************
Data: Write-back with Write Allocate
****************************************
Total number of references: 9
Hits: 4
Misses: 5
Memory References: 6
```

III. Trace File Description

The .trace files that will be used as input to your cache simulator contain some variable number of lines (no more than 500), where each line has the following format:

```
<access_type> <byte_address> <IorD>
```

The access type can be either 'R' for reading or 'W' for writing. The byte address is simply the decimal representation of the 32-bit address of the reference. The type of the reference can be either 'I' for instruction or 'D' for data.

Note: MIPS is word-aligned, which means that we can only legitimately access every 4^{th} byte. All provided test files will contain word-aligned references, but you do not need to check to make sure the references are word-aligned.

IV. Suggested Development Approach

In this project, you will not be given any starting file so it is suggested that you start as soon as possible to allow yourself time to plan the structure of your program.

The first thing you should work on is the parsing of the command line arguments. Note that the command line arguments may be specified in any order. You may assume that the block size and number of sets provided as arguments to the program will always be powers-of-two. After you have extracted the arguments, work on the unified cache. First, echo the values back to the user as shown in the sample output. You should also calculate and print the number of offset bits, index bits, and tag bits, as discussed in lecture 11. You may assume that the addresses in the test file correspond to 32-bit binary addresses.

```
Block size: 16
Number of sets: 64
Associativity: 2
Number of offset bits: 4
Number of index bits: 6
Number of tag bits: 22
```

Next, you should implement a write-through, no write allocate unified cache and supporting functionality for reading from and writing to the cache. There are several ways to implement the cache in your C code, but the most straightforward is probably through the use of two-dimensional arrays (or a flattened one-dimensional array).

Side note: In C, the most conventional rule for array creation is that arrays may only be statically allocated with sizes determined by constants known at compile-time. That is, an array can only be statically declared as

int myarray[c]; /* c is a literal value or otherwise constant */

The array sizes in this program are determined by information provided to you by the user during run-time. Because these values can only be known at run-time, they are not compile-time constants and, therefore, cannot be used to statically allocate arrays in C. In this case, generally speaking, you would need to create a dynamically-sized array based on the input parameters using the malloc() function.

However**,** this restriction is only a widely-implemented convention and not always present.

Variable length arrays (“statically” allocated with run-time values) are an *optional* feature in a C11 compiler, mandatory in C99, and not available at all in C90. As long as your code works with gcc on linprog, that's fine. But you should know that this compile-time restriction may be present and variable length arrays are not a "guaranteed" feature of C.

For a write-through, no write allocate cache, we have the following properties:

- When a block is present in the cache (hit), a read simply grabs the data for the processor.
- When a block is present in the cache (hit), a write will update both the cache and main memory (i.e. we are writing *through* to main memory).
- When a block is not present in the cache (miss), a read will cause the block to be pulled from main memory into the cache, replacing the **least recently used block** if necessary.
- When a block is not present in the cache (miss), a write will update the block in main memory but we do not bring the block into the cache (this is why it is called “no write allocate”).

After you have implemented the write-through unified cache, you should implement the write-back unified cache. One way to approach this is to have two separate caches, one for write-through and the other for write-back, which are both updated independently every time a reference is simulated.

For a write-back, write allocate cache, we have the following properties:

- When a block is present in the cache (hit), a read simply grabs the data for the processor.
- When a block is present in the cache (hit), a write will update only the cache block and set the dirty bit for the block. The dirty bit indicates that the cache entry is not in sync with main memory and will need to be written back to main memory when the block is evicted from the cache.
- When a block is not present in the cache (miss), a read will cause the block to be pulled from main memory into the cache, replacing the **least recently used block** if necessary. If the block being evicted is dirty, the block’s contents must be written back to main memory.
- When a block is not present in the cache (miss), a write will cause the block to be pulled from main memory into the cache, replacing the **least recently used block** if necessary. When the block is pulled into the cache, it should immediately be marked as “dirty”. If the block being evicted is dirty, the block’s contents must be written back to main memory.

Using these properties, you must calculate the number of hits and misses as well as the memory references made by the sequence of addresses in the trace file. A memory reference is defined as an access to main memory in order to either update or read a block. You do not

need to model the data or main memory, we are just simulating the effect of the references, not dealing with actual data.

Once you have finished all of this for the unified model, then perform the same actions for the split model. Each side of the split should contain half the number of specified sets (e.g. if the user specifies 16 sets, the instruction cache has 8 sets and the data cache has 8 sets). All other specs remain the same.

V. Miscellaneous

You must submit proj3.c through Blackboard before August 1 at 11:59 PM. Be sure to include your name and FSUID in a comment at the top of the file. All submissions will be tested with plagiarism detection software – any cases of academic dishonesty will result in a grade of zero for the project. Late submissions may be submitted up to two days late for a 10% penalty each day.