1 Purpose

In this Project, you will be implementing a simulation of a cache. Actually, you won’t actually be simulating the cache itself, since you won’t store any data; however, you will be keeping track of the various cache lines, and will be able to tell me if a given access is a hit or a miss. You will support various cache line sizes and numbers of cache lines; you will also support both 1-way and 2-way set associative caches.

To break this down into simpler steps, I have defined a set of utility functions that you must implement. Some of them deal directly with the cache state, and others are simply mathematical tools which you might find useful in your cache implementation.

When we grade your code, some of our test cases will just test the utility functions - so you can get some partial credit even without implementing the cache. Moreover, you can get most of the credit even if you don’t support the 2-way set associative cache. However, to receive full credit, you will have to implement all of the pieces. I encourage you to implement this project one function at a time, and test each piece as you go - which will maximize the amount of credit you can earn.

1.1 Required Filenames to Turn in

Name your assembly language file asm4.s.

1.2 New Things to Try

In this Project, you will get experience with several new parts of MIPS assembly. First of all, you will be defining a struct for yourself, and using it; I’ll provide you a pointer to a buffer in memory, and you will fill it however you like.

Second, you will get a little experience with at least one (maybe more) assembly instructions; see below for details.

Third, you will get a lot more experience with defining (and calling) functions.

1.3 LRU

In the 2-way set-associative version of the cache, you will sometimes need to decide which of the two entries to discard (because you need to insert a new
entry). We will implement the LRU (Least Recently Used) strategy, which simply means that we will discard whichever of the two entries was accessed the least recently.

In real hardware, LRU is implemented with a single bit: you set the bit to 0 or 1, to indicate which one is newer. This is a good design for hardware, since it requires very little storage. But in code, it’s a little awkward. Therefore, my recommended implementation, in this situation, is that you have two slots, and define the first one to be the more-recently used one. (If there is an access that hits the address in the second slot, then swap them.)

However, the details of the implementation are up to you - if you prefer to do something different, that’s fine. However, note that when I call the readStoredTags() and getStoredValidBits() functions (see below), I will expect (however you actually store this information) that the more-recently-used information be returned in $v0 and the less in $v1.

1.4 Allowable Instructions
When writing MIPS assembly, the only instructions that you are allowed to use (so far) are:

- add, addi, sub, addu, addiu
- and, andi, or, ori, xor, xori, nor
- beq, bne, j
- jal, jr
- slt, slti
- sll, sra, srl, sllv, srav, srlv
- clo, clz
- lw, lh, lb, sw, sh, sb
- la
- syscall
- mult, div, mfhi, mflo

While MIPS has many other useful instructions (and the assembler recognizes many pseudo-instructions), do not use them! We want you to learn the fundamentals of how assembly language works - you can use fancy tricks after this class is over.

1.5 Special Comment Requirement
In this project, you will be defining the layout of a object, which represents the state of the cache (see below for details). You must document how this object is arranged, in a block comment at the top of your file.
2 Tasks

Your file must declare a set of functions, as detailed below. In this project, you
won’t have any global variables provided by the testcase; instead, everything
that you need to know will be provided through parameters.

Likewise, you must not define any global variables in your code. To
enforce this, I will (in some testcases) actually have two active cache “data
structures,” with different parameters and cache state.

2.1 Properly Saving Registers

As before, you must properly save registers (both sX and tX). I will be using
my testcases to check everything that I can!

2.2 No Output for You!

In this project, you won’t be printing anything. I’ll take care of all of it - you
simply need to implement the required functions.

2.3 Simple Helper Functions

Your first job is to write a series of simple helper functions. These functions
don’t have anything to do with the cache code, specifically, but each one will be
useful to you in writing the cache code later on. So I encourage you to write
these first.

Note that a lot of these functions are quite trivial - only a line or two of C
code. They won’t be hard to write in MIPS, either. Feel free to cut-n-paste
heavily, particularly for your prologue and epilogue.

2.3.1 shiftRightArithmeticV()

Implement the following function:

```c
int shiftRightArithmeticV(int val, int amount)
{
    return val >> amount; // use an arithmetic shift!
}
```

This function probably won’t be necessary for your cache code, but it’s a
trivial exercise in a new assembly instruction. Look at page A-56 of the textbook
Appendix to find an instruction which will be useful for this function. (I don’t
think that zyBooks includes page numbers. But the PDF that I’ve posted on
the class website does!)
2.3.2 \textbf{int buildMaskV(int bits)}

Implement a function which, given a parameter in the range \([0, 31]\) (inclusive) will return the mask which has that number of bits turned on. That is, if \(\text{bits}=3\), then return \(0x7 = 111\) binary.

You are permitted to do this however you can - but I can tell you that a loop isn't necessary. You can do this in just a handful of instructions. How? Here's a hint: every mask that you might build is one less than a power of 2.

2.3.3 \textbf{int logBase2(int)}

Implement a function which returns the base-2 logarithm of a number. You may assume that the number is nonzero, and also that it is exactly a power of 2.

Take a look at page A-52 to find an instruction that might help you. Or, if you prefer, write a loop - it takes a lot more instructions and is slower, but doesn't require you to learn a new instruction. Your choice!

2.4 \textbf{Cache Functions}

Now that you’ve written the simple helper functions, you will now write a series of functions that work on “cache” objects. I will provide the memory buffer for your objects - you will fill them up with whatever you want.

To do this, you will need to define what the fields of your object are; give them names, and define exactly how they will be laid out in memory. I do not care how you do this, but you must document your design in a comment at the top of your file.

Your object can include whatever fields you find useful (provided that you can fit them into the memory size that I provide). I would encourage you to store both basic properties (like the number of cache lines) and more complex, derived properties (like the number of bits in the tag) - but the details are all up to you.

Now, how will I give you access to this memory buffer? I will pass it as the first parameter to each of the functions below; this is how assembly implements the \textit{this} pointer in Java/C++, or the \textit{self} parameter in Python.

The first time that we use an object, I will call the \texttt{cacheInit()} method that you define; this is, basically, the object constructor. You may assume that the object is filled with zeroes when I pass it to you; fill it up with whatever you want.

For the rest of the methods, I will pass you that same pointer again; you may assume that the pointer I pass is always one that you’ve initialized previously with \texttt{cacheInit()}.

2.4.1 \texttt{cacheInit()}

This is the constructor for the cache objects. You may assume that the pointer that I provide (the first argument) points to a buffer which has at least \(64+\text{numCacheLine}\) words.
This method has 4 parameters:

- **cache** - the pointer to the object you should initialize
- **int numCacheLines** - how many cache lines there are in the entire cache (not the number of sets!). You may assume that this is a power of 2, and that it is no smaller than 4.
- **int cacheLineSize** - the size of each cache line, in bytes. You should assume that this is exactly a power of 2, and that it is it no smaller than 4.
- **int setSize** - the number of cache lines per set. It will be either 1 or 2.

### 2.4.2 int getIndex(cache, int addr)

This method extracts the ‘index’ part of the address out of an address. It must do it based on the properties of this particular cache object. Remember, ‘index’ is the part of the address which selects which set must hold the data from this address.

### 2.4.3 int getTag(cache, int addr)

See above. This returns the tag for the address.

### 2.4.4 (int,int) readStoredTags(cache, int index)

This method reads the stored tags (1 or 2) for the set with the given index.

If setSize=1, then return the tag in $v0 and set $v1 to zero. If setSize=2, then the more-recently-used tag should be in $v0 and the less in $v1. (See the discussion of LRU near the top of this spec.)

If a cache line is not valid, then return 0 for that tag.

You may assume that the index is valid for the given cache. You are **not** required to do error checking to confirm this.

### 2.4.5 (int,int) getStoredValidBits(cache, int index)

This method is the same as the previous, except that it returns the valid bits for the set, instead of the tags. (Return them in $v0 and $v1, for simplicity. Yes, it would definitely be possible to return 2 bits of information in a single value - but we won’t bother with that.)

### 2.4.6 int cacheAccess(cache, int addr)

This is the core of the cache simulation. This method models a single access to some particular address; since we don’t actually track the contents of any of the cache lines, we don’t care whether the access is a read or a write. Instead, this method simply tells you the address where the access occurs.
Your code must look into the current cache state: if the address is a hit, then return 1; if it is a miss, return 0.

In addition, your code must update the cache state. Before you return, this cache line must now be present in the cache; discard an old cache line (silently) if necessary.

If setSize=2, then the just-touched cache line must be in the “most-recently-used” position. If the set previously contained one valid entry, move that one into the other slot. If the set previously contained two valid entries, discard the older of the two to make space (if necessary).

2.4.7 cacheAccess() Example Pseudocode

I have provided some example code below. You are not required to implement your code exactly like this; after all, I don’t tell you what information your data structure should hold. However, this is a good example implementation - so you may simply implement it this way if you desire. (If you do something different, then make sure that the results of the function are equivalent, from the perspective of the testcase.)

(code on next page)
int cacheAccess(cache, int addr) {
    // this is the index of the *set* we’re accessing
    int indx = getIndex(cache, addr);

    // this is the *line* index, since we might have multiple
    // lines per set. This is the index of the first line in
    // the set.
    int line = indx * cache->setSize;

    if ( cache line [line] is valid and matches tag )
        return 1; // hit, no changes necessary

    if (cache->setSize == 2 &&
        cache line [line+1] is valid and matches tag )
    {
        // swap the two entries. In a *REAL* cache line, we would have a
        // 1-bit flag which tells you which is the "most recent" value;
        // swapping 1 bit is cheaper than swapping two words. But in
        // code, it’s simpler to just swap them.
        swap(cache line [line], cache line [line+1]);
        return 1;
    }

    // cache miss. But was there *anything* that we should save?
    if (cache->setSize == 2 && first line in set is valid)
        cache line [line+1] = cache line [line];

        ... set cache line [line] to 'valid' and having the correct tag ...

    return 0; // MISS
}

3 Running Your Code

You should always run your code using the grading script before you turn it in. However, while you are writing (or debugging) your code, it often handy to run the code yourself.

3.1 Running With Mars (GUI)

To launch the Mars application (as a GUI), open the JAR file that you downloaded from the Mars website. You may be able to just double-click it in your operating system; if not, then you can run it by typing the following command:

    java -jar <marsJarFileName>
This will open a GUI, where you can edit and then run your code. Put your code, plus one testcase, in some directory. Open your code in the Mars editor; you can edit it there. When it’s ready to run, assemble it (F3), run it (F5), or step through it one instruction at a time (F7). You can even step backwards in time (F8)!

3.1.1 Running the Mars GUI the First Time

The first time that you run the Mars GUI, you will need to go into the Settings menu, and set two options:

- **Assemble all files in directory** - so your code will find, and link with, the testcase

- **Initialize Program Counter to 'main' if defined** - so that the program will begin with `main()` (in the testcase) instead of the first line of code in your file.

3.2 Running Mars at the Command Line

You can also run Mars without a GUI. This will only print out the things that you explicitly print inside your program (and errors, of course)\(^2\). However, it’s an easy way to test simple fixes. (And of course, it’s how the grading script works.) Perhaps the nicest part of it is that (unlike the GUI, as far as I can tell), you can tell Mars exactly what files you want to run - so multiple testcases in the directory is OK.

To run Mars at the command line, type the following command:

```java
java -jar <marsJarFileName> sm <testcaseName>.s <yourSolution>.s
```

4 A Note About Grading

Your code will be tested automatically. Therefore, your code must:

- Use exactly the filenames that we specify (remember that names are case sensitive).

- **Not** use any other files (unless allowed by the project spec) - since our grading script won’t know to use them.

- Follow the spec precisely (don’t change any names, or edit the files I give you, unless the spec says to do so).

---

1. Why can’t you put multiple testcases in the directory at the same time? As far as I can tell (though I’m just learning Mars myself), the Mars GUI only runs in two modes: either (a) it runs only one file, or (b) it runs **all** of the files in the same directory. If you put multiple testcases in the directory, it will get duplicate-symbol errors.

2. Mars has lots of additional options that allow you to dump more information, but I haven’t investigated them. If you find something useful, be sure to share it with the class!
• (In projects that require output) match the required output exactly! Any extra spaces, blank lines misspelled words, etc. will cause the testcase to fail.

To make it easy to check, I have provided the grading script. I strongly recommend that you download the grading script and all of the testcases, and use them to test your code from the beginning. You want to detect any problems early on!

4.1 mips_checker.pl

In addition to downloading grade_asm4, you should also download mips_checker.pl, and put it in the same directory. The grading script will call the checker script.

4.2 Testcases

You can find a set of testcases for this project at

http://lecturer-russ.appspot.com/classes/cs252/fall18/asm/asm4/

You can also find them on Lectura at /home/russelll/cs252f18_website/asm/asm4/.

For assembly language programs, the testcases will be named test.*.s. For C programs, the testcases will be named test.*.c. For Java programs, the testcases will be named Test.*.java. (You will only have testcases for the languages that you have to actually write for each project, of course.)

Each testcase has a matching output file, which ends in .out; our grading script needs to have both files available in order to test your code.

For many projects, we will have “secret testcases,” which are additional testcases that we do not publish until after the solutions have been posted. These may cover corner cases not covered by the basic testcase, or may simply provide additional testing. You are encouraged to write testcases of your own, in order to better test your code.

4.3 Automatic Testing

We have provided a testing script (in the same directory), named grade_asm4, along with a helper script, mips_checker.pl. Place both scripts, all of the testcase files (including their .out files), and your program files in the same directory. (I recommend that you do this on Lectura, or a similar department machine. It might also work on your Mac or Linux box, but no promises!)

4.4 Writing Your Own Testcases

The grading script will grade your code based on the testcases it finds in the current directory. Start with the testcases I provide - however, I encourage you to write your own as well. If you write your own, simply name your testcases using the same pattern as mine, and the grading script will pick them up.
While you normally cannot share code with friends and classmates, testcases are the exception. We encourage you to share you testcases - ideally by posting them on Piazza. Sometimes, I may even pick your testcase up to be part of the official set, when I do the grading!

5 Turning in Your Solution

You must turn in your code using D2L, using the Assignment folder for this project. Turn in only your program; do not turn in any testcases or other files.

5.1 Late Work

I will set up a separate folder in D2L for Late work. Please do not put any files in that folder unless you intend to take a late day. (If you put files in both folders, we’ll ignore the regular folder and only use your files from the Late Day folder.)