# ECE 2500 Project 1

**Fall 2017**

**Total points:** 100

**Version:** 1 (Any updates will be announced on Canvas.)

**Deadline:** October 31st at 9:00 PM

**Late Policy:**Projects have strict deadlines and have to be digitally submitted at 9 PM on the day they are due. If submitted before 10 PM, a 5% reduction in grade will apply and the project will still be accepted. If submitted by 9 PM of the day after due date, a 20% reduction in grade will be applied and the project will still be accepted. No project will be accepted after that point without a letter from Dean’s Office. In order for your projects to be accepted, you need to complete the submission process. Uploading files but not making them available to the instructor, submitting the wrong version, and other technical or non-technical issues do not make you eligible for a late submission.

**Note 1**: For this project, we will use Moss, a tool developed by Stanford University to detect similarities in code structure across both sections. Moss cannot be defeated by changes to variable and function names, function ordering, formatting changes, and comments. Any strong similarities flagged by Moss will be carefully examined and possibly submitted to the Office of Undergraduate Academic Integrity.

**Note 2**: For the sake of making it easy for the GTA to compile and run your source code, you must include basic instructions for running your code in your submission. Visual Studio is the preferred development environment. If you prefer using another IDE then port your code to Visual Studio after debugging. For this project you are allowed to use **standard** libraries for fundamental needs such as linked lists, parsing, etc.

**Project Description:**

Build a MIPS core instruction *assembler* using either C++ or Python code. An assembler reads the human-readable assembly instructions of the MIPS ISA and converts them to the corresponding 32-bit machine code (in hex). Manual assembly is very tedious and error prone, so it makes sense to write an assembler program. The input to the assembler is a text file named \*.s, where \* is the name of an assembly file in the same folder as the assembler executable. The output of the assembler is another text file called \*.obj (same name but different extension). The executable will be named myAssembler and is called from the console as follows:

myAssembler \*.obj

This command will have myAssembler read \*.s and generate \*.obj. If there is an existing \*.obj file simply overwrite it.

Your grade will be mostly based on the correctness of the machine code in the output file \*.obj. You can use the console to print out messages for debugging purposes. The myAssembler requirements are:

* The input of myAssembler is a text file containing MIPS assembly code. Each line will contain the MIPS assembly language instructions, possibly preceded by a label on the previous line if and only if that instruction is the target of a branch instruction.
* The output of myAssembler is a text file containing one line per instruction. Each line contains a 32-bit instruction encoded as 8 hexadecimal digits **without** a leading “0x”. Since object code is *relocatable* you don’t really need to know the absolute addresses of the instructions. Assume addresses start at 0 treat all addresses as **relative**.
* myAssembler should be able to assemble all the instructions in the **core** instruction set with the exception of j and jal instructions. The core instructions are summarized in the top left table of the MIPS reference card, also known as the Green Card.
* myAssembler should read a line at a time and print an error message **to the command line** for each line of the assembly code file that contains an error. In other words, myAssembler should report all errors found in an input file with error messages to the command line and then **exit without producing an output file**. Examples of errors are: lines that don’t contain assembly instructions or a label, or encountering an instruction that cannot be assembled. The error message should say “Cannot assemble <string> at line <number>”. **Grading Scheme:**

|  |  |
| --- | --- |
| **Component** | **Points** |
| The skeleton of the assembler | 30 |
| R-type instructions | 10 |
| I-type arithmetic and logical instructions | 10 |
| Branch instructions | 20 |
| Error checking | 10 |
| Documentation and 1-page report | 20 |

The code should be well structured and documented. A one-page report should give a high-level description of your solution including algorithms and data structures used. If needed, you can add a maximum of one page to your report.

Hint for debugging: Qtspim can help check that you have generated the correct output. When you load the assembly code into QtSpim, it gets assembled and loaded into the simulator memory. You can see the machine code in hex format under the Text tab, in the second column of the table. Use Qtspim as a guide when in doubt about what your assembler should do in certain situations. In the QtSpim Simulator setting, you need to check Enable Delayed Branches. Otherwise, the numbers you expect to be encoded as the immediate for the target would be different from the lecture notes.

The assembly instructions from the test cases should be preceded by the following assembler directives at the start of the file in order to have QtSpim assemble and load your instructions correctly:

.globl main

.data

.text

main:

< your assembly instructions>

ori $v0, $0, 10

syscall

When this is done you will see the assembled code appear in the Text tab beginning at address 0x00400024. The ori and syscall instructions are the return from main(). Be sure to always use the Qtspim File -> Reinitialize and Load File command after changing the assembler source code.

There are a set of test input (.s) files provided on the Project 1 assignment page along with the desired outputs (.obj) for test cases without errors. Please note that your final submission will be run on similar test cases so you have to follow the input and output protocols exactly. We may add to the test suite, and you should also generate some of your own test cases to check for as many correct and incorrect scenarios as possible.