ECE 2500 Project 1
Fall 2018

Total points: 100

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

Deadline: Thursday October 25th at 11:59 PM

Late Policy: Projects have strict deadlines and have to be digitally submitted at 11:59 PM on the day they are due. If submitted before 2 AM the next day, a 5% reduction in grade will apply and the project will still be accepted. If submitted by 11:59 PM on the day after the 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 submissions. 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 TA to compile and run your source code, you must provide build instructions with your submission. For this project you are allowed to use standard libraries.

Project Description:

Build a MIPS *core instruction disassembler* using a programming language of your choice. As the name implies, a disassembler is the inverse of an assembler: It reads the binary (formatted as hexadecimal) instructions executed by a MIPS processor and converts them to symbolic assembly source code read and written by people. Manual disassembly is very tedious and error prone, so it makes sense to write a disassembler program. The input to the disassembler is a text file named `*.obj`, where * is the name of a machine code file in the same folder as the disassembler executable. The output of the disassembler is another text file called `*.s` (same name but different extension). The executable will be named `myDisassembler` and is called from the console as follows:

```
myDisassembler *.obj
```

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

Your grade will be mostly based on the correctness of the object code in the output file `*.s`. You can use the console to print out messages for debugging purposes. The `myDisassembler` requirements are:

- The input of `myDisassembler` is a text file containing MIPS machine code. 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.

- The output of `myDisassembler` is a text file containing one line per instruction. Each line will contain the MIPS assembly language corresponding to the object code, possibly preceded by a label on the previous line if and only if that instruction is the target of a branch instruction.

- `myDisassembler` should be able to disassemble all the instructions in the **core** instruction set with the exception of `j`, `jal`, and `jr` instructions. The core instructions are summarized in the top left table of the MIPS reference card, also known as the Green Card.

- In the case of `bne` and `beq` instructions, a label should be generated with a format **"`Addr_####:`"** where **"`####`"** is the address (rather than the relative offset) of the branch target, expressed in hexadecimal, starting at 0000 and incremented by 0004 after each instruction. The label also needs to be generated for the instruction at that address. The simplest way to do this is by taking two passes over the object code. The first pass looks for branch instructions and records the addresses of all branch targets in a list. The second pass uses this list to generate labels at the beginning of any instruction that is the target of a branch instruction. At most one label should appear on any instruction even if the label is referenced by more than one branch instruction.

- `myDisassembler` should read a line at a time and print an error message **to the command line** for each line of the machine code file that contains an error. In other words, `myDisassembler` should report all errors found in an input file with error messages to the command line and then **exit without producing an output file**. The error checking can be done on the first pass. Examples of errors are: lines that don't contain 8 hexadecimal digits, or encountering an instruction that cannot be disassembled. The error message should say **"`Cannot disassemble <string> at line <number>`"**.

- Immediate values in the `myDisassembler` output should appear as signed decimal numbers, and registers should be mapped to the names

appearing in the Name (leftmost) column of the Green Card's "Register Name, Number, Use, Call Convention" table (e.g. `$t0, $s0`).

Grading Scheme:

Component	Points
The skeleton of the disassembler	20
R-type instructions	15
I-type arithmetic and logical instructions	20
Branch instructions and labels	20
Error checking	5
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 you have generated 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 disassembler 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 created for 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 (`.obj`) files provided on Canvas along with the desired outputs (`.s`) for test cases without errors. One of the test files contains errors and hence has no .s file. 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.