1. Project Description:
   * This project is a MIPS compiler. It takes assembly code from a file, “asm.dat”, and converts it into machine code and data segment. The entire compiler is written in the MIPS assembly language, to assemble MIPS assembly code.
   * The compiler handles labels, integers, comments, all standard data types (.ascii, .word, etc.), and converts the CORE INSTRUCTION SET of MIPS assembly.
   * All three instruction types are supported and addresses from labels in branch and jump commands are properly converted.
   * The syntax interpreted by the compiler is very similar to that used by Mars.
2. The project was implemented in 3 main parts:
   * The fileProcessor
     1. This part reads and loads the source file into a buffer in memory.
     2. Then the dataProcessor is called (see below)
     3. Then after the data segment is processed each relevant line is picked out.
     4. fileProcessor determines if the next relevant source data is a label or an instruction
        1. If it’s a label, it is added to label buffer
        2. If it’s an instruction, it is passed to processLine (see below)
     5. After all lines have been processed, then the labels must be processed
        1. All instructions that had a label added their information to a label reference buffer
        2. Each entry has the label needed looked up and it’s address updated with what the label points to from the label buffer
     6. After all labels have been processed, then the outputs are made
        1. Code output in “instructions.dat”
        2. Data output in “dataSegment.dat”
   * The dataProcessor
     1. The dataProcessor finds each line of relevant data from the source file
     2. Writes the label to the label reference buffer (if there is one)
     3. Determines what type of data the line is
     4. Processes the data on the line and adds all data to the data buffer correctly
     5. Return once the .text segment begins in the source file
   * The lineProcessor
     1. This part finds the instruction type and format
     2. Then converts the arguments (registers, immediate, etc.) into numbers
     3. If an argument is a label the instruction and label are added to the label reference buffer so that later the instruction can be updated with the address that the label points to
     4. The parts of the instruction are combined into one hex value and put into the code buffer
3. The process taken to tackle the program was to start with the small individual pieces of the program and work up to the larger flow controlling ones.
   * First was lineProcessor
     1. We wanted to be able to process individual lines first and make sure that our methods and techniques work on the actual bulk of the compilation before sorting through data and files.
     2. The label processing part of this came later
   * Next was the fileProcessor
     1. This was a higher in the control hierarchy, and so we started small
     2. First it could read a file and pass lines to line processor
     3. Then it could ignore extra whitespace
     4. Then it could support comments, etc.
     5. Then labels were implemented, which involved several buffers working out of fileProcessor. This made it much more of its own part and not just the line iterator.
     6. Then the data segment part was developed (see below)
     7. Then the output was made to finalize the whole project.
   * The dataProcessor came after we had already functioning instruction processing capabilities, and as such already had much of the functions needed to carry out processing lines in the data segment of the source file.
     1. Similar to how we had done the line and file processing we broke this into a hierarchy of functions.
     2. First came the individual data processors. These would be the ones that dealt with the data once it was known what the type was (there was a different one for each new data type).
     3. Then came the general line processor, which dealt with
        1. the labels similar to the line processor above
        2. finding the data type and function to call to process the data
     4. After that came the general data processor that found the relevant lines of instruction to pass to the line processor and ignored non-relevant things.
4. The two biggest challenges were passing values around correctly through function inputs and outputs, and the label processing system.
   * Inputs/arguments and outputs for functions
     1. Due to how we split functions many had to have standardized arguments and return types.
     2. Determining which were needed and how many we were allowed to take and give (i.e. limited $a and $v registers), guided much of the function design and height of the function tree.
     3. Remembering which registers were what for which functions became a great pain as more and more were added.
     4. Abstraction helped a great deal by adding more functions to layer over complex functions to make them appear simple.
   * The label processing method changed several times during the planning phase.
     1. Originally (as you could see in the checkpoint) we were going to loop through the source twice to find all the functions, and then convert all the instructions.
     2. This was much more complicated and would have resulted in much overlap as the instruction count needed to be known for the label loop, but if you could tell what was/wasn’t an instruction you might as well convert it there.
     3. The solution was to convert the functions first and use a system of buffers to go back after the code was all done and add back in the missing label values once they were all known.
5. I have learned a lot about function structuring, MIPS assembly, bitwise arithmetic, and ascii.
   * Functions were pretty abstract to me before this project, but once I had to make my own and manage all the registers and stacks myself they became very real. Recursion was always annoying to me and I never understood why it was needed when iteration was so much easier in other languages. Assembly taught me that recursive functions are not that hard and that stacks make everything easier.
   * MIPS assembly goes without saying really. I started vaguely aware of assembly, but now that I’ve done this project I feel like I’ve known it for a long time now and am very comfortable in the environment.
   * Bitwise arithmetic before the project was another fringe subject. I had never learned a very low level language before that needed bitwise operations. This was a shock when every function in some way used and, or, andi, ori, sll, srl, etc. I now feel that I have a stronger grasp on how computers in general work on the most fundamental level of data processing.
   * Ascii was again kind of a fringe idea. I know that chars had values but never in my life have I had to reference a table for values as much as I did here. We were told that our greensheet was going to be super helpful for MIPS, as it turns out I have used it way more simply because it has an ascii table on it. Most valuable data on it.