**MIPS Assembler**

**High Level Description**

The file is first parsed by code that was supplied to us. That code takes the labels, names, and register/offset/constants found in the unassembled instructions and puts them in a structure and then into a vector. This implementation of the MIPS assembler makes two passes over those structures. The first pass goes through the unassembled file and finds all of the labels. It then creates a table (which is really a QList of QPairs) which is later used to determine the values to place in the branch instructions. The assembler then makes a second pass over the file, and actually assembles the instructions, plugging in the correct values for the branch instructions and simply assembling the rest. The assembled instructions are parsed one at a time, and the result of each instruction assembly is output to a text file with the same name as the input file.

**Assembling Process**

The assembler has a different if-else statement for each instruction name. When an “add” instruction is run through the assembler for example, an if-else catches that instruction name, assigns the input values to the correct variables in memory, and then passes them to the logical assembler. Depending on whether the instruction is “R-type” or “I-type” the logic is slightly different. Bit shifting is used to put each part of the instruction into a 32-bit unsigned integer. Once the instruction has been assembled it is returned to the main function and output to the output file.

**Assembling Algorithm**

My assembler pseudo-code algorithm for R-Type instructions looks like this:

R-Type Assemble Function()

assembled = assembled | (opcode << 26);

assembled = assembled | (($rs & lower5BitMask) << 21);

assembled = assembled | (($rt & lower5BitMask) << 16);

assembled = assembled | (($rd & lower5BitMask) << 11);

assembled = assembled | ((shamt & lower5BitMask) << 6);

assembled = assembled | ((funct & lower6BitMask) << 0);

return assembled;

As you can see, I simply get all of the necessary values to put into the assembled instruction, and then using bit masks, I logically shift and combine each piece of the assembled instruction. I-type instructions follow a similar algorithm that accounts for labels.

**Dealing with Labels and Branches**

In order to ensure that the labels work correctly, I first had to make a pass over the file and create a symbol table that holds the name and location of each label in the unassembled file. I then had to check each beq and bne instruction to see if the immediate value was a number or a label. If the value was a label I look in the symbol table and determine that the immediate the difference of (Position in the file of the symbol – Current Position of the Instruction – 1). The result of this goes into the assembled result as the immediate. If the immediate is not a label and is just a number, I put the number into the resulting branch instruction.