Palmer Robins

Programming #1 – Design Document

Overview: This program receives a file of binary strings and checks that file to ensure syntactic correctness, including that its length equals 32 bits, and that the instruction it represents is valid. Then, the program converts each string into its corresponding MIPS assembly instruction, outputting both the binary and assembly instructions to the command line. The classes used in this implementation include BinaryParser, Opcode Table, Register Table, and Instruction. Finally, a file named Binary.cpp executes the conversion process.

Classes:

1. BinaryParser: This class opens the input file and checks for syntax correctness. Parsing each binary string, it stores its opcode, function field and any register/immediate values in its list of instructions. Then, it uses the instruction, which contains all the data that is needed, into its corresponding assembly string.
2. Instruction: As mentioned in BinaryParser, this is the place we can store information about all of the individual instructions included in the input file. Along with all the information listed above, it stores the binary string (encoding) as well as the final assembly string.
3. Register Table: This class stores data corresponding to the MIPS register system. If a register exists in MIPS, both its name and number are listed in a specific entry.
4. Opcode Table: All of the MIPS instructions that the program supports for translation are listed here. We can store information that is needed concerning each instruction, like its name and any registers that will have to be present within the binary string. Any info needed about an instruction we can translate will be accessible through its corresponding entry in Opcode Table.

Organization: Binary.cpp reads the filename from the command line, which it then passes to the Binary Parser that is created just afterward. In Binary Parser, we open the file, assure that it opened correctly, and then check the file's syntax. Iterating through the file instruction by instruction, we check to make sure all opcode values are contained within our Opcode Table, along with assuring that all register values are contained within our Register Table. After checking syntax, we read in the opcode for that instruction, which is always contained in the first six digits. We also read in the final 6 digits, where the function field will be (if an RTYPE). Using these two fields, we can get the corresponding opcode by sending them to Opcode Table and iterating through our list, checking for matching opcodes and (if needed) function fields. Once we have a valid opcode, we can use the Opcode Table to also get the opcode's name (in the form it will be when printed) and its instruction type. We send the line, without the opcode, into the decode function that matches the instruction type, where the instruction class instance is set, including any register values or immediate values included in the instruction. After checking that the process worked correctly, we send this instruction instance into the function to write the string containing the MIPS assembly that corresponds to its instruction type. That string is set as the MIPS assembly string in the instruction instance after it is returned. Finally,

we add that instruction into our vector of instructions, and Binary.cpp prints it, after which we can continue to the next line, where the process is repeated.

Adding A New Instruction: The first step is to add the instruction to the Opcode Table. If all of that info is correct, this should be the only change you need to make to this code.