Project 1 Report

Alex Lee March 1, 2023

In this project, an *.obj file is required as an input with several hexadecimal instructions located inside of it. To extract and decompile these instructions, the file needs to be read from the command line then loaded into an appropriate list. If the file cannot be found or does not have the extension .obj, then an error is thrown. Once all the instructions are in the list, error checks are made to ensure that the length is of the right size and the code is a hexadecimal. If not, then an error will be printed out and a build file will not be made.

To decompile each line of code, a lookup table is created using unorganized lists where each opcode has a key. One list is for “primary” names; they simply use their opcode as their instruction. The other list is for “secondary” names; in this case the opcode is 0x0 and thus requires the function part of the instruction. The key to each instruction is their hexadecimal counterpart. Lastly, a list for register names is created to assign each register name ($v0, $a0, $t0, etc.) to their corresponding decimal value as their key. These three lists will help to determine each code’s MIPS instruction.

Next, a bitmask is used to see what the opcode is then binary shifted until only the specific portion of the code is seen. This determines if the opcode is an “R” instruction or “I” instruction. If it is an “R” instruction, it is checked if the “secondary” list of codes is required. If it is, then a bitmask is used on the function part of the hexadecimal instruction and compared against the “secondary” list to see which instruction is required. If the opcode is not 0x0, then the “primary” list is checked for the opcode name. If the opcode is not found, then an error is printed and the instructions are not pushed to the *.asm file.

Once the code is found, it is put into a Pair with their instruction type (‘R’ or ‘I’). This lets the program know how to interpolate the remaining instruction. In decompiling R instructions, only rs, rt, and shamt are required to be converted into a register/number. In decompling I instructions, only rs, rt, and immediate are required to be converted into a register/address/offset. Each register is converted into a string using the lookup table. If it is not found in the lookup table, an error is printed. From here, conditional statements are used to determine how to assemble the MIPS instruction. In R instructions, ‘sll’ and ‘srl’ are differentiated from the rest of the R-type instructions requiring this format: “name rd, rt, shamt”. The rest of the R functions follow this format: “name rd, rs, rt”. For I type instructions, ‘lbu’, ‘lhu’, ‘ll’, ‘lw’, ‘sb’, ‘sc’, ‘sh’, and ‘sw’ are grouped in load/store type instructions and ‘beq’ and ‘bne’ are grouped in branch type instructions. Load/Store type instructions follow this format: “name, rs, imm(rs)”. Branch instructions are special since they require an offset and a label. To determine the label name, the offset is added to the current line number and multiplied by 4 then converted into hexadecimal. This is added onto “Addr_” to give “Addr_XXXX” where XXXX is the address. Then, the instruction is constructed to this format: “name rs, rt, Addr_XXXX”. The numerical address XXXX is then added to a list called branchAddresses to be used later in the program. If XXXX is either negative or larger than 16 bits, then an error is printed.

Each instruction is added to a vector called decompiledCode. A vector is chosen for its simplicity and resemblance of use to an array, but with extra functionality. After every instruction is added, the label’s are added to the corresponding decimal address. The addresses in branchAddresses are sorted from least to greatest then loaded from back to front. This is to avoid having to change the index of decompiledCode each time a new label is added to the array. The location to be inserted in decompiledCode is calculated by dividing XXXX from branchAddress by 4 to get the correct line to place it in. Then, at that index “Addr_XXXX:” is placed into decompiledCode.

Finally, each index of decompiledCode is written into a file named *.asm where * comes from the *.obj file. The .asm file can be found in the same directory as the .obj.