William Lin

March 24, 2017

Dr. Martinez

CSCI 2214-02

Project 1 Report

1. The project was successful as long as the input file is strictly formatted to comply with the exact syntax of my program. Or else the program may throw up errors. All parts of my program work and I can encode all types of instructions including load, store, branch, ALU, ALUi, JR, and J type instructions.
2. You will find the documentation of my code on a separate page of this document. I have a main.cpp file, a Parser class and an Instructions class. I also use the tools class which is made readily available to CS majors by Dr. Fischer. The main class just creates an instance of the Parser class with the test file to encode, the instruction file and the output file that the encoding will be outputted to.
3. **Step 1:** main creates a Parser object and passes it the input file and a file containing the instructions.

**Step 2:** In the Parser object constructer, it opens the input file containing the instructions. It also creates and instance of the Instructions class object with the instructions text file. The instructions class just reads in all the instructions and stores them into two vectors. One vector for the names of the instruction and one for the opcode of the instruction. The instruction text file also indicates which instructions are R type instructions so those are specified in the instruction text file and the Instruction class will see if the instruction that it is storing is a rtype or not. If it is it makes a reference that this instruction it just saved is a r type. This will be used to check of the instruction is a rtype or not. Back to the parser.

**Step 3:** It then reads character by character without skipping whitespace and send to lower case. If the character read in is alpha, a digit or any of these character: ‘#’, ‘(’, ‘)’, ‘-’ then store that character onto the end of a temporary string named temp. If the char read was a colon ‘:’, then push the temp string back into a labels vector that stores label names. Also push back the line number in a vector that stores int. This vector will store the line numbers each label is at. Whenever the char read in is a newline character it will push whatever is in string temp back onto the vector line. And then push line into a vector of vector strings called allLines. allLines stores every line of the instruction file after the instructions have been parsed and broken up.

**Step 4:** The Parser object has not been constructed and now it is back to main. main then takes that Parser object and called encode() which main will pass the target output file to.

**Step 5:** encode opens the output file and gets ready to write the output to it.

**Step 6:** encode then loops through allLine and parses each vector <string> line by sending it to the lineEncode function with the line number which will return an unsigned int that represent the binary code before being converted to hex.

**Step 7:** Create an unsigned int named toReturn and set it equal to 0. So toReturn now holds 32 bits of 0. We push the rs1, rs2, rd, and imm bits onto this unsigned int variable. We also create an int named cursor that will keep track of where in the toReturn variable the next bit to be pushed onto the toReturn variable is going. We set cursor = 32 because it is the left most position of the first bit.

**Step 8:** The opcode is always the first thing in the line and is always 6 bits. So, we find the opcode for the instruction and use bit operations to insert the 6 bit opcode into a toReturn unsigned int. The opcode right now has it’s 6 bits to the right most bit position. We have to move it to the left most so we take the opcode bits and left shift it cursor minus 6 (the bit size of the opcode) bits. Then we set the toReturn variable to the AND of the maskedOpcode with the toReturn variable. Now we have the opcode in the first 6 bits of toReturn and this will happen no matter what type of instruction type it is.

**Step 9:** lineEncode checks if the line contains 2 words, then it is a j type instruction so go to step J. If the line contains 3 words then it could be either load store or branch instructions so go to step B. To check if the instruction is an rType or I type the opcode is passed to a isRtype function in Instructions class object and if it is an Rtype then true will be returned. So, if the instruction is an Rtype go to Step R but if not then it is an itype instruction so go to step I.

**Step J:** If the opcode is J or JAL then all we do is find the label and calculate the label offset based on the line number the label is at. By taking (line that the label is found at \* 4) – (the line number \* 4 – 4). We then take that label offset number AND it with the number 0x3FFFFFF and store it back into lbl. Then we push it back onto toReturn by using toReturn |= lbl. However, if the opcode is JR or JALR then we just find the register at the second index of line and get the number of register using reg.susbtr(1). We push the register encoding onto the end of toReturn and leave the rest 0 because there are no rs2 or rd or imm. Go to Step 10.

**Step B:** If line at index 1 contains an opening parenthesis then we know it is a store but if at index 2 it contains an opening parenthesis then we know it is a load. If there are no parenthesis found then we know it is a branch instruction. If a parenthesis is found, the register inside the parenthesis is the rs1 and the other register is rs2. The imm is found and the bit operation to push them onto the toReturn variable is done. If it is a branch instruction then rs1 is found and the label offset is found and set to the imm. The rs1 is pushed onto toReturn and then a 5 bits are skipped to keep rs2 equal to 0 because rs2 is not used. The imm is pushed on and toReturn is complete for branch type instructions. Go to Step 10.

**Step R:** We move the cursor to down 6 spaces because the op code is left at 0 for R type instructions. We start by putting rs1 and rs1 onto the toReturn variable and then finish it off by adding the 11 bit func op code at the end. Go to Step 10.

**Step I:** Find rs1 and rs2 and push them onto toReturn. Then find the imm by accessing the last word in the line which should be the imm. Push the imm on the toReturn variable and go to step 10.

**Step 10:** Return toReturn which will be stored printed out in hex to the output file and the console. Repeat from step 7 until all lines have been encoded.

1. You will found my project out attached page that reads Output.

**Documentation**

main.cpp

* Creates an instance of Parser class and uses the Parser class to encode and output encoding.

Instructions

* Data Members:
  + vector<string> instrName : stores the instruction Name. Must be used with the vector of opCodes
  + vector<int> opCode : stores the opcode number of each instruction in instrName
  + vector<string> rTypes : stores a copy of the instruction names that are r types
* Functions:
  + Instructions(string) : opens the instructions file and parses through line by line and each line, it sends to parseInstr(). This will build the Instructions object with the given instructions from the instruction file.
  + vector<string> parseInstr(string) : takes a line from the instruction file and checks to see if it is an R type and then stores the instruction name and op code. If it is an r type I also stores a copy of the instruction name into the rTypes vector for later use. Returns the broken instruction name in vector position 0 and the opcode in 1.
  + int getOpCode(string) : return the opcode of the instruction that was passed into getOpCode.
  + bool isRType(string) : takes in a name of a instruction to check if it is an RType. Looks through the rType vector to see if there is a match. If there is then true is returned. Else false is returned.

Parser

* Data Members:
  + vector< vector<string> > allLines : this vector of vector<string> will store every line broken up. Also, when the lines are parsed, the labels are taken out and stored.
  + vector<string> labels : this will store all the labels by their name.
  + vector<string> lblIndex : stores the line number for each label in labels.
  + Instructions instr : and instance of the Instructions class object which will be used for finding the opcode of the instruction and checking what if it is an RType instr.
* Functions:
  + Parser(string, string) : Takes the input file name and the instruction file name. Creates a Instructions object with the instruction file name. Then reads in the input file and breaks up the input file by line then by the words in a line and stores them into allLines.
  + void encode(string) : encode takes in the output file name and opens it and gets ready to write the encoded instructions to it. It calls lineEncode by passing it a line of instruction to encode and the line number. The returned encoding is printed to the output file and also the console.
  + unsigned int lineEncode(vector<string>, int) : takes in the vector<string> line and the int line number and encodes the line of instruction and returns the unsigned int of the encoding that will be printed in hex in encode.
  + int getLblIndex(string) : takes the label name it is passed and finds it in the vector<string> labels and returns the line index of the label.

**Output**

**Test file:**

TOP:BEQZ r1,STOP

lh r3,-4(r12)

jr r1

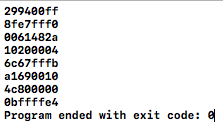
sub r2,r4,r5

addi r3,r4,3

j top

stop:sb 4(r4),r3

**Output:**

****

**In binary:**

0010 1001 1001 0100 0000 0000 1111 1111

1000 1111 1110 0111 1111 1111 1111 0000

0000 0000 0110 0001 0100 1000 0010 1010

0001 0000 0010 0000 0000 0000 0000 0100

0110 1100 0110 0111 1111 1111 1111 1011

1010 0001 0110 1001 0000 0000 0001 0000

0100 1100 1000 0000 0000 0000 0000 0000

0000 1011 1111 1111 1111 1111 1110 0100