Lab3 (RISC-V Disassembler)

Approach: -

Input: -

- Read the input file line by line.
- Convert hex to binary and make vector<string> to store binary code of instruction.

Disassembler: -

- Iterate through vector<string> and find opcode.
- Find the instruction type from opcode by calling the getOpcode function that returns the type of instruction, i.e., I, R, etc.
- Now call the function corresponding to the instruction to dessemble.
  E.g.: - If R-type call RType(...) function, it will return assembly syntax for the line.
- Store the output in vector<string> output.

Print Output:-

- Print the output store in the output (vector<string>)

Description of function:- (Eg:- RType) (Explaining one, all other similar)

```
string RType(string &line) {
    string funct3 = get_funct3(line);
    string rs1 = get_rs1(line);
    string rs2 = get_rs2(line);
    string rd = get_rd(line);
    string funct7 = line.substr(0, 7);
    string instr = "";
    if (funct3=="000") {
        if (funct7=="0000000") instr = "add";
        else if (funct7=="0100000") instr = "sub";
        else return "Invalid instruction of R type";
    }
    else if (funct3=="001") instr = "sll";
    else if (funct3=="010") instr = "slt";
    else if (funct3=="011") instr = "sltu";
    else if (funct3=="100") instr = "xor";
    else if (funct3=="101") {
        if (funct7=="0000000") instr = "srl";
        else if (funct7=="0100000") instr = "sra";
        else return "Invalid instruction of R type";
    }
    else if (funct3=="110") instr = "or";
    else if (funct3=="111") instr = "and";
    else return "Invalid instruction of R type";
    return instr + " x" + rd + ", x" + rs1 + ", x" + rs2;
}
```

- R-Type instruction have rs1, rs2, rd, funct3, funct7.
- Get those from the binary code as per the RISC-V card.
- Now, on the basis of funct3 and funct7, get the instruction name.
- By rs1, rs2, and rd, get which registers are used.
- At last, return combining instruction name and registers.
- get_rs1, get_rs2, get_rd:- return the decimal corresponding to 5 binary bits.
- If none, funct3/funct7 matches return Invalid Instruction.

- Similarly, for other types of instruction where imm also used, we convent imm into signed decimal. Also some places add ‘0’ to imm.

```
string get_funct3(string &line) {
    return line.substr(17, 3);
}

string get_rs1(string &line) {
    string rs1 = line.substr(12, 5);
    // convert binary to decimal
    int dec = 0;
    for (int i = 0; i < rs1.size(); i++) {
        dec += (rs1[i] - '0') * (1 << (rs1.size() - i - 1));
    }
    return to_string(dec);
}
```

Two code files: -

- disassembler.cpp: - Part1, Part2, Part3
- part4.cpp: - Part4

Changes done for Part4: -
For B/J instruction: -

- When we get converted hex code to assembly code, map the ‘offset+pc’ to label (Here pc is i*4, where i is ith line in the input starting from 0).
- Now change offset to label and push into output.
- For each line check if (i*4) is in the map, then add the label corresponding to i*4 in the front and push to output.
- Label starting from L0 and increase, i.e., L0, L1, L2, … .

// changin the offset in B Type (similar in J Type) and storing label in map

```
else if (opcode == "B") {
    // rename line number with lable name using map
    string imm;
    imm = line.substr(0, 1) + line.substr(24, 1) + line.substr(1, 6) + line.substr(20, 4) + "0";
    // convert binary to decimal
    int dec = 0;
    dec = - (imm[0] - '0') * (1 << 12);
    for (int i = 1; i < imm.size(); i++) {
        dec += (imm[i] - '0') * (1 << (imm.size() - i - 1));
    }
    if (label.find(dec+i*4) == label.end()) {
        label[dec+i*4] = textLabel + to_string(c);
        c++;
    }
    output.push_back(BType(line) + label[dec+i*4]);
}
```

// Adding label in front if (i*4) in map

```
if (label.find(i*4) != label.end()) {
    string temp = output.back();
    output.pop_back();
    output.push_back(label[i*4] + " : " + temp);
}
```

Testing the code: -
Made test cases: -

- First, using the given code in the Assignment Problem
- Made test cases from the assembly examples given in the RISC-V simulator: - complexMul.s, factorial.s,
- Compare the output of my code and the code given in the simulator.

A minute difference is there. Sometimes, in a simulator, the imm value is in hex format but returned in decimal format in my code.
This is because the hex code for both is the same, Eg: -

- addi x12, x0, 0xff :- 0ff00613
- addi x12, x0, 255 :- 0ff00613