```
//  main.cpp
//  isa-comp-arch
//
//  Created by Avanish Shrestha on 5/15/17.
//  Copyright © 2017 Avanish Shrestha. All rights reserved.
//

#include <iostream>
#include <string>
#include <vector>
#include <sstream>
#include <cmath>
#include <stdio.h>
#include <iomanip>

using namespace std;

struct Register {
    int index;                      // 0 - 7
    string name;                    // R0 - R7
    int value;                      // Initialized with 0
};

struct Instruction {
    string opcode;                  // [MOV, ADD, SUB, MUL, DIV]
    string first_operand;           // [R0 - R7]
    string second_operand;          // [R0 - R7] || Integer value (-32767) to
(+32767)
    int clock_cycle;                // [MOV = 1, ADD = 2, SUB = 2, MUL = 3,
DIV = 5]
};

struct BinaryInst {
    string opcode;                  // 03-bit: opcode; 2^3 = 8 Opcodes
    string optype;                  // 02-bit: 2nd operand type; 00:register;
01: value;
    string first_operand;           // 03-bit: binary first operand ( 2^3 = 8
Registers)
    string value;                   // 16-bit: value (-32767) to (+32767)
};

struct ClockCycle {
    int number;                     // Clock cycle number
    vector<string> stages;          // Stored as columns, not rows
};

struct Hazard {
    string inOperand;               // Operand 1 for instruction i
    int instructionA;               // Instruction i
    int instructionB;               // Instruction i+1
    string solvedBy;                // Forwarding || Stall
};

#pragma mark - Functions
/*
 *      Functions - ISA
 */
```

```
// Get a register struct, from a name s; search in vector<Register>
Register getRegister(string s, vector<Register> &registers);

// Get current value in a register with a name s; search in vector<Register>
int getValue(string s, vector<Register> &registers);

// Set new value for a register with a name s; search in vector<Register>
void setValue(string s, int value, vector<Register> &registers);

// Check if the value for second operand exceeds the limit
// Returns true if we can continue
bool checkValue(Instruction &instruction, vector<Register> &registers);

// Update a register based on the given instruction; search in
vector<Register>
// Returns a formatted string, to be printed later
string updateRegister(Instruction &instruction, vector<Register> &registers,
int bx);

// Checks for a opcode with a name s
// Returns true if opcode is one of [MOV, ADD, SUB, MUL, DIV]
bool checkOpcode(string s);

// Checks whether the operand is register or value based on a name s; search
in vector<Register>
// Returns true if it is a register, false if it is a value
bool checkOperand(string s, vector<Register> &registers);

// Encodes the opcode from high-lvl language to binary
// Returns 3-bit binary encoded string for the opcode
string encodeOpcode(string s);

// Encodes the operation type based on the type of second operand in the
instruction
// If 2nd operand = register, it returns 00; if value, it returns 01
string encodeOptype(string s, vector<Register> &registers);

// Encodes the first operand from high-lvl language to binary
// Returns 3-bit binary encoded string for the first operand
string encodeFirstOperand(string s, vector<Register> &registers);

// Returns the correct clock cycle based on the type of operation; opcode
name s
// [MOV = 1, ADD = 2, SUB = 2, MUL = 3, DIV = 5]
int getClockCycle(string s);

// Convert decimal to binary
// input    = decimal integer
// bit      = bit size for the binary
// type     = boolean (true=unsigned, false=signed)
// Returns bit-sized binary string after conversion
string decimalToBinary(int input, int bit, bool type);

/*
 *      Functions - Vector functions for split
 */
```

```
void split(const string &s, char delim, vector<string> &elems);
vector<string> split(const string &s, char delim);

// Convert integer to string for printing purpose
string intToString(int input);


int main(int argc, const char * argv[])
{
    vector<Register> registers;
    vector<Instruction> decodedInstructions;
    vector<BinaryInst> encodedInstructions;
    vector<ClockCycle> clockCycles;
    vector<Hazard> hazards;

    int nRegisters = 16;
    int mem = 0;            // memory

    string input;
    vector<string> inputs;
    vector<string> results;

    bool needStall = false;

#pragma mark - App Details

    cout << "\n\n";
    cout.width(50); cout << right << "Instruction Set Architecture\n";
    cout.width(50); cout << right << "============================\n";
    cout << "\n\n";
    cout << "It is a 24-bit ISA and can handle any 16-bit Architecture
operations.\n";
    cout << "There are sixteen 16-bit GPRs (R0-R15), plus two 32 and 16-bit
additional registers.\n";
    cout << "'RM' and 'RE' are meant for storing 32-bit result after a
multiplication\n";
    cout << "and 16-bit remainder after a division respectively.\n";
    cout << endl;

    cout << "Structure:\n";
    cout << "----------\n";
    cout << "+-------+-------+-------+------------------------+\n";
    cout << "|  3-bit|  1-bit|  4-bit|         16-bit         |\n";
    cout << "| opcode|control|operand|          value         |\n";
    cout << "+-------+-------+-------+------------------------+\n";
    cout << "|   000 |     0 |  0000 |    0000000000000000    |\n";
    cout << "+-------+-------+-------+------------------------+\n";
    cout << endl;

    cout << "Steps:\n";
    cout << "------\n";
    cout << "1. Select the opcode <'MOV', 'ADD', 'SUB', 'MUL', 'DIV', or
'END' to end code>\n";
    cout << "2. Select the first operand <R0 - R15>\n";
    cout << "3. Select the second operand<R0...R15> or <a decimal value (-
32767) to (+32767)>\n";
```

```
    cout << "4. Register R0 can also be used as a memory location, to get the
value from a memory.\n";
    cout << endl;

    cout << "Example, 'MOV R0 10' or 'ADD R1 R0'\n";
    cout << "Type 'END' to end the code\n\n";

#pragma mark - Register Initialization

    cout << "Initializing the registers.. \n";
    for (int i = 0; i < nRegisters; i++) {
        Register r;
        r.index = i;
        r.name = "R" + intToString(r.index);
        r.value = 0;
        registers.push_back(r);
    }
    cout << nRegisters << " registers initialized successfully.\n\n";

    do {
        cout << "Input a value for memory location: ";
        cin >> mem;
    } while (mem < -32767 || mem > 32767);
    cout << endl;

#pragma mark - Registers print

    cout << "Current value of the registers: \n";
    for (int i = 0; i < registers.size(); i++) {
        Register &r = registers.at(i);
        if (i % 2 == 0) {
            cout.width(10); cout << r.name << " = " << r.value;
        } else {
            cout.width(10); cout << r.name << " = " << r.value << endl;
        }
    }
    cout << endl;

#pragma mark - Instruction Fetch

    cout << "Enter the instructions:\n";
    cout << "=======================\n";
    cout << endl;

    do {
        label:
        getline(cin, input);
        transform(input.begin(), input.end(), input.begin(), ::toupper);
        inputs = split(input, ' ');

        if (inputs.size() == 0) {
            goto label;
        } else if (inputs.size() == 1 || inputs[0] == "END") {

        } else if (inputs.size() == 3) {
            if (checkOpcode(inputs[0])) {
                if (checkOperand(inputs[1], registers)) {
```

```
                    // save the high-lvl language form in decoded_instruction
                    Instruction i;
                    i.opcode = inputs[0];
                    i.first_operand = inputs[1];
                    i.second_operand = inputs[2];
                    i.clock_cycle = getClockCycle(i.opcode);

                    if (checkValue(i, registers)) {
                        decodedInstructions.push_back(i);

                        string res = updateRegister(i, registers, mem);
                        results.push_back(res);

                        // save the binary instruction in encoded_instruction
                        BinaryInst bi;
                        bi.opcode = encodeOpcode(inputs[0]);
                        bi.optype = encodeOptype(inputs[2], registers);
                        bi.first_operand = encodeFirstOperand(inputs[1],
registers);

                        int tempValue = 0;

                        if (bi.optype == "0") {
                            // second operand is a register; get value from
the register
                            bi.value = decimalToBinary(getValue(inputs[2],
registers), 16, false);
                        } else {
                            if (i.second_operand == "[R0]") {
                                tempValue = mem;
                            } else {
                                if (inputs[2].at(0) == '-') {
                                    tempValue = (-1) *
atoi((inputs[2].substr(1)).c_str());
                                } else {
                                    tempValue = atoi(inputs[2].c_str());
                                }
                            }

                            bi.value = decimalToBinary(tempValue, 16, false);
                        }

                        encodedInstructions.push_back(bi);
                    } else {
                        cout << "The result exceeds 16-bit.\n";
                    }
                } else {
                    cout << "Incorrect FIRST-OPERAND.\n";
                    goto label;
                }
            } else {
                cout << "Incorrect OPCODE.\n";
                goto label;
            }
        } else {
            cout << "Invalid instruction.\n";
            goto label;
```

```
        }
    } while (input != "END");

#pragma mark - ISA Calculation

    cout << endl;
    cout << setw(75) << setfill('-') << "" << endl;
    cout << setfill(' ');

    cout.width(05); cout << "PC";
    cout.width(20); cout << "Decoded";
    cout.width(35); cout << "Encoded (24-bit)";
    cout.width(15); cout << "Clock Cycle" << endl;
    cout << setw(75) << setfill('-') << "" << endl;
    cout << setfill(' ');

    int totalClockCycle = 0;

    for (int i = 0; i < decodedInstructions.size(); i++) {
        Instruction ins = decodedInstructions.at(i);
        BinaryInst bins = encodedInstructions.at(i);

        string a = ins.opcode + " " + ins.first_operand + ", " +
ins.second_operand;
        string b = bins.opcode + " " + bins.optype + " " + bins.first_operand
+ " " + bins.value;

        totalClockCycle += ins.clock_cycle;

        cout.width(05); cout << right << i+1;
        cout.width(20); cout << right << a;
        cout.width(35); cout << right << b;
        cout.width(15); cout << right << ins.clock_cycle;
        cout << endl;
    }

    cout << setw(75) << setfill('-') << "" << endl;
    cout << setfill(' ') << endl;

    cout << "Step-by-Step result for the Registers:\n";
    cout << "-------------------------------------\n";

    for (int i = 0; i < results.size(); i++) {
        cout << results.at(i) << endl;
    }

    cout << endl;
    cout << "CPI = " << (double)totalClockCycle/decodedInstructions.size() <<
endl;
    cout << endl;

    // Pipeline calculation
    for (int i = 0; i < decodedInstructions.size(); i++) {
        Instruction ins = decodedInstructions.at(i);
        Instruction prev_ins;

        if (i == 0) {
```

```
            needStall = false;
        } else {
            needStall = false;
            prev_ins = decodedInstructions.at(i-1);
            // hazard might occur depending on the opcode
            if (checkOperand(ins.second_operand, registers)
                && ins.second_operand == prev_ins.first_operand) {

                if (prev_ins.opcode == "MOV" && prev_ins.second_operand ==
"[RO]") {
                    // data harazd requiring stall
                    if (ins.opcode == "ADD" || ins.opcode == "SUB" ||
ins.opcode == "MUL" || ins.opcode == "DIV") {
                        needStall = true;
                        Hazard hz;
                        hz.inOperand = ins.second_operand;
                        hz.instructionA = i;
                        hz.instructionB = i+1;
                        hz.solvedBy = "Stall";
                        hazards.push_back(hz);
                    }
                    // no data hazard; MOV followed by MOV
                    else {

                    }
                    // prev_ins.opcode == "ADD" | "SUB" | "MUL" | "DIV"
                } else {
                    // RAW hazard solved by forwarding
                    Hazard hz;
                    hz.inOperand = ins.second_operand;
                    hz.instructionA = i;
                    hz.instructionB = i+1;
                    hz.solvedBy = "Forwarding";
                    hazards.push_back(hz);
                }
            }
        }

        int counter = 1;
        int j = i;
        int clockCounter = i+4;

        do {
            //create new CC
            if (clockCycles.size() <= j) {
                ClockCycle c;
                c.number = j;

                for (int k = 0; k < i; k++) {
                    c.stages.push_back("");
                }
                clockCycles.push_back(c);
            }

            if (counter == 1) {
                // IF Stage
                ClockCycle &c = clockCycles.at(j);
```

```
    if (i == 0) {
        c.stages.push_back("IF");
        counter++;
    } else {
        if (c.stages.at(i-1) == "ST") {
            c.stages.push_back("ST");
            clockCounter++;
        } else if (c.stages.at(i-1) == "IF") {
            c.stages.push_back(" ");
            clockCounter++;
        } else {
            c.stages.push_back("IF");
            counter++;
        }

    }
} else if (counter == 2) {
    // ID Stage
    ClockCycle &c = clockCycles.at(j);
    if (i == 0) {
        c.stages.push_back("ID");
        counter++;
    } else {
        if (c.stages.at(i-1) == "ST") {
            c.stages.push_back("ST");
            clockCounter++;
        } else {
            c.stages.push_back("ID");
            counter++;
        }

    }
} else if (counter == 3) {
    // EX Stage
    ClockCycle &c = clockCycles.at(j);
    if (i == 0) {
        c.stages.push_back("EX");
        counter++;
    } else {
        if (needStall) {
            c.stages.push_back("ST");
            clockCounter++;
            needStall = false;
        } else {
            c.stages.push_back("EX");
            counter++;
        }

    }
} else if (counter == 4) {
    // MEM Stage
    ClockCycle &c = clockCycles.at(j);
    if (i == 0) {
        c.stages.push_back("MEM");
        counter++;
    } else {
        c.stages.push_back("MEM");
```

```
                counter++;
            }
        } else if (counter == 5) {
            // WB Stage
            ClockCycle &c = clockCycles.at(j);
            if (i == 0) {
                c.stages.push_back("WB");
                counter++;
            } else {
                c.stages.push_back("WB");
                counter++;
            }
        }
        j++;
    } while (j <= clockCounter);
}

// Show pipeline
cout.width(3); cout << right << "NO.";
cout.width(20); cout << right << "Instruction";
for (int i = 0; i < clockCycles.size(); i++) {
    ClockCycle c = clockCycles.at(i);
    cout.width(5);  cout << right << c.number+1;
}
cout << endl;

int length = 3 + 20 + ((int)clockCycles.size() * 5);
cout << setw(length) << setfill('-') << "" << endl;
cout << setfill(' ');

for (int i = 0; i < decodedInstructions.size(); i++) {

    cout.width(3); cout << right << i+1;

    Instruction ins = decodedInstructions.at(i);
    Instruction prev_ins;
    if (i > 0) {
        prev_ins = decodedInstructions.at(i-1);
    }

    string s = "";

    s = ins.opcode + " " + ins.first_operand + ", " + ins.second_operand;
    cout.width(20); cout << right << s;
    for (int j = i; j > 0; j--) {
        cout.width(5); cout << right << "";
    }

    for (int k = 0; k < clockCycles.size(); k++) {
        if (k >= i) {
            ClockCycle c = clockCycles.at(k);
            string stage = c.stages.at(i);
            cout.width(5); cout << right << stage;
        }
    }
    cout << endl;
}
```

```
    cout << endl;

    for (int i = 0; i < hazards.size(); i++) {
        Hazard h = hazards.at(i);
        cout << h.inOperand << " in instruction " << h.instructionA << " and
" << h.instructionB << " caused RAW hazard and is solved by => " <<
h.solvedBy << endl;
    }
    cout << endl;

    return 0;
}
```