```
// Instruction Decoder F06.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#define SCORING_PC      0x80
#define SCORING_IR      0x40
#define SCORING_ACC     0x20
#define SCORING_MAR     0x10
#define SCORING_MEM1    0x08
#define SCORING_MEM2    0x04
#define QUESTION_PT_TOTAL   10
using namespace std;

#define HALT_OPCODE 0x19

void fetchNextInstruction(void);
void executeInstruction(void);
void printRegisterValues(void);
void initializePC_IR_ACC_MAR(int PC_val, char IR_val, char ACC_val, int MAR_val);
void verifyOpcode(char *instruction, int scoring, int correct_PC, unsigned char correct_IR↙
    , unsigned char correct_ACC, int correct_MAR, int mem_addr_1, int mem_addr_2, char    ↙
    correct_mem_addr_1, char correct_mem_addr_2);

unsigned char memory[65536];
unsigned char ACC=0;
unsigned char IR=0;
unsigned int MAR=0;
unsigned int PC=0;
unsigned int test_number = 0;
unsigned int Old_PC;
unsigned int number_of_operand_bytes = 0;
int total_points=0;
int total_possible_points=0;

int _tmain(int argc, _TCHAR* argv[])
{
    int scoring_mask = 0;
    char username[30];
//////////// Make sure we get the student's name on top of the output /////////
    cin >> username;
    cout << "Results of Instruction Decoder Project\n\n";
    cout << "Student username: " << username << "\n\n";

//////////////////////// Execution/testing of instruction /////////////////////
    initializePC_IR_ACC_MAR(0, 0, 0, 0);
    memory[0] = 0x09;
    memory[1] = 0x05;
    unsigned int target_PC=2;
    unsigned char target_IR=memory[PC];
    unsigned char target_ACC=0x05;
    unsigned int target_MAR=0;
    scoring_mask = SCORING_PC|SCORING_IR|SCORING_ACC;
    verifyOpcode("LOAD ACC,5", scoring_mask, target_PC, target_IR, target_ACC, target_MAR,↙
     -1, -1, 0, 0);
//////////////////////// Execution/testing of instruction /////////////////////
    initializePC_IR_ACC_MAR(0, 0, 0, 0);
    memory[0] = 0x0d;
    memory[1] = 0x0a;
    memory[2] = 0x54;
    scoring_mask = SCORING_PC|SCORING_IR|SCORING_MAR;
    target_PC=3;
    target_IR=memory[PC];
    target_ACC=0;
    target_MAR=0x0a54;
    verifyOpcode("LOAD MAR,0x0a54", scoring_mask, target_PC, target_IR, target_ACC,       ↙
    target_MAR, -1, -1, 0, 0);
//////////////////////// Execution/testing of instruction /////////////////////
    initializePC_IR_ACC_MAR(2, 0, 0x34, 0);
    memory[2] = 0x00;
    memory[3] = 0x10;
    memory[4] = 0x00;
    memory[0x1000] = 0x00;
    scoring_mask = SCORING_PC|SCORING_IR|SCORING_ACC|SCORING_MEM1;
    target_PC=5;
    target_IR=memory[PC];
    target_ACC=0x34;
    target_MAR=0;
    verifyOpcode("STORE ACC,0x1000", scoring_mask, target_PC, target_IR, target_ACC,    ↙
    target_MAR, 0x1000, -1, 0x34, 0);
//////////////////////// Execution/testing of instruction /////////////////////
    initializePC_IR_ACC_MAR(10, 0, 0x55, 0);
    memory[10] = 0x86;
    memory[11] = 0x0f;
    scoring_mask = SCORING_PC|SCORING_IR|SCORING_ACC;
    target_PC=12;
    target_IR=memory[PC];
    target_ACC=0x05;
    target_MAR=0;
    verifyOpcode("AND ACC,0x0F", scoring_mask, target_PC, target_IR, target_ACC,    ↙
    target_MAR, -1, -1, 0, 0);
//////////////////////// Execution/testing of instruction /////////////////////
    initializePC_IR_ACC_MAR(10, 0, 0x55, 0);
    memory[10] = 0x87;
    memory[11] = 0x0c;
    memory[12] = 0x00;
    memory[0x0c00] = 0x0f;
    scoring_mask = SCORING_PC|SCORING_IR|SCORING_ACC|SCORING_MEM1;
    target_PC=13;
    target_IR=memory[PC];
    target_ACC=0x05;
    target_MAR=0;
    verifyOpcode("AND ACC,[0x0C00]", scoring_mask, target_PC, target_IR, target_ACC,    ↙
    target_MAR, 0x0c00, -1, 0x0f, 0);
//////////////////////// Execution/testing of instruction /////////////////////
    initializePC_IR_ACC_MAR(10, 0, 0, 0x5555);
    memory[10] = 0x8a;
    memory[11] = 0x00;
    memory[12] = 0xff;
    memory[0x0c00] = 0x0f;
    scoring_mask = SCORING_PC|SCORING_IR|SCORING_MAR;
    target_PC=13;
    target_IR=memory[PC];
    target_ACC=0;
    target_MAR=0x0055;
    verifyOpcode("AND MAR,0x00ff", scoring_mask, target_PC, target_IR, target_ACC,    ↙
    target_MAR, -1, -1, 0, 0);
//////////////////////// Execution/testing of instruction /////////////////////
    initializePC_IR_ACC_MAR(10, 0, 0x55, 0);
    memory[10] = 0x96;
    memory[11] = 0x0f;
    scoring_mask = SCORING_PC|SCORING_ACC;
    target_PC=12;
    target_IR=memory[PC];
    target_ACC=0x5f;
    target_MAR=0;
    verifyOpcode("OR ACC,0x0F", scoring_mask, target_PC, target_IR, target_ACC, target_MAR↙
    , -1, -1, 0, 0);
//////////////////////// Execution/testing of instruction /////////////////////
    initializePC_IR_ACC_MAR(10, 0, 0x55, 0);
    memory[10] = 0xA6;
    memory[11] = 0x0f;
    scoring_mask = SCORING_ACC;
    target_PC=12;
    target_IR=memory[PC];
    target_ACC=0x5a;
    target_MAR=0;
    verifyOpcode("XOR ACC,0x0F", scoring_mask, target_PC, target_IR, target_ACC,    ↙
    target_MAR, -1, -1, 0, 0);
//////////////////////// Execution/testing of instruction ////////////////////
    initializePC_IR_ACC_MAR(10, 0, 0x55, 0);
    memory[10] = 0xB6;
    memory[11] = 0x0f;
    scoring_mask = SCORING_ACC;
    target_PC=12;
    target_IR=memory[PC];
    target_ACC=0x64;
    target_MAR=0;
    verifyOpcode("ADD ACC,0x0F", scoring_mask, target_PC, target_IR, target_ACC,    ↙
    target_MAR, -1, -1, 0, 0);
//////////////////////// Execution/testing of instruction ////////////////////
    initializePC_IR_ACC_MAR(10, 0, 0x55, 0);
    memory[10] = 0xC6;
    memory[11] = 0x0f;
    scoring_mask = SCORING_ACC;
    target_PC=12;
    target_IR=memory[PC];
    target_ACC=0x46;
    target_MAR=0;
    verifyOpcode("SUB ACC,0x0F", scoring_mask, target_PC, target_IR, target_ACC,    ↙
    target_MAR, -1, -1, 0, 0);
//////////////////////// Execution/testing of instruction ////////////////////
    initializePC_IR_ACC_MAR(25, 0, 0, 0);
    memory[25] = 0x10;
    memory[26] = 0x10;
    memory[27] = 0x00;
    scoring_mask = SCORING_PC|SCORING_IR;
    target_PC=0x1000;
    target_IR=memory[PC];
    target_ACC=0;
    target_MAR=0;
    verifyOpcode("BRA 0x1000", scoring_mask, target_PC, target_IR, target_ACC, target_MAR,↙
     -1, -1, 0, 0);
//////////////////////// Execution/testing of instruction ////////////////////
    initializePC_IR_ACC_MAR(25, 0, 0, 0);
    memory[25] = 0x11;
    memory[26] = 0x10;
    memory[27] = 0x00;
    scoring_mask = SCORING_PC|SCORING_ACC;
    target_PC=0x1000;
    target_IR=memory[PC];
    target_ACC=0;
    target_MAR=0;
    verifyOpcode("BRZ 0x1000", scoring_mask, target_PC, target_IR, target_ACC, target_MAR,↙
     -1, -1, 0, 0);
//////////////////////// Execution/testing of instruction ////////////////////
    initializePC_IR_ACC_MAR(25, 0, 5, 0);
    memory[25] = 0x11;
    memory[26] = 0x10;
    memory[27] = 0x00;
    scoring_mask = SCORING_PC;
    target_PC=28;
    target_IR=memory[PC];
    target_ACC=5;
    target_MAR=0;
    verifyOpcode("BRZ 0x1000", scoring_mask, target_PC, target_IR, target_ACC, target_MAR,↙
     -1, -1, 0, 0);
//////////////////////// Execution/testing of instruction ////////////////////
    initializePC_IR_ACC_MAR(25, 0, 5, 0);
    memory[25] = 0x12;
    memory[26] = 0x10;
    memory[27] = 0x00;
    scoring_mask = SCORING_PC|SCORING_ACC;
    target_PC=0x1000;
    target_IR=memory[PC];
    target_ACC=5;
    target_MAR=0;
    verifyOpcode("BNE 0x1000", scoring_mask, target_PC, target_IR, target_ACC, target_MAR,↙
     -1, -1, 0, 0);
//////////////////////// Execution/testing of instruction /////////////////////
    initializePC_IR_ACC_MAR(25, 0, 0, 0);
    memory[25] = 0x12;
    memory[26] = 0x10;
    memory[27] = 0x00;
    scoring_mask = SCORING_PC;
    target_PC=28;
    target_IR=memory[PC];
    target_ACC=0;
    target_MAR=0;
    verifyOpcode("BNE 0x1000", scoring_mask, target_PC, target_IR, target_ACC, target_MAR,↙
     -1, -1, 0, 0);
//////////////////////// Execution/testing of instruction /////////////////////
    initializePC_IR_ACC_MAR(25, 0, 0xFF, 0);
    memory[25] = 0x13;
    memory[26] = 0x10;
    memory[27] = 0x00;
    scoring_mask = SCORING_PC;
    target_PC=0x1000;
    target_IR=memory[PC];
    target_ACC=0xFF;
    target_MAR=0;
    verifyOpcode("BLT 0x1000", scoring_mask, target_PC, target_IR, target_ACC, target_MAR,↙
     -1, -1, 0, 0);
//////////////////////// Execution/testing of instruction /////////////////////
    initializePC_IR_ACC_MAR(25, 0, 5, 0);
    memory[25] = 0x13;
    memory[26] = 0x10;
    memory[27] = 0x00;
    scoring_mask = SCORING_PC;
    target_PC=28;
    target_IR=memory[PC];
    target_ACC=5;
    target_MAR=0;
    verifyOpcode("BLT 0x1000", scoring_mask, target_PC, target_IR, target_ACC, target_MAR,↙
     -1, -1, 0, 0);
//////////////////////// Execution/testing of instruction /////////////////////
    initializePC_IR_ACC_MAR(25, 0, 0xFF, 0);
    memory[25] = 0x15;
    memory[26] = 0x10;
    memory[27] = 0x00;
    scoring_mask = SCORING_PC;
    target_PC=28;
    target_IR=memory[PC];
    target_ACC=0xFF;
    target_MAR=0;
    verifyOpcode("BGT 0x1000", scoring_mask, target_PC, target_IR, target_ACC, target_MAR,↙
     -1, -1, 0, 0);
//////////////////////// Execution/testing of instruction /////////////////////
    initializePC_IR_ACC_MAR(25, 0, 5, 0);
    memory[25] = 0x15;
    memory[26] = 0x10;
    memory[27] = 0x00;
    scoring_mask = SCORING_PC;
    target_PC=0x1000;
    target_IR=memory[PC];
    target_ACC=5;
    target_MAR=0;
    verifyOpcode("BGT 0x1000", scoring_mask, target_PC, target_IR, target_ACC, target_MAR,↙
     -1, -1, 0, 0);
//////////////////////// Execution/testing of instruction ////////////////////
    initializePC_IR_ACC_MAR(25, 0, 0, 0);
    memory[25] = 0x15;
    memory[26] = 0x10;
    memory[27] = 0x00;
    scoring_mask = SCORING_PC;
    target_PC=28;
    target_IR=memory[PC];
    target_ACC=0;
    target_MAR=0;
    verifyOpcode("BGT 0x1000", scoring_mask, target_PC, target_IR, target_ACC, target_MAR,↙
     -1, -1, 0, 0);
//////////////////////// Execution/testing of instruction ////////////////////
    initializePC_IR_ACC_MAR(0xFFFE, 0, 0, 0);
    memory[0xFFFE] = 0x18;
    scoring_mask = SCORING_PC;
    target_PC=0xFFFF;
    target_IR=memory[PC];
    target_ACC=0;
    target_MAR=0;
    verifyOpcode("NOP", scoring_mask, target_PC, target_IR, target_ACC, target_MAR, -1, -1↙
    , 0, 0);
//////////////////////// Execution/testing of instruction ////////////////////
    initializePC_IR_ACC_MAR(0xFFFF, 0, 0, 0);
    memory[0xFFFF] = 0x19;
    scoring_mask = SCORING_PC;
    target_PC=0;
    target_IR=memory[PC];
    target_ACC=0;
    target_MAR=0;
    verifyOpcode("HALT", scoring_mask, target_PC, target_IR, target_ACC, target_MAR, -1, -↙
    1, 0, 0);


//////////////////////////// Output final score //////////////////////////////
    cout << "\n**** Code score using test suite:\n  "<< total_points << " out of " <<     ↙
    total_possible_points << ".\n";

    int adj_points = 0;
    cin >> adj_points;

    int question_points[5];
    cin >> question_points[0] >> question_points[1] >> question_points[2] >>           ↙
    question_points[3] >> question_points[4];

    cout << "Adjustment points: " << adj_points << "\n";
    cout << "Question 1 points: " << question_points[0] << " out of 1\n";
    cout << "Question 2 points: " << question_points[1] << " out of 2\n";
    cout << "Question 3 points: " << question_points[2] << " out of 3\n";
    cout << "Question 4 points: " << question_points[3] << " out of 2\n";
    cout << "Question 5 points: " << question_points[4] << " out of 2\n";
    int question_total = 0;
    for (int i=0; i<5; i++) question_total += question_points[i];
    cout << "\n**** Final score adjusted for code:\n  " << (total_points + adj_points +   ↙
    question_total) << " out of " << (total_possible_points + QUESTION_PT_TOTAL) << ".\n\n↙
    ";
    return 0;
}

//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
//////////////////////// Insert student functions here////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////////////

void fetchNextInstruction(void)
{
    IR = memory[PC];
    Old_PC = PC;
    PC++;       // Increment PC past the opcode
    // Now we have to see if there was an operand and if so, how big is it
    if(IR & 0x80)
    {
        switch(IR & 0x0c)   // Check on operands for the destination
        {
            case 0x00:
                switch(IR & 0x03)   // Check on operands for the source
                {
                    case 0: // Source indirect/destination indirect
                        break;
                    case 1: // Source ACC/destination indirect
                        break;
                    case 2: // Source 8-bit operand constant/destination indirect
                        PC++;
                        break;
                    case 3: // Source 16-bit operand address/destination indirect
                        PC += 2;
                        break;
                    default:
                        break;
                }
                break;
            case 0x04:
                switch(IR & 0x03)   // Check on operands for the source
                {
                    case 0: // Source indirect/destination ACC
                        break;
                    case 1: // Source ACC/destination ACC
                        break;
                    case 2: // Source 8-bit operand constant/destination ACC
                        PC++;
                        break;
                    case 3: // Source 16-bit operand address/destination ACC
                        PC += 2;
                        break;
                    default:
                        break;
                }
                break;
            case 0x08:
                switch(IR & 0x03)   // Check on operands for the source
                {
                    case 0: // Source indirect/destination MAR
                        break;
                    case 1: // Source ACC/destination MAR
                        break;
                    case 2: // Source 16-bit operand constant/destination MAR
                        PC += 2;
                        break;
                    case 3: // Source 16-bit operand address/destination MAR
                        PC += 2;
                        break;
                    default:
                        break;
                }
                break;
            case 0x0c:
                switch(IR & 0x03)   // Check on operands for the source
                {
                    case 0: // Source indirect/destination 16-bit operand address
```

```
                        PC += 2;
                        break;
                    case 1: // Source ACC/destination 16-bit operand address
                        PC += 2;
                        break;
                    case 2: // Source 8-bit operand constant/destination 16-bit operand  ↙
    address
                        PC += 3;
                        break;
                    case 3: // Source 16-bit operand address/destination 16-bit operand  ↙
    address
                        PC += 4;
                        break;
                    default:
                        break;
                }
                break;
            default:
                break;
        }
    }
    else if((IR & 0xf0) == 0)   // Now do memory transfer operations
    {
        switch(IR & 0x7)
        {
            case 0: // register = ACC/operand = 16-bit address
                PC +=2;
                break;
            case 1: // register = ACC/operand = 8-bit constant
                PC++;
                break;
            case 2: // register = ACC/operand = indirect (MAR) addressing
                break;
            case 3: // register = ACC/operand = not defined
                break;
            case 4: // register = MAR/operand = 16-bit address
                PC +=2;
                break;
            case 5: // register = MAR/operand = 16-bit constant
                PC +=2;
                break;
            case 6: // register = MAR/operand = indirect (MAR) addressing
                break;
            case 7: // register = MAR/operand = not defined
                break;
            default:
                break;
        }
    }
    else if((IR & 0xf8) == 0X10)    // Now do branches (They always have a 16-bit operand)
        PC+=2;
    else
    {
    }
    number_of_operand_bytes = PC - Old_PC - 1;
    PC &= 0xFFFF; // Verify that we haven't gone past end of memory
}
void executeInstruction(void)
{
    int address;
    // Check if it's a math function
    if ((IR & 0x80) == 0x80)
    {
    // Grab the destination and source values
        int destination, source;
        switch(IR & 0x0c)   // Isolate destination id
        {
            case 0x0:
                destination = memory[MAR];
                break;
            case 0x4:
                destination = ACC;
                break;
            case 0x8:
                destination = MAR;
                break;
            case 0xc:
                destination = memory[((memory[Old_PC + 1] << 8) + memory[Old_PC + 2])];
                break;
        }
        switch(IR & 0x03)   // Isolate source id
        {
            case 0x0:
                source = memory[MAR];
                break;
            case 0x1:
                source = ACC;
                break;
            case 0x2:
                // MAR is only 16-bit mathematical operation, so do this
                if((IR & 0x0c) == 0x8)  // MAR is the destination, 16-bit operand
                    source = (memory[PC - 2] << 8) + memory[PC -1];
                // else if 8 bits, do this
                else
                    source = memory[PC - 1];
                break;
            case 0x3:
                // MAR is only 16-bit mathematical operation, so do this
                if((IR & 0x0c) == 0x8)  // MAR is the destination, 16-bit operand
                {
                    address = ((memory[Old_PC + 1] << 8) + memory[Old_PC + 2]);
                    source = (memory[address] << 8) + memory[address + 1];
                }
                // else if 8 bits, do this
                else
                    source = memory[((memory[Old_PC + 1] << 8) + memory[Old_PC + 2])];
                break;
        }

    // Process the operation
        switch (IR&0x70)
        {
            case 0x00:  // AND function
                destination &= source;
                break;
            case 0x10:
                destination |= source;
                break;
            case 0x20:
                destination ^= source;
                break;
            case 0x30:
                destination += source;
                break;
            case 0x40:
                destination -= source;
                break;
            case 0x50:
                destination++;
                break;
            case 0x60:
                destination--;
                break;
            case 0x70:
                destination = !destination;
                break;
            default:
                break;
        }
    // Store destination
        switch(IR & 0x0c)   // Isolate destination id
        {
            case 0x0:
                memory[MAR] = destination & 0xff;
                break;
            case 0x4:
                ACC = destination & 0xff;
                break;
            case 0x8:
                MAR = destination & 0xffff;
                break;
            case 0xc:
                memory[((memory[Old_PC + 1] << 8) + memory[Old_PC + 2])] = destination &  ↙
    0xff;
                break;
        }
    }

    // Check if this is a memory function
    else if ((IR & 0xf0) == 0)
    {
        if ((IR & 0x08) == 0)   // Store function
        {
            if ((IR & 0x04) == 0)   // Storing from ACC
            {
                switch(IR & 0x03)
                {
                    case 0:             // Storing ACC to address
                        memory[((memory[Old_PC + 1] << 8) + memory[Old_PC + 2])] = ACC;
                        break;
                    case 1:             // Storing ACC to constant -- doesn't make sense
                        break;
                    case 2:             // Storing ACC using MAR as pointer
                        memory[MAR] = ACC;
                        break;
                    case 3:             // Not used
                        break;
                    default:
                        break;
                }
            }
            else                    // Storing from MAR
            {
                switch(IR & 0x03)
                {
                    case 0:             // Storing MAR to address
                        memory[((memory[Old_PC + 1] << 8) + memory[Old_PC + 2])] = (MAR >>↙
     8) & 0xff;
                        memory[((memory[Old_PC + 1] << 8) + memory[Old_PC + 2]) + 1] = MAR↙
     & 0xff;
                        break;
                    case 1:             // Storing MAR to constant -- doesn't make sense
                        break;
                    case 2:             // Storing MAR using MAR as pointer
                        memory[MAR] = (MAR >> 8) & 0xff;
                        memory[MAR + 1] = MAR & 0xff;
                        break;
                    case 3:             // Not used
                        break;
                    default:
                        break;
                }
            }
        }
        else                    // Load function
        {
            if ((IR & 0x04) == 0)   // Loading from ACC
            {
                switch(IR & 0x03)
                {
                    case 0:             // Loading ACC from address
                        ACC = memory[((memory[Old_PC + 1] << 8) + memory[Old_PC + 2])];
                        break;
                    case 1:             // Loading ACC from constant
                        ACC = memory[Old_PC + 1];
                        break;
                    case 2:             // Loading ACC using MAR as pointer
                        ACC = memory[MAR];
                        break;
                    case 3:             // Not used
                        break;
                    default:
                        break;
                }
            }
            else                    // Loading to MAR
            {
                int Old_MAR = MAR;
                switch(IR & 0x03)
                {
                    case 0:             // Loading MAR from address
                        MAR = memory[((memory[Old_PC + 1] << 8) + memory[Old_PC + 2])];
                        MAR <<=8;
                        MAR += memory[((memory[Old_PC + 1] << 8) + memory[Old_PC + 2]) + ↙
    1];
                        break;
                    case 1:             // Loading MAR from constant
                        MAR = memory[Old_PC + 1];
                        MAR <<=8;
                        MAR += memory[Old_PC + 2];
                        break;
                    case 2:             // Loading MAR using MAR as pointer
                        MAR = memory[Old_MAR];
                        MAR <<=8;
                        MAR += memory[Old_MAR + 1];
                        break;
                    case 3:             // Not used
                        break;
                    default:
                        break;
                }
            }
        }
    }

    // Check branch function
    else if ((IR & 0xF8) == 0x10)
    {
        address = (memory[Old_PC + 1] << 8) + memory[Old_PC + 2];
        switch (IR & 0x07)
        {
            case 0: // BRA (Unconditional branch/branch always)
                PC = address;
                break;
            case 1: // BRZ (Branch if ACC = 0)
                if (ACC == 0) PC = address;
                break;
            case 2: // BNE (Branch if ACC != 0)
```

```
                if (ACC != 0) PC = address;
                break;
            case 3: // BLT (Branch if ACC < 0)
                if ((ACC & 0x80) != 0) PC = address;
                break;
            case 4: // BLE (Branch if ACC <= 0)
                if (((ACC & 0x80) != 0) || (ACC == 0)) PC = address;
                break;
            case 5: // BGT (Branch if ACC > 0)
                if (((ACC & 0x80) == 0) && (ACC != 0)) PC = address;
                break;
            case 6: // BGE (Branch if ACC > 0)
                if ((ACC & 0x80) == 0) PC = address;
                break;
            default:
                break;
        }
    }

    // Otherwise, it's a special opcode or an illegal opcode
    else
    // The following is the logic code for a special opcode or an illegal opcode
    {
        if(IR == 0x18) // Then it's a NOP -- do nothing
        {}
        else if(IR == 0x19) // Then it's a HALT -- framework will halt for us
        {}
        else // Otherwise it's an illegal opcode -- you can print an error message if you ↙
    want to
        {}
    }
}

//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
////////// This code is to execute the test suite and generate output ////////////
//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////

void printRegisterValues(void)
{
    cout << setfill('0');
    cout << "PC = " << hex << setw(4) << PC;
    cout << ", ACC = " << hex << setw(2) << int(ACC);
    cout << ", IR = " << hex << setw(2) << int(IR);
    cout << ", and MAR = " << hex << setw(4) << MAR;
    cout << "\n";
    cout << setfill(' ');
}


//////////////////////////////// verifyOpcode() //////////////////////////////
// verifyOpcode() executes a single instruction, compares the results with //
// the "correct" values sent to it, and outputs the results of the test.   //
// In addition, it keeps track of the points accumulated and the total     //
// points possible.  The oddest thing about the whole routine is the method//
// of scoring.  A single byte is sent called "scoring".  This byte         //
// indicates which elements are of concern in scoring. The bit-by-bit      //
// description of scoring is as follows:                                   //
//                                                                         //
//                        +---+---+---+---+---+---+---+---+                //
//                        |   |   |   |   |   |   |   |   |                //
//                        +---+---+---+---+---+---+---+---+                //
//                          ^   ^   ^   ^   ^   ^                          //
//                          |   |   |   |   |   |                          //
//      correct PC ---------+   |   |   |   |   |                          //
//      correct IR -------------+   |   |   |   |                          //
//      correct ACC ----------------+   |   |   |                          //
//      correct MAR --------------------+   |   |                          //
//      correct memory[mem_addr_1] ---------+   |                          //
//      correct memory[mem_addr_2] -------------+                          //
//                                                                         //
// A zero is placed in bit positions that we don't care about.  If a 1 is  //
// in a bit position, it is used in the calculation of the points.  The    //
// number of 1's in the scoring byte indicates how many points the         //
// instruction is worth.                                                   //
/////////////////////////////////////////////////////////////////////////////
void verifyOpcode(char *instruction, int scoring, int correct_PC, unsigned char correct_IR↙
    , unsigned char correct_ACC, int correct_MAR, int mem_addr_1, int mem_addr_2, char    ↙
    correct_mem_addr_1, char correct_mem_addr_2)
{
    char pc_check = ' ';
    char ir_check = ' ';
    char acc_check = ' ';
    char mar_check = ' ';
    char mem1_check = ' ';
    char mem2_check = ' ';
    int points=0;
    int possible_points=0;


// Indicate the test we're on
    test_number++;

// See how many points are possible with this test
    int mask = 0x80;
    for(int i=0; i<8; i++)
    {
        if(scoring & mask) possible_points++;
        mask >>= 1;
    }
    points = possible_points;

// Make the column headings
    cout << "**** Test " << test_number << " Results (" << instruction << ")\n";
    cout << "          PC   IR  ACC   MAR";
    if (mem_addr_1 != -1) cout << "  mem[" << setfill('0') << hex << setw(4) << mem_addr_1↙
     << "]";
    if (mem_addr_2 != -1) cout << "  mem[" << setfill('0') << hex << setw(4) << mem_addr_2↙
     << "]";
    cout << "\n";

// Generate the columns representing the correct results we should see
    cout << "Initial: " << setfill('0') << hex << setw(4) << PC;
    cout << "  " << hex << setw(2) << int(IR);
    cout << "   " << hex << setw(2) << int(ACC);
    cout << "   " << hex << setw(4) << MAR;
    if (mem_addr_1 != -1) cout << "     " << hex << setw(2) << int(correct_mem_addr_1);
    if (mem_addr_2 != -1) cout << "         " << hex << setw(2) << int(correct_mem_addr_2)↙
    ;
    cout << "\n";

// Generate the columns representing the correct results we should see
    cout << " Target: " << setfill('0') << hex << setw(4) << correct_PC;
    cout << "  " << hex << setw(2) << int(correct_IR);
    cout << "   " << hex << setw(2) << int(correct_ACC);
    cout << "   " << hex << setw(4) << correct_MAR;
    if (mem_addr_1 != -1) cout << "     " << hex << setw(2) << int(correct_mem_addr_1);
    if (mem_addr_2 != -1) cout << "         " << hex << setw(2) << int(correct_mem_addr_2)↙
    ;
    cout << "\n";
```

```cpp
// Execute the instruction
    fetchNextInstruction();
    executeInstruction();

// Here we put a star next to the outputs that are in error
    if(correct_PC != PC)
    {
        pc_check='*';
        if(scoring & SCORING_PC) points--;
    }
    if(correct_IR != IR)
    {
        ir_check='*';
        if(scoring & SCORING_IR) points--;
    }
    if(correct_ACC != ACC)
    {
        acc_check='*';
        if(scoring & SCORING_ACC) points--;
    }
    if(correct_MAR != MAR)
    {
        mar_check='*';
        if(scoring & SCORING_MAR) points--;
    }
    if(mem_addr_1 != -1)
    {
        if(correct_mem_addr_1 != memory[mem_addr_1])
        {
            mem1_check='*';
            if(scoring & SCORING_MEM1) points--;
        }
    }
    if(mem_addr_2 != -1)
    {
        if(correct_mem_addr_2 != memory[mem_addr_2])
        {
            mem2_check='*';
            if(scoring & SCORING_MEM2) points--;
        }
    }

// Generate the columns of actual data after execution of the instruction
    cout << " Actual: " << setfill('0') << hex << setw(4) << PC << pc_check;
    cout << " " << hex << setw(2) << int(IR) << ir_check;
    cout << "  " << hex << setw(2) << int(ACC) << acc_check;
    cout << "  " << hex << setw(4) << MAR << mar_check;
    if (mem_addr_1 != -1) cout << "    " << hex << setw(2) << int(memory[mem_addr_1]) <<  ↙
    mem1_check;
    if (mem_addr_2 != -1) cout << "        " << hex << setw(2) << int(memory[mem_addr_2]) ↙
    << mem2_check;
    cout << "\n";

// Now let's output the score they received
    cout << dec << "   Score: " << points << " out of " << possible_points << "\n\n";
    total_points += points;
    total_possible_points += possible_points;

}

void initializePC_IR_ACC_MAR(int PC_val, char IR_val, char ACC_val, int MAR_val)
{
    PC = PC_val;
    IR = IR_val;
    ACC = ACC_val;
    MAR = MAR_val;
}
```