```
/*
 ------------------------------------------------------------------------
 PROGRAM NAME: Interpreter
 PROGRAMMER:   Samuel Jentsch/James Francis
 CLASS:        CSC 214.001, Spring 2014
 INSTRUCTOR:   Dr. Strader
 DATE STARTED: April 18, 2014
 DUE DATE:     April 23, 2014

 PROGRAM PURPOSE:
 Take input file with machine code and interpret and execute the
 commands indicated in the file.


 VARIABLE DICTIONARY:

 char *memory[kMAX_LINES]- simulates program memory (including stack)
 long currentLine - keeps track of current line being executed.
 long stackPointer - keeps track of current stack top.

 const char *opcodes[kNUM_OPS] - numer of opcodes.

 //Architecture simulators
 //Update strings to show state of hardware.
 char *PC - simulates program counter (holds next instruction).
 char *IR - simulates IR.
 char *CC - simulates CC. Updated based on top of stack.

 int jump - checks if a jump is occurred and prevents extra instruction
            incrementation.

 ------------------------------------------------------------------------
 */


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define kNUM_OPS 19     //Total number of operations.
#define kMAX_LINES 128  //1111111 base 2 is 127 base 10.

char *memory[kMAX_LINES];
long currentLine;
long stackPointer;

//Constants (Assembly Syntax)
const char *opcodes[kNUM_OPS];

//Architecture simulators
//Update strings to show state of hardware.
char *PC;
char *IR;
char *CC;
```

```
int jump;
/*----------------------Method Stubs----------------------*/

//Program execution
void readFile(FILE *machineFile);
void executeProgram();
int checkCharacter(char c);

void refreshArray();

//Memory/Architecture display
void printMemory();
void printArchitectureState();

//Line parsing
int parseLine(char *line);

//String manipulation
void cutString(char *string);
long binaryStringToDecimal(char* binary);
long powerOfTwo(long power);
char* decimalToBinaryString(long decimal);
void reverseString(char* string);
char* padBinaryToLength(char *binString, int length);
char* twoCompForDecimal(long decimal);
long decimalForTwoComp(char *twoComp);
char characterForOctal(char *octal1, char* octal2);
char* octalToBinaryConversion(char *convertOctal, int length);

//Opcodes
char* returnOpcodeForLine(char *line);
int getOpcodeIndex(char *opcode);
char* returnAddressForLine(char* line);

//Architecture
void updatePC();
void updateIR();
void updateCC();

//Operations
void push(char *address);
void pop(char *address);
void add();
void subtract();
void binaryAnd();
void binaryOr();
void binaryNot();
void binaryShiftLeft();
void binaryShiftRight();
void testSkip();
void inputCharacter(char *address);
void outputCharacter(char *address);
void inputInteger(char *address);
void outputInteger(char *address);
void jumpJMP(char *address);
void jumpJEQ(char *address);
```

```
void jumpJGT(char *address);
void jumpJLT(char *address);

/*----------------------Initialization----------------------*/
int main(int argc, const char * argv[]) {
    //Initialize necessary data values.

    opcodes[0] = "00000"; //Push
    opcodes[1] = "00001"; //Pop
    opcodes[2] = "00010"; //Add
    opcodes[3] = "00011"; //Subtract
    opcodes[4] = "00100"; //And
    opcodes[5] = "00101"; //Or
    opcodes[6] = "00110"; //Not (complement)
    opcodes[7] = "00111"; //testskip
    opcodes[8] = "01000"; //Shift left
    opcodes[9] = "01001"; //Shift right
    opcodes[10] = "01010"; //Jump JMP
    opcodes[11] = "01011"; //Jump JEQ
    opcodes[12] = "01100"; //Jump JGT
    opcodes[13] = "01101"; //Jump JLT
    opcodes[14] = "01110"; //input character
    opcodes[15] = "01111"; //output character
    opcodes[16] = "10000"; //input integer
    opcodes[17] = "10001"; //output integer
    opcodes[18] = "11111"; //halt program

    currentLine = 0;
    stackPointer = kMAX_LINES - 1;
    jump = 0;

    PC = malloc(12);
    PC = "0000000";
    CC = malloc(2);
    CC = "00";
    IR = malloc(12);
    IR = "000000000000";

    //FILE *machineFile = fopen("/Users/SamJ/Google Drive/SFA/Courses/
        2014_Spring/CSC_214/Lab7/Test.dat", "r");
    FILE *machineFile = fopen("../instr/proj1b.dat", "r");
    readFile(machineFile);

    return 0;
}//main

void refreshArray() {
    //Refresh array if item has been modified or deleted.
    //Refresh opcodes for use in line parsing.

    opcodes[0] = "00000"; //Push
    opcodes[1] = "00001"; //Pop
    opcodes[2] = "00010"; //Add
    opcodes[3] = "00011"; //Subtract
    opcodes[4] = "00100"; //And
    opcodes[5] = "00101"; //Or
```

```
    opcodes[6] = "00110"; //Not (complement)
    opcodes[7] = "00111"; //testskip
    opcodes[8] = "01000"; //Shift left
    opcodes[9] = "01001"; //Shift right
    opcodes[10] = "01010"; //Jump JMP
    opcodes[11] = "01011"; //Jump JEQ
    opcodes[12] = "01100"; //Jump JGT
    opcodes[13] = "01101"; //Jump JLT
    opcodes[14] = "01110"; //input character
    opcodes[15] = "01111"; //output character
    opcodes[16] = "10000"; //input integer
    opcodes[17] = "10001"; //output integer
    opcodes[18] = "11111"; //halt program
}

/*----------------------File Management----------------------*/
void readFile(FILE *machineFile) {
    //-------------------------------------------------------
    //Preconditions: File pointer to machineFile passed as
    //parameter.
    //Postconditions: File read line by line and added to the
    //memory array. If error in line is found, program is not
    //executed.
    //Read the file line by line and add to the memory array.
    //If file is good, execute the program with the initialized
    //memory.
    //-------------------------------------------------------

    if(machineFile == NULL) {
        printf("Error, assembly file not found.");
        exit(1);
    }

    char inputString[256];
    int sIndex = 0;

    char c;
    int flag = 0;
    int goodFile = 1;
    long badLine = -1;

    do {

        c = fgetc(machineFile);

        if (c != EOF && checkCharacter(c) == 1) {
            inputString[sIndex] = c;
            inputString[sIndex + 1] = '\0';
            sIndex++;
            flag = 0;
        } else if(flag != 1) {
            //inputString now contains 1 line.
            memory[currentLine] = malloc(15);

            if (strlen(inputString) != 12) {
                goodFile = 0;
```

```
                badLine = currentLine;
            }

            strcpy(memory[currentLine], inputString);
            currentLine++;
            sIndex = 0;
            flag = 1;
        }

    } while (c != EOF);

    if (goodFile == 1) {
        executeProgram();
    } else {
        printf("Incorrect line format. Line %ld\n", badLine);
    }

}//readFile

int checkCharacter(char c) {
    //-------------------------------------------------------------
    //Preconditions: character passed as parameter.
    //Postconditions: Checks to see if character is 0 or 1.
    //Returns 1 if c is a 0 or 1 (valid character for file).
    //-------------------------------------------------------------

    int isGood = 0;

    if (c == '0' || c == '1') {
        isGood = 1;
    }

    return isGood;
}

/*------------------------Execution-----------------------*/
void executeProgram() {
    //-------------------------------------------------------------
    //Preconditions: Readfile verifies file for correctness.
    //Postconditions: Program is executed line by line from
    //memory array.
    //-------------------------------------------------------------

    char *line = malloc(15);

    int halt = 0;
    currentLine = 0;

    while (memory[currentLine] != NULL && halt == 0) {
        strcpy(line, memory[currentLine]);
        halt = parseLine(line);

        printArchitectureState();
        if (jump == 0) {
            currentLine++;
        } else {
```

```
            jump = 0;
        }
    }
}//executeProgram

/*-------------------------Memory-------------------------*/
void printMemory() {
    //-----------------------------------------------------------
    //Preconditions: Memory array.
    //Postconditions: Memory array and current position of stack
    //and PC displayed to console.
    //-----------------------------------------------------------

    int i;

    printf("%7s      %12s\n", "Address", "Value");

    char *lineNumber = malloc(7);

    for (i = 0; i < kMAX_LINES; i++) {

        lineNumber = padBinaryToLength(decimalToBinaryString(i), 7);

        if (memory[i] != NULL) {
            printf("%7s      ", lineNumber);
            printf("%12s", memory[i]);
        }

        else if (i == stackPointer - 1 || i == stackPointer - 2) {
            printf("%7s      ", ".......");
            printf("............\n");
        }

        if (i == stackPointer) {
            printf("%7s      ", lineNumber);
            if (memory[i] == NULL) {
                printf("%12s", "Empty");
            }
            printf("  <---Stack Pointer");
        }

        long PCLong = binaryStringToDecimal(PC);
        if (i == PCLong) {
            printf("  <---PC");
        }

        if (memory[i] != NULL || i == stackPointer) {
            printf("\n");
        }
    }
}

void printArchitectureState() {
    //-----------------------------------------------------------
    //Preconditions: PC, CC, IR class variables.
    //Postconditions: Architecture state and memory displayed.
```

```
    //-------------------------------------------------------------

    printf("PC: %s\n", PC);
    printf("CC: %s\n", CC);
    printf("IR: %s\n", IR);
    printf("Memory: \n");
    printMemory();

    printf("\n");
}//printArchitectureState

/*----------------------Line Parsing----------------------*/
int parseLine(char *line) {
    //-------------------------------------------------------------
    //Preconditions: Line to parse passed as string.
    //Postconditions: Correct method is executed based on line
    //being parsed.
    //-------------------------------------------------------------

    int halt = 0;

    char *opcode = malloc(5);
    strncpy(opcode, line, 5);

    char *address = malloc(7);
    strncpy(address, line + 5, 7);

    int opIndex = getOpcodeIndex(opcode);

    switch (opIndex) {
        case 0:
            //Push
            //printf("PUSH\n");
            push(address);
            break;
        case 1:
            //Pop
            //printf("POP\n");
            pop(address);
            break;
        case 2:
            //Add
            //printf("ADD\n");
            add();
            break;
        case 3:
            //Subtract
            //printf("SUBTRACT\n");
            subtract();
            break;
        case 4:
            //And
            binaryAnd();
            break;
        case 5:
            //Or
```

```
            binaryOr();
            break;
        case 6:
            //Not
            binaryNot();
            break;
        case 7:
            //testSkip
            testSkip();
            break;
        case 8:
            //Shift left
            binaryShiftLeft();
            break;
        case 9:
            //Shift right
            binaryShiftRight();
            break;
        case 10:
            //Jump JMP (No condition)
            jumpJMP(address);
            break;
        case 11:
            //Jump JEQ (If equal to)
            jumpJEQ(address);
            break;
        case 12:
            //Jump JGT (If greater than)
            jumpJGT(address);
            break;
        case 13:
            //Jump JLT (If less than)
            jumpJLT(address);
            break;
        case 14:
            //Input character
            inputCharacter(address);
            break;
        case 15:
            //Output character
            outputCharacter(address);
            break;
        case 16:
            //Input integer
            inputInteger(address);
            break;
        case 17:
            //Output integer
            outputInteger(address);
            break;
        case 18:
            //Halt program
            halt = 1;
            break;
        default:
            printf("INVALID OPCODE LINE %ld\n", currentLine);
```

```
            halt = 1;
            break;
        }

        updatePC();
        updateIR();
        updateCC();

        return halt;

    }//parseLine

    /*-------------------String Management--------------------*/
    void cutString(char *string) {
        //-----------------------------------------------------
        //Preconditions: string label passed as parameter.
        //Postconditions: String is iterated through and trailing
        //space or a new line is cut off.
        //-----------------------------------------------------

        long length = strlen(string);
        int i;

        for (i = 0; i < length; i++) {
            if (string[i] == ' ' || string[i] == '\n') {
                string[i] = '\0';
                break;
            }
        }

    }//cutString

    long binaryStringToDecimal(char* binary) {
        //-----------------------------------------------------------
        //Preconditions: Binary representation of a number passed as
        //parameter as string.
        //Postconditions: Decimal representation of number passed
        //returned as string.
        //Get decimal value for binary string passed.
        //-----------------------------------------------------------

        long decimal = 0;

        long strLength = strlen(binary);

        long i;
        for (i = 0; i < strLength; i++) {
            if (binary[i] == '1') {
                decimal += powerOfTwo((strLength-1) - i);
            }
        }

        //printf("%s base 2 is %lu base 10.\n", binary, decimal);

        return decimal;
    }
```

```
char* decimalToBinaryString(long decimal) {
    //-------------------------------------------------------------
    //Preconditions: Decimal value passed as long.
    //Postconditions: Binary representation of number passed
    //returned as a string.
    //Get the binary string representation of the decimal number
    //passed.
    //-------------------------------------------------------------

    char* binString = malloc(256);

    if (decimal == 0) {
        binString[0] = '0';
        binString[1] = '\0';
    }

    int stringIndex = 0;
    int remainder;
    while (decimal != 0) {
        remainder = decimal % 2;
        if (remainder == 1) {
            binString[stringIndex] = '1';
        } else {
            binString[stringIndex] = '0';
        }

        decimal = decimal / 2;

        stringIndex++;
        binString[stringIndex] = '\0';
    }

    reverseString(binString);

    //printf("%lu base 10 is %s base 2.\n", decimal, binString);

    return binString;
}

char* twoCompForDecimal(long decimal) {
    //-------------------------------------------------------------
    //Preconditions: Decimal number passed as long.
    //Postconditions: Returns the two's complement representation
    //of the number passed as a string.
    //Returns the 2's complement representation of the number
    //passed.
    //-------------------------------------------------------------

    char* binString = malloc(256);
    binString = decimalToBinaryString(decimal);

    char *padded = malloc(256);
    padded = padBinaryToLength(binString, 12);

    char *twoComp = malloc(256);
```

```
    int i = 0;
    for (i = 0; i < strlen(padded); i++) {
        if (padded[i] == '0') {
            twoComp[i] = '1';
        } else {
            twoComp[i] = '0';
        }
    }

    long temp = binaryStringToDecimal(twoComp);

    temp++;
    twoComp = decimalToBinaryString(temp);

    //printf("%lu as a 2's comp value is: %s\n", decimal, twoComp);

    return twoComp;
}

long decimalForTwoComp(char *twoComp) {
    //-------------------------------------------------------------
    //Preconditions: String binary representation of a negative
    //two's complement number.
    //Postconditions: Returns the decimal as a positive long
    //for the two's complement binary string passed.
    //Get the positive decimal value for the two's complement
    //binary representation passed.
    //-------------------------------------------------------------

    long temp = binaryStringToDecimal(twoComp);
    temp--;

    char *binString = malloc(256);
    binString = decimalToBinaryString(temp);

    char *positiveBinary = malloc(256);

    int i = 0;
    for (i = 0; i < strlen(binString); i++) {
        if (binString[i] == '0') {
            positiveBinary[i] = '1';
        } else {
            positiveBinary[i] = '0';
        }
    }

    long positiveDecimal = binaryStringToDecimal(positiveBinary);

    //printf("%s as a positive base 10 value is: %lu", twoComp, positiveDecimal);

    return positiveDecimal;
}

long powerOfTwo(long power) {
    //-------------------------------------------------------------
```

```
    //Preconditions: Power as long passed as parameter.
    //Postconditions: Returns 2 to the power passed.
    //Return 2 to the power passed.
    //-------------------------------------------------------

    long decimal = 1;

    int i;
    for (i = 1; i <= power; i++) {
        decimal = decimal * 2;
    }

    return decimal;
}

void reverseString(char* string) {
    long length = strlen(string);

    char *reversed = malloc(256);

    long i;
    int count = 0;
    for(i = length - 1; i >= 0; i--) {
        reversed[count] = string[i];
        count++;
    }

    strcpy(string, reversed);
}


char* padBinaryToLength(char *binString, int length) {
    //-------------------------------------------------------
    //Preconditions: string to pad passed as char*, length to
    //pad to passed as int.
    //Postconditions: binString is padded with 0s to match
    //length and returned.
    //-------------------------------------------------------

    char *padded = malloc(length + 1);

    long padNumber = length - strlen(binString);
    int i;
    for (i = 0; i < padNumber; i++) {
        strcat(padded, "0");
    }

    strcat(padded, binString);

    return padded;
}//padBinaryToLength

char characterForOctal(char *octal1, char* octal2) {
    //-------------------------------------------------------
    //Preconditions: octal1 and octal2 strings passed as parameter.
    //Postconditions: Character for octal returned.
```

```
	//Return the character for the octal row and column passed
	//based on the BCD for the system.
	//-------------------------------------------------------

	char c = 0;

	long row = binaryStringToDecimal(octal1);
	long column = binaryStringToDecimal(octal2);

	char bcd[8][8];

	bcd[0][0] = '0';
	bcd[0][1] = '1';
	bcd[0][2] = '2';
	bcd[0][3] = '3';
	bcd[0][4] = '4';
	bcd[0][5] = '5';
	bcd[0][6] = '6';
	bcd[0][7] = '7';

	bcd[1][0] = '8';
	bcd[1][1] = '9';
	bcd[1][2] = '[';
	bcd[1][3] = '#';
	bcd[1][4] = '@';
	bcd[1][5] = ':';
	bcd[1][6] = '>';
	bcd[1][7] = '?';


	bcd[2][0] = ' ';
	bcd[2][1] = 'A';
	bcd[2][2] = 'B';
	bcd[2][3] = 'C';
	bcd[2][4] = 'D';
	bcd[2][5] = 'E';
	bcd[2][6] = 'F';
	bcd[2][7] = 'G';


	bcd[3][0] = 'H';
	bcd[3][1] = 'I';
	bcd[3][2] = '&';
	bcd[3][3] = '.';
	bcd[3][4] = ']';
	bcd[3][5] = '(';
	bcd[3][6] = '<';
	bcd[3][7] = '\\';


	bcd[4][0] = '^';
	bcd[4][1] = 'J';
	bcd[4][2] = 'K';
	bcd[4][3] = 'L';
	bcd[4][4] = 'M';
	bcd[4][5] = 'N';
```

```
    bcd[4][6] = 'O';
    bcd[4][7] = 'P';


    bcd[5][0] = 'Q';
    bcd[5][1] = 'R';
    bcd[5][2] = '-';
    bcd[5][3] = '$';
    bcd[5][4] = '*';
    bcd[5][5] = ')';
    bcd[5][6] = ';';
    bcd[5][7] = '\'';


    bcd[6][0] = '\'';
    bcd[6][1] = '/';
    bcd[6][2] = 'S';
    bcd[6][3] = 'T';
    bcd[6][4] = 'U';
    bcd[6][5] = 'V';
    bcd[6][6] = 'W';
    bcd[6][7] = 'X';


    bcd[7][0] = 'Y';
    bcd[7][1] = 'Z';
    bcd[7][2] = '<';
    bcd[7][3] = '.';
    bcd[7][4] = '%';
    bcd[7][5] = '=';
    bcd[7][6] = '"';
    bcd[7][7] = '!';

    c = bcd[row][column];

    return c;

}//characterForOctal

char* octalToBinaryConversion(char *convertOctal, int length) {
    //-------------------------------------------------------
    //Preconditions: octal number to convert as string,
    //length for binary representation as int.
    //Postconditions: convertOctal is converted into binary
    //representation and is padded with 0s to match the
    //length passed.
    //-------------------------------------------------------

    char *binString;
    binString = (char*)malloc(length+1);

    int maxOctal = length/3;
    long remaining = maxOctal - strlen(convertOctal);
    int i;
    for (i = 0; i < remaining; i++) {
        strcat(binString, "000");
```

```
    }

    for (i = 0; i < strlen(convertOctal); i++) {
        switch(convertOctal[i]){
            case '0':
                strcat(binString, "000");
                break;
            case '1':
                strcat(binString, "001");
                break;
            case '2':
                strcat(binString, "010");
                break;
            case '3':
                strcat(binString, "011");
                break;
            case '4':
                strcat(binString, "100");
                break;
            case '5':
                strcat(binString, "101");
                break;
            case '6':
                strcat(binString, "110");
                break;
            case '7':
                strcat(binString, "111");
                break;
            default:
                printf("\nInvalid octal digit %c ", convertOctal[i]);
        }
    }

    return binString;
}//octalToBinaryConversion


/*-------------------------Opcode-------------------------*/
int getOpcodeIndex(char *opcode) {
    //-----------------------------------------------------
    //Preconditions: string passed as parameter.
    //Postconditions: Returns index of operator matching
    //string passed if string is found, -1 if no match is
    //discovered.
    //-----------------------------------------------------

    refreshArray();
    cutString(opcode);
    int i;
    int operationIndex = -1;
    for (i = 0; i < kNUM_OPS && operationIndex == -1; i++) {
        if(strcmp(opcode, opcodes[i]) == 0) {
            //Match
            operationIndex = i;
        }
    }
```

```
    return operationIndex;
}//getOpcodeIndex

/*-------------------------Architecture-------------------------*/
void updatePC() {
    //------------------------------------------------------------
    //Preconditions: Memory.
    //Postconditions: Memory is examined and current instruction
    //being executed is stored in PC.
    //------------------------------------------------------------

    char *lineNumber = malloc(7);
    PC = malloc(12);

    lineNumber = padBinaryToLength(decimalToBinaryString(currentLine), 7);
    strcpy(PC, lineNumber);
}

void updateIR() {
    //------------------------------------------------------------
    //Preconditions: Memory.
    //Postconditions: Memory is examined and current line is
    //loaded into IR.
    //------------------------------------------------------------

    IR = malloc(12);
    strcpy(IR, memory[currentLine]);
}

void updateCC() {
    //------------------------------------------------------------
    //Preconditions: stack.
    //Postconditions: Value on stack is examined and CC is
    //updated appropriately.
    //------------------------------------------------------------

    CC = malloc(2);
    if (stackPointer == kMAX_LINES - 1) {
        //Stack is empty
        strcpy(CC, "00");
    } else {
        if (memory[stackPointer + 1][0] == '1') {
            //Negative
            strcpy(CC, "10");
        } else if(binaryStringToDecimal(memory[stackPointer + 1]) != 0) {
            strcpy(CC, "01");
        } else {
            strcpy(CC, "00");
        }
    }
}

/*-------------------------Operations-------------------------*/
void push(char *address) {
    //------------------------------------------------------------
```

```
    //Preconditions: address passed as parameter.
    //Postconditions: value at address is pushed onto stack top.
    //Push value at address onto stack.
    //-------------------------------------------------------

    char *valueAtAddress = malloc(12);
    strcpy(valueAtAddress, memory[binaryStringToDecimal(address)]);

    memory[stackPointer] = malloc(12);
    strcpy(memory[stackPointer], valueAtAddress);

    stackPointer--;
}

void pop(char *address) {
    //-------------------------------------------------------
    //Preconditions: address passed as parameter.
    //Postconditions: value on stack top is removed and stored
    //at address.
    //Pop top of stack and store in address.
    //-------------------------------------------------------

    long lineNumber = binaryStringToDecimal(address);

    char *stackTop = malloc(256);
    stackTop = memory[++stackPointer];

    if (stackTop == NULL) {
        printf("STACK TOP AT LINE %ld is null\n", stackPointer);
    }

    if (memory[lineNumber] == NULL) {
        printf("MEMORY at LINE %ld is null.\n", lineNumber);
    }

    strcpy(memory[lineNumber], stackTop);

    memory[stackPointer] = NULL;
}

void add() {
    //-------------------------------------------------------
    //Preconditions: Two values on stack.
    //Postconditions: Values added.
    //Add top two values in stack and push result on stack.
    //-------------------------------------------------------

    char *op1 = memory[stackPointer + 1];
    char *op2 = memory[stackPointer + 2];

    long op1Long;
    long op2Long;

    if (op1[0] == '1') {
        //Negative
        op1Long = decimalForTwoComp(op1);
```

```
        op1Long = op1Long * -1;
    } else {
        //Positive
        op1Long = binaryStringToDecimal(op1);
    }

    if (op2[0] == '1') {
        //Negative
        op2Long = decimalForTwoComp(op2);
        op2Long = op2Long * -1;
    } else {
        //Positive
        op2Long = binaryStringToDecimal(op2);
    }

    //Pop both off stack
    memory[stackPointer + 1] = NULL;
    memory[stackPointer + 2] = NULL;
    stackPointer += 2;

    long sum = op1Long + op2Long;

    printf("SUM %ld\n", sum);

    char *sumString = malloc(256);

    if (sum < 0) {
        //Negative
        sum *= -1;
        sumString = twoCompForDecimal(sum);
    } else {
        //Positive
        sumString = padBinaryToLength(decimalToBinaryString(sum), 12);
    }

    //Pop sum on stack
    memory[stackPointer] = malloc(12);
    strcpy(memory[stackPointer], sumString);
    stackPointer--;
}

void subtract() {
    //-------------------------------------------------------------
    //Preconditions: Two values on stack.
    //Postconditions: Difference found.
    //Subtract top two values in stack and push result on stack.
    //Stack form: OP_2
    //            OP_1
    //-------------------------------------------------------------
    char *op2 = memory[stackPointer + 1];
    char *op1 = memory[stackPointer + 2];

    long op1Long;
    long op2Long;

    if (op1[0] == '1') {
```

```
        //Negative
        op1Long = decimalForTwoComp(op1);
        op1Long = op1Long * -1;
    } else {
        //Positive
        op1Long = binaryStringToDecimal(op1);
    }

    if (op2[0] == '1') {
        //Negative
        op2Long = decimalForTwoComp(op2);
        op2Long = op2Long * -1;
    } else {
        //Positive
        op2Long = binaryStringToDecimal(op2);
    }

    //Pop both off stack
    memory[stackPointer + 1] = NULL;
    memory[stackPointer + 2] = NULL;
    stackPointer += 2;

    long difference = op1Long - op2Long;

    char *differenceString = malloc(256);

    if (difference < 0) {
        //Negative
        difference *= -1;
        differenceString = twoCompForDecimal(difference);
    } else {
        //Positive
        differenceString = padBinaryToLength(decimalToBinaryString(difference),
            12);
    }

    //Pop difference on stack
    memory[stackPointer] = malloc(12);
    strcpy(memory[stackPointer], differenceString);
    stackPointer--;

}

void binaryAnd() {
    //-------------------------------------------------------------
    //Preconditions: Two values on stack.
    //Postconditions: Values are ANDed together. (union found)
    //Two values on top of stack are ANDed together.
    //-------------------------------------------------------------

    //AND top two values in stack and push result on stack.
    char *op1 = memory[stackPointer + 1];
    char *op2 = memory[stackPointer + 2];

    long op1Long;
    long op2Long;
```

```
    if (op1[0] == '1') {
        //Negative
        op1Long = decimalForTwoComp(op1);
        op1Long = op1Long * -1;
    } else {
        //Positive
        op1Long = binaryStringToDecimal(op1);
    }

    if (op2[0] == '1') {
        //Negative
        op2Long = decimalForTwoComp(op2);
        op2Long = op2Long * -1;
    } else {
        //Positive
        op2Long = binaryStringToDecimal(op2);
    }

    //Pop both off stack
    memory[stackPointer + 1] = NULL;
    memory[stackPointer + 2] = NULL;
    stackPointer += 2;

    long unionSet = 0;

    unionSet = op1Long & op2Long;

    char *unionString = malloc(256);

    if (unionSet < 0) {
        //Negative
        unionSet *= -1;
        unionString = twoCompForDecimal(unionSet);
    } else {
        //Positive
        unionString = padBinaryToLength(decimalToBinaryString(unionSet), 12);
    }

    printf("Union of %s and %s is %s.\n", op1, op2, unionString);

    //Pop sum on stack
    memory[stackPointer] = malloc(12);
    strcpy(memory[stackPointer], unionString);
    stackPointer--;
}

void binaryOr() {
    //-------------------------------------------------------
    //Preconditions: Two values on stack.
    //Postconditions: Values are ORed together. (intersection found)
    //Two values on top of stack are ORed together.
    //-------------------------------------------------------

    //OR top two values in stack and push result on stack.
    char *op1 = memory[stackPointer + 1];
```

```
    char *op2 = memory[stackPointer + 2];

    long op1Long;
    long op2Long;

    if (op1[0] == '1') {
        //Negative
        op1Long = decimalForTwoComp(op1);
        op1Long = op1Long * -1;
    } else {
        //Positive
        op1Long = binaryStringToDecimal(op1);
    }

    if (op2[0] == '1') {
        //Negative
        op2Long = decimalForTwoComp(op2);
        op2Long = op2Long * -1;
    } else {
        //Positive
        op2Long = binaryStringToDecimal(op2);
    }

    //Pop both off stack
    memory[stackPointer + 1] = NULL;
    memory[stackPointer + 2] = NULL;
    stackPointer += 2;

    long intersectionSet = 0;

    intersectionSet = op1Long | op2Long;

    char *intersectionString = malloc(256);

    if (intersectionSet < 0) {
        //Negative
        intersectionSet *= -1;
        intersectionString = twoCompForDecimal(intersectionSet);
    } else {
        //Positive
        intersectionString = padBinaryToLength(decimalToBinaryString
            (intersectionSet), 12);
    }

    printf("Intersection of %s and %s is %s.\n", op1, op2, intersectionString);

    //Pop sum on stack
    memory[stackPointer] = malloc(12);
    strcpy(memory[stackPointer], intersectionString);
    stackPointer--;
}

void binaryNot() {
    //-------------------------------------------------------
    //Preconditions: value on stack.
    //Postconditions: complements value on top of stack.
```

```
    //-----------------------------------------------------------

    //Complement top value in stack.
    char *op1 = memory[stackPointer + 1];

    long op1Long;

    if (op1[0] == '1') {
        //Negative
        op1Long = decimalForTwoComp(op1);
        op1Long = op1Long * -1;
    } else {
        //Positive
        op1Long = binaryStringToDecimal(op1);
    }

    //Pop off stack
    memory[stackPointer + 1] = NULL;
    stackPointer += 1;

    long complement = 0;

    complement = ~op1Long;

    char *complementString = malloc(256);

    if (complement < 0) {
        //Negative
        complement *= -1;
        complementString = twoCompForDecimal(complement);
    } else {
        //Positive
        complementString = padBinaryToLength(decimalToBinaryString(complement),
            12);
    }

    printf("Complement of %s is %s.\n", op1, complementString);

    //Pop complement on stack
    memory[stackPointer] = malloc(12);
    strcpy(memory[stackPointer], complementString);
    stackPointer--;
}

void binaryShiftLeft() {
    //-----------------------------------------------------------
    //Preconditions: value on stack.
    //Postconditions: value on top of stack is shifted 1 bit left.
    //shifts value on top of stack left one bit.
    //-----------------------------------------------------------

    //Binary shift left word on top of stack.
    char *op1 = memory[stackPointer + 1];

    long op1Long;
```

```
    if (op1[0] == '1') {
        //Negative
        op1Long = decimalForTwoComp(op1);
        op1Long = op1Long * -1;
    } else {
        //Positive
        op1Long = binaryStringToDecimal(op1);
    }

    //Pop off stack
    memory[stackPointer + 1] = NULL;
    stackPointer += 1;

    long shifted = 0;

    shifted = op1Long << 1;

    char *shiftedString = malloc(256);

    if (shifted < 0) {
        //Negative
        shifted *= -1;
        shiftedString = twoCompForDecimal(shifted);
    } else {
        //Positive
        shiftedString = padBinaryToLength(decimalToBinaryString(shifted), 12);
    }

    printf("Shift left of %s is %s.\n", op1, shiftedString);

    //Pop complement on stack
    memory[stackPointer] = malloc(12);
    strcpy(memory[stackPointer], shiftedString);
    stackPointer--;
}

void binaryShiftRight() {
    //-----------------------------------------------------------
    //Preconditions: stack with value.
    //Postconditions: value on top of stack is shifted right.
    //shifts value on top of stack right one bit.
    //-----------------------------------------------------------

    //Binary shift right word on top of stack.
    char *op1 = memory[stackPointer + 1];

    long op1Long;

    if (op1[0] == '1') {
        //Negative
        op1Long = decimalForTwoComp(op1);
        op1Long = op1Long * -1;
    } else {
        //Positive
        op1Long = binaryStringToDecimal(op1);
    }
```

```
    //Pop off stack
    memory[stackPointer + 1] = NULL;
    stackPointer += 1;

    long shifted = 0;

    shifted = op1Long >> 1;

    char *shiftedString = malloc(256);

    if (shifted < 0) {
        //Negative
        shifted *= -1;
        shiftedString = twoCompForDecimal(shifted);
    } else {
        //Positive
        shiftedString = padBinaryToLength(decimalToBinaryString(shifted), 12);
    }

    printf("Shift right of %s is %s.\n", op1, shiftedString);

    //Pop complement on stack
    memory[stackPointer] = malloc(12);
    strcpy(memory[stackPointer], shiftedString);
    stackPointer--;
}

void testSkip() {
    //-------------------------------------------------------------
    //Preconditions: CC stored as class variable.
    //Postconditions: Skips next instruction if CC is 00.
    //Skip next instruction if cc is 00.
    //-------------------------------------------------------------

    printf("*****TEST SKIP*****\n");
    if (strcmp(CC, "00") == 0) {
        currentLine += 2;
        jump = 1;
        updatePC();
        updateIR();
    }
}//testSkip

void inputCharacter(char *address) {
    //-------------------------------------------------------------
    //Preconditions: address passed as parameter.
    //Postconditions: Character taken from input and stored at
    //address.
    //Take character from input and store at address.
    //Characters are stored in octal.
    //-------------------------------------------------------------

    long lineNumber = binaryStringToDecimal(address);
```

```
    printf("Enter a character value in octal: ");

    char octal[12];
    int sIndex = 0;

    char c;

    do {
        c = fgetc(stdin);
        if (c != EOF && c != '\n' && c != ' ') {
            octal[sIndex] = c;
            octal[sIndex + 1] = '\0';
            sIndex++;
        } else {
            break;
        }
    } while (c != EOF);

    printf("ENTERED %s\n", octal);

    char *binString = malloc(256);

    binString = octalToBinaryConversion(octal, 12);

    strcpy(memory[lineNumber], padBinaryToLength(binString, 12));
}

void outputCharacter(char *address) {
    //-------------------------------------------------------
    //Preconditions: address passed as parameter.
    //Postconditions: outputs character stored at address to
    //console.
    //Output character stored at address.
    //0,1,2,3,4,5,(6,7,8),(9,10,11)
    //-------------------------------------------------------

    long lineNumber = binaryStringToDecimal(address);

    // printf("LINENUMBER %ld\n", lineNumber);

    char *octalRow = malloc(5);
    strncpy(octalRow, memory[lineNumber]+6, 3);

    char *octalColumn = malloc(5);
    strncpy(octalColumn, memory[lineNumber] + 9, 3);

    // printf("MEMORY %s\n", memory[lineNumber]);
    // printf("OctalRow %s\n", octalRow);
    // printf("OctalColumn %s\n", octalColumn);

    char c = characterForOctal(octalRow, octalColumn);

    printf("\nOUTPUT: %c\n\n", c);
}

void inputInteger(char *address) {
```

```
    //-------------------------------------------------------------
    //Preconditions: address passed as parameter.
    //Postconditions: stores integer input at address.
    //Input integer and store at address.
    //-------------------------------------------------------------

    printf("Enter an integer value in decimal: ");
    int d;
    scanf("%d", &d);
    printf("ENTERED %d\n", d);

    char *binaryRep = malloc(256);
    if (d < 0) {
        d = d * -1;
        binaryRep = twoCompForDecimal(d);
    } else {
        binaryRep = decimalToBinaryString(d);
    }

    printf("BINARY: %s\n", binaryRep);

    long lineNumber = binaryStringToDecimal(address);
    printf("STORE IN %ld\n", lineNumber);
    strcpy(memory[lineNumber],padBinaryToLength(binaryRep, 12));
}

void outputInteger(char *address) {
    //-------------------------------------------------------------
    //Preconditions: address passed as parameter.
    //Postconditions: integer stored at address output to console.
    //Output integer and store at address.
    //-------------------------------------------------------------

    long lineNumber = binaryStringToDecimal(address);

    long value;

    if (memory[lineNumber][0] == '1') {
        //Negative
        long temp = decimalForTwoComp(memory[lineNumber]);
        value = temp * -1;
    } else {
        //Positive
        value = binaryStringToDecimal(memory[lineNumber]);
    }

    printf("\nOUTPUT: %ld\n\n", value);
}

void jumpJMP(char *address) {
    //-------------------------------------------------------------
    //Preconditions: address passed as parameter.
    //Postconditions: Jump to address.
    //Jump. CC not a factor.
    //-------------------------------------------------------------
```

```
    long lineNumber = binaryStringToDecimal(address);
    printf("*****JUMP*****\n LINE %ld\n", lineNumber);
    currentLine = lineNumber;

    updatePC();
    updateIR();
    jump = 1;
}

void jumpJEQ(char *address) {
    //-------------------------------------------------------------
    //Preconditions: address passed as parameter.
    //Postconditions:jump JEQ executed.
    //Jump if CC is equal to 0.
    //-------------------------------------------------------------

    long lineNumber = binaryStringToDecimal(address);
    printf("*****JUMP*****\n LINE %ld\n", lineNumber);
    if (strcmp(CC, "00") == 0) {
        currentLine = lineNumber;
        updatePC();
        updateIR();
        jump = 1;
    }
}

void jumpJGT(char *address) {
    //-------------------------------------------------------------
    //Preconditions: address passed as parameter.
    //Postconditions: jumpJGT executed.
    //Jump if CC is greater than 0.
    //-------------------------------------------------------------

    long lineNumber = binaryStringToDecimal(address);
    printf("*****JUMP*****\n LINE %ld\n", lineNumber);
    if (strcmp(CC, "01") == 0) {
        currentLine = lineNumber;
        updatePC();
        updateIR();
        jump = 1;
    }
}

void jumpJLT(char *address) {
    //-------------------------------------------------------------
    //Preconditions: address passed as parameter.
    //Postconditions: jump JLT executed.
    //Jump if CC is less than 0.
    //-------------------------------------------------------------

    long lineNumber = binaryStringToDecimal(address);
    printf("*****JUMP*****\n LINE %ld\n", lineNumber);
    if (strcmp(CC, "10") == 0) {
        currentLine = lineNumber;
        updatePC();
        updateIR();
```

```
            jump = 1;
        }
    }
```