```
/*
 ----------------------------------------------------------------------
 PROGRAM NAME: Two Pass Assembler
 PROGRAMMER:   Samuel Jentsch/James Francis
 CLASS:        CSC 214.001, Spring 2014
 INSTRUCTOR:   Dr. Strader
 DATE STARTED: April 2, 2014
 DUE DATE:     April 7, 2014

 PROGRAM PURPOSE:

 Read an assembly file and parse the file into machine code.
 Report any errors that are encountered.
 Print machine code to file in no errors were detected.


 VARIABLE DICTIONARY:

 //Constants (Assembly Syntax, used for parsing file)
 const char *operations[kNUM_OPS] - Holds all operations.
 const char *opcodes[kNUM_OPS] - Holds all binary opcodes.
 const char *directives[kNUM_DIRECTIVES] - Holds all directives.

 //Machine code Holder
 char *machineCodeHolder[kMAX_LINES] - Holds machine code for program.
 line by line for processing/printing.
 int machineCodeIndex - Index of the last added line in the
 machineCodeHolder.
 int addressIndex - Used to count the number of actual code lines
 (doesn't count error lines). Used for memory addresses.
 int secondPassLabelsForLine[kMAX_LINES] - Used to hold the index
 in the symbol table corresponding to a line in the machine code.
 Used to append appropriate address for label on second pass.

 //Symbol Table
 char *symbolTable[kMAX_SYMBOLS] - Holds the symbols encountered
 in the program to create the symbol table.
 int symbolTableIndex - Tracks the last added symbol table index.
 char *symbolTableAddresses[kMAX_SYMBOLS] - Holds the binary addresses
 associated with each label in the symbol table.

 //Undefined table
 int errorTableLineNumbers[kMAX_LINES][kMAX_ERRORS] - Used to hold the
 line numbers with undefined labels.
 char *errorTable[kMAX_LINES] - Used to hold the undefined labels
 (undefined symbol table).
 int errorTableIndex - Tracks the last added error table index.

 //Multiply defined table
 int multiplyDefinedTableLineNumbers[kMAX_LINES][kMAX_ERRORS] - Used to
 hold line number associated with multiply defined labels.
 char *multiplyDefinedTable[kMAX_LINES] - Holds the labels that are
 multiply defined.
 int multiplyDefinedTableIndex - Tracks the last added multiply defined
 table index.
```

```
 //Output File
 FILE *machineFile - Used for file output to machine file.

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

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

#define kNUM_OPS 19     //Total number of operations.
#define kMAX_SYMBOLS 50 //Maximum number of labels
#define kMAX_LINES 100  //Maximum assembly file size
#define kNUM_DIRECTIVES 5 //Total number of assembler directives
#define kMAX_ERRORS 5   //Maximum number of errors.
#define kWORD_LENGTH = 12; //Length of word (12 bits).

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

//Machine code Holder
char *machineCodeHolder[kMAX_LINES];
int machineCodeIndex;
int addressIndex;
int secondPassLabelsForLine[kMAX_LINES];

//Holds Assembly code for final printing.
char *assemblyCodeHolder[kMAX_LINES];
int assemblyCodeIndex;

//Symbol Table
char *symbolTable[kMAX_SYMBOLS];
int symbolTableIndex;
char *symbolTableAddresses[kMAX_SYMBOLS];

//Undefined table
int errorTableLineNumbers[kMAX_LINES][kMAX_ERRORS];
char *errorTable[kMAX_LINES];
int errorTableIndex;

//Multiply defined table
int multiplyDefinedTableLineNumbers[kMAX_LINES][kMAX_ERRORS];
char *multiplyDefinedTable[kMAX_LINES];
int multiplyDefinedTableIndex;

//Output File
FILE *machineFile;
int ERROR;

//File handling
void readFile(FILE *assemblyFile);
void outputToMachineFile(FILE *machineFile);

//Code processing
```

```
int processStringInput(char *stringInput, int lineNumber);
void secondPass();
void printMachineCodeHolder();
void addMachineCodeWithoutOperand(char *machineCode);
void addMachineCodeWithOperand(char *machineCode, char *operand);
void addMachineCodeForOperandError();

//Assember Directives
int isAssemblerDirective(char *mnemonic);
char* dataForDirective(char *directive, char *data);
int getIndexForDirective(char *directive);

//Operators
int isOperator(char* mnemonic);
int hasOperand(int operationIndex);
int getMnemonicIndex(char *mnemonic);
char* convertMnemonic(char *mnemonic);

//Symbol Table
int handleLabel(char *label);
void saveLineNumberForLabel(char *label);
void printSymbolTable();
void cutString(char *label);
int getLabelIndex(char *label);

//Undefined
int getUndefinedLabelIndex(char* label);
void handleUndefinedSymbols();
void addLineNumberForUndefinedSymbol(int symbolIndex, int lineNumber);
void printUndefinedSymbolTable();

//Multiply Defined
int getMultipleDefinedLabelIndex(char* label);
void handleMultipleDefinedSymbols();
void addLineNumberForMultipleDefinedSymbol(int symbolIndex, int lineNumber);
void printMultipleDefinedSymbolTable();
void addMultipleDefinedSymbol(char *label);
int secondPassLabelsForMultipleLine[kMAX_LINES];

//Conversions/String manipulation
char* decimalBinaryConversion(int n, int length);
char* octalToBinaryConversion(char *convertOctal, int length);
char* padBinaryToLength(char *binString, int length);
void cutComments(char* assemblyLine);

/*--------------------Initialization--------------------*/
int main(int argc, const char * argv[]) {
    //---------------------------------------------------
    //Initializes constants and arrays used in program.
    //Opens files and begins file read/parsing process.
    //---------------------------------------------------

    operations[0] = "PSH"; //Push
    operations[1] = "POP"; //Pop
    operations[2] = "ADD"; //Add
    operations[3] = "SUB"; //Subtract
```

```
    operations[4] = "AND"; //And
    operations[5] = "IOR"; // Or
    operations[6] = "NOT"; //Not (complement)
    operations[7] = "TSS"; //testskip
    operations[8] = "SHL"; //Shift left
    operations[9] = "SHR"; //Shift right
    operations[10] = "JMP"; //Jump
    operations[11] = "JEQ"; //Jump
    operations[12] = "JGT"; //Jump
    operations[13] = "JLT"; //Jump
    operations[14] = "INC"; //input character
    operations[15] = "OTC"; //output character
    operations[16] = "INI"; //input integer
    operations[17] = "OTI"; //output integer
    operations[18] = "HLT"; //halt program

    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

    directives[0] = ".nd"; //End of listing
    directives[1] = ".dw"; //Define integer word
    directives[2] = ".ds"; //Define storage
    directives[3] = ".dc"; //Define character
    directives[4] = ".st"; //Start of listing

    symbolTableIndex = 0;
    machineCodeIndex = 0;
    multiplyDefinedTableIndex = 0;
    addressIndex = 0; //Index for addresses (labels).
    errorTableIndex = 0;
    assemblyCodeIndex = 0;

    //Set to no error.
    ERROR = 0;

    //Intialize array for line correspondence
    int i;
    for (i = 0; i < kMAX_LINES; i++) {
        secondPassLabelsForLine[i] = -1;
```

```
        secondPassLabelsForMultipleLine[i] = -1;
    }

    //Initialize error line array
    //and initialze multiply defined error array.
    int j;
    for (i = 0; i < kMAX_LINES; i++) {
        for (j = 0; j < kMAX_ERRORS; j++) {
            errorTableLineNumbers[i][j] = -1;
            multiplyDefinedTableLineNumbers[i][j] = -1;
        }
    }

    FILE *assemblyFile = fopen("/Users/SamJ/Google Drive/SFA/Courses/2014_Spring/
        CSC_214/Lab6/DataFile.dat", "r");
    //FILE *assemblyFile = fopen("./proj1a.dat", "r");
    machineFile = fopen("/Users/SamJ/Google Drive/SFA/Courses/2014_Spring/
        CSC_214/Lab6/assembly", "wb");
    //machineFile = fopen("./lab6Machine.dat", "wb");

    readFile(assemblyFile);
    fclose(assemblyFile);
    fclose(machineFile);

    return 0;
}//main

/*----------------------File Management----------------------*/
void readFile(FILE *assemblyFile) {
    //-------------------------------------------------------
    //Preconditions: File pointer to a file containing
    //assembly file to be assembled.
    //Postconditions: File is parsed line by line and
    //assembled. Symbol table, undefined symbol table, and
    //machine code are displayed to console. If there are no
    //errors, the machine code is output to a file.
    //-------------------------------------------------------

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

    char inputString[256];
    fgets(inputString, 256, assemblyFile);

    int continueProcessing = 0;
    int lineNumber = 0;
    do {
        assemblyCodeHolder[assemblyCodeIndex] = malloc(128);
        strcpy(assemblyCodeHolder[assemblyCodeIndex], inputString);
        assemblyCodeIndex++;
        continueProcessing = processStringInput(inputString, lineNumber);
        fgets(inputString, 256, assemblyFile);
        lineNumber++;
    }while (strcmp(inputString, ".nd") != 0 && continueProcessing != -1);
```

```
    handleUndefinedSymbols();
    printUndefinedSymbolTable();

    handleMultipleDefinedSymbols();
    printMultipleDefinedSymbolTable();
    printSymbolTable();
    printf("FIRST PASS: \n");
    printMachineCodeHolder();
    printf("SECOND PASS: \n");

    secondPass();

    printMachineCodeHolder();

    if (ERROR == 0) {
        printf("\nNo error detected in program. Outputing to file.\n");
        outputToMachineFile(machineFile);
    } else {
        printf("\nError detected in program. Not outputing to file.\n");
    }
}//readFile

void outputToMachineFile(FILE *machineFile) {
    //------------------------------------------------------
    //Preconditions: Pointer to created file to write machine
    //code to passed as a parameter. Array containing machine
    //code line by line as global variable to print.
    //Postconditions: Array containing machine code is printed
    //line by line to the machineFile.
    //------------------------------------------------------

    int i;
    for (i = 0; i < machineCodeIndex; i++) {
        fputs(machineCodeHolder[i], machineFile);
        fputc('\n', machineFile);
    }

}//outputToMachineFile

/*----------------------Input Parsing----------------------*/
int processStringInput(char *stringInput, int lineNumber) {
    //------------------------------------------------------
    //Main logic for parsing assembly lines. Contains logic
    //for different cases outlined below. Formats determined
    //at different stages of the method are indicated.
    //Preconditions: String of assembly file line passed as
    //parameter.
    //Postconditions: The string is parsed, errors are
    //determined, and the symbol tables/machine code holder
    //is updated appropriately.
    //------------------------------------------------------

    cutComments(stringInput); //Remove comments from string.
    char *splitLine[10];
    char *split;
```

```
int index = 0;
split = strtok (stringInput," ");
while (split != NULL) {
    splitLine[index] = split;
    split = strtok(NULL, " ");
    index++;
}

//Used for error handling. Number of elements in the string can be
//used to check the string at certain points. If the number of elements
//doesn't match the string format at a logic point, there is an error.
int numberOfElements = index;

//Splitline now contains each part of the line read as a separate string.

/*-------------------------------------------------------------------------*/
//Logic for parsing:
//1. A single assembler directive (either .st or .nd) is valid.
//
//2.If the first token is a valid operator, then the next token should be
//an operand (label/address). Unless the operator doesn't have operands,
//like ADD.
//
//3. If the first token is NOT a valid operator, then the next token either
//has to be a valid operator (for a jump mark) or an assembler directive
//(to mark a label storing data).
//
//4. If the token mentioned above is a valid operator, the first case
//should be followed.
//
//5. If the token mentioned above is an assembler directive, the
//appropriate conversion should take place (either a character or integer)
//to create the binary representation of the data and print it to the
//machine file.
/*-------------------------------------------------------------------------*/

int isValidAssemblerDirective;
int isValidOperator;
int isValidLabel;


isValidAssemblerDirective = isAssemblerDirective(splitLine[0]);
if (isValidAssemblerDirective == 1) {
    //Line is an assembler directive (either start or end of listing)
    //FORMAT: *** (assembler directive)
    printf("Line %d: is a single assembler directive\n", lineNumber);

    if (getIndexForDirective(splitLine[0]) == 0) {
        //Directive is .nd. End the program.
        return -1;
    } else {
        return 0;
    }
}

isValidOperator = isOperator(splitLine[0]);
```

```
    if(isValidOperator == 1) {
        //Line is an operator, either with or without an operand.
        //FORMAT 1: *** (Operation without operand)
        //FORMAT 2: *** ****** (Operation with an operand)

        int opIndex = getMnemonicIndex(splitLine[0]);
        int opHasOperand = hasOperand(opIndex);
        char *operationCode = convertMnemonic(splitLine[0]);
        if (opHasOperand == 0) {
            //Operation does not require an operand.
            //FORMAT 1: *** (Operation without operand)

            if (numberOfElements > 1) {
                //ERROR, WRONG # OPERANDS
                //FORMAT: *** (Invalid Operation)
                //UUUUU
                printf("ERROR, WRONG NUMBER OPERANDS %s //FORMAT: *** (Too many
                    Operands)\n", splitLine[0]);
                addMachineCodeForOperandError();
            }

            printf("Line %d: %s is an operation that does not need an operand.
                Opcode: %s\n", lineNumber, splitLine[0], operationCode);

            char *padded = padBinaryToLength(operationCode, 12);
            addMachineCodeWithoutOperand(padded);
        } else {
            //Operation requires an operand.
            //FORMAT 2: *** ****** (Operation with an operand)

            if (numberOfElements != 2) {
                //ERROR, WRONG # OPERANDS
                //FORMAT: *** ______ (Missing Operand)
                //UUUUU
                printf("ERROR, WRONG NUMBER OPERANDS %s  //FORMAT: *** ______
                    (Missing Operand)\n\t OR (Too many operands)\n", splitLine[0]
                    );
                addMachineCodeForOperandError();
                return 0;
            }

            isValidLabel = handleLabel(splitLine[1]);
            if (isValidLabel == 1) {
                //Label is valid and has been added to the symbol table
                //by handleLabel.

                printf("Line %d: %s is an operation with a VALID operand. Opcode:
                    %s\n", lineNumber, splitLine[0], operationCode);

                addMachineCodeWithOperand(operationCode, splitLine[1]);
            }
        }
        return 0;
    }
```

```
if (hasOperand(getMnemonicIndex(splitLine[1])) == 1 && numberOfElements <= 2)
    {
    //ERROR INVALID OPERATOR
    //FORMAT 1: *** (Invalid Operation)
    //FORMAT 2: *** ****** (Invalid Operation, Operand)
    printf("ERROR, INVALID OPERATOR %s//FORMAT 1: *** (Invalid Operation)  //
        FORMAT 2: *** ****** (Invalid Operation, Operand)\n", splitLine[0]);
    addMachineCodeForOperandError();
    return 0;
}

isValidLabel = handleLabel(splitLine[0]);
if (isValidLabel == 1) {
    //Line begins with a label. There are 3 possibilities.
    //FORMAT 1: ****** *** (Label, Operation)
    //FORMAT 2: ****** *** ****** (Label, Operation, Operand)
    //FORMAT 3: ****** *** ______ (Label, Directive, Data value)

    //Save line number in the symbol table. (If a line begins with a label,
        the line number should be saved
    //as the label address)
    saveLineNumberForLabel(splitLine[0]);


    isValidAssemblerDirective = isAssemblerDirective(splitLine[1]);
    if (isValidAssemblerDirective == 1) {
        //FORMAT 3: ****** *** ______ (Label, Directive, Data value)
        char *machineData = dataForDirective(splitLine[1], splitLine[2]);
        //Print machine data to file.
        printf("Line %d: is a label, followed by an assembler directive,
            followed by data value %s\n", lineNumber, machineData);

        secondPassLabelsForMultipleLine[machineCodeIndex] = getLabelIndex
            (splitLine[0]);
        addMachineCodeWithoutOperand(machineData);

        return 0;
    }

    isValidOperator = isOperator(splitLine[1]);

    if (isValidOperator == 1) {
        //FORMAT 1: ****** *** (Label, Operation)
        //OR
        //FORMAT 2: ****** *** ****** (Label, Operation, Operand)

        int opIndex = getMnemonicIndex(splitLine[1]);
        int opHasOperand = hasOperand(opIndex);
        char *operationCode = convertMnemonic(splitLine[1]);
        if (opHasOperand == 0) {
            //Operation does not require an operand.
            //FORMAT 1: ****** *** (Label, Operation)

            if (numberOfElements > 2) {
                //ERROR, WRONG # OPERANDS
                //FORMAT: ****** *** (Label, Invalid Operation)
```

```
                    //UUUUU
                    printf("ERROR, WRONG NUMBER OPERANDS %s //FORMAT: ****** ***
                        (Label, Invalid Operation)\n", splitLine[1]);
                    addMachineCodeForOperandError();
                    return 0;
                }

                printf("Line %d: is a label followed by %s that does not need an
                    operand. Opcode: %s\n", lineNumber, splitLine[1],
                    operationCode);

                secondPassLabelsForMultipleLine[machineCodeIndex] = getLabelIndex
                    (splitLine[0]);
                addMachineCodeWithoutOperand(padBinaryToLength(operationCode, 7))
                    ;

            } else {
                //Operation requires an operand.
                //FORMAT 2: ****** *** ****** (Label, Operation, Operand)
                isValidLabel = handleLabel(splitLine[2]);
                if (isValidLabel == 1) {
                    //Label is valid and has been added to the symbol table
                    //by handleLabel.

                    if (numberOfElements < 3) {
                        //ERROR, WRONG # OPERANDS
                        //FORMAT: ****** *** ****** (Label, Invalid Operation,
                            Operand)
                        //UUUUU
                        printf("ERROR, WRONG NUMBER OPERANDS %s //FORMAT: ******
                            *** ****** (Label, Invalid Operation, Operand)\n",
                            splitLine[1]);
                        addMachineCodeForOperandError();
                        return 0;
                    }

                    printf("Line %d: is a label followed by %s with a VALID
                        operand. Opcode: %s\n", lineNumber, splitLine[1],
                        operationCode);

                    addMachineCodeWithOperand(operationCode, splitLine[2]);
                    secondPassLabelsForMultipleLine[machineCodeIndex - 1] =
                        getLabelIndex(splitLine[0]);
                }
            }
        }

        return 0;
    }

    return 0;

}//processStringInput

void secondPass() {
    //-----------------------------------------------------
```

```
    //Preconditions: First pass has occurred. Error labels
    //have been removed.
    //Postconditions: Second pass takes place. Addresses for
    //labels are written to the machine code holder to
    //complete the machine code.
    //------------------------------------------------------

    int i;
    for (i = 0; i < machineCodeIndex; i++) {
        if (secondPassLabelsForLine[i] != -1) {
            //if (strlen(machineCodeHolder[i]) < 12) {
            strcat(machineCodeHolder[i], symbolTableAddresses
                [secondPassLabelsForLine[i]]);
            //}
        }
    }
}//secondPass

void addMachineCodeWithoutOperand(char *machineCode) {
    //------------------------------------------------------
    //Preconditions: String of machine code passed as
    //parameter.
    //Postconditions: Adds the machineCode to the next line
    //of the machine code holder. Increments the index.
    //------------------------------------------------------

    machineCodeHolder[machineCodeIndex] = malloc(128);
    strcpy(machineCodeHolder[machineCodeIndex], machineCode);
    machineCodeIndex++;
    addressIndex++;
}//addMachineCodeWithoutOperand

void addMachineCodeWithOperand(char *machineCode, char *operand) {
    //------------------------------------------------------
    //Preconditions: String of machine code passed as
    //parameter.
    //Postconditions: Adds the machineCode to the next line
    //of the machine code holder and sets the index for the
    //line to match the label for the operand so the correct
    //address is swapped during the second pass.
    //Increments the index.
    //------------------------------------------------------

    machineCodeHolder[machineCodeIndex] = malloc(128);
    strcpy(machineCodeHolder[machineCodeIndex], machineCode);
    secondPassLabelsForMultipleLine[machineCodeIndex] = getLabelIndex(operand);
    secondPassLabelsForLine[machineCodeIndex] = getLabelIndex(operand);
    machineCodeIndex++;
    addressIndex++;
}//addMachineCodeWithOperand

void addMachineCodeForOperandError() {
    //------------------------------------------------------
    //Postconditions: Adds line representation for operator
    //error.
    //------------------------------------------------------
```

```
    machineCodeHolder[machineCodeIndex] = malloc(128);
    strcpy(machineCodeHolder[machineCodeIndex], "???????????");
    //An error occurred.
    ERROR = 1;
    machineCodeIndex++;
}//addMachineCodeForOperandError

void printMachineCodeHolder() {
    //-------------------------------------------------------
    //Preconditions: Global machineCodeHolder
    //Postconditions: Machine code is displayed to console.
    //-------------------------------------------------------

    printf("------------MACHINE CODE-------------\n");
    printf("-------------------------------------\n");

    int i;
    for (i = 0; i < machineCodeIndex; i++) {
        printf("| %12s | %18s\n", machineCodeHolder[i], assemblyCodeHolder[i+1]);
    }
    printf("-------------------------------------\n");
}//printMachineCodeHolder

/*---------------------Directives---------------------*/
int isAssemblerDirective(char *mnemonic) {
    //-------------------------------------------------------
    //Preconditions: char *mneonic passed as parameter.
    //directives array containing valid directives.
    //Postconditions: directives array is checked to see if
    //the mnemonic passed is a valid directive. 1 is returned
    //if the mnemonic is a directive.
    //-------------------------------------------------------

    int isDirective = 0; //0 is false

    cutString(mnemonic);

    if (strlen(mnemonic) > 3) {
        return isDirective;
    }

    int i;
    for (i = 0; i < kNUM_DIRECTIVES && isDirective == 0; i++) {
        if(strcmp(mnemonic, directives[i]) == 0) {
            isDirective = 1; //Set to true
        }
    }

    return isDirective;
}//isAssemblerDirective

int getIndexForDirective(char *directive) {
    //-------------------------------------------------------
    //Preconditions: char* directive passed as parameter.
    //Postconditions: Checks each index of directives array
```

```
    //for a match to the string passed. Index of directive
    //is returned if a match is found, -1 is returned if not.
    //-----------------------------------------------------

    int directiveIndex = -1;

    cutString(directive);

    int i;
    for(i = 0; i < kNUM_DIRECTIVES && directiveIndex == -1; i++) {
        if (strcmp(directives[i], directive) == 0) {
            directiveIndex = i;
        }
    }

    return directiveIndex;
}//getIndexForDirective

char* dataForDirective(char *directive, char *data) {
    //-----------------------------------------------------
    //Preconditions: Directive and data to be created passed
    //as parameter.
    //Postconditions: The correct string representation for
    //the data parameter is created and added to the machine
    //code at the current line. The type of data created is
    //based on the directive type (dw, ds, dc).
    //-----------------------------------------------------

    //Returns a string representation of the correct data
    //for the data type (char or int).
    char *machineData;

    cutString(data);

    int directiveIndex = getIndexForDirective(directive);
    int i;
    for(i = 0; i < kNUM_DIRECTIVES && directiveIndex == -1; i++) {
        if (strcmp(directives[i], directive) == 0) {
            directiveIndex = i;
        }
    }

    // directives[0] = ".nd"; //End of listing
    // directives[1] = ".dw"; //Define integer word
    // directives[2] = ".ds"; //Define storage
    // directives[3] = ".dc"; //Define character
    // directives[4] = ".st"; //Start of listing

    if (directiveIndex == 1) {
        //.dw: define integer word.
        int decimal = atoi(data);
        machineData = decimalBinaryConversion(decimal, 12);
    } else if(directiveIndex == 2) {
        //.ds: define storage space.
        int decimal = atoi(data); //amount of space to reserve.
        int i;
```

```
        machineData = "000000000000";
        for (i = 0; i < decimal - 1; i++) {
            assemblyCodeHolder[assemblyCodeIndex] = malloc(128);
            strcpy(assemblyCodeHolder[assemblyCodeIndex], "\n");
            assemblyCodeIndex++;
            addMachineCodeWithoutOperand(machineData);
        }
    } else if(directiveIndex == 3) {
        //.dc: define character.
        machineData = octalToBinaryConversion(data, 12);
    } else {
        //Either .nd or .st: Not used correctly.
    }
    return machineData;
}//dataForDirective

/*----------------------Operators----------------------*/
int isOperator(char *mnemonic) {
    //-------------------------------------------------------
    //Preconditions: string passed as parameter.
    //Postconditions: Checks array operations to see if string
    //passed matches a valid operation mnemonic. Returns
    //1 if mnemonic is an operator, 0 otherwise.
    //-------------------------------------------------------

    int isMnemonic = 0; //0 is false

    cutString(mnemonic);

    if (strlen(mnemonic) > 3) {
        return isMnemonic;
    }

    int i;
    for (i = 0; i < kNUM_OPS && isMnemonic == 0; i++) {
        if(strcmp(mnemonic, operations[i]) == 0) {
            isMnemonic = 1; //Set to true
        }
    }

    return isMnemonic;
}//isOperator

int hasOperand(int operationIndex) {
    //-------------------------------------------------------
    //Preconditions: operationIndex passed as parameter.
    //Postconditions: Returns 0 if operation has requires an
    //operand, 0 if not.
    //-------------------------------------------------------

    //Operations with operands:
    //PSH 0
    //POP 1
    //JUMPS 10-13
    //INC 14
    //OTC 15
```

```
    //INI 16
    //OTI 17

    int hasOperand = 0;

    int indicesWithOperands[] = {0, 1, 10, 11, 12, 13, 14, 15, 16, 17};

    int i;
    for (i = 0; i < 10; i++) {
        if(indicesWithOperands[i] == operationIndex)
            hasOperand = 1;
    }

    return hasOperand;
}//hasOperand

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

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

    return operationIndex;
}//getMnemonicIndex

char* convertMnemonic(char *mnemonic) {
    //-------------------------------------------------------
    //Preconditions: string mneonic to convert to binary
    //opcode representation passed as parameter.
    //Postconditions: Gets the binary string representation
    //for the operation passed and returns the string.
    //-------------------------------------------------------

    char *opCode = malloc(32);

    int operationIndex = getMnemonicIndex(mnemonic);

    if (operationIndex != -1) {
        strcpy(opCode, opcodes[operationIndex]);
    }

    return opCode;
}//convertMnemonic
```

```
/*---------------------Labels/Symbol Table---------------------*/
int handleLabel(char *label) {
    //-------------------------------------------------------
    //Preconditions: Label to add to symbol table passed as
    //parameter.
    //Postconditions: Adds label to appropriate index of
    //symbol table if not already present. Returns 1 if label
    //is valid, 0 if not.
    //-------------------------------------------------------

    //-----------------------------------------//
    //Checks to see if label has valid format.
    //If valid, label is added to the symbol table
    //if not already present in the table.
    //-----------------------------------------//

    int isLabel = 0; //0 is false
    cutString(label);

    if (strlen(label) > 6 || strlen(label) == 0) {
        //Invalid size
        return isLabel;
    }


    if (symbolTableIndex == 0) {
        symbolTable[symbolTableIndex] = malloc(32);
        strcpy(symbolTable[symbolTableIndex], label);
        symbolTableIndex++;
        isLabel = 1;
    }

    int labelIndex = getLabelIndex(label);

    if (labelIndex == -1) {
        //Add the label to the symbol table.
        symbolTable[symbolTableIndex] = malloc(32);
        strcpy(symbolTable[symbolTableIndex], label);
        symbolTableIndex++;
        isLabel = 1;
    } else {
        isLabel = 1;
    }

    return isLabel;
}//handleLabel

int getLabelIndex(char *label) {
    //-------------------------------------------------------
    //Preconditions: string label passed as parameter.
    //Postconditions: index of label passed in symbol table
    //returned if found, -1 is returned of no match discovered.
    //-------------------------------------------------------

    int labelIndex = -1;
```

```
    cutString(label);

    int i;
    for (i = 0; i < symbolTableIndex && labelIndex == -1; i++) {
        if(strcmp(label, symbolTable[i]) == 0) {
            labelIndex = i; //Set to true
        }
    }

    return labelIndex;
}//getLabelIndex

void saveLineNumberForLabel(char *label) {
    //-------------------------------------------------------
    //Preconditions: Label encountered passed as string
    //parameter.
    //Postconditions: Adds binary address location for label
    //to symbolTableAddresses at the label index to use for
    //2nd pass.
    //Handles adding the address for a label (the labels
    //location to swap in during 2nd pass).
    //-------------------------------------------------------

    int labelIndex = getLabelIndex(label);
    if (labelIndex != -1) {
        if (symbolTableAddresses[labelIndex] == NULL) {
            symbolTableAddresses[labelIndex] = decimalBinaryConversion
                (addressIndex, 7);
        } else {
            //ERROR, MULTIPLY DEFINED LABEL
            //MMMMMMM
            printf("ERROR, MULTIPLY DEFINED LABEL %s\n", label);
            addMultipleDefinedSymbol(label);
        }
    }
}//saveLineNumberForLabel

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

    long length = strlen(label);
    int i;

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

void printSymbolTable() {
```

```
    //-------------------------------------------------------
    //Preconditions: symbolTable array, global variable.
    //Postconditions: Each value of the symbolTable array
    //containing a non-null value is printed along with the
    //address corresponding to that label.
    //-------------------------------------------------------

    printf("\n-------SYMBOL TABLE--------\n");
    printf("|   LABEL   |    ADDRESS  |\n");
    printf("---------------------------\n");

    char *placeHolder = "   |   ";
    int i;
    for (i = 0; i < kMAX_SYMBOLS; i++) {
        if (symbolTable[i] != NULL) {
            printf("| %6s %s %7s |\n", symbolTable[i], placeHolder,
                symbolTableAddresses[i]);
        }
    }
    printf("---------------------------\n");
}//printSymbolTable

/*--------------------Undefined Symbol Table--------------------*/
int getUndefinedLabelIndex(char* label) {
    //-------------------------------------------------------
    //Preconditions: string label passed as parameter.
    //Postconditions: Returns the location of the label in
    //the undefined symbol table if found, returns -1 if not
    //present in undefined symbol table.
    //-------------------------------------------------------

    int labelIndex = -1;
    cutString(label);

    int i;
    for (i = 0; i < kMAX_LINES && labelIndex == -1; i++) {
        if(errorTable[i] != NULL && strcmp(label, errorTable[i]) == 0) {
            labelIndex = i; //Set to true
        }
    }

    return labelIndex;
}//getUndefinedLabelIndex

void handleUndefinedSymbols() {
    //-------------------------------------------------------
    //Preconditions: machineCodeHolder used to hold machine
    //code representation.
    //Postconditions: Swaps in UUUUUUU for undefined symbols
    //in machine code. Removed undefined symbols from symbol
    //table.
    //-------------------------------------------------------

    int i;
    for (i = 0; i < machineCodeIndex; i++) {
        if (secondPassLabelsForLine[i] != -1) {
```

```
            if(symbolTableAddresses[secondPassLabelsForLine[i]] == NULL) {
                //ERROR: Undefined Symbol
                strcat(machineCodeHolder[i], "UUUUUUU");
                //Error occurred.
                ERROR = 1;

                int index = getUndefinedLabelIndex(symbolTable
                    [secondPassLabelsForLine[i]]);

                if (index == -1) {
                    errorTable[errorTableIndex] = malloc(32);
                    strcpy(errorTable[errorTableIndex], symbolTable
                        [secondPassLabelsForLine[i]]);
                    //Get the index now that the symbol is in the table.
                    index = getUndefinedLabelIndex(symbolTable
                        [secondPassLabelsForLine[i]]);
                } else {
                    //Already in undefined symbol table.
                }

                printf("ERROR: Undefined symbol %s. Line %d\n", errorTable[index]
                    , i);

                addLineNumberForUndefinedSymbol(index, i);

                secondPassLabelsForLine[i] = -1;
                errorTableIndex++;
            }
        }
    }

    int j;
    for (i = 0; i < machineCodeIndex; i++) {
        for (j = 0; j < machineCodeIndex; j++) {
            if (symbolTable[j] != NULL && errorTable[i] != NULL && strcmp
                (symbolTable[j], errorTable[i]) == 0) {
                symbolTable[j] = NULL;
            }
        }
    }

}//handleUndefinedSymbols

void addLineNumberForUndefinedSymbol(int symbolIndex, int lineNumber) {
    //-------------------------------------------------------
    //Preconditions: symbolIndex and lineNumber to add passed
    //as parameter.
    //Postconditions: Adds the lineNumber referencing the
    //undefined symbol to the array containing the line
    //references for that symbol (symbolIndex).
    //-------------------------------------------------------

    int inserted = 0;
    int i;
    for (i = 0; i < kMAX_ERRORS && inserted == 0; i++) {
        if (errorTableLineNumbers[symbolIndex][i] == -1) {
```

```
            errorTableLineNumbers[symbolIndex][i] = lineNumber;
            //secondPassLabelsForLine[i] = -1;
            inserted = 1;
        }
    }
}//addLineNumberForUndefinedSymbol


void printUndefinedSymbolTable() {
    //-----------------------------------------------------
    //Preconditions: errorTable array, global variable
    //Postconditions: Each index of errorTable that is not
    //NULL is displayed along with each line referencing
    //the symbol contained at that index of the table.
    //-----------------------------------------------------

    printf("\n-------UNDEFINED TABLE--------\n");
    printf("|   LABEL   |  LINE NUMBERS  |\n");
    printf("-----------------------------\n");

    int i;
    for (i = 0; i < kMAX_LINES; i++) {
        if (errorTable[i] != NULL) {
            printf("| %6s    | ", errorTable[i]);
            int j = 0;
            while (errorTableLineNumbers[i][j] != -1) {
                printf("%2d ", errorTableLineNumbers[i][j]);
                j++;
            }
            printf("|\n");
        }
    }
    printf("-----------------------------\n");
}//printSymbolTable

/*---------------Multiply Defined Symbol Table-------------*/
int getMultipleDefinedLabelIndex(char* label) {
    //-----------------------------------------------------
    //Preconditions: string label passed as parameter.
    //Postconditions: Returns the index of the label passed
    //in the Multiple Defined Table if present, -1 if not
    //found.
    //-----------------------------------------------------

    int labelIndex = -1;
    cutString(label);

    int i;
    for (i = 0; i < kMAX_LINES && labelIndex == -1; i++) {
        if(multiplyDefinedTable[i] != NULL && strcmp(label, multiplyDefinedTable
            [i]) == 0) {
            labelIndex = i; //Set to true
        }
    }

    return labelIndex;
```

```
}

void addMultipleDefinedSymbol(char *label) {
    //-------------------------------------------------------
    //Preconditions: char* label passed as parameter.
    //Postconditions: label is added to the Multiple Defined
    //Table and the index for the table is incremented.
    //-------------------------------------------------------

    cutString(label);
    int labelIndex = getMultipleDefinedLabelIndex(label);

    if (labelIndex == -1) {
        //Add the label to the symbol table.
        multiplyDefinedTable[multiplyDefinedTableIndex] = malloc(32);
        strcpy(multiplyDefinedTable[multiplyDefinedTableIndex], label);
        multiplyDefinedTableIndex++;
    }

}

void handleMultipleDefinedSymbols() {
    //-------------------------------------------------------
    //Preconditions: Machine code holder.
    //Postconditions: Adds MMMMMMMMMM to lines of the machine
    //code holder where multiply defined symbols are
    //referenced. Removes multiply defined symbols from the
    //symbol table.
    //-------------------------------------------------------

    int i;
    for (i = 0; i < machineCodeIndex; i++) {
        int index = -1;
        if (secondPassLabelsForLine[i] != -1 && symbolTable
            [secondPassLabelsForLine[i]] != NULL) {
            index = getMultipleDefinedLabelIndex(symbolTable
                [secondPassLabelsForLine[i]]);
        }

        if (secondPassLabelsForMultipleLine[i] != -1 && symbolTable
            [secondPassLabelsForMultipleLine[i]] != NULL) {
            index = getMultipleDefinedLabelIndex(symbolTable
                [secondPassLabelsForMultipleLine[i]]);
        }

        if (index != -1) {
            //ERROR: Undefined Symbol
            strcpy(machineCodeHolder[i], "MMMMMMMMMMM");
            //Error occurred.
            ERROR = 1;
            printf("ERROR: Multiple Defined symbol %s. Line %d\n",
                multiplyDefinedTable[index], i);
            addLineNumberForMultipleDefinedSymbol(index, i);
            secondPassLabelsForLine[i] = -1;
        }
    }
```

```
    int j;
    for (i = 0; i < machineCodeIndex; i++) {
        for (j = 0; j < machineCodeIndex; j++) {
            if (symbolTable[j] != NULL && multiplyDefinedTable[i] != NULL &&
                strcmp(symbolTable[j], multiplyDefinedTable[i]) == 0) {
                symbolTable[j] = NULL;
            }
        }
    }

}

void addLineNumberForMultipleDefinedSymbol(int symbolIndex, int lineNumber) {
    //-------------------------------------------------------
    //Preconditions: symbolIndex and lineNumber passed as
    //parameters.
    //Postconditions: adds the lineNumber referencing the
    //symbol at symbol index to the array that holds the
    //lines referenced by multiply defined labels at that
    //index.
    //-------------------------------------------------------

    int inserted = 0;
    int i;

    printf("Inserting linenumber %d in index %d\n", lineNumber, symbolIndex);
    for (i = 0; i < kMAX_ERRORS && inserted == 0; i++) {
        if (multiplyDefinedTableLineNumbers[symbolIndex][i] == -1) {
            multiplyDefinedTableLineNumbers[symbolIndex][i] = lineNumber;
            inserted = 1;
        }
    }
}

void printMultipleDefinedSymbolTable() {
    //-------------------------------------------------------
    //Preconditions: multiplyDefinedTable containing the
    //characters that are multiply defined in the program.
    //Postconditions: Prints each index of the multiply
    //defined table that is not null along with the line
    //numbers that reference that label.
    //-------------------------------------------------------

    printf("\n-------MULTIPLY DEFINED--------\n");
    printf("|   LABEL   |  LINES REFERENCED  |\n");
    printf("-----------------------------\n");

    int i;
    for (i = 0; i < kMAX_LINES; i++) {
        if (multiplyDefinedTable[i] != NULL) {
            printf("| %6s    | ", multiplyDefinedTable[i]);
            int j = 0;
            while (multiplyDefinedTableLineNumbers[i][j] != -1) {
                printf("%2d ", multiplyDefinedTableLineNumbers[i][j]);
                j++;
```

```
            }
            printf("|\n");
        }
    }
    printf("-----------------------------\n");
}

/*--------------------Conversions---------------------*/
char* decimalBinaryConversion(int convertInt, int length) {
    //-----------------------------------------------------
    //Preconditions: Decimal integer to convert along with
    //the length of the binary string to return.
    //Postconditions: integer is converted to a binary
    //representation as a char* and padded with 0s to match
    //the length passed. String is returned.
    //-----------------------------------------------------

    int c;
    int d;
    int count = 0;

    char *binString;

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

    for (c = length - 1; c >= 0 ; c--)
    {
        d = convertInt >> c;

        if (d & 1)
            *(binString+count) = 1 + '0';
        else
            *(binString+count) = 0 + '0';

        count++;
    }
    *(binString+count) = '\0';

    return  binString;
}//decimalBinaryConversion

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

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);

    strcat(padded, binString);

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

```
    return padded;
}//padBinaryToLength

void cutComments(char *assemblyLine) {
    //-------------------------------------------------------
    //Preconditions: string containing comment passed as
    //parameter.
    //Postconditions: string is cut off where comment occurs.
    //-------------------------------------------------------

    long length = strlen(assemblyLine);
    int i;

    for (i = 0; i < length; i++) {
        if (assemblyLine[i] == '/') {
            assemblyLine[i] = '\0';
            break;
        }
    }
}//cutComments
```