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

/*
System's Programming Phase 3 Algorithm

Author: Jesus M. Morales
Due Date: 04/13/2018

Remarks:
*Updated conditions to accept the exact amout of parameters on the main menu.
*Updated Pass1 to print Memory Addresses as HEX numbers instead of Decimals.
*Moved several variables from Pass1 to global variables to be able to use them
in Pass2 and vice-versa.
*Updated TOKEN structure to be able to use the same structure to tokenize the
source file from Pass1 and intermediate file from Pass2.

*/

typedef struct
{
    char label[10];
    int memoryAddress;
}LABELS;

typedef struct
{
    int memoryAddress;
    char *label;
    char *mnemonic;
    char *operand;
    int errorCode;
} TOKEN;

typedef struct
{
    char mnemonic[5];
    char opcode[5];
} OPCODE;

void breakupLine(char *input, char *command, char *param1, char *param2, int
  *numParams);
int searchLabelLocation(char *inputLabel);
void printError(char **messageOutput, int errorCode);
void loadFile(char *fileName);
void executeFile();
void debugFile();
void dumpFile();
void helpFile();
void assembleFile();
void passOne(char * fileName);
void passTwo(char * fileName);

LABELS labelStructure[500];
OPCODE opcodeStructure[] = { { "ADD", "18" },{ "AND","58" },{ "COMP", "28" },
  { "DIV", "24" },
{ "J", "3C" },{ "JEQ", "30" },{ "JGT", "34" },{ "JLT", "38" },
{ "JSUB", "48" },{ "LDA", "00" },{ "LDCH", "50" },{ "LDL", "08" },
{ "LDX", "04" },{ "MUL", "20" },{ "OR", "44" },{ "RD", "D8" },
{ "RSUB", "4C" },{ "STA", "0C" },{ "STCH", "54" },{ "STL", "14" },
{ "STX", "10" },{ "SUB", "1C" },{ "TD", "E0" },{ "TIX", "2C" },{ "WD", "DC" } };

int programLenght;
int errorFound;
int numberOfLabels;
int numMnemonics = 25;

int main(void)
{
    char input[50];
    char command[50];
    char param1[50];
    char param2[50];

    printf("Hello welcome to Jesus Morales Personal SIC Machine\n \n");
    while (1)
    {
        int numParams = 0;
        int len = 0;
        printf("Command ----> ");
        fgets(input, 50, stdin);

        len = strlen(input) - 1;
        if (input[len] == '\n')
        {
            input[len] = '\0';
        }

        breakupLine(input, command, param1, param2, &numParams);
        numParams--;

        if (strcmp(command, "load") == 0)
        {
            if (param2[0] != '\0')
            {
                printf("LOAD only requieres one parameter\n");
            }
            else if (param1[0] == '\0')
            {
                printf("LOAD requieres one parameter\n");
            }
            else if (param2[0] == '\0')
            {
                loadFile(param1);
            }
        }
        else if (strcmp(command, "execute") == 0)
        {
            if (param2[0] != '\0' || param1[0] != '\0')
            {
                printf("EXECUTE doesn't need any parameters\n");
            }
            else
            {
                executeFile();
            }
        }
        else if (strcmp(command, "debug") == 0)
        {
            if (param2[0] != '\0' || param1[0] != '\0')
            {
                printf("DEBUG doesn't need any parameters\n");
            }
            else
            {
                debugFile();
            }
        }
        else if (strcmp(command, "dump") == 0)
        {
            if (param1[0] == '\0' || param2[0] == '\0')
            {
                printf("DUMP needs two parameter only\n");
            }
            else if (numParams > 3)
            {
                printf("DUMP needs two parameter only\n");
            }
            else
            {
                dumpFile();
            }
        }
        else if (strcmp(command, "help") == 0)
        {
```

```c
            if (param1[0] != '\0')
            {
                printf("HELP does not need parameters\n");
            }
            else
            {
                helpFile();
            }
        }
        else if (strcmp(command, "assemble") == 0)
        {
            if (param2[0] != '\0')
            {
                printf("ASSEMBLE needs a file name \n");
            }
            else if (param1[0] == '\0')
            {
                printf("ASSEMBLE needs only one file name\n");
            }
            else if (param2[0] == '\0')
            {
                assembleFile();
            }
        }
        else if (strcmp(command, "dir") == 0)
        {
            if (param2[0] != '\0' || param1[0] != '\0')
            {
                printf("DIRECTORY doesn't need any parameters \n");
            }
            else
            {
                system("ls");
            }
        }
        else if (strcmp(command, "exit") == 0)
        {
            break;
        }
        else
        {
            printf("Invalid Command , for any help type 'help' to display the 
              command list. \n \n");
        }

        numParams = 0;
    }
    return 0;
}
```

```c

void breakupLine(char *input, char *command, char *param1, char *param2, int 
  *numParams)
{
    command[0] = param1[0] = param2[0] = '\0';
    *numParams = sscanf(input, "%s %s %s %*s", command, param1, param2);
}
int searchLabelLocation(char *inputLabel)
{
    char input[100];
    char *tokenizer;

    int memoryLocation;

    FILE *symbol_table = fopen("symbolTable.txt", "r");

    while (fgets(input, 100, symbol_table))
    {
        input[strcspn(input, "\n")] = '\0';
        tokenizer = strtok(input, "\t");
        memoryLocation = (int)strtol(tokenizer, NULL, 16);
        tokenizer = strtok(NULL, "\t");
        if (strcmp(tokenizer, inputLabel) == 0)
        {
            fclose(symbol_table);
            return memoryLocation;
        }
    }
    fclose(symbol_table);
        return 00000;

}
void printError(char** messageOutput,int errorCode)
{
    if (errorCode == 1)
    {
        strcpy(*messageOutput, "\t ** DUPLICATE LABEL ** ");
    }
    else if(errorCode == 2)
    {
        strcpy(*messageOutput, "\t ** ILLEGAL LABEL ** ");
    }
    else if (errorCode == 3)
    {
        strcpy(*messageOutput, "\t ** ILLEGAL OPERATION ** ");
    }
    else if (errorCode == 4)
    {
        strcpy(*messageOutput, "\t ** ILLEGAL DATA STORAGE DIRECTIVE ** ");
    }
    else if (errorCode == 5)
    {
        strcpy(*messageOutput, "\t ** MISSING START DIRECTIVE ** ");

    }
    else if (errorCode == 6)
    {
        strcpy(*messageOutput, "\t ** MISSING END DIRECTIVE ** ");
    }
    else if (errorCode == 7)
    {
        strcpy(*messageOutput, "\t **TOO MANY SYMBOLS ** ");
    }
    else if (errorCode == 8)
    {
        strcpy(*messageOutput, "\t ** PROGRAM TOO LONG ** ");
    }



}
void loadFile(char *param1)
{
    printf("Loading file:  %s\n", param1);
    passOne(param1);
    passTwo(param1);
    printf("The Programg lenght of this file is:   %d Bytes\n\n",
      programLenght);
    programLenght = 0;

}
void executeFile()
{
    printf(" is not yet avaibalbe.\n");
}
void debugFile()
{
    printf("debug is not avaialabe.\n");
}
void dumpFile()
{
    printf("dump is not avaiblable.\n");
}
void helpFile()
{
    printf("\n");
    printf("\tWelcome to the Help menu. \n");
    printf("\tCommands are the following: \n \n");
    printf("\tload [file_name]\n");
    printf("\texecute \n");
    printf("\tdebug \n");
    printf("\tdump [start] [end] \n");
    printf("\thelp \n");
    printf("\tassemble [file_name] \n");
    printf("\tdirectory \n");
    printf("\texit \n\n");
    printf("\t**ALL COMMANDS ARE CASE SENSITIVE.**\n\n");
}
void assembleFile()
{
    printf("assemble not avaibalbe. \n");
}
void passOne(char *param1)
{
    char input[500];
    char *tokenizer = input;

    char *startingLoct;
    int start = 0;
    int locctr = 0;
    int memLenght = 0;

    int index = 0;

    int labelPresentFlag = 0;
    int duplicateLabelFlag = 0;
    int illegalLabelFlag = 0;
    int illegalOperationFlag = 0;
    int missingDataDirectiveFlag = 0;
    int missingStartFlag = 0;
    int missingEndFlag = 0;
    int tooManyLabelsFlag = 0;
    int programTooLongFlag = 0;
    int errorCode = 0;

    FILE *source_file, *symbol_file, *intermediate_file, *opcode_file;
    TOKEN sourceFileTokenizer;


    source_file = fopen(param1, "r");
    intermediate_file = fopen("intermediate.txt", "w");
    symbol_file = fopen("symbolTable.txt", "w");

    if (source_file == NULL)
    {
        printf("Error openning file does not exist: %s\n", param1);
        return;
    }

    sourceFileTokenizer.label = (char *)malloc(6);
    sourceFileTokenizer.mnemonic = (char *)malloc(6);
    sourceFileTokenizer.operand = (char *)malloc(6);

    errorFound = 0;
    numberOfLabels = 0;
    while (fgets(input, 500, source_file))
    {
        labelPresentFlag = 0;
        duplicateLabelFlag = 0;
        illegalLabelFlag = 0;
        illegalOperationFlag = 0;
        missingDataDirectiveFlag = 0;
        missingStartFlag = 0;
        missingEndFlag = 0;
        tooManyLabelsFlag = 0;
        programTooLongFlag = 0;
        errorCode = 0;
        memLenght = 0;

        /*  Check if label is present in the string line    */
        if (input[0] == ' ' || input[0] == '\t')
        {
            labelPresentFlag = 0;
        }
        else
        {
            labelPresentFlag = 1;
        }

        /*  Check if comment is present in the string line  */
        if (input[0] == '.')
        {
            continue;
        }

        /*  Tokenize the input string   */
        tokenizer = strtok(input, " \t\r\n\v\f");

        /*  Remove of the trailing newLine at the end of the string */
        int counter = 0;
        while (input[counter - 1] != '\n')
        {
            counter++;
        }
        input[counter] = '\0';

        /*  If there is a label */
        if (labelPresentFlag == 1)
        {
            /*  Tokenize the label into the structure   */
            strcpy(sourceFileTokenizer.label, tokenizer);

            /*  Tokenize the mnemonic into the structure    */
            tokenizer = strtok(NULL, " \t\r\n\v\f");
            strcpy(sourceFileTokenizer.mnemonic, tokenizer);

            /*  Tokenize the operand of the mnemonic into the structure */
            tokenizer = strtok(NULL, " \t\r\n\v\f");
            strcpy(sourceFileTokenizer.operand, tokenizer);

            /*  Add the labels to the structure to create a list of existing labels/symbols */
            strcpy(labelStructure[numberOfLabels].label, sourceFileTokenizer.label);
            labelStructure[numberOfLabels].memoryAddress = locctr;

            /*  Check if there are labels in the list   */
            if (numberOfLabels > 0)
            {
                /*  Check if limit of labels has been reached   */
                if (numberOfLabels > 500)
                {
                    tooManyLabelsFlag = 1;
                }

                /*  Inefficiently scan the label/symbol list to check for duplicate labels/symbols     */
                for (int i = 0; i < numberOfLabels; i++)
                {
                    if (strcmp(labelStructure[i].label, sourceFileTokenizer.label) == 0)
                    {
                        duplicateLabelFlag = 1;
                    }
                }
            }

            /*  Check if the label is legal */
            if (!isalpha(sourceFileTokenizer.label[0]))
            {
                illegalLabelFlag = 1;
            }

            /*  Check if we have a START directive in the beginning of the program */
            if (index == 0 && strcmp(sourceFileTokenizer.mnemonic, "START") != 0)
            {
                missingStartFlag = 1;
                locctr = 0;
            }

            /*  Check if we have a END directive in the end of the program  */
            if (missingEndFlag == 1 && errorCode == 0)
            {
                if (strcmp(sourceFileTokenizer.mnemonic, "END") != 0)
                {
                    missingEndFlag = 1;
                }
            }

            /*  If directive START initialize LOCCTR to the starting address */
            if (strcmp(sourceFileTokenizer.mnemonic, "START") == 0) // if start directive initialize locct to the start(convert the string to integer)
            {

                startingLoct = sourceFileTokenizer.operand;
                start = (int)strtol(startingLoct, NULL, 16);
                locctr = start;
            }

            /*  Check if program is too long    */
            if (locctr > 32000)
            {
                programTooLongFlag = 1;
            }
            /* Lenght size in memory from the directives to increment the LOCCTR */
            if (strcmp(sourceFileTokenizer.mnemonic, "WORD") == 0)
            {
                memLenght += 3;
            }
            if (strcmp(sourceFileTokenizer.mnemonic, "RESB") == 0)
            {
                memLenght += (int)strtol(sourceFileTokenizer.operand, NULL, 10);

            }
            if (strcmp(sourceFileTokenizer.mnemonic, "RESW") == 0)
            {
                memLenght += 3 * (int)strtol(sourceFileTokenizer.operand, NULL, 10);
            }

            if (strcmp(sourceFileTokenizer.mnemonic, "BYTE") == 0)
            {
                /*  Check if operand is set to read a string (C) or a hexadecimal (X) */
                if (sourceFileTokenizer.operand[0] == 'C')
                {

                    int bufferSpace = 0;
                    int counter = 2;
                    while (sourceFileTokenizer.operand[counter] != '\'' && bufferSpace < 30)
                    {
                        bufferSpace++;
                        counter++;
                    }

                    memLenght += bufferSpace;

                }
                else if (sourceFileTokenizer.operand[0] == 'X')
                {
                    char hexInput[10];
                    int bufferSpace = 0;
                    int counter = 3;
                    while (sourceFileTokenizer.operand[counter] != '\'' && bufferSpace < 10)
                    {
                        hexInput[bufferSpace] = sourceFileTokenizer.operand[counter];
                        bufferSpace++;
                        counter++;
                    }

                    memLenght = (int)strtol(hexInput, NULL, 10);
                }

                /* Check for errors in the input for the BYTE directive */
                else
                {
                    illegalOperationFlag = 1;
                }

                if (sourceFileTokenizer.operand[1] != '\'' || sourceFileTokenizer.operand[strlen(sourceFileTokenizer.operand) - 1] != '\'')
                {
                    missingDataDirectiveFlag = 1;
                }
            }

            /* Error Flag conditions    */
            if (duplicateLabelFlag == 1 && errorCode == 0)
            {
                errorCode = 1;
                errorFound = 1;
            }
            else if (illegalLabelFlag == 1 && errorCode == 0)
            {
                errorCode = 2;
                errorFound = 1;
            }
            else if (illegalOperationFlag == 1 && errorCode == 0)
            {
                errorCode = 3;
                errorFound = 1;
            }
            else if (missingDataDirectiveFlag == 1 && errorCode == 0)
            {
                errorCode = 4;
                errorFound = 1;
            }
            else if (missingStartFlag == 1 && errorCode == 0)
            {
                errorCode = 5;
                errorFound = 1;
            }
            else if (missingEndFlag == 1 && errorCode == 0)
            {
                errorCode = 6;
                errorFound = 1;
            }
            else if (tooManyLabelsFlag == 1 && errorCode == 0)
            {
                errorCode = 7;
                errorFound = 1;
            }
            else if (programTooLongFlag == 1 && errorCode == 0)
            {
                errorCode = 8;
                errorFound = 1;
            }

            /*Print to the intermediate file and symbol file */
            fprintf(intermediate_file, "%X\t%s\t%s\t%s\t%d\n", locctr, 
              sourceFileTokenizer.label, sourceFileTokenizer.mnemonic, 
              sourceFileTokenizer.operand, errorCode);
            fprintf(symbol_file, "%X\t%s\n", locctr, sourceFileTokenizer.label);
```

```

            /* Search for the mnemonic in the operand table and add 3 to it */
            for (int i = 0; i < numMnemonics; i++)
            {
                if (strcmp(opcodeStructure[i].mnemonic, 
                  sourceFileTokenizer.mnemonic) == 0)
                {
                    locctr += 3;
                }
            }

            /*Update the memory locations after LOCCTR is printed in the file 
              */
            if (strcmp(sourceFileTokenizer.mnemonic, "BYTE") == 0 || strcmp 
              (sourceFileTokenizer.mnemonic, "RESB") == 0 || strcmp 
              (sourceFileTokenizer.mnemonic, "RESW") == 0 || strcmp 
              (sourceFileTokenizer.mnemonic, "WORD") == 0)
            {
                locctr += memLenght;
            }

            /*  Increment the number of labels in the system    */
            numberOfLabels++;
        }

        /*If there is no label in the input line do the same as above but 
          without labels */
        else
        {
            /*  Tokenize the mnemonic into the structure    */
            strcpy(sourceFileTokenizer.mnemonic, tokenizer);

            if (strcmp(sourceFileTokenizer.mnemonic, "RSUB") != 0)
            {
                tokenizer = strtok(NULL, " \t\r\n\v\f");
                strcpy(sourceFileTokenizer.operand, tokenizer);

            }
            else
            {
                strcpy(sourceFileTokenizer.label, " ");
                strcpy(sourceFileTokenizer.operand, " ");
            }


            /*  Tokenize the operand into the structure */


            /*  Check if we have a START directive in the beginning of the 
```

```c
              program */
            if (index == 0 && strcmp(sourceFileTokenizer.mnemonic, "START") != 0)
            {
                missingStartFlag = 1;
                locctr = 0;
            }

            /*  Check if we have a END directive in the end of the program  */
            if (missingEndFlag == 1 && errorCode == 0)
            {
                if (strcmp(sourceFileTokenizer.mnemonic, "END") != 0)
                {
                    missingEndFlag = 1;
                }
            }

            /*  If directive START initialize LOCCTR to the starting address */
            if (strcmp(sourceFileTokenizer.mnemonic, "START") == 0) // if start directive initialize locct to the start(convert the string to integer)
            {
                start = atoi(sourceFileTokenizer.operand);
                locctr = start;
            }

            /*  Check if program is too long    */
            if (locctr > 32000)
            {
                programTooLongFlag = 1;
            }

            /* Lenght size in memory from the directives to increment the LOCCTR */
            if (strcmp(sourceFileTokenizer.mnemonic, "WORD") == 0)
            {
                memLenght += 3;
            }
            if (strcmp(sourceFileTokenizer.mnemonic, "RESB") == 0)
            {
                memLenght += (int)strtol(sourceFileTokenizer.operand, NULL, 10);

            }
            if (strcmp(sourceFileTokenizer.mnemonic, "RESW") == 0)
            {
                memLenght += 3 * (int)strtol(sourceFileTokenizer.operand, NULL, 10);
            }

            if (strcmp(sourceFileTokenizer.mnemonic, "BYTE") == 0)
            {
                /*  Check if operand is set to read a string (C) or a hexadecimal (X) */
                if (sourceFileTokenizer.operand[0] == 'C')
                {
                    int bufferSpace = 0;
                    int counter = 2;
                    while (sourceFileTokenizer.operand[counter] != '\'' && bufferSpace < 30)
                    {
                        bufferSpace++;
                        counter++;
                    }
                    memLenght += bufferSpace;

                }
                else if (sourceFileTokenizer.operand[0] == 'X')
                {
                    char hexInput[16];
                    int bufferSpace = 0;
                    int counter = 3;
                    while (sourceFileTokenizer.operand[counter] != '\'' && bufferSpace < 16)
                    {
                        hexInput[bufferSpace] = sourceFileTokenizer.operand[counter];
                        bufferSpace++;
                        counter++;
                    }

                    memLenght = (int)strtol(hexInput, NULL, 10);
                }

                /* Check for errors in the input for the BYTE directive */
                else
                {
                    illegalOperationFlag = 1;
                }

                if (sourceFileTokenizer.operand[1] != '\'' || sourceFileTokenizer.operand[strlen(sourceFileTokenizer.operand) - 1] != '\'')
                {
                    missingDataDirectiveFlag = 1;
                }
            }

```

```c
            /* Error Flag conditions    */
            if (duplicateLabelFlag == 1 && errorCode == 0)
            {
                errorCode = 1;
                errorFound = 1;
            }
            else if (illegalLabelFlag == 1 && errorCode == 0)
            {
                errorCode = 2;
                errorFound = 1;
            }
            else if (illegalOperationFlag == 1 && errorCode == 0)
            {
                errorCode = 3;
                errorFound = 1;
            }
            else if (missingDataDirectiveFlag == 1 && errorCode == 0)
            {
                errorCode = 4;
                errorFound = 1;
            }
            else if (missingStartFlag == 1 && errorCode == 0)
            {
                errorCode = 5;
                errorFound = 1;
            }
            else if (missingEndFlag == 1 && errorCode == 0)
            {
                errorCode = 6;
                errorFound = 1;
            }
            else if (tooManyLabelsFlag == 1 && errorCode == 0)
            {
                errorCode = 7;
                errorFound = 1;
            }
            else if (programTooLongFlag == 1 && errorCode == 0)
            {
                errorCode = 8;
                errorFound = 1;
            }

            /*Print to the intermediate file and symbol file */
            fprintf(intermediate_file, "%X\t\t\t%s\t%s\t%d\n", locctr,
              sourceFileTokenizer.mnemonic, sourceFileTokenizer.operand,
              errorCode);

            /* Search for the mnemonic in the operand table and add 3 to it */
            for (int i = 0; i < numMnemonics; i++)
```

```c
                {
                    if (strcmp(opcodeStructure[i].mnemonic, 
                      sourceFileTokenizer.mnemonic) == 0)
                    {
                        locctr += 3;
                    }
                }

                /*Update the memory locations after LOCCTR is printed in the file 
                  */
                if (strcmp(sourceFileTokenizer.mnemonic, "BYTE") == 0 || strcmp 
                  (sourceFileTokenizer.mnemonic, "RESB") == 0 || strcmp 
                  (sourceFileTokenizer.mnemonic, "RESW") == 0 || strcmp 
                  (sourceFileTokenizer.mnemonic, "WORD") == 0)
                {
                    locctr += memLenght;
                }
            }
            index++;
        }

        programLenght = locctr - start;
        programLenght = programLenght - 4;

    printf("Pass One complete successfully. \n");

    fprintf(intermediate_file, "\n\n\t Printing Error Code List: \n\n");
    fprintf(intermediate_file, "*==================================================* 
      \n");
    fprintf(intermediate_file, "\tNo Error = 0\n");
    fprintf(intermediate_file, "\tDuplicate Label = 1\n");
    fprintf(intermediate_file, "\tIllegal Label = 2\n");
    fprintf(intermediate_file, "\tIllegal Operation = 3\n");
    fprintf(intermediate_file, "\tIllegal Data Storage Directive = 4\n");
    fprintf(intermediate_file, "\tMissing START Directive = 5\n");
    fprintf(intermediate_file, "\tMissing END Directive = 6\n");
    fprintf(intermediate_file, "\tToo Many Symbols = 7\n");
    fprintf(intermediate_file, "\tProgram Too Long = 8\n");
    fprintf(intermediate_file, "*==================================================* 
      \n");

    fclose(intermediate_file);
    fclose(source_file);
    fclose(symbol_file);
    }
    void passTwo(char *param1)
    {
        char input[100];
        char sourceInput[500];

        char *tokenizer;
        char *objectCode_string;
        char *errorMessage;

        int startingAddress;
        int operandAddress;
        int objectCode_decimal;
        int objectLineLenght = 0;

        int newLineFlag = 1;
        int labelPresentFlag = 0;

        FILE *intermediateFile, *symbolTable, *objectFile, *listingFile, 
          *sourceFile;
        TOKEN intermediateFileTokenizer;

        objectFile = fopen("objectFile.txt", "w");
        listingFile = fopen("listingFile.txt", "w");
        intermediateFile = fopen("intermediate.txt", "r");
        symbolTable = fopen("symbolTable.txt", "r");
        sourceFile = fopen(param1,"r");

        if (sourceFile == NULL)
        {
            printf("Intermediate file did not opened correctly \n");
            return;
        }

        intermediateFileTokenizer.label = (char *)malloc(6);
        intermediateFileTokenizer.mnemonic = (char *)malloc(6);
        intermediateFileTokenizer.operand = (char *)malloc(6);
        errorMessage = (char *)malloc(256);

        while (fgets(input, 100, intermediateFile))
        {
            memset(intermediateFileTokenizer.label, '\0', 6);
            memset(intermediateFileTokenizer.mnemonic, '\0', 6);
            memset(intermediateFileTokenizer.operand, '\0', 6);
            memset(errorMessage, '\0', 256);

            fgets(sourceInput, 500, sourceFile);

            /*  Check it the source line is a comment   */
            if (sourceInput[0] == '.')
            {
                while (sourceInput[0] == '.')
                {
                    fprintf(listingFile, "%s", sourceInput);
                    fgets(sourceInput, 500, sourceFile);
                }
            }

            labelPresentFlag = 0;

            tokenizer = strtok(input, "\t");
            intermediateFileTokenizer.memoryAddress = (int)strtol(tokenizer, NULL, 16);   ////save address
            tokenizer = strtok(NULL, "\t");

            for (int i = 0; i < numberOfLabels; i++)
            {
                if (strcmp(labelStructure[i].label, tokenizer) == 0)
                {
                    labelPresentFlag = 1;
                    break;
                }
            }

            if (labelPresentFlag == 1)
            {
                strcpy(intermediateFileTokenizer.label, tokenizer); ////save label
                tokenizer = strtok(NULL, "\t");
            }

            strcpy(intermediateFileTokenizer.mnemonic, tokenizer); ////save mnemonic

            if (strcmp(intermediateFileTokenizer.mnemonic, "RSUB") != 0)
            {
                tokenizer = strtok(NULL, "\t");
                strcpy(intermediateFileTokenizer.operand, tokenizer); /////save operand
                tokenizer = strtok(NULL, " \t");
                intermediateFileTokenizer.errorCode =(int)strtol(tokenizer,NULL,10); ////save errorcode
            }
            else
            {
                tokenizer = strtok(NULL, " \t");
                intermediateFileTokenizer.errorCode = (int)strtol(tokenizer, NULL, 10); ////save errorcode
                objectCode_string = "4C0000";

                if (newLineFlag == 1)
                {
                    fprintf(objectFile, "\n");
                    fprintf(objectFile, "T%s", objectCode_string);
                    newLineFlag = 0;
```

```c
                objectLineLenght++;
            }
            else
            {
                fprintf(objectFile, "%s", objectCode_string);
                objectLineLenght++;
            }

            if (intermediateFileTokenizer.errorCode == 0)
            {
                fprintf(listingFile, "%X\t%s\t%s",
                  intermediateFileTokenizer.memoryAddress,
                  objectCode_string,sourceInput);
                continue;
            }
            else
            {
                printError(&errorMessage, intermediateFileTokenizer.errorCode);
                fprintf(listingFile, "%s\n", errorMessage);
                continue;
            }
        }

        /*  Check if the object file size limit has been reached    */
        if (objectLineLenght == 10)
        {
            newLineFlag = 1;
            objectLineLenght = 0;
        }

        /*  Check it the intermidiate line is a START   */
        if (strcmp(intermediateFileTokenizer.mnemonic, "START") == 0 ||
          intermediateFileTokenizer.errorCode == 5)
        {
            fprintf(objectFile,"H%_%s%06X%06X", intermediateFileTokenizer.label,
               intermediateFileTokenizer.memoryAddress, programLenght);
            if (intermediateFileTokenizer.errorCode == 0)
            {
                fprintf(listingFile, "%X\t\t%s",
                  intermediateFileTokenizer.memoryAddress,sourceInput);
            }
            else
            {
                printError(&errorMessage,intermediateFileTokenizer.errorCode);
                fprintf(listingFile, "%s\n", errorMessage);
            }
            startingAddress = intermediateFileTokenizer.memoryAddress;
        }

        /*  Check it the intermidiate line is a RESW    */
        else if (strcmp(intermediateFileTokenizer.mnemonic, "RESW") == 0 ||
          strcmp(intermediateFileTokenizer.mnemonic, "RESB") == 0 ||
          intermediateFileTokenizer.errorCode == 4)
        {
            if (intermediateFileTokenizer.errorCode == 0)
            {
                fprintf(listingFile, "%X\t\t%s",
                  intermediateFileTokenizer.memoryAddress,sourceInput);
            }
            else
            {
                printError(&errorMessage, intermediateFileTokenizer.errorCode);
                fprintf(listingFile, "%s\n", errorMessage);
            }
        }

        /*  Check it the intermidiate line is a WORD    */
        else if (strcmp(intermediateFileTokenizer.mnemonic, "WORD") == 0 ||
          intermediateFileTokenizer.errorCode == 4)
        {
            objectCode_string = intermediateFileTokenizer.operand;
            objectCode_decimal = (int)strtol(objectCode_string,NULL,10);
            if (strcmp(intermediateFileTokenizer.operand, "0") == 0)
            {
                fprintf(objectFile, "%06X", objectCode_decimal);
                newLineFlag = 1;
                objectLineLenght = 0;

                if (intermediateFileTokenizer.errorCode == 0)
                {
                    fprintf(listingFile, "%X\t%06X\t%s",
                     intermediateFileTokenizer.memoryAddress,
                     objectCode_decimal,sourceInput);
                    continue;
                }
                else
                {
                    printError(&errorMessage,
                     intermediateFileTokenizer.errorCode);
                    fprintf(listingFile, "%s\n", errorMessage);
                    continue;
                }
            }
            else
            {
                if (intermediateFileTokenizer.errorCode == 0)
                {
                    fprintf(listingFile, "%X\t%06X\t%s",
                         intermediateFileTokenizer.memoryAddress, objectCode_decimal, sourceInput);
                }
                else
                {
                    printError(&errorMessage, intermediateFileTokenizer.errorCode);
                    fprintf(listingFile, "%s\n", errorMessage);
                }
            }

            if (newLineFlag == 1)
            {
                fprintf(objectFile, "\n");
                fprintf(objectFile, "T%06X%06X",intermediateFileTokenizer.memoryAddress, objectCode_decimal);
                newLineFlag = 0;
                objectLineLenght++;
            }
            else
            {
                fprintf(objectFile, "%06X", objectCode_decimal);
                objectLineLenght++;
            }
        }

        /*  Check it the intermidiate line is a BYTE    */
        else if(strcmp(intermediateFileTokenizer.mnemonic, "BYTE") == 0 || intermediateFileTokenizer.errorCode == 3)
        {
            if(intermediateFileTokenizer.operand[0] == 'C')
            {
                char copyHEX[10];
                char convertedHEX[10];

                int inputIndex = 2;
                int outputIndex = 0;

                while (intermediateFileTokenizer.operand[inputIndex] != '\'')
                {
                    copyHEX[outputIndex] = intermediateFileTokenizer.operand[inputIndex];
                    inputIndex++;
                    outputIndex++;
                }

                sprintf(convertedHEX, "%X%X%X", copyHEX[0], copyHEX[1], copyHEX[2]);
```

```c

                    if (intermediateFileTokenizer.errorCode == 0)
                    {
                        fprintf(listingFile, "%X\t%s\t%s",
                        intermediateFileTokenizer.memoryAddress,
                        convertedHEX,sourceInput);
                    }
                    else
                    {
                        printError(&errorMessage,
                        intermediateFileTokenizer.errorCode);
                        fprintf(listingFile, "%s\n", errorMessage);
                    }

                    if (newLineFlag == 1)
                    {
                        fprintf(objectFile, "\n");
                        fprintf(objectFile, "T%06X%s",
                        intermediateFileTokenizer.memoryAddress, convertedHEX);
                        newLineFlag = 0;
                        objectLineLenght++;

                    }
                    else
                    {
                        fprintf(objectFile, "%s", convertedHEX);
                        objectLineLenght++;
                    }
                }
                else if (intermediateFileTokenizer.operand[0] == 'X')
                {
                    char copyHEX[10];

                    int inputIndex = 2;
                    int outputIndex = 0;

                    while (intermediateFileTokenizer.operand[inputIndex] != '\'')
                    {
                        copyHEX[outputIndex] = intermediateFileTokenizer.operand
                        [inputIndex];
                        inputIndex++;
                        outputIndex++;
                    }
                    copyHEX[outputIndex] = '\0';

                    if (intermediateFileTokenizer.errorCode == 0)
                    {
                        fprintf(listingFile, "%X\t%s\t%s",
                        intermediateFileTokenizer.memoryAddress,
                          copyHEX,sourceInput);
                    }
                    else
                    {
                        printError(&errorMessage, 
                          intermediateFileTokenizer.errorCode);
                        fprintf(listingFile, "%s\n", errorMessage);
                    }

                    if (newLineFlag == 1)
                    {
                        fprintf(objectFile, "\n");
                        fprintf(objectFile, "T%06X%s", 
                          intermediateFileTokenizer.memoryAddress, copyHEX);
                        newLineFlag = 0;
                        objectLineLenght++;
                    }
                    else
                    {
                        fprintf(objectFile, "%s", copyHEX);
                        objectLineLenght++;
                    }
                }
                else
                {
                    printError(&errorMessage, intermediateFileTokenizer.errorCode);
                    fprintf(listingFile, "%s\n", errorMessage);
                }
            }

            /*  Check it the intermidiate line is a END */
            else if (strcmp(intermediateFileTokenizer.mnemonic, "END") == 0)
            {
                operandAddress = searchLabelLocation
                  (intermediateFileTokenizer.operand);
                fprintf(objectFile,"\n");
                fprintf(objectFile,"E%06X", operandAddress);

                if (intermediateFileTokenizer.errorCode == 0)
                {
                    fprintf(listingFile, "%s", sourceInput);
                }
                else
                {
                    printError(&errorMessage, intermediateFileTokenizer.errorCode);
                    fprintf(listingFile, "%s\n", errorMessage);
                }
                break;
            }
```

```c

        /*  Else it is a regular mnemonic and just requieres normal handling */
        else
        {
                char *OpcodeExtracted;
                int OpcodeConverted;
                char objectCode[10];

                operandAddress = searchLabelLocation(intermediateFileTokenizer.operand);

                for (int i = 0; i < numMnemonics; i++)
                {
                    if (strcmp(opcodeStructure[i].mnemonic, intermediateFileTokenizer.mnemonic) == 0)
                    {
                        OpcodeExtracted = opcodeStructure[i].opcode;
                        break;
                    }
                }

                OpcodeConverted = (int)strtol(OpcodeExtracted, NULL, 16);
                sprintf(objectCode, "%02X%04X", OpcodeConverted, operandAddress);

                if (intermediateFileTokenizer.errorCode == 0)
                {
                    fprintf(listingFile, "%X\t%06s\t%s", intermediateFileTokenizer.memoryAddress, objectCode,sourceInput);
                }
                else
                {
                    printError(&errorMessage, intermediateFileTokenizer.errorCode);
                    fprintf(listingFile, "%s\n", errorMessage);
                }

                if (newLineFlag == 1)
                {
                    fprintf(objectFile, "\n");
                    fprintf(objectFile, "T%06X%06s", intermediateFileTokenizer.memoryAddress, objectCode);
                    newLineFlag = 0;
                    objectLineLenght++;
                }
                else
                {
                    fprintf(objectFile, "%06s", objectCode);
                    objectLineLenght++;
                }
        }
    }

    /*  Close all files */
    fclose(objectFile);
    fclose(listingFile);
    fclose(intermediateFile);
    fclose(symbolTable);

    /*  Check if there were errors on Pass 1 if so delete the object file   */
    if (errorFound == 1)
    {
        if (remove("objectFile.txt") == 0)
        {
            printf("Program Has Errors.\n");
        }
    }
    else
    {
        printf("Pass Two complete successfully. \n");
    }
}

```