**summary of a laboratory project in systems programming:**

This project was built by Max Freeman : as well by his project partner Itai Baldev: In the file we will list briefly (at an explanatory level) all the files and actions we built and the structure of the project: The structure of the project in brief:

We divided the project into several parts, in the first part we perform the pre-assembler (where we also have the macro table), where we actually get rid of all the macro definitions, and interpret them into real code. Then we go over the file extracted from the pre-assembler stage, and start building the symbol table. If an error is found, we will continue to go through the entire file to find more errors, and finally stop. It is important to note that we chose not to code at all in this part but only to find errors and build the symbol table. Finally, we will perform another pass on the unzipped file, only this time we have a built-in symbol table, and the assumption that the file is correct (except for errors that can only be found in the second pass). In this part we actually code, and we do this by coding instruction sentences in a separate file (cmd), and by coding instruction sentences in a separate file (data), finally we combine these files into one ob file, which at the beginning has the contents of the cmd file, and at the end the content of the data file. At the same time we write to the ext,ent files, as requested in the course booklet. Documentation of the project files:

**Assembler.c**

The main file in the project, which of course has no .h file and is the one that activates the algorithm written in the booklet. Errors/comments/screen output/final result files will all be output from it only. will also include the following actions:

• **symbolTable createSymbolTable(char\* fileName, flags\* status)**

This function Receives a pointer to the file and performs a first pass of the assembly on it, thereby checking the integrity of the file, and builds the symbol table for the second pass. In addition, it receives a pointer to the structure where we have flags (such as whether an error was found, whether an entry prompt was found, etc.)

**• int encodeAssembly(char\* fileName, symbolTable table, flags\* status)**

This function Receiving a pointer to the file and performing a second pass of the assembly on it, it assumes that the file is correct (except for errors such as an undefined label, since it is the one responsible for finding it), and encodes the file into the output files ob, ent, ext.

**GeneralFunctions.c**

A file that contains all the auxiliary functions that we use in the various files. At the beginning of the file defines of color printing are defined. The functions it has are:

**void getFileWithExtension(char\* fileName, char\* extension, char\* fileWithExtension) •**

This function takes a file name and its extension, and centers it in fileWithExtension. The parameters it receives are:

file\_name - the name of the file

extent - the extension of the file

fileWithExtension - the string to write to

**• FILE\* openFile(char\* file\_name, char\* file\_extension, char\* mode)**

This function opens a file with the requested extension according to the project guidelines The parameters it receives are:

file\_name - the name of the file

file\_extension - the extension of the file

mode - the mode with which the file should be opened Returns a pointer to the opened file

**• void deleteFile(char\* file\_name, char\* file\_extension)**

This function takes a filename and its extension, and deletes the file. The parameters it accepts are:

file\_name - the name of the file

file\_extension - the extension of the file does not return anything

**• void removeSpacesAndTabs(char line[MAX\_LINE\_LENGTH])**

This function deletes the spaces at the beginning and end of a string. The parameter it receives is:

line - the string to be changed (with the maximum size for a line as requested in the assignment) does not return anything

**• void throwError(char\* errorMsg, int numberOfLine)**

This function throws an error and prints it on the screen. The parameters it receives are:

errorMsg - the error message to be printed

numberOfLine - the number of the line where the error occurred does not return anything

**• void throwWarning(char\* warningMsg, int numberOfLine)**

This function throws a warning and prints it on the screen. The parameters it receives are:

warningMsg - The warning message to be printed

numberOfLine - the number of the line where the error occurred does not return anything

**• void reservedWord (char\* word)**

This function checks if the given word is a reserved word in assembly. The parameters it receives are:

word - the word to test

**• int foundEmptySentence (char line[MAX\_LINE\_LENGTH])**

This function checks if a line contains only blank characters (spaces/tabs) The parameter it receives is:

line - the line for testing (with the maximum size for a line as requested in the assignment) Returns 1 if it found a non-empty character, otherwise returns 0

**• int firstCharIsDot(char \*line)**

This function checks if the first character in the line is a period The parameters it receives is:

line - compare to test If it finds a dot at the beginning of the string it returns 1 otherwise it returns 0

**• void cutColonFromLabel(char \*labelName)**

This function deletes the colon at the end of the label The parameter it receives is:

labelName – the name of the label does not return anything

**• int isNumber(char \*number)**

This function checks if a string represents a valid number The parameter it receives is:

number - the string to check If the string represents a valid number then return 1 otherwise return 0

**• int convertToNumber(char\* numberString, int\* number)**

This function converts a given string to an int The parameters it receives are: numberString - the string to convert

number - points to the place we want to convert to If it successfully converted the string it will return 0 otherwise it will return 1

**macroTable.c:**

In this file are all the functions for handling the one-way linked list in which there is information about each macro. The structure of each link in the list is as follows: char \*name int startIndex; int length; struct Mnode\* next; Where name represents the name of the macro, startIndex represents the location of the start of the macro in the file, length represents the length of the macro (bytes), and next points to the next element in the linked list. Its functions and their actions:

**• macroNode createMacroNode(char\* name)**

This function is used to allocate memory and create a new macro block The parameter it receives is:

name - the name of the macro Returns a pointer to the

new empty macro

• **void InsertMacroNode(struct MNode\*\* head, struct MNode\* newNode)**

This function is used to insert a new macro into the macro list The parameters it receives are:

head – a pointer to the current head of the linked list, we need a pointer since we want to change it.

newNode - the new macro added to the list does not return anything

**• struct MNode\* SearchNode(struct MNode\* head, char \*name)**

This function is used to search for a specific macro in a linked list The parameters it receives are:

head - the current head of the linked list

name - the name to search for Returns the found node, otherwise returns NULL if it cannot be found

**• void freeMacroList(struct MNode\* node)**

This function is used to release the macro list from the heap memory The parameter it receives is:

node - the current node in the linked list does not return anything

**• int isValidMacroName (char\* name)**

This function is used to check the correctness of the name of a macro, according to the conditions we received in the assignment. The parameter it receives is:

name – the name to test Returns 1 if the name is correct, 0 otherwise

**• getName (struct MNode\* node),**

**getStartIndex(struct MNode\* node),**

**getLength(struct MNode\* node)**

These functions are responsible for getting the macro fields.

The parameter they receive are:

node – pointer to the struct of a macro, whose fields we want to access. Returns the requested field.

**• setLength (struct MNode\* node, int length)**

**, setStartIndex (struct MNode\* node, int startIndex**)

These functions are responsible for changing the macro fields. The parameter they receive are:

node – pointer to the struct of a macro, whose fields we want to access.

Length/startIndex - this will be the new variable that we will change, depending on the function. They don't return anything

**PreAssembler.c:**

Our pre-assembler file where the macros are separated from the source files into files a.m Its functions:

**• void WriteMacroToOutputFile(macroNode macro, FILE\* inputFile, FILE\* outputFile)**

This function is used to write the contents of a macro to the output file Its parameters are: macro - the macro to be written

- inputFile The source input file

- outputFile The output file am to write to does not return anything

**• int addToMacroList(macroNode\* head, char\* line, FILE\* inputFile)**

This function is used to create a new macro in the linked list, by extracting its name, and its contents from the input file. .

- head The head of the linked list of the macro table

line – the current line in the input file

- inputFile The input source file Return 1 if an error is found, 0 otherwise

**• int preAssemble(char\* file\_name)**

This function is the main one in the pre-assembler stage, and is responsible for creating an am file with expanded macros. Its parameter is:

- file\_name The name of the input file Return 0 if everything went well, and if an error was found return 1.

**RegisterTable.c:**

is a file that contains information, and help functions on handling registers At the beginning we defined a large array, in which all the registers are defined, and their number. In addition, the file contains functions for handling registers:

**• int isRegisterName(char \*name)**

The function checks if the received name is a valid register name or not Its parameter is: name - the name to search for Returns 1 if it found the name in the table otherwise returns 0

**• int getRegisterNum(char \*name)**

The function checks if the received name is a valid register name or not and returns its number Its parameter is: name - the name to search for Returns the register number if it found the name in the table otherwise returns 0

**AddressingMode.c:**

is a file that handles the names of the directives and the different types of addresses. At the beginning it contains an array in which there are all the correct instructions. The functions it has are:

• **int isInstructionName(char \*name)**

This function checks if a given string is the name of a valid directive in the directive table. Its parameter is:

name - the string to check If it found the prompt in the prompt table it will return 1 otherwise it will return 0

**• int getFirstDelimIndex(char \*str, char delim)**

This function returns the index of the first occurrence of the separator we are looking for Its parameters are:

str - the string to search on

delim - the search delimiter Returns the index of the first occurrence of the given separator or 0 if it is not found

**• enum addressingMode getAddressingMode(char \*operand, int numberOfLine)**

This function returns the address type of the given operand Their parameters are:

operand - the operand to get its addressing method

numberOfLine - the current line in the input file Returns the address type of the given operand, if it does not find one, return error(1-)

In addition, in the h file of this file, we have an enum type structure that stores the types of addresses according to the numbers they receive, it looks like this:

**typedef enum addressingMode { immediateAddress = 0, direct address\\ directAddress = 1, Record access address \\ addressAccess = 2, direct register address \\ directRegisterAddress = 3, error \\ error = -1 } addressingMode;**

**SymbolTable.c:**

In this file we handle everything related to symbols in the project. We implemented the symbol table as a one-way linked list of symbols, and each symbol contains its name, its address, and its type**:**

**typedef struct symbol\* symbolTable; /\*\* Next symbol in table \*/ symbolTable next; /\*\* Address of the symbol \*/ int address; /\*\* name (symbol name) is a string (aka char\*) \*/ char \*name; /\*\* Symbol type \*/ symbolType type; };**

Its functions and their actions:

**• int isLabel(char \*name)**

This function checks if the received symbol ends with a colon Its parameter is name – the name of the symbol Returns 1 if a colon is found in the last character of the string otherwise 0

**• int validLabelName(char \*name)**

This function checks the conditions of the symbol as defined in the assignment: if the first character is alpha and all the rest are alphanumeric, and the string is not a reserved word and its length is not longer than 30 characters The parameter of is: name - the string to check Returns 1 if all the above conditions were met otherwise returns 0

**• void setType(symbolTable symbol, symbolType type)**

This function inserts the symbol type into the symbol table Its parameters are: symbol – the symbol to which we update the type

type – the type that symbol will receive does not return anything

**• void setAddress(symbolTable symbol, int address)**

This function inserts the address of the symbol into the symbol table Its parameters are: symbol – the symbol to which we update the address address - the address that symbol will receive does not return anything

**• symbolTable findInTable(symbolTable symbol, char \*name)**

This function searches the symbol table for the symbol of the name it received Its parameters are:

symbol – the current symbol that we are checking (initially it will be the head, and it will change because the function works recursively)

name – the string I compare to the symbol we are looking for If we found the symbol the symbol table will return it otherwise it will return NULL

**• symbolTable createSymbol(char\* name, int address**)

This function creates a null symbol Its parameters are:

name – a string that is the name of the symbol

address - the address that symbol will receive Returns the newly created cell in the symbol table with all its fields.

**• void shiftHead(symbolTable\* head)**

This function moves the head of the linked list one forward, that is, deletes the head. Its parameter is:

head – the cell that needs to be moved does not return anything

**• void InsertSymbolNode(symbolTable\* head, char \*label, int address)**

This function is used to insert a new symbol into the symbol table The parameters it receives are: head - the current head of the linked list

Label - the string of the name of the new symbol

address – the address of the new symbol does not return anything

**• int getAddress(symbolTable symbol)**

This function receives from the symbol table the address of the attached symbol Its parameter is: symbol – the symbol we access the address with Returns the symbol address

**• symbolType getType(symbolTable symbol)**

This function receives from the symbol table the type of the attached symbol Its parameter is: symbol – the symbol we access for typing Returns the symbol type

**• char\* getSymbolName(symbolTable symbol)**

This function receives the name of the symbol from the symbol table Its parameter is:

symbol – the symbol we access there Returns the name of the symbol

**• void updateDataSymbols(symbolTable table, int IC)**

This function updates the instruction counter with the data counter of the symbols according to their type as we were asked in the assignment. Its parameters are:

table – the symbol table whose symbol we want to update

IC – the instruction counter does not return anything.

**• void freeNode(symbolTable node)**

This function frees a memory cell from the symbol table (our linked list) Its parameter is:

node – the memory cell in the linked list that we want to release does not return anything.

**• void freeTable(symbolTable table)**

This function frees from memory the table of symbols we created Its parameter is:

table – the table of symbols we want to release from memory does not return anything.

**AssemblySentence.c:**

This file handles the am file after the macros are retired In this file we will go through the file line by line and analyze it word by word and act accordingly. In addition, at the beginning of the header file we defined a data structure called flags, which contains flags of information for the first pass. It contains an error field, which says if an error was found in the first pass, in addition to a foundExtern field which says if any extern definition was found (by which we know whether to open the file in the second pass), as well as foundEntry. It contains finalIC, finalDC, which are called the final IC and DC, for use in the second pass. In order to do this we used the following functions and data sets:

**• int checkFirstCharacter (char\* line, char c)**

This function checks if the first character in the line received from the file is any character. Its parameters are:

line - the line we are checking

c- the character we are checking Returns 1 if c is the first letter in the string line (after ignoring spaces) otherwise returns 0

**• void fixDataInstruction (char\* line, char parsedLine[MAX\_LINE\_LENGTH \* 2])**

This function is a function that handles the line before doing doData, and it does this by adding spaces after commas, so strtok won't skip adjacent commas.

line - the line that we arrange

parsedLine – the string to which we will write the corrected line, its length is MAX\_LINE\_LENGTH \* 2, since we are adding spaces and we don't want to exceed the line limits. returns nothing

**• int doData(symbolTable\* table, char \* line, int \*DC, int numberOfLine, symbolTable symbol)**

This function updates the data counter (DC) according to the amount of numbers found in the line containing the instruction .data Its parameters are: table – our symbol table

line - a string that contains the complete instruction we received in the line,

DC - our data counter pointer

numberOfLine – the current line number in the input file symbol - a symbol created if received in a row Returns 0 if an error is found in the line we will print the error with the number of the line otherwise 1

**• int doString(symbolTable\* table, char \* line, int \*DC, int numberOfLine, symbolTable symbol)**

This function updates the data counter (DC) according to the number of characters found in the line containing the .string instruction Its parameters are: table – our symbol table

line – a string that contains the complete instruction we received in the line

DC - our data counter integer

numberOfLine – the current line number in the input file symbol - symbol if received in line Returns: if an error is found in the line we will print the error with the line number and return 0, otherwise return 1

**• int doStruct(symbolTable\* table, char \* line, int \*DC, int numberOfLine, symbolTable symbol)**

This function updates the data counter (DC) according firstly to the single number that Hick receives after the word struct. In addition, it will also add to the data counter the amount of characters found in the line containing the instruction. struct Its parameters are:

table – our symbol table

line – a string that contains the complete instruction we received in the line

DC - our data counter integer

numberOfLine – the current line number in the input file symbol - symbol if received in line returns 0 if an error is found print the error and the line number, otherwise return 1

**• int doEntry(symbolTable\* table, char \* line, int \*DC, int numberOfLine, symbolTable symbol)**

This function checks if the symbol received with the prompt is a valid symbol Its parameters are:

table – our symbol table

line – a string that contains the complete instruction we received in the line

DC - our data counter integer

numberOfLine – the current line number in the input file symbol - symbol if received in line Returns 1 if the symbol is valid otherwise prints an error and the line number and returns 0

**• int doExtern(symbolTable\* table, char \* line, int \*DC, int numberOfLine, symbolTable symbol)**

This function checks if the extern directive is according to the instructions of the assignment and operates according to the instructions of the assignment Its parameters are:

table – our symbol table

line – a string that contains the complete instruction we received in the line

DC - our data counter integer

numberOfLine – the current line number in the input file symbol - symbol if received in line returns 1 if the prompt is valid and according to the assignment conditions otherwise prints an error and the line number and returns 0

**• int checkValidOperand(char\* operand , int numberOfLine)**

This function checks if the set operand is valid and the assignment conditions are met Its parameters are:

operand – the received operand string

numberOfLine – the current line number in the input file The rules operand returns 1 and meets the conditions of the assignment, otherwise prints an error and the line number and returns 0

**• int isValidCommandSentence(int operandNum, char\* restOfLine , int numberOfLine, char\* firstOperand, char\* secondOperand)**

This function checks if the instruction received in the row is a valid instruction and meets all the assignment conditions of the project Its parameters are:

operandNum – the number of operands that the command receives

restOfLine – the string that is received after the instruction word

numberOfLine – the current line number in the input file

firstOperand - a string containing the first operand in the instruction

secondOperand - a string containing the second operand in the instruction The function will return 1 if the additional operands of the command are correct, otherwise it will return 0

**• int UpdateICforCommandSentence(char\* command, int operandNum , int\* IC, int numberOfLine, char\* firstOperand, char\* secondOperand)**

This function updates the instruction counter (IC) if the instruction line is valid and the instruction and its operands match the assignment conditions Its parameters are:

Command – a string that contains the complete instruction we received in the line

operandNum – the number of operands received with the instruction IC - Pointer of our instruction counter

numberOfLine – the current line number in the input file

firstOperand - a string containing the first operand in the instruction secondOperand - a string containing the second operand in the instruction Returns 0 if all the conditions and tests were accepted according to the terms of the assignment, otherwise returns 1

**• int doCommandSentence(char \*command, int \*IC,int numberOfLine,symbolTable symbol)**

This function executes the instruction line across the assignment conditions and performs all the necessary tests accordingly Its parameters are:

Command – a string that contains the complete instruction we received in the line

IC - Pointer of our instruction counter

numberOfLine – the current line number in the input file

symbol - symbol if received in line Returns 1 An error was found in the command line otherwise returns 0 (depending on what the function UpdateICforCommandSentence returns)

One of the data structures we have in this file is struct inst which contains the following fields: { const char \*name; int(\*doInstructions)(symbolTable\* table,char \*line, int \*DC, int numberOfLine,symbolTable symbol); This structure is a structure that matches each name of an instruction to the appropriate functions above. (If .data is found, doData is called, etc.)

**• int validInstructions(symbolTable\* table,char \*instruction,int \*DC, int numberOfLine,symbolTable symbol)**

This function checks if the instruction line received in the row is a valid instruction and if so, performs the necessary operations according to the conditions of the assignment to execute an instruction line according to the instruction type Its parameters are:

Table- points to the top of the symbol table

instruction – a string that contains the instruction we received in the line

DC - our data meter vendor

numberOfLine – the current line number in the input file

symbol - symbol if received in line Returns 0 if an error is found in the instruction line (and of course prints it), otherwise 1

**• void iCCounter(addressingMode address, addressingMode prevAddress, int \*IC)**

This function updates the instruction counter according to the type of address received in the instruction line Its parameters are:

address – a number that represents the type of address received for the second operand

prevAddress – a number that represents the address type of the first operand

IC - Pointer of our instruction counter does not return anything

**• symbolTable handleLabelDefinition(symbolTable\* table, char\* labelName, flags\* status, int IC, int numberOfLine)**

This function is responsible for handling new symbol definitions. It checks for errors, and adds the new symbol to the symbol table. Its parameters are:

table – our symbol table

labelName – a string that initializes the name of the symbol

status – our flags structure, where we know if there is an error

IC - Integer of our instruction counter

numberOfLine – the current line number in the input file Returns the generated symbol if no error is found, otherwise returns NULL and changes the error flag to 1.

**• symbolTable createSymbolTable(char\* fileName, flags\* status)**

This function is the main one in the assembleySentence file, and it is the one responsible for passing the am file, creating the symbol table and detecting errors. She analyzes the words received in the line and performs the necessary tests according to the instructions of the assignment. Its parameters are:

fileName – string of the name of the file

status – pointer to a structure containing flags about the assembler status. If the file is not opened, return or an error was found in the file, return NULL and update the status, otherwise return the table of symbols obtained from the file

**SecondPass.c:**

After the first pass, a symbol table was built and most of the errors were found. The purpose of this file is to encode the assembly code for ob, ext, ent files. This file has the following functions:

**• void writeToFile(char\* content, FILE\* file, int IC)**

This function is responsible for writing to the ob file. Its parameters are: Content – ​​a string representing two base 32 characters that will be written to the file.

file – pointer to the ob file

IC - the value of the current IC, it will also be written to the file. returns nothing.

**• void toBase32 (int num, char\* base**)

This function is responsible for converting a number to base32, and writing it to an attached string. Its parameters are:

num – the number we convert

base – pointer to the string in which we write the converted result returns nothing. We have divided the file into parts, so that each part is responsible for something different. The following part is responsible for coding the instructions:

**• void encodeData(symbolTable table, int\* IC, int numberOfLine, FILE\* outFile)**

This function is responsible for coding directive data Its parameters are:

Table – the table of symbols built in the first pass

IC – pointer to the instruction counter

numberOfLine – the current line number in the input file outFile – pointer to the data file where we encode our information instructions.

**• void encodeString(symbolTable table, int\* IC, int numberOfLine, FILE\* outFile)**

This function is responsible for encoding the string directive Its parameters are: Table – the table of symbols built in the first pass

IC – pointer to the instruction counter

numberOfLine – the current line number in the input file

outFile – pointer to the data file where we encode our information instructions.

**• void encodeStruct(symbolTable table, int\* IC, int numberOfLine, FILE\* outFile)**

This function is responsible for encoding the struct directive Its parameters are: Table – the table of symbols built in the first pass

IC – pointer to the instruction counter

numberOfLine – the current line number in the input file

outFile – pointer to the data file where we encode our information instructions.

**• void handleEntryAndExtern (char\* firstWord, symbolTable table, int\* IC, int numberOfLine, FILE\* entFile, flags\* status)**

This function is responsible for handling entry and external directives. If this is an entry directive, it will check for errors and write to the .ent file And if this is an extern instruction, it will check for errors and write to the .ext file Its parameters are:

firstWord – this is the type of instruction written, it will be .extern/.entry

Table – the table of symbols built in the first pass

IC – pointer to the instruction counter

numberOfLine – the current line number in the input file

entFile – pointer to the ent file where we code entity directives Status – pointer to the structure that contains our flags in the assembler.

**• void handleInstructions(symbolTable table, char \*instruction, int \*IC, int numberOfLine, FILE\* outFile)**

This function is responsible for handling and encoding instructions. Its parameters are:

Table – the table of symbols built in the first pass

Instruction – a string containing the instruction type

IC – pointer to the instruction counter

numberOfLine – the current line number in the input file

outFile – pointer to the data file where we encode information instructions The following part is responsible for encoding commands:

**• int encodeNoneOperandsCommand(symbolTable table, char \*command, int \*IC,int numberOfLine, FILE\* outFile)**

This function is responsible for handling and encoding commands with 0 operands. Its parameters are:

Table – the table of symbols built in the first pass

command – a string containing the command in the line

IC – pointer to the instruction counter

numberOfLine – the current line number in the input file

outFile – pointer to the cmd file where we code instruction instructions

**• int encodeOneOperandsCommand (symbolTable table, char \*command, int \*IC,int numberOfLine, FILE\* outFile, FILE\* extFile)**

This function is responsible for handling and encoding commands with a single operand. Its parameters are:

Table – the table of symbols built in the first pass

command – a string containing the command in the line

IC – pointer to the instruction counter

numberOfLine – the current line number in the input file

outFile – pointer to the cmd file where we code instruction instructions

extFile - pointer to the ext file where we write appearances of externals.

The following part is responsible for encoding operands according to their addressing method:

• **int handleImmediateAddress(char\* operand, int\* IC, int numberOfLine, FILE\* outFile)**

This function is responsible for handling and encoding operands with immediate address. Its parameters are:

operand – a string containing the operand

IC – pointer to the data counter

numberOfLine – the current line number in the input file

outFile – pointer to the cmd file where we code commands

**• int handleDirectAddress (symblTable table, char\* operand, int\* IC, int numberOfLine, FILE\* outFile, FILE\* extFile)**

This function is responsible for handling and encoding direct-addressed operands. Its parameters are:

Table – the table of symbols built in the first pass

operand – a string containing the operand

IC – pointer to the data counter

numberOfLine – the current line number in the input file

outFile – pointer to the cmd file where we code commands

extFile – pointer to the .ext file where we encode instances of externals

**• int handleAddressAccess (symblTable table, char\* operand, int\* IC, int numberOfLine, FILE\* outFile, FILE\* extFile)**

This function is responsible for handling and encoding operands with access to the record. Its parameters are:

Table – the table of symbols built in the first pass

operand – a string containing the operand

IC – pointer to the data counter

numberOfLine – the current line number in the input file

outFile – pointer to the cmd file where we code commands

extFile – pointer to the .ext file where we encode instances of externals

**• int handleAddressingAccesses (addressingMode operandMode, symblTable table, int isDest, char\* operand, int\* IC, int numberOfLine, FILE\* outFile, FILE\* extFile)**

This function is responsible for handling and coding additional information words, by calling the appropriate functions according to the addressing methods of the operands. The parameters it receives are:

operandMode – the addressing method of the operand

Table – the table of symbols built in the first pass

isDest – a boolean variable that says whether the sent operand is a source or a destination

operand – a string containing the operand

IC – pointer to the data counter

numberOfLine – the current line number in the input file

outFile – pointer to the cmd file where we code commands

extFile – pointer to the .ext file where we encode instances of externals General functions of the second pass:

**• void handleFinalOutputFiles (char\* fileName, FILE\* cmdFile, FILE\* dataFile, flags\* status**)

This function is responsible for combining the two .cmd and .data files into an .obj file that first contains the commands and then the data. The parameters it receives are:

filename – the name of the file to be moved to

cmdFile – pointer to the coding file of the commands

Datafile – pointer to the encoding file of the information

Status – a structure that contains flags that contain information about the assembler's status

**• openFiles(char\* fileName, FILE\*\* inputFile, FILE\*\* cmdFile, FILE\*\* dataFile, FILE\*\* entFile, FILE\*\* extFile, flags\* status)**

This function is responsible for opening all the files used by us in the second pass. The function accepts pointers to file pointers, as it needs to change the variables. The parameters it receives are:

fileName – the name of the file to be moved to

inputFile – pointer of the am input file

cmdFile – pointer of the command encoding file

dataFile – pointer to the encoding file of the information

entFile – pointer to a file containing encoding of entity labels

extFile – pointer to a file containing encoding of externals labels

Status – a structure that contains flags that contain information about the assembler's status

**• encodeAssembly(char\* fileName, symbolTable table, flags\* status)**

This function is the main one in the secondPass file, and is responsible for the transition of the input file, and its encoding. The parameters it receives are: fileName – the name of the file to be moved to Table – the table of symbols built in the first pass Status – a structure that contains flags that contain information about the assembler's status