

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <ctype.h>
5
6  /*
7   System's Programming Phase 3 Algorithm
8
9   Author: Jesus M. Morales
10  Due Date: 04/13/2018
11
12  Remarks:
13  *Updated conditions to accept the exact amount of parameters on the main menu.
14  *Updated Pass1 to print Memory Addresses as HEX numbers instead of Decimals.
15  *Moved several variables from Pass1 to global variables to be able to use them
16  in Pass2 and vice-versa.
17  *Updated TOKEN structure to be able to use the same structure to tokenize the
18  source file from Pass1 and intermediate file from Pass2.
19
20  */
21
22  typedef struct
23  {
24      char label[10];
25      int memoryAddress;
26  } LABELS;
27
28  typedef struct
29  {
30      int memoryAddress;
31      char *label;
32      char *mnemonic;
33      char *operand;
34      int errorCode;
35  } TOKEN;
36
37  typedef struct
38  {
39      char mnemonic[5];
40      char opcode[5];
41  } OPCODE;
42
43  void breakupLine(char *input, char *command, char *param1, char *param2, int
44      *numParams);
45  int searchLabelLocation(char *inputLabel);
46  void printError(char **messageOutput, int errorCode);
47  void loadFile(char *fileName);
48  void executeFile();
49  void debugFile();
```

```
49 void dumpFile();
50 void helpFile();
51 void assembleFile();
52 void passOne(char * fileName);
53 void passTwo(char * fileName);
54
55 LABELS labelStructure[500];
56 OPCODE opcodeStructure[] = { { "ADD", "18" }, { "AND", "58" }, { "COMP", "28" },
    { "DIV", "24" },
57 { "J", "3C" }, { "JEQ", "30" }, { "JGT", "34" }, { "JLT", "38" },
58 { "JSUB", "48" }, { "LDA", "00" }, { "LDCH", "50" }, { "LDL", "08" },
59 { "LDX", "04" }, { "MUL", "20" }, { "OR", "44" }, { "RD", "D8" },
60 { "RSUB", "4C" }, { "STA", "0C" }, { "STCH", "54" }, { "STL", "14" },
61 { "STX", "10" }, { "SUB", "1C" }, { "TD", "E0" }, { "TIX", "2C" }, { "WD", "DC" } };
62
63 int programLength;
64 int errorFound;
65 int numberOfLabels;
66 int numMnemonics = 25;
67
68 int main(void)
69 {
70     char input[50];
71     char command[50];
72     char param1[50];
73     char param2[50];
74
75     printf("Hello welcome to Jesus Morales Personal SIC Machine\n\n");
76     while (1)
77     {
78         int numParams = 0;
79         int len = 0;
80         printf("Command ----> ");
81         fgets(input, 50, stdin);
82
83         len = strlen(input) - 1;
84         if (input[len] == '\n')
85         {
86             input[len] = '\0';
87         }
88
89         breakupLine(input, command, param1, param2, &numParams);
90         numParams--;
91
92         if (strcmp(command, "load") == 0)
93         {
94             if (param2[0] != '\0')
95             {
96                 printf("LOAD only requieres one parameter\n");
```

```
97     }
98     else if (param1[0] == '\0')
99     {
100         printf("LOAD requieres one parameter\n");
101     }
102     else if (param2[0] == '\0')
103     {
104         loadFile(param1);
105     }
106 }
107 else if (strcmp(command, "execute") == 0)
108 {
109     if (param2[0] != '\0' || param1[0] != '\0')
110     {
111         printf("EXECUTE doesn't need any parameters\n");
112     }
113     else
114     {
115         executeFile();
116     }
117 }
118 else if (strcmp(command, "debug") == 0)
119 {
120     if (param2[0] != '\0' || param1[0] != '\0')
121     {
122         printf("DEBUG doesn't need any parameters\n");
123     }
124     else
125     {
126         debugFile();
127     }
128 }
129 else if (strcmp(command, "dump") == 0)
130 {
131     if (param1[0] == '\0' || param2[0] == '\0')
132     {
133         printf("DUMP needs two parameter only\n");
134     }
135     else if (numParams > 3)
136     {
137         printf("DUMP needs two parameter only\n");
138     }
139     else
140     {
141         dumpFile();
142     }
143 }
144 else if (strcmp(command, "help") == 0)
145 {
```

```
146         if (param1[0] != '\0')
147         {
148             printf("HELP does not need parameters\n");
149         }
150         else
151         {
152             helpFile();
153         }
154     }
155     else if (strcmp(command, "assemble") == 0)
156     {
157         if (param2[0] != '\0')
158         {
159             printf("ASSEMBLE needs a file name \n");
160         }
161         else if (param1[0] == '\0')
162         {
163             printf("ASSEMBLE needs only one file name\n");
164         }
165         else if (param2[0] == '\0')
166         {
167             assembleFile();
168         }
169     }
170     else if (strcmp(command, "dir") == 0)
171     {
172         if (param2[0] != '\0' || param1[0] != '\0')
173         {
174             printf("DIRECTORY doesn't need any parameters \n");
175         }
176         else
177         {
178             system("ls");
179         }
180     }
181     else if (strcmp(command, "exit") == 0)
182     {
183         break;
184     }
185     else
186     {
187         printf("Invalid Command , for any help type 'help' to display the  ↗\n\n");
188     }
189     numParams = 0;
190 }
191 return 0;
192 }
193 }
```

```
194
195 void breakupLine(char *input, char *command, char *param1, char *param2, int  ↗
    *numParams)
196 {
197     command[0] = param1[0] = param2[0] = '\0';
198     *numParams = sscanf(input, "%s %s %s %s", command, param1, param2);
199 }
200 int searchLabelLocation(char *inputLabel)
201 {
202     char input[100];
203     char *tokenizer;
204
205     int memoryLocation;
206
207     FILE *symbol_table = fopen("symbolTable.txt", "r");
208
209     while (fgets(input, 100, symbol_table))
210     {
211         input[strcspn(input, "\n")] = '\0';
212         tokenizer = strtok(input, "\t");
213         memoryLocation = (int)strtol(tokenizer, NULL, 16);
214         tokenizer = strtok(NULL, "\t");
215         if (strcmp(tokenizer, inputLabel) == 0)
216         {
217             fclose(symbol_table);
218             return memoryLocation;
219         }
220     }
221     fclose(symbol_table);
222     return 00000;
223 }
224
225 void printError(char** messageOutput, int errorCode)
226 {
227     if (errorCode == 1)
228     {
229         strcpy(*messageOutput, "\t ** DUPLICATE LABEL ** ");
230     }
231     else if (errorCode == 2)
232     {
233         strcpy(*messageOutput, "\t ** ILLEGAL LABEL ** ");
234     }
235     else if (errorCode == 3)
236     {
237         strcpy(*messageOutput, "\t ** ILLEGAL OPERATION ** ");
238     }
239     else if (errorCode == 4)
240     {
241         strcpy(*messageOutput, "\t ** ILLEGAL DATA STORAGE DIRECTIVE ** ");
```

```
242     }
243     else if (errorCode == 5)
244     {
245         strcpy(*messageOutput, "\t ** MISSING START DIRECTIVE ** ");
246     }
247     else if (errorCode == 6)
248     {
249         strcpy(*messageOutput, "\t ** MISSING END DIRECTIVE ** ");
250     }
251     else if (errorCode == 7)
252     {
253         strcpy(*messageOutput, "\t **TOO MANY SYMBOLS ** ");
254     }
255     else if (errorCode == 8)
256     {
257         strcpy(*messageOutput, "\t ** PROGRAM TOO LONG ** ");
258     }
259 }
260
261
262
263 }
264 void loadFile(char *param1)
265 {
266     printf("Loading file: %s\n", param1);
267     passOne(param1);
268     passTwo(param1);
269     printf("The Programg lenght of this file is:  %d Bytes\n\n",
270           programLenght);
271     programLenght = 0;
272 }
273 void executeFile()
274 {
275     printf(" is not yet avaibalbe.\n");
276 }
277 void debugFile()
278 {
279     printf("debug is not avaiabale.\n");
280 }
281 void dumpFile()
282 {
283     printf("dump is not avaiblable.\n");
284 }
285 void helpFile()
286 {
287     printf("\n");
288     printf("\tWelcome to the Help menu. \n");
289     printf("\tCommands are the following: \n \n");
```

```
290     printf("\tload [file_name]\n");
291     printf("\texecute \n");
292     printf("\tdebug \n");
293     printf("\tdump [start] [end] \n");
294     printf("\thelp \n");
295     printf("\tassemble [file_name] \n");
296     printf("\tdirectory \n");
297     printf("\texit \n\n");
298     printf("\t**ALL COMMANDS ARE CASE SENSITIVE.**\n\n");
299 }
300 void assembleFile()
301 {
302     printf("assemble not avaibalbe. \n");
303 }
304 void passOne(char *param1)
305 {
306     char input[500];
307     char *tokenizer = input;
308
309     char *startingLocct;
310     int start = 0;
311     int locctr = 0;
312     int memLenght = 0;
313
314     int index = 0;
315
316     int labelPresentFlag = 0;
317     int duplicateLabelFlag = 0;
318     int illegalLabelFlag = 0;
319     int illegalOperationFlag = 0;
320     int missingDataDirectiveFlag = 0;
321     int missingStartFlag = 0;
322     int missingEndFlag = 0;
323     int tooManyLabelsFlag = 0;
324     int programTooLongFlag = 0;
325     int errorCode = 0;
326
327     FILE *source_file, *symbol_file, *intermediate_file, *opcode_file;
328     TOKEN sourceFileTokenizer;
329
330
331     source_file = fopen(param1, "r");
332     intermediate_file = fopen("intermediate.txt", "w");
333     symbol_file = fopen("symbolTable.txt", "w");
334
335     if (source_file == NULL)
336     {
337         printf("Error opening file does not exist: %s\n", param1);
338         return;
```

```
339     }
340
341     sourceFileTokenizer.label = (char *)malloc(6);
342     sourceFileTokenizer.mnemonic = (char *)malloc(6);
343     sourceFileTokenizer.operand = (char *)malloc(6);
344
345     errorFound = 0;
346     numberOfLabels = 0;
347     while (fgets(input, 500, source_file))
348     {
349         labelPresentFlag = 0;
350         duplicateLabelFlag = 0;
351         illegalLabelFlag = 0;
352         illegalOperationFlag = 0;
353         missingDataDirectiveFlag = 0;
354         missingStartFlag = 0;
355         missingEndFlag = 0;
356         tooManyLabelsFlag = 0;
357         programTooLongFlag = 0;
358         errorCode = 0;
359         memLenght = 0;
360
361         /* Check if label is present in the string line */
362         if (input[0] == ' ' || input[0] == '\t')
363         {
364             labelPresentFlag = 0;
365         }
366         else
367         {
368             labelPresentFlag = 1;
369         }
370
371         /* Check if comment is present in the string line */
372         if (input[0] == '.')
373         {
374             continue;
375         }
376
377         /* Tokenize the input string */
378         tokenizer = strtok(input, " \t\r\n\v\f");
379
380         /* Remove of the trailing newLine at the end of the string */
381         int counter = 0;
382         while (input[counter - 1] != '\n')
383         {
384             counter++;
385         }
386         input[counter] = '\0';
387     }
```



```
388     /* If there is a label */
389     if (labelPresentFlag == 1)
390     {
391         /* Tokenize the label into the structure */
392         strcpy(sourceFileTokenizer.label, tokenizer);
393
394         /* Tokenize the mnemonic into the structure */
395         tokenizer = strtok(NULL, " \\t\\r\\n\\v\\f");
396         strcpy(sourceFileTokenizer.mnemonic, tokenizer);
397
398         /* Tokenize the operand of the mnemonic into the structure */
399         tokenizer = strtok(NULL, " \\t\\r\\n\\v\\f");
400         strcpy(sourceFileTokenizer.operand, tokenizer);
401
402         /* Add the labels to the structure to create a list of existing      ↗
           labels/symbols */
403         strcpy(labelStructure[numberOfLabels].label,                      ↗
           sourceFileTokenizer.label);
404         labelStructure[numberOfLabels].memoryAddress = locctr;
405
406         /* Check if there are labels in the list */
407         if (numberOfLabels > 0)
408         {
409             /* Check if limit of labels has been reached */
410             if (numberOfLabels > 500)
411             {
412                 tooManyLabelsFlag = 1;
413             }
414
415             /* Inefficiently scan the label/symbol list to check for      ↗
           duplicate labels/symbols */
416             for (int i = 0; i < numberOfLabels; i++)
417             {
418                 if (strcmp(labelStructure[i].label,                      ↗
           sourceFileTokenizer.label) == 0)
419                 {
420                     duplicateLabelFlag = 1;
421                 }
422             }
423         }
424
425         /* Check if the label is legal */
426         if (!isalpha(sourceFileTokenizer.label[0]))
427         {
428             illegalLabelFlag = 1;
429         }
430
431         /* Check if we have a START directive in the beginning of the      ↗
           program */
```

```
432     if (index == 0 && strcmp(sourceFileTokenizer.mnemonic, "START") != 0) ↗
433     {
434         missingStartFlag = 1;
435         locctr = 0;
436     }
437
438     /* Check if we have a END directive in the end of the program */
439     if (missingEndFlag == 1 && errorCode == 0)
440     {
441         if (strcmp(sourceFileTokenizer.mnemonic, "END") != 0)
442         {
443             missingEndFlag = 1;
444         }
445     }
446
447     /* If directive START initialize LOCCTR to the starting address ↗
448     */
449     if (strcmp(sourceFileTokenizer.mnemonic, "START") == 0) // if start ↗
450     {
451         startingLoc = sourceFileTokenizer.operand;
452         start = (int)strtol(startingLoc, NULL, 16);
453         locctr = start;
454     }
455
456     /* Check if program is too long */
457     if (locctr > 32000)
458     {
459         programTooLongFlag = 1;
460     }
461     /* Length size in memory from the directives to increment the LOCCTR ↗
462     */
463     if (strcmp(sourceFileTokenizer.mnemonic, "WORD") == 0)
464     {
465         memLength += 3;
466     }
467     if (strcmp(sourceFileTokenizer.mnemonic, "RESB") == 0)
468     {
469         memLength += (int)strtol(sourceFileTokenizer.operand, NULL, 10);
470     }
471     if (strcmp(sourceFileTokenizer.mnemonic, "RESW") == 0)
472     {
473         memLength += 3 * (int)strtol(sourceFileTokenizer.operand, NULL, ↗
474                                     10);
```

```
475
476     if (strcmp(sourceFileTokenizer.mnemonic, "BYTE") == 0)
477     {
478         /* Check if operand is set to read a string (C) or a      ↗
479            hexadecimal (X) */
480         if (sourceFileTokenizer.operand[0] == 'C')
481         {
482             int bufferSpace = 0;
483             int counter = 2;
484             while (sourceFileTokenizer.operand[counter] != '\\' &&      ↗
485                   bufferSpace < 30)
486             {
487                 bufferSpace++;
488                 counter++;
489             }
490             memLenght += bufferSpace;
491         }
492     }
493     else if (sourceFileTokenizer.operand[0] == 'X')
494     {
495         char hexInput[10];
496         int bufferSpace = 0;
497         int counter = 3;
498         while (sourceFileTokenizer.operand[counter] != '\\' &&      ↗
499               bufferSpace < 10)
500         {
501             hexInput[bufferSpace] = sourceFileTokenizer.operand      ↗
502             [counter];
503             bufferSpace++;
504             counter++;
505         }
506         memLenght = (int)strtol(hexInput, NULL, 10);
507     }
508     /* Check for errors in the input for the BYTE directive */
509     else
510     {
511         illegalOperationFlag = 1;
512     }
513
514     if (sourceFileTokenizer.operand[1] != '\\' ||                  ↗
515         sourceFileTokenizer.operand[strlen      ↗
516         (sourceFileTokenizer.operand) - 1] != '\\')
517     {
518         missingDataDirectiveFlag = 1;
519     }
```

```
518     }
519
520     /* Error Flag conditions */
521     if (duplicateLabelFlag == 1 && errorCode == 0)
522     {
523         errorCode = 1;
524         errorFound = 1;
525     }
526     else if (illegalLabelFlag == 1 && errorCode == 0)
527     {
528         errorCode = 2;
529         errorFound = 1;
530     }
531     else if (illegalOperationFlag == 1 && errorCode == 0)
532     {
533         errorCode = 3;
534         errorFound = 1;
535     }
536     else if (missingDataDirectiveFlag == 1 && errorCode == 0)
537     {
538         errorCode = 4;
539         errorFound = 1;
540     }
541     else if (missingStartFlag == 1 && errorCode == 0)
542     {
543         errorCode = 5;
544         errorFound = 1;
545     }
546     else if (missingEndFlag == 1 && errorCode == 0)
547     {
548         errorCode = 6;
549         errorFound = 1;
550     }
551     else if (tooManyLabelsFlag == 1 && errorCode == 0)
552     {
553         errorCode = 7;
554         errorFound = 1;
555     }
556     else if (programTooLongFlag == 1 && errorCode == 0)
557     {
558         errorCode = 8;
559         errorFound = 1;
560     }
561
562     /*Print to the intermediate file and symbol file */
563     fprintf(intermediate_file, "%X\t%s\t%s\t%s\t%d\n", locctr,
564            sourceFileTokenizer.label, sourceFileTokenizer.mnemonic,
565            sourceFileTokenizer.operand, errorCode);
566     fprintf(symbol_file, "%X\t%s\n", locctr, sourceFileTokenizer.label);
```

```
565
566     /* Search for the mnemonic in the operand table and add 3 to it */
567     for (int i = 0; i < numMnemonics; i++)
568     {
569         if (strcmp(opcodeStructure[i].mnemonic,
570                    sourceFileTokenizer.mnemonic) == 0)
571         {
572             locctr += 3;
573         }
574     }
575     /*Update the memory locations after LOCCTR is printed in the file
576     */
577     if (strcmp(sourceFileTokenizer.mnemonic, "BYTE") == 0 || strcmp
578         (sourceFileTokenizer.mnemonic, "RESB") == 0 || strcmp
579         (sourceFileTokenizer.mnemonic, "RESW") == 0 || strcmp
580         (sourceFileTokenizer.mnemonic, "WORD") == 0)
581     {
582         locctr += memLenght;
583     }
584     /* Increment the number of labels in the system */
585     numberOfLabels++;
586 }
587
588 /*If there is no label in the input line do the same as above but
589 without labels */
590 else
591 {
592     /* Tokenize the mnemonic into the structure */
593     strcpy(sourceFileTokenizer.mnemonic, tokenizer);
594
595     if (strcmp(sourceFileTokenizer.mnemonic, "RSUB") != 0)
596     {
597         tokenizer = strtok(NULL, " \t\r\n\v\f");
598         strcpy(sourceFileTokenizer.operand, tokenizer);
599     }
600     else
601     {
602         strcpy(sourceFileTokenizer.label, " ");
603         strcpy(sourceFileTokenizer.operand, " ");
604     }
605
606     /* Tokenize the operand into the structure */
607
608     /* Check if we have a START directive in the beginning of the
```



```
650
651     if (strcmp(sourceFileTokenizer.mnemonic, "BYTE") == 0)
652     {
653         /* Check if operand is set to read a string (C) or a      ↗
           hexadecimal (X) */
654         if (sourceFileTokenizer.operand[0] == 'C')
655         {
656             int bufferSpace = 0;
657             int counter = 2;
658             while (sourceFileTokenizer.operand[counter] != '\\' &&      ↗
           bufferSpace < 30)
659             {
660                 bufferSpace++;
661                 counter++;
662             }
663             memLenght += bufferSpace;
664
665         }
666         else if (sourceFileTokenizer.operand[0] == 'X')
667         {
668             char hexInput[16];
669             int bufferSpace = 0;
670             int counter = 3;
671             while (sourceFileTokenizer.operand[counter] != '\\' &&      ↗
           bufferSpace < 16)
672             {
673                 hexInput[bufferSpace] = sourceFileTokenizer.operand      ↗
           [counter];
674                 bufferSpace++;
675                 counter++;
676             }
677
678             memLenght = (int)strtol(hexInput, NULL, 10);
679         }
680
681         /* Check for errors in the input for the BYTE directive */
682         else
683         {
684             illegalOperationFlag = 1;
685         }
686
687         if (sourceFileTokenizer.operand[1] != '\\' ||      ↗
           sourceFileTokenizer.operand[strlen      ↗
           (sourceFileTokenizer.operand) - 1] != '\\')
688         {
689             missingDataDirectiveFlag = 1;
690         }
691     }
692
```

```
693      /* Error Flag conditions */
694      if (duplicateLabelFlag == 1 && errorCode == 0)
695      {
696          errorCode = 1;
697          errorFound = 1;
698      }
699      else if (illegalLabelFlag == 1 && errorCode == 0)
700      {
701          errorCode = 2;
702          errorFound = 1;
703      }
704      else if (illegalOperationFlag == 1 && errorCode == 0)
705      {
706          errorCode = 3;
707          errorFound = 1;
708      }
709      else if (missingDataDirectiveFlag == 1 && errorCode == 0)
710      {
711          errorCode = 4;
712          errorFound = 1;
713      }
714      else if (missingStartFlag == 1 && errorCode == 0)
715      {
716          errorCode = 5;
717          errorFound = 1;
718      }
719      else if (missingEndFlag == 1 && errorCode == 0)
720      {
721          errorCode = 6;
722          errorFound = 1;
723      }
724      else if (tooManyLabelsFlag == 1 && errorCode == 0)
725      {
726          errorCode = 7;
727          errorFound = 1;
728      }
729      else if (programTooLongFlag == 1 && errorCode == 0)
730      {
731          errorCode = 8;
732          errorFound = 1;
733      }
734
735      /*Print to the intermediate file and symbol file */
736      fprintf(intermediate_file, "%X\t\t\t%s\t%s\t%d\n", locctr,
737          sourceFileTokenizer.mnemonic, sourceFileTokenizer.operand,
738          errorCode);
739
740      /* Search for the mnemonic in the operand table and add 3 to it */
741      for (int i = 0; i < numMnemonics; i++)
```



```

740     {
741         if (strcmp(opcodeStructure[i].mnemonic,
742                    sourceFileTokenizer.mnemonic) == 0)
743         {
744             locctr += 3;
745         }
746     }
747     /*Update the memory locations after LOCCTR is printed in the file
748     */
749     if (strcmp(sourceFileTokenizer.mnemonic, "BYTE") == 0 || strcmp
750         (sourceFileTokenizer.mnemonic, "RESB") == 0 || strcmp
751         (sourceFileTokenizer.mnemonic, "RESW") == 0 || strcmp
752         (sourceFileTokenizer.mnemonic, "WORD") == 0)
753     {
754         locctr += memLenght;
755     }
756     index++;
757 }
758
759 programLenght = locctr - start;
760 programLenght = programLenght - 4;
761
762 printf("Pass One complete successfully. \n");
763
764 fprintf(intermediate_file, "\n\n\t Printing Error Code List: \n\n");
765 fprintf(intermediate_file, "*=====*\n");
766 fprintf(intermediate_file, "\tNo Error = 0\n");
767 fprintf(intermediate_file, "\tDuplicate Label = 1\n");
768 fprintf(intermediate_file, "\tIllegal Label = 2\n");
769 fprintf(intermediate_file, "\tIllegal Operation = 3\n");
770 fprintf(intermediate_file, "\tIllegal Data Storage Directive = 4\n");
771 fprintf(intermediate_file, "\tMissing START Directive = 5\n");
772 fprintf(intermediate_file, "\tMissing END Directive = 6\n");
773 fprintf(intermediate_file, "\tToo Many Symbols = 7\n");
774 fprintf(intermediate_file, "\tProgram Too Long = 8\n");
775 fprintf(intermediate_file, "*=====*\n");
776
777 fclose(intermediate_file);
778 fclose(source_file);
779 fclose(symbol_file);
780 }
781 void passTwo(char *param1)
782 {
783     char input[100];
784     char sourceInput[500];

```

```
782
783     char *tokenizer;
784     char *objectCode_string;
785     char *errorMessage;
786
787     int startingAddress;
788     int operandAddress;
789     int objectCode_decimal;
790     int objectLineLenght = 0;
791
792     int newLineFlag = 1;
793     int labelPresentFlag = 0;
794
795     FILE *intermediateFile, *symbolTable, *objectFile, *listingFile,      ↗
        *sourceFile;
796     TOKEN intermediateFileTokenizer;
797
798     objectFile = fopen("objectFile.txt", "w");
799     listingFile = fopen("listingFile.txt", "w");
800     intermediateFile = fopen("intermediate.txt", "r");
801     symbolTable = fopen("symbolTable.txt", "r");
802     sourceFile = fopen(param1, "r");
803
804     if (sourceFile == NULL)
805     {
806         printf("Intermediate file did not opened correctly \n");
807         return;
808     }
809
810     intermediateFileTokenizer.label = (char *)malloc(6);
811     intermediateFileTokenizer.mnemonic = (char *)malloc(6);
812     intermediateFileTokenizer.operand = (char *)malloc(6);
813     errorMessage = (char *)malloc(256);
814
815     while (fgets(input, 100, intermediateFile))
816     {
817         memset(intermediateFileTokenizer.label, '\0', 6);
818         memset(intermediateFileTokenizer.mnemonic, '\0', 6);
819         memset(intermediateFileTokenizer.operand, '\0', 6);
820         memset(errorMessage, '\0', 256);
821
822         fgets(sourceInput, 500, sourceFile);
823
824         /* Check it the source line is a comment */
825         if (sourceInput[0] == '.')
826         {
827             while (sourceInput[0] == '.')
828             {
829                 fprintf(listingFile, "%s", sourceInput);
```

```
830         fgets(sourceInput, 500, sourceFile);
831     }
832 }
833
834 labelPresentFlag = 0;
835
836 tokenizer = strtok(input, "\\t");
837 intermediateFileTokenizer.memoryAddress = (int)strtol(tokenizer, NULL, 16);    ///save address
838 tokenizer = strtok(NULL, "\\t");
839
840 for (int i = 0; i < numberOfLabels; i++)
841 {
842     if (strcmp(labelStructure[i].label, tokenizer) == 0)
843     {
844         labelPresentFlag = 1;
845         break;
846     }
847 }
848
849 if (labelPresentFlag == 1)
850 {
851     strcpy(intermediateFileTokenizer.label, tokenizer); ///save label
852     tokenizer = strtok(NULL, "\\t");
853 }
854
855 strcpy(intermediateFileTokenizer.mnemonic, tokenizer); ///save mnemonic
856
857 if (strcmp(intermediateFileTokenizer.mnemonic, "RSUB") != 0)
858 {
859     tokenizer = strtok(NULL, "\\t");
860     strcpy(intermediateFileTokenizer.operand, tokenizer); ///save operand
861     tokenizer = strtok(NULL, " \\t");
862     intermediateFileTokenizer.errorCode = (int)strtol(tokenizer, NULL, 10);    ///save errorcode
863 }
864 else
865 {
866     tokenizer = strtok(NULL, " \\t");
867     intermediateFileTokenizer.errorCode = (int)strtol(tokenizer, NULL, 10);    ///save errorcode
868     objectCode_string = "4C0000";
869
870     if (newLineFlag == 1)
871     {
872         fprintf(objectFile, "\\n");
873         fprintf(objectFile, "T%s", objectCode_string);
874         newLineFlag = 0;
```

```
875         objectLineLenght++;
876     }
877     else
878     {
879         fprintf(objectFile, "%s", objectCode_string);
880         objectLineLenght++;
881     }
882
883     if (intermediateFileTokenizer.errorCode == 0)
884     {
885         fprintf(listingFile, "%X\t%s\t%s",
886                 intermediateFileTokenizer.memoryAddress,
887                 objectCode_string, sourceInput);
888         continue;
889     }
890     else
891     {
892         printError(&errorMessage, intermediateFileTokenizer.errorCode);
893         fprintf(listingFile, "%s\n", errorMessage);
894         continue;
895     }
896 }
897
898 /* Check if the object file size limit has been reached */
899 if (objectLineLenght == 10)
900 {
901     newLineFlag = 1;
902     objectLineLenght = 0;
903 }
904
905 /* Check it the intermediate line is a START */
906 if (strcmp(intermediateFileTokenizer.mnemonic, "START") == 0 ||
907     intermediateFileTokenizer.errorCode == 5)
908 {
909     fprintf(objectFile, "H%_s%06X%06X", intermediateFileTokenizer.label,
910             intermediateFileTokenizer.memoryAddress, programLenght);
911     if (intermediateFileTokenizer.errorCode == 0)
912     {
913         fprintf(listingFile, "%X\t\t%s",
914                 intermediateFileTokenizer.memoryAddress, sourceInput);
915     }
916     else
917     {
918         printError(&errorMessage, intermediateFileTokenizer.errorCode);
919         fprintf(listingFile, "%s\n", errorMessage);
920     }
921     startingAddress = intermediateFileTokenizer.memoryAddress;
922 }
```

```

919  /* Check it the intermediate line is a RESW */
920  else if (strcmp(intermediateFileTokenizer.mnemonic, "RESW") == 0 ||
          strcmp(intermediateFileTokenizer.mnemonic, "RESB") == 0 ||
          intermediateFileTokenizer.errorCode == 4)
921  {
922      if (intermediateFileTokenizer.errorCode == 0)
923      {
924          fprintf(listingFile, "%X\t\t%s",
                  intermediateFileTokenizer.memoryAddress, sourceInput);
925      }
926      else
927      {
928          printError(&errorMessage, intermediateFileTokenizer.errorCode);
929          fprintf(listingFile, "%s\n", errorMessage);
930      }
931  }
932
933  /* Check it the intermediate line is a WORD */
934  else if (strcmp(intermediateFileTokenizer.mnemonic, "WORD") == 0 ||
          intermediateFileTokenizer.errorCode == 4)
935  {
936      objectCode_string = intermediateFileTokenizer.operand;
937      objectCode_decimal = (int)strtol(objectCode_string, NULL, 10);
938      if (strcmp(intermediateFileTokenizer.operand, "0") == 0)
939      {
940          fprintf(objectFile, "%06X", objectCode_decimal);
941          newLineFlag = 1;
942          objectLineLength = 0;
943
944          if (intermediateFileTokenizer.errorCode == 0)
945          {
946              fprintf(listingFile, "%X\t\t%06X\t%s",
                      intermediateFileTokenizer.memoryAddress,
                      objectCode_decimal, sourceInput);
947              continue;
948          }
949          else
950          {
951              printError(&errorMessage,
                          intermediateFileTokenizer.errorCode);
952              fprintf(listingFile, "%s\n", errorMessage);
953              continue;
954          }
955      }
956      else
957      {
958          if (intermediateFileTokenizer.errorCode == 0)
959          {
960              fprintf(listingFile, "%X\t\t%06X\t%s",

```

```

        intermediateFileTokenizer.memoryAddress, objectCode_decimal,
        sourceInput);
961     }
962     else
963     {
964         printError(&errorMessage,
965                     intermediateFileTokenizer.errorCode);
966         fprintf(listingFile, "%s\n", errorMessage);
967     }
968
969     if (newLineFlag == 1)
970     {
971         fprintf(objectFile, "\n");
972         fprintf(objectFile, "T%06X%
973             06X", intermediateFileTokenizer.memoryAddress,
974             objectCode_decimal);
975         newLineFlag = 0;
976         objectLineLenght++;
977     }
978     else
979     {
980         fprintf(objectFile, "%06X", objectCode_decimal);
981         objectLineLenght++;
982     }
983
984     /* Check it the intermediate line is a BYTE */
985     else if(strcmp(intermediateFileTokenizer.mnemonic, "BYTE") == 0 ||
986             intermediateFileTokenizer.errorCode == 3)
987     {
988         if(intermediateFileTokenizer.operand[0] == 'C')
989         {
990             char copyHEX[10];
991             char convertedHEX[10];
992
993             int inputIndex = 2;
994             int outputIndex = 0;
995
996             while (intermediateFileTokenizer.operand[inputIndex] != '\')
997             {
998                 copyHEX[outputIndex] = intermediateFileTokenizer.operand
999                 [inputIndex];
1000                 inputIndex++;
1001                 outputIndex++;
1002             }
1003
1004             sprintf(convertedHEX, "%X%X%X", copyHEX[0], copyHEX[1], copyHEX
1005                 [2]);

```

```
1002
1003     if (intermediateFileTokenizer.errorCode == 0)
1004     {
1005         fprintf(listingFile, "%X\t%s\t%s",
1006                 intermediateFileTokenizer.memoryAddress,
1007                 convertedHEX, sourceInput);
1008     }
1009     else
1010     {
1011         printError(&errorMessage,
1012                 intermediateFileTokenizer.errorCode);
1013         fprintf(listingFile, "%s\n", errorMessage);
1014     }
1015
1016     if (newLineFlag == 1)
1017     {
1018         fprintf(objectFile, "\n");
1019         fprintf(objectFile, "T%06X%s",
1020                 intermediateFileTokenizer.memoryAddress, convertedHEX);
1021         newLineFlag = 0;
1022         objectLineLenght++;
1023     }
1024     else
1025     {
1026         fprintf(objectFile, "%s", convertedHEX);
1027         objectLineLenght++;
1028     }
1029 }
1030 else if (intermediateFileTokenizer.operand[0] == 'X')
1031 {
1032     char copyHEX[10];
1033
1034     int inputIndex = 2;
1035     int outputIndex = 0;
1036
1037     while (intermediateFileTokenizer.operand[inputIndex] != '\0')
1038     {
1039         copyHEX[outputIndex] = intermediateFileTokenizer.operand
1040             [inputIndex];
1041         inputIndex++;
1042         outputIndex++;
1043     }
1044     copyHEX[outputIndex] = '\0';
1045
1046     if (intermediateFileTokenizer.errorCode == 0)
1047     {
1048         fprintf(listingFile, "%X\t%s\t%s",
1049                 intermediateFileTokenizer.memoryAddress,
```

```
        copyHEX,sourceInput);
1045     }
1046     else
1047     {
1048         printError(&errorMessage,
1049                  intermediateFileTokenizer.errorCode);
1049         fprintf(listingFile, "%s\n", errorMessage);
1050     }
1051
1052     if (newLineFlag == 1)
1053     {
1054         fprintf(objectFile, "\n");
1055         fprintf(objectFile, "T%06X%s",
1056                  intermediateFileTokenizer.memoryAddress, copyHEX);
1056         newLineFlag = 0;
1057         objectLineLenght++;
1058     }
1059     else
1060     {
1061         fprintf(objectFile, "%s", copyHEX);
1062         objectLineLenght++;
1063     }
1064 }
1065 else
1066 {
1067     printError(&errorMessage, intermediateFileTokenizer.errorCode);
1068     fprintf(listingFile, "%s\n", errorMessage);
1069 }
1070 }
1071
1072 /* Check it the intermediate line is a END */
1073 else if (strcmp(intermediateFileTokenizer.mnemonic, "END") == 0)
1074 {
1075     operandAddress = searchLabelLocation
1076                     (intermediateFileTokenizer.operand);
1076     fprintf(objectFile, "\n");
1077     fprintf(objectFile, "E%06X", operandAddress);
1078
1079     if (intermediateFileTokenizer.errorCode == 0)
1080     {
1081         fprintf(listingFile, "%s", sourceInput);
1082     }
1083     else
1084     {
1085         printError(&errorMessage, intermediateFileTokenizer.errorCode);
1086         fprintf(listingFile, "%s\n", errorMessage);
1087     }
1088     break;
1089 }
```



```
1090
1091     /* Else it is a regular mnemonic and just requieres normal handling */
1092     else
1093     {
1094         char *OpcodeExtracted;
1095         int OpcodeConverted;
1096         char objectCode[10];
1097
1098         operandAddress = searchLabelLocation
1099             (intermediateFileTokenizer.operand);
1100
1101         for (int i = 0; i < numMnemonics; i++)
1102         {
1103             if (strcmp(opcodeStructure[i].mnemonic,
1104                 intermediateFileTokenizer.mnemonic) == 0)
1105             {
1106                 OpcodeExtracted = opcodeStructure[i].opcode;
1107                 break;
1108             }
1109         }
1110
1111         OpcodeConverted = (int)strtol(OpcodeExtracted, NULL, 16);
1112         sprintf(objectCode, "%02X%04X", OpcodeConverted,
1113             operandAddress);
1114
1115         if (intermediateFileTokenizer.errorCode == 0)
1116         {
1117             fprintf(listingFile, "%X\t%06s\t%s",
1118                 intermediateFileTokenizer.memoryAddress,
1119                 objectCode, sourceInput);
1120         }
1121         else
1122         {
1123             printError(&errorMessage,
1124                 intermediateFileTokenizer.errorCode);
1125             fprintf(listingFile, "%s\n", errorMessage);
1126         }
1127
1128         if (newLineFlag == 1)
1129         {
1130             fprintf(objectFile, "\n");
1131             fprintf(objectFile, "T%06X%06s",
1132                 intermediateFileTokenizer.memoryAddress, objectCode);
1133             newLineFlag = 0;
1134             objectLineLenght++;
1135         }
1136         else
1137         {
1138
```

```
1131         fprintf(objectFile, "%06s", objectCode);
1132         objectLineLenght++;
1133     }
1134 }
1135 }
1136
1137 /* Close all files */
1138 fclose(objectFile);
1139 fclose(listingFile);
1140 fclose(intermediateFile);
1141 fclose(symbolTable);
1142
1143 /* Check if there were errors on Pass 1 if so delete the object file */
1144 if (errorFound == 1)
1145 {
1146     if (remove("objectFile.txt") == 0)
1147     {
1148         printf("Program Has Errors.\n");
1149     }
1150 }
1151 else
1152 {
1153     printf("Pass Two complete successfully. \n");
1154 }
1155 }
1156
```