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

/*
System's Programming Phase 4 Algorithm

Author: Jesus M. Morales
Due Date: 05/04/2018

Remarks:
*Added to the project the fetchObj2Memory() functions that basically opens the object file
and adds the data into memory.
*Implemented the Execute command.
*Implemented the Load command by adding to the project the fetchObj2Memory() function in which the data of the object file
is added to memory.
*Implemented the Dump command

* This are the 3 modifications I did to my project for Phase 4; I did the project from a single file so that is why I keep
adding more to the program since I only have this main project.

*/

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];
} OPCODE1;

```

```
47  void breakupLine(char *input, char *command, char *param1, char *param2, int
      *numParams);
48  int searchLabelLocation(char *inputLabel);
49  void printError(char **messageOutput, int errorCode);
50  void loadFile(char *fileName);
51  void executeFile();
52  void debugFile();
53  void dumpFile(char *param1, char *param2);
54  void helpFile();
55  void assembleFile();
56  void passOne(char * fileName);
57  void passTwo(char * fileName);
58  void fetchObj2Memory();
59
60  LABELS labelStructure[500];
61  OPCODE1 opcodeStructure[] = { { "ADD", "18" },{ "AND","58" },{ "COMP", "28" },
      { "DIV", "24" },
62  { "J", "3C" },{ "JEQ", "30" },{ "JGT", "34" },{ "JLT", "38" },
63  { "JSUB", "48" },{ "LDA", "00" },{ "LDCH", "50" },{ "LDL", "08" },
64  { "LDX", "04" },{ "MUL", "20" },{ "OR", "44" },{ "RD", "D8" },
65  { "RSUB", "4C" },{ "STA", "0C" },{ "STCH", "54" },{ "STL", "14" },
66  { "STX", "10" },{ "SUB", "1C" },{ "TD", "E0" },{ "TIX", "2C" },{ "WD", "DC" } };
67
68  int programLenght;
69  int errorFound;
70  int numberOfLabels;
71  int numMnemonics = 25;
72  int startLoc = -1;
73  int programEnd_int;
74
75
76  int main(void)
77  {
78
79      SICInit();
80      char input[50];
81      char command[50];
82      char param1[50];
83      char param2[50];
84
85      printf("Hello welcome to Jesus Morales Personal SIC Machine\n \n");
86      while (1)
87      {
88          int numParams = 0;
89          int len = 0;
90          printf("Command ----> ");
91          fgets(input, 50, stdin);
92
93          len = strlen(input) - 1;
```

```
 94          if (input[len] == '\n')
 95          {
 96              input[len] = '\0';
 97          }
 98
 99          breakupLine(input, command, param1, param2, &numParams);
100          numParams--;
101
102          if (strcmp(command, "load") == 0)
103          {
104              if (param2[0] != '\0')
105              {
106                  printf("LOAD only requieres one parameter\n");
107              }
108              else if (param1[0] == '\0')
109              {
110                  printf("LOAD requieres one parameter\n");
111              }
112              else if (param2[0] == '\0')
113              {
114                  loadFile(param1);
115              }
116          }
117          else if (strcmp(command, "execute") == 0)
118          {
119              if (param2[0] != '\0' || param1[0] != '\0')
120              {
121                  printf("EXECUTE doesn't need any parameters\n");
122              }
123              else
124              {
125                  executeFile();
126              }
127          }
128          else if (strcmp(command, "debug") == 0)
129          {
130              if (param2[0] != '\0' || param1[0] != '\0')
131              {
132                  printf("DEBUG doesn't need any parameters\n");
133              }
134              else
135              {
136                  debugFile();
137              }
138          }
139          else if (strcmp(command, "dump") == 0)
140          {
141              if (param1[0] == '\0' || param2[0] == '\0')
142              {
```

```
143                 printf("DUMP needs two parameter only\n");
144             }
145             else if (numParams > 3)
146             {
147                 printf("DUMP needs two parameter only\n");
148             }
149             else
150             {
151                 dumpFile(param1,param2);
152             }
153         }
154         else if (strcmp(command, "help") == 0)
155         {
156             if (param1[0] != '\0')
157             {
158                 printf("HELP does not need parameters\n");
159             }
160             else
161             {
162                 helpFile();
163             }
164         }
165         else if (strcmp(command, "assemble") == 0)
166         {
167             if (param2[0] != '\0')
168             {
169                 printf("ASSEMBLE needs a file name \n");
170             }
171             else if (param1[0] == '\0')
172             {
173                 printf("ASSEMBLE needs only one file name\n");
174             }
175             else if (param2[0] == '\0')
176             {
177                 assembleFile();
178             }
179         }
180         else if (strcmp(command, "dir") == 0)
181         {
182             if (param2[0] != '\0' || param1[0] != '\0')
183             {
184                 printf("DIRECTORY doesn't need any parameters \n");
185             }
186             else
187             {
188                 system("ls");
189             }
190         }
191         else if (strcmp(command, "exit") == 0)
```

```
192         {
193             break;
194         }
195         else
196         {
197             printf("Invalid Command , for any help type 'help' to display the
                  command list. \n \n");
198         }
199
200         numParams = 0;
201     }
202     return 0;
203 }
204
205 void breakupLine(char *input, char *command, char *param1, char *param2, int
      *numParams)
206 {
207     command[0] = param1[0] = param2[0] = '\0';
208     *numParams = sscanf(input, "%s %s %s %*s", command, param1, param2);
209 }
210 int searchLabelLocation(char *inputLabel)
211 {
212     char input[100];
213     char *tokenizer;
214
215     int memoryLocation;
216
217     FILE *symbol_table = fopen("symbolTable.txt", "r");
218
219     while (fgets(input, 100, symbol_table))
220     {
221         input[strcspn(input, "\n")] = '\0';
222         tokenizer = strtok(input, "\t");
223         memoryLocation = (int)strtol(tokenizer, NULL, 16);
224         tokenizer = strtok(NULL, "\t");
225         if (strcmp(tokenizer, inputLabel) == 0)
226         {
227             fclose(symbol_table);
228             return memoryLocation;
229         }
230     }
231     fclose(symbol_table);
232     return 00000;
233
234 }
235 void printError(char** messageOutput, int errorCode)
236 {
237     if (errorCode == 1)
238     {
```

```
239         strcpy(*messageOutput, "\t ** DUPLICATE LABEL ** ");
240     }
241     else if (errorCode == 2)
242     {
243         strcpy(*messageOutput, "\t ** ILLEGAL LABEL ** ");
244     }
245     else if (errorCode == 3)
246     {
247         strcpy(*messageOutput, "\t ** ILLEGAL OPERATION ** ");
248     }
249     else if (errorCode == 4)
250     {
251         strcpy(*messageOutput, "\t ** ILLEGAL DATA STORAGE DIRECTIVE ** ");
252     }
253     else if (errorCode == 5)
254     {
255         strcpy(*messageOutput, "\t ** MISSING START DIRECTIVE ** ");
256 
257     }
258     else if (errorCode == 6)
259     {
260         strcpy(*messageOutput, "\t ** MISSING END DIRECTIVE ** ");
261     }
262     else if (errorCode == 7)
263     {
264         strcpy(*messageOutput, "\t **TOO MANY SYMBOLS ** ");
265     }
266     else if (errorCode == 8)
267     {
268         strcpy(*messageOutput, "\t ** PROGRAM TOO LONG ** ");
269     }
270 
271 
272 
273 }
274 void loadFile(char *param1)
275 {
276     printf("Loading file:  %s\n", param1);
277     passOne(param1);
278     passTwo(param1);
279     printf("\n");
280 
281     fetchObj2Memory();
282     programLenght = 0;
283 }
284 void executeFile()
285 {
286     SICRun(&programEnd_int, 0);
287 }
```

```c
void debugFile()
{
    printf("debug is not avaialabe.\n");
}
void dumpFile(char *param1, char *param2)
{
    int startingAddress = (int)strtol(param1, NULL, 16);
    int endingAddress = (int)strtol(param2, NULL, 16);

    BYTE value;
    int index = 0;

    printf("%X: ", startingAddress);
    for (int i = startingAddress; i <= endingAddress; i++)
    {
        if (index == 16)
        {
            printf("\n%X: ", i);
            index = 0;
        }

        GetMem(i, &value, 0);
        printf("%02X ", value);
        index++;
    }
    printf("\n\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];
```

```
337     char *tokenizer = input;
338
339     char *startingLoct;
340     int start = 0;
341     int locctr = 0;
342     int memLenght = 0;
343
344     int index = 0;
345
346     int labelPresentFlag = 0;
347     int duplicateLabelFlag = 0;
348     int illegalLabelFlag = 0;
349     int illegalOperationFlag = 0;
350     int missingDataDirectiveFlag = 0;
351     int missingStartFlag = 0;
352     int missingEndFlag = 0;
353     int tooManyLabelsFlag = 0;
354     int programTooLongFlag = 0;
355     int errorCode = 0;
356
357     FILE *source_file, *symbol_file, *intermediate_file, *opcode_file;
358     TOKEN sourceFileTokenizer;
359
360
361     source_file = fopen(param1, "r");
362     intermediate_file = fopen("intermediate.txt", "w");
363     symbol_file = fopen("symbolTable.txt", "w");
364
365     if (source_file == NULL)
366     {
367         printf("Error openning file does not exist: %s\n", param1);
368         return;
369     }
370
371     sourceFileTokenizer.label = (char *)malloc(6);
372     sourceFileTokenizer.mnemonic = (char *)malloc(6);
373     sourceFileTokenizer.operand = (char *)malloc(6);
374
375     errorFound = 0;
376     numberOfLabels = 0;
377     while (fgets(input, 500, source_file))
378     {
379         labelPresentFlag = 0;
380         duplicateLabelFlag = 0;
381         illegalLabelFlag = 0;
382         illegalOperationFlag = 0;
383         missingDataDirectiveFlag = 0;
384         missingStartFlag = 0;
385         missingEndFlag = 0;
```

```
386         tooManyLabelsFlag = 0;
387         programTooLongFlag = 0;
388         errorCode = 0;
389         memLenght = 0;
390 
391         /*  Check if label is present in the string line    */
392         if (input[0] == ' ' || input[0] == '\t')
393         {
394             labelPresentFlag = 0;
395         }
396         else
397         {
398             labelPresentFlag = 1;
399         }
400 
401         /*  Check if comment is present in the string line  */
402         if (input[0] == '.')
403         {
404             continue;
405         }
406 
407         /*  Tokenize the input string   */
408         tokenizer = strtok(input, " \t\r\n\v\f");
409 
410         /*  Remove of the trailing newLine at the end of the string */
411         int counter = 0;
412         while (input[counter - 1] != '\n')
413         {
414             counter++;
415         }
416         input[counter] = '\0';
417 
418         /*  If there is a label */
419         if (labelPresentFlag == 1)
420         {
421             /*  Tokenize the label into the structure   */
422             strcpy(sourceFileTokenizer.label, tokenizer);
423 
424             /*  Tokenize the mnemonic into the structure    */
425             tokenizer = strtok(NULL, " \t\r\n\v\f");
426             strcpy(sourceFileTokenizer.mnemonic, tokenizer);
427 
428             /*  Tokenize the operand of the mnemonic into the structure */
429             tokenizer = strtok(NULL, " \t\r\n\v\f");
430             strcpy(sourceFileTokenizer.operand, tokenizer);
431 
432             /*  Add the labels to the structure to create a list of existing
                  labels/symbols */
433             strcpy(labelStructure[numberOfLabels].label,
```

```
                sourceFileTokenizer.label);
434             labelStructure[numberOfLabels].memoryAddress = locctr;
435
436             /*  Check if there are labels in the list   */
437             if (numberOfLabels > 0)
438             {
439                 /*  Check if limit of labels has been reached   */
440                 if (numberOfLabels > 500)
441                 {
442                     tooManyLabelsFlag = 1;
443                 }
444
445                 /*  Inefficiently scan the label/symbol list to check for
                      duplicate labels/symbols     */
446                 for (int i = 0; i < numberOfLabels; i++)
447                 {
448                     if (strcmp(labelStructure[i].label,
                        sourceFileTokenizer.label) == 0)
449                     {
450                         duplicateLabelFlag = 1;
451                     }
452                 }
453             }
454
455             /*  Check if the label is legal */
456             if (!isalpha(sourceFileTokenizer.label[0]))
457             {
458                 illegalLabelFlag = 1;
459             }
460
461             /*  Check if we have a START directive in the beginning of the
                  program */
462             if (index == 0 && strcmp(sourceFileTokenizer.mnemonic, "START") !=
                  0)
463             {
464                 missingStartFlag = 1;
465                 locctr = 0;
466             }
467
468             /*  Check if we have a END directive in the end of the program  */
469             if (missingEndFlag == 1 && errorCode == 0)
470             {
471                 if (strcmp(sourceFileTokenizer.mnemonic, "END") != 0)
472                 {
473                     missingEndFlag = 1;
474                 }
475             }
476
477             /*  If directive START initialize LOCCTR to the starting address
```

```
                */
478             if (strcmp(sourceFileTokenizer.mnemonic, "START") == 0) // if start
                directive initialize locct to the start(convert the string to
                integer)
479             {
480
481                 startingLoct = sourceFileTokenizer.operand;
482                 start = (int)strtol(startingLoct, NULL, 16);
483                 locctr = start;
484             }
485
486             /*  Check if program is too long    */
487             if (locctr > 32000)
488             {
489                 programTooLongFlag = 1;
490             }
491             /* Lenght size in memory from the directives to increment the LOCCTR
                   */
492             if (strcmp(sourceFileTokenizer.mnemonic, "WORD") == 0)
493             {
494                 memLenght += 3;
495             }
496             if (strcmp(sourceFileTokenizer.mnemonic, "RESB") == 0)
497             {
498                 memLenght += (int)strtol(sourceFileTokenizer.operand, NULL, 10);
499
500             }
501             if (strcmp(sourceFileTokenizer.mnemonic, "RESW") == 0)
502             {
503                 memLenght += 3 * (int)strtol(sourceFileTokenizer.operand, NULL,
                    10);
504             }
505
506             if (strcmp(sourceFileTokenizer.mnemonic, "BYTE") == 0)
507             {
508                 /*  Check if operand is set to read a string (C) or a
                    hexadecimal (X) */
509                 if (sourceFileTokenizer.operand[0] == 'C')
510                 {
511
512                     int bufferSpace = 0;
513                     int counter = 2;
514                     while (sourceFileTokenizer.operand[counter] != '\'' &&
                        bufferSpace < 30)
515                     {
516                         bufferSpace++;
517                         counter++;
518                     }
519
```

```
520                     memLenght += bufferSpace;
521
522                 }
523                 else if (sourceFileTokenizer.operand[0] == 'X')
524                 {
525                     char hexInput[10];
526                     int bufferSpace = 0;
527                     int counter = 3;
528                     while (sourceFileTokenizer.operand[counter] != '\'' &&
                        bufferSpace < 10)
529                     {
530                         hexInput[bufferSpace] = sourceFileTokenizer.operand
                        [counter];
531                         bufferSpace++;
532                         counter++;
533                     }
534
535                     memLenght = (int)strtol(hexInput, NULL, 10);
536                 }
537
538                 /* Check for errors in the input for the BYTE directive */
539                 else
540                 {
541                     illegalOperationFlag = 1;
542                 }
543
544                 if (sourceFileTokenizer.operand[1] != '\'' ||
                      sourceFileTokenizer.operand[strlen
                      (sourceFileTokenizer.operand) - 1] != '\'')
545                 {
546                     missingDataDirectiveFlag = 1;
547                 }
548             }
549
550             /* Error Flag conditions    */
551             if (duplicateLabelFlag == 1 && errorCode == 0)
552             {
553                 errorCode = 1;
554                 errorFound = 1;
555             }
556             else if (illegalLabelFlag == 1 && errorCode == 0)
557             {
558                 errorCode = 2;
559                 errorFound = 1;
560             }
561             else if (illegalOperationFlag == 1 && errorCode == 0)
562             {
563                 errorCode = 3;
564                 errorFound = 1;
```

```
565             }
566             else if (missingDataDirectiveFlag == 1 && errorCode == 0)
567             {
568                 errorCode = 4;
569                 errorFound = 1;
570             }
571             else if (missingStartFlag == 1 && errorCode == 0)
572             {
573                 errorCode = 5;
574                 errorFound = 1;
575             }
576             else if (missingEndFlag == 1 && errorCode == 0)
577             {
578                 errorCode = 6;
579                 errorFound = 1;
580             }
581             else if (tooManyLabelsFlag == 1 && errorCode == 0)
582             {
583                 errorCode = 7;
584                 errorFound = 1;
585             }
586             else if (programTooLongFlag == 1 && errorCode == 0)
587             {
588                 errorCode = 8;
589                 errorFound = 1;
590             }
591
592             /*Print to the intermediate file and symbol file */
593             fprintf(intermediate_file, "%X\t%s\t%s\t%s\t%d\n", locctr,
                  sourceFileTokenizer.label, sourceFileTokenizer.mnemonic,
                  sourceFileTokenizer.operand, errorCode);
594             fprintf(symbol_file, "%X\t%s\n", locctr, sourceFileTokenizer.label);
595
596             /* Search for the mnemonic in the operand table and add 3 to it */
597             for (int i = 0; i < numMnemonics; i++)
598             {
599                 if (strcmp(opcodeStructure[i].mnemonic,
                      sourceFileTokenizer.mnemonic) == 0)
600                 {
601                     locctr += 3;
602                 }
603             }
604
605             /*Update the memory locations after LOCCTR is printed in the file
                  */
606             if (strcmp(sourceFileTokenizer.mnemonic, "BYTE") == 0 || strcmp
                  (sourceFileTokenizer.mnemonic, "RESB") == 0 || strcmp
                  (sourceFileTokenizer.mnemonic, "RESW") == 0 || strcmp
                  (sourceFileTokenizer.mnemonic, "WORD") == 0)
```

```
607             {
608                 locctr += memLenght;
609             }
610 
611             /*  Increment the number of labels in the system    */
612             numberOfLabels++;
613         }
614 
615         /*If there is no label in the input line do the same as above but
              without labels */
616         else
617         {
618             /*  Tokenize the mnemonic into the structure    */
619             strcpy(sourceFileTokenizer.mnemonic, tokenizer);
620 
621             if (strcmp(sourceFileTokenizer.mnemonic, "RSUB") != 0)
622             {
623                 tokenizer = strtok(NULL, " \t\r\n\v\f");
624                 strcpy(sourceFileTokenizer.operand, tokenizer);
625 
626             }
627             else
628             {
629                 strcpy(sourceFileTokenizer.label, " ");
630                 strcpy(sourceFileTokenizer.operand, " ");
631             }
632 
633 
634             /*  Tokenize the operand into the structure */
635 
636 
637             /*  Check if we have a START directive in the beginning of the
                  program */
638             if (index == 0 && strcmp(sourceFileTokenizer.mnemonic, "START") !=
                  0)
639             {
640                 missingStartFlag = 1;
641                 locctr = 0;
642             }
643 
644             /*  Check if we have a END directive in the end of the program  */
645             if (missingEndFlag == 1 && errorCode == 0)
646             {
647                 if (strcmp(sourceFileTokenizer.mnemonic, "END") != 0)
648                 {
649                     missingEndFlag = 1;
650                 }
651             }
652 
```

```
653             /*  If directive START initialize LOCCTR to the starting address
                  */
654             if (strcmp(sourceFileTokenizer.mnemonic, "START") == 0) // if start
                  directive initialize locct to the start(convert the string to
                  integer)
655             {
656                 start = atoi(sourceFileTokenizer.operand);
657                 locctr = start;
658             }
659
660             /*  Check if program is too long    */
661             if (locctr > 32000)
662             {
663                 programTooLongFlag = 1;
664             }
665
666             /* Lenght size in memory from the directives to increment the LOCCTR
                    */
667             if (strcmp(sourceFileTokenizer.mnemonic, "WORD") == 0)
668             {
669                 memLenght += 3;
670             }
671             if (strcmp(sourceFileTokenizer.mnemonic, "RESB") == 0)
672             {
673                 memLenght += (int)strtol(sourceFileTokenizer.operand, NULL, 10);
674
675             }
676             if (strcmp(sourceFileTokenizer.mnemonic, "RESW") == 0)
677             {
678                 memLenght += 3 * (int)strtol(sourceFileTokenizer.operand, NULL,
                      10);
679             }
680
681             if (strcmp(sourceFileTokenizer.mnemonic, "BYTE") == 0)
682             {
683                 /*  Check if operand is set to read a string (C) or a
                      hexadecimal (X) */
684                 if (sourceFileTokenizer.operand[0] == 'C')
685                 {
686                     int bufferSpace = 0;
687                     int counter = 2;
688                     while (sourceFileTokenizer.operand[counter] != '\'' &&
                       bufferSpace < 30)
689                     {
690                         bufferSpace++;
691                         counter++;
692                     }
693                     memLenght += bufferSpace;
694
```

```
695                 }
696                 else if (sourceFileTokenizer.operand[0] == 'X')
697                 {
698                     char hexInput[16];
699                     int bufferSpace = 0;
700                     int counter = 3;
701                     while (sourceFileTokenizer.operand[counter] != '\'' &&
                        bufferSpace < 16)
702                     {
703                         hexInput[bufferSpace] = sourceFileTokenizer.operand
                        [counter];
704                         bufferSpace++;
705                         counter++;
706                     }
707 
708                     memLenght = (int)strtol(hexInput, NULL, 10);
709                 }
710 
711                 /* Check for errors in the input for the BYTE directive */
712                 else
713                 {
714                     illegalOperationFlag = 1;
715                 }
716 
717                 if (sourceFileTokenizer.operand[1] != '\'' ||
                      sourceFileTokenizer.operand[strlen
                      (sourceFileTokenizer.operand) - 1] != '\'')
718                 {
719                     missingDataDirectiveFlag = 1;
720                 }
721             }
722 
723             /* Error Flag conditions    */
724             if (duplicateLabelFlag == 1 && errorCode == 0)
725             {
726                 errorCode = 1;
727                 errorFound = 1;
728             }
729             else if (illegalLabelFlag == 1 && errorCode == 0)
730             {
731                 errorCode = 2;
732                 errorFound = 1;
733             }
734             else if (illegalOperationFlag == 1 && errorCode == 0)
735             {
736                 errorCode = 3;
737                 errorFound = 1;
738             }
739             else if (missingDataDirectiveFlag == 1 && errorCode == 0)
```

```
740             {
741                 errorCode = 4;
742                 errorFound = 1;
743             }
744             else if (missingStartFlag == 1 && errorCode == 0)
745             {
746                 errorCode = 5;
747                 errorFound = 1;
748             }
749             else if (missingEndFlag == 1 && errorCode == 0)
750             {
751                 errorCode = 6;
752                 errorFound = 1;
753             }
754             else if (tooManyLabelsFlag == 1 && errorCode == 0)
755             {
756                 errorCode = 7;
757                 errorFound = 1;
758             }
759             else if (programTooLongFlag == 1 && errorCode == 0)
760             {
761                 errorCode = 8;
762                 errorFound = 1;
763             }
764
765             /*Print to the intermediate file and symbol file */
766             fprintf(intermediate_file, "%X\t\t\t%s\t%s\t%d\n", locctr,
                  sourceFileTokenizer.mnemonic, sourceFileTokenizer.operand,
                  errorCode);
767
768             /* Search for the mnemonic in the operand table and add 3 to it */
769             for (int i = 0; i < numMnemonics; i++)
770             {
771                 if (strcmp(opcodeStructure[i].mnemonic,
                      sourceFileTokenizer.mnemonic) == 0)
772                 {
773                     locctr += 3;
774                 }
775             }
776
777             /*Update the memory locations after LOCCTR is printed in the file
                  */
778             if (strcmp(sourceFileTokenizer.mnemonic, "BYTE") == 0 || strcmp
                  (sourceFileTokenizer.mnemonic, "RESB") == 0 || strcmp
                  (sourceFileTokenizer.mnemonic, "RESW") == 0 || strcmp
                  (sourceFileTokenizer.mnemonic, "WORD") == 0)
779             {
780                 locctr += memLenght;
781             }
```

```
782         }
783         index++;
784     }
785
786     programLenght = locctr - start;
787     programLenght = programLenght - 4;
788
789     fprintf(intermediate_file, "\n\n\t Printing Error Code List: \n\n");
790     fprintf(intermediate_file,
          "*================================================*\n");
791     fprintf(intermediate_file, "\tNo Error = 0\n");
792     fprintf(intermediate_file, "\tDuplicate Label = 1\n");
793     fprintf(intermediate_file, "\tIllegal Label = 2\n");
794     fprintf(intermediate_file, "\tIllegal Operation = 3\n");
795     fprintf(intermediate_file, "\tIllegal Data Storage Directive = 4\n");
796     fprintf(intermediate_file, "\tMissing START Directive = 5\n");
797     fprintf(intermediate_file, "\tMissing END Directive = 6\n");
798     fprintf(intermediate_file, "\tToo Many Symbols = 7\n");
799     fprintf(intermediate_file, "\tProgram Too Long = 8\n");
800     fprintf(intermediate_file,
          "*================================================*\n");
801
802     fclose(intermediate_file);
803     fclose(source_file);
804     fclose(symbol_file);
805 }
806 void passTwo(char *param1)
807 {
808     char input[100];
809     char sourceInput[500];
810
811     char *tokenizer;
812     char *objectCode_string;
813     char *errorMessage;
814
815     int startingAddress;
816     int operandAddress;
817     int objectCode_decimal;
818     int objectLineLenght = 0;
819
820     int newLineFlag = 1;
821     int labelPresentFlag = 0;
822
823     FILE *intermediateFile, *symbolTable, *objectFile, *listingFile,
          *sourceFile;
824     TOKEN intermediateFileTokenizer;
825
826     objectFile = fopen("objectFile.txt", "w");
827     listingFile = fopen("listingFile.txt", "w");
```

```
828     intermediateFile = fopen("intermediate.txt", "r");
829     symbolTable = fopen("symbolTable.txt", "r");
830     sourceFile = fopen(param1, "r");
831
832     if (sourceFile == NULL)
833     {
834         printf("Intermediate file did not opened correctly \n");
835         return;
836     }
837
838     intermediateFileTokenizer.label = (char *)malloc(6);
839     intermediateFileTokenizer.mnemonic = (char *)malloc(6);
840     intermediateFileTokenizer.operand = (char *)malloc(6);
841     errorMessage = (char *)malloc(256);
842
843     while (fgets(input, 100, intermediateFile))
844     {
845         memset(intermediateFileTokenizer.label, '\0', 6);
846         memset(intermediateFileTokenizer.mnemonic, '\0', 6);
847         memset(intermediateFileTokenizer.operand, '\0', 6);
848         memset(errorMessage, '\0', 256);
849
850         fgets(sourceInput, 500, sourceFile);
851
852         /*  Check it the source line is a comment   */
853         if (sourceInput[0] == '.')
854         {
855             while (sourceInput[0] == '.')
856             {
857                 fprintf(listingFile, "%s", sourceInput);
858                 fgets(sourceInput, 500, sourceFile);
859             }
860         }
861
862         labelPresentFlag = 0;
863
864         tokenizer = strtok(input, "\t");
865         intermediateFileTokenizer.memoryAddress = (int)strtol(tokenizer, NULL,
            16);   ////save address
866         tokenizer = strtok(NULL, "\t");
867
868         for (int i = 0; i < numberOfLabels; i++)
869         {
870             if (strcmp(labelStructure[i].label, tokenizer) == 0)
871             {
872                 labelPresentFlag = 1;
873                 break;
874             }
875         }
```

```
876
877         if (labelPresentFlag == 1)
878         {
879             strcpy(intermediateFileTokenizer.label, tokenizer); ////save label
880             tokenizer = strtok(NULL, "\t");
881         }
882
883         strcpy(intermediateFileTokenizer.mnemonic, tokenizer); ////save mnemonic
884
885         if (strcmp(intermediateFileTokenizer.mnemonic, "RSUB") != 0)
886         {
887             tokenizer = strtok(NULL, "\t");
888             strcpy(intermediateFileTokenizer.operand, tokenizer); /////save
                operand
889             tokenizer = strtok(NULL, " \t");
890             intermediateFileTokenizer.errorCode = (int)strtol(tokenizer, NULL,
                10); ////save errorcode
891         }
892         else
893         {
894             tokenizer = strtok(NULL, " \t");
895             intermediateFileTokenizer.errorCode = (int)strtol(tokenizer, NULL,
                10); ////save errorcode
896             objectCode_string = "4C0000";
897
898             if (newLineFlag == 1)
899             {
900                 fprintf(objectFile, "\n");
901                 fprintf(objectFile, "T%s", objectCode_string);
902                 newLineFlag = 0;
903                 objectLineLenght++;
904             }
905             else
906             {
907                 fprintf(objectFile, "%s", objectCode_string);
908                 objectLineLenght++;
909             }
910
911             if (intermediateFileTokenizer.errorCode == 0)
912             {
913                 fprintf(listingFile, "%X\t%s\t%s",
                      intermediateFileTokenizer.memoryAddress, objectCode_string,
                      sourceInput);
914                 continue;
915             }
916             else
917             {
918                 printError(&errorMessage, intermediateFileTokenizer.errorCode);
919                 fprintf(listingFile, "%s\n", errorMessage);
```

```
920                 continue;
921             }
922         }
923 
924         /*  Check if the object file size limit has been reached    */
925         if (objectLineLenght == 10)
926         {
927             newLineFlag = 1;
928             objectLineLenght = 0;
929         }
930 
931         /*  Check it the intermidiate line is a START   */
932         if (strcmp(intermediateFileTokenizer.mnemonic, "START") == 0 ||
              intermediateFileTokenizer.errorCode == 5)
933         {
934             fprintf(objectFile, "H%_%s%06X%06X",
                      intermediateFileTokenizer.label,
                      intermediateFileTokenizer.memoryAddress, programLenght);
935 
936             if (intermediateFileTokenizer.errorCode == 0)
937             {
938                 fprintf(listingFile, "%X\t\t%s",
                          intermediateFileTokenizer.memoryAddress, sourceInput);
939             }
940             else
941             {
942                 printError(&errorMessage, intermediateFileTokenizer.errorCode);
943                 fprintf(listingFile, "%s\n", errorMessage);
944             }
945             startingAddress = intermediateFileTokenizer.memoryAddress;
946         }
947 
948         /*  Check it the intermidiate line is a RESW    */
949         else if (strcmp(intermediateFileTokenizer.mnemonic, "RESW") == 0 ||
                  strcmp(intermediateFileTokenizer.mnemonic, "RESB") == 0 ||
                  intermediateFileTokenizer.errorCode == 4)
950         {
951             if (intermediateFileTokenizer.errorCode == 0)
952             {
953                 fprintf(listingFile, "%X\t\t%s",
                          intermediateFileTokenizer.memoryAddress, sourceInput);
954             }
955             else
956             {
957                 printError(&errorMessage, intermediateFileTokenizer.errorCode);
958                 fprintf(listingFile, "%s\n", errorMessage);
959             }
960         }
961 
```

```
962         /*  Check it the intermidiate line is a WORD    */
963         else if (strcmp(intermediateFileTokenizer.mnemonic, "WORD") == 0 ||
              intermediateFileTokenizer.errorCode == 4)
964         {
965             objectCode_string = intermediateFileTokenizer.operand;
966             objectCode_decimal = (int)strtol(objectCode_string, NULL, 10);
967             if (strcmp(intermediateFileTokenizer.operand, "0") == 0)
968             {
969                 fprintf(objectFile, "%06X", objectCode_decimal);
970                 newLineFlag = 1;
971                 objectLineLenght = 0;
972
973                 if (intermediateFileTokenizer.errorCode == 0)
974                 {
975                     fprintf(listingFile, "%X\t%06X\t%s",
                         intermediateFileTokenizer.memoryAddress, objectCode_decimal,
                          sourceInput);
976                     continue;
977                 }
978                 else
979                 {
980                     printError(&errorMessage,
                         intermediateFileTokenizer.errorCode);
981                     fprintf(listingFile, "%s\n", errorMessage);
982                     continue;
983                 }
984             }
985             else
986             {
987                 if (intermediateFileTokenizer.errorCode == 0)
988                 {
989                     fprintf(listingFile, "%X\t%06X\t%s",
                         intermediateFileTokenizer.memoryAddress, objectCode_decimal,
                          sourceInput);
990                 }
991                 else
992                 {
993                     printError(&errorMessage,
                         intermediateFileTokenizer.errorCode);
994                     fprintf(listingFile, "%s\n", errorMessage);
995                 }
996             }
997
998             if (newLineFlag == 1)
999             {
1000                fprintf(objectFile, "\n");
1001                fprintf(objectFile, "T%06X%06X",
                      intermediateFileTokenizer.memoryAddress, objectCode_decimal);
1002                newLineFlag = 0;
```

```
1003                objectLineLenght++;
1004            }
1005            else
1006            {
1007                fprintf(objectFile, "%06X", objectCode_decimal);
1008                objectLineLenght++;
1009            }
1010        }
1011
1012        /*  Check it the intermidiate line is a BYTE    */
1013        else if (strcmp(intermediateFileTokenizer.mnemonic, "BYTE") == 0 ||
              intermediateFileTokenizer.errorCode == 3)
1014        {
1015            if (intermediateFileTokenizer.operand[0] == 'C')
1016            {
1017                char copyHEX[10];
1018                char convertedHEX[10];
1019
1020                int inputIndex = 2;
1021                int outputIndex = 0;
1022
1023                while (intermediateFileTokenizer.operand[inputIndex] != '\'')
1024                {
1025                    copyHEX[outputIndex] = intermediateFileTokenizer.operand
                        [inputIndex];
1026                    inputIndex++;
1027                    outputIndex++;
1028                }
1029
1030                sprintf(convertedHEX, "%X%X%X", copyHEX[0], copyHEX[1], copyHEX
                      [2]);
1031
1032                if (intermediateFileTokenizer.errorCode == 0)
1033                {
1034                    fprintf(listingFile, "%X\t%s\t%s",
                        intermediateFileTokenizer.memoryAddress, convertedHEX,
                        sourceInput);
1035                }
1036                else
1037                {
1038                    printError(&errorMessage,
                        intermediateFileTokenizer.errorCode);
1039                    fprintf(listingFile, "%s\n", errorMessage);
1040                }
1041
1042                if (newLineFlag == 1)
1043                {
1044                    fprintf(objectFile, "\n");
1045                    fprintf(objectFile, "T%06X%s",
```

```
                        intermediateFileTokenizer.memoryAddress, convertedHEX);
1046                    newLineFlag = 0;
1047                    objectLineLenght++;
1048
1049                }
1050                else
1051                {
1052                    fprintf(objectFile, "%s", convertedHEX);
1053                    objectLineLenght++;
1054                }
1055            }
1056            else if (intermediateFileTokenizer.operand[0] == 'X')
1057            {
1058                char copyHEX[10];
1059
1060                int inputIndex = 2;
1061                int outputIndex = 0;
1062
1063                while (intermediateFileTokenizer.operand[inputIndex] != '\'')
1064                {
1065                    copyHEX[outputIndex] = intermediateFileTokenizer.operand
                        [inputIndex];
1066                    inputIndex++;
1067                    outputIndex++;
1068                }
1069                copyHEX[outputIndex] = '\0';
1070
1071                if (intermediateFileTokenizer.errorCode == 0)
1072                {
1073                    fprintf(listingFile, "%X\t%s\t%s",
                        intermediateFileTokenizer.memoryAddress, copyHEX,
                        sourceInput);
1074                }
1075                else
1076                {
1077                    printError(&errorMessage,
                        intermediateFileTokenizer.errorCode);
1078                    fprintf(listingFile, "%s\n", errorMessage);
1079                }
1080
1081                if (newLineFlag == 1)
1082                {
1083                    fprintf(objectFile, "\n");
1084                    fprintf(objectFile, "T%06X%s",
                        intermediateFileTokenizer.memoryAddress, copyHEX);
1085                    newLineFlag = 0;
1086                    objectLineLenght++;
1087                }
1088                else
```

```
1089                {
1090                    fprintf(objectFile, "%s", copyHEX);
1091                    objectLineLenght++;
1092                }
1093            }
1094            else
1095            {
1096                printError(&errorMessage, intermediateFileTokenizer.errorCode);
1097                fprintf(listingFile, "%s\n", errorMessage);
1098            }
1099        }
1100
1101        /*  Check it the intermidiate line is a END */
1102        else if (strcmp(intermediateFileTokenizer.mnemonic, "END") == 0)
1103        {
1104            operandAddress = searchLabelLocation
                  (intermediateFileTokenizer.operand);
1105            fprintf(objectFile, "\n");
1106            fprintf(objectFile, "E%06X", operandAddress);
1107
1108            if (intermediateFileTokenizer.errorCode == 0)
1109            {
1110                fprintf(listingFile, "%s", sourceInput);
1111            }
1112            else
1113            {
1114                printError(&errorMessage, intermediateFileTokenizer.errorCode);
1115                fprintf(listingFile, "%s\n", errorMessage);
1116            }
1117            break;
1118        }
1119
1120        /*  Else it is a regular mnemonic and just requieres normal handling
              */
1121        else
1122        {
1123            char *OpcodeExtracted;
1124            int OpcodeConverted;
1125            char objectCode[10];
1126
1127            operandAddress = searchLabelLocation
                  (intermediateFileTokenizer.operand);
1128
1129            for (int i = 0; i < numMnemonics; i++)
1130            {
1131                if (strcmp(opcodeStructure[i].mnemonic,
                      intermediateFileTokenizer.mnemonic) == 0)
1132                {
1133                    OpcodeExtracted = opcodeStructure[i].opcode;
```

```
1134                        break;
1135                    }
1136                }
1137
1138                OpcodeConverted = (int)strtol(OpcodeExtracted, NULL, 16);
1139                sprintf(objectCode, "%02X%04X", OpcodeConverted, operandAddress);
1140
1141                if (intermediateFileTokenizer.errorCode == 0)
1142                {
1143                    fprintf(listingFile, "%X\t%06s\t%s",
                          intermediateFileTokenizer.memoryAddress, objectCode,
                          sourceInput);
1144                }
1145                else
1146                {
1147                    printError(&errorMessage, intermediateFileTokenizer.errorCode);
1148                    fprintf(listingFile, "%s\n", errorMessage);
1149                }
1150
1151                if (newLineFlag == 1)
1152                {
1153                    fprintf(objectFile, "\n");
1154                    fprintf(objectFile, "T%06X%06s",
                          intermediateFileTokenizer.memoryAddress, objectCode);
1155                    newLineFlag = 0;
1156                    objectLineLenght++;
1157                }
1158                else
1159                {
1160                    fprintf(objectFile, "%06s", objectCode);
1161                    objectLineLenght++;
1162                }
1163            }
1164        }
1165
1166        /*  Close all files */
1167        fclose(objectFile);
1168        fclose(listingFile);
1169        fclose(intermediateFile);
1170        fclose(symbolTable);
1171
1172        /*  Check if there were errors on Pass 1 if so delete the object file   */
1173        if (errorFound == 1)
1174        {
1175            if (remove("objectFile.txt") == 0)
1176            {
1177                printf("Program Has Errors.\n");
1178            }
1179        }
```

```
1180  }
1181  void fetchObj2Memory()
1182  {
1183      char input[100];
1184      int inputLenght;
1185
1186      char programName_string[10];
1187      char programStart_string[6];
1188      char programEnd_string[6];
1189      char programLenght_string[6];
1190      int programStart_int;
1191      int programLenght_int;
1192
1193      char objectLineAddress_string[6];
1194      char objectLineLenght_string[2];
1195      char objectLineCode_string[6];
1196      int objectLineAddress_int = 0;
1197      int objectLineLenght_int;
1198      int objectLineCode_int;
1199
1200      int index = 1;
1201      BYTE value;
1202
1203      FILE * object_file = fopen("objectFile_Working.txt", "r");
1204
1205      fgets(input, 100, object_file);
1206      for (int i = 0; input[index] != '_'; i++) // get name of program
1207      {
1208          programName_string[i] = input[index];
1209          index++;
1210      }
1211      index++;
1212
1213      for (int i = 0; i < 6; i++)                 // get starting address of
            program
1214      {
1215          programStart_string[i] = input[index];
1216          index++;
1217      }
1218      programStart_int = (int)strtol(programStart_string, NULL, 16);
1219
1220      for (int i = 0; i < 6; i++)                 // get lenght of program
1221      {
1222          programLenght_string[i] = input[index];
1223          index++;
1224      }
1225      programLenght_int = (int)strtol(programLenght_string, NULL, 16);
1226
1227      index = 1;
```

```
1228      fgets(input, 100, object_file);
1229
1230      while (input[0] != 'E')
1231      {
1232          for (int i = 0; i < 6; i++) // get the memory location of the record
1233          {
1234              objectLineAddress_string[i] = input[index];
1235              index++;
1236          }
1237          objectLineAddress_int = (int)strtol(objectLineAddress_string, NULL, 16);
1238
1239          for (int i = 0; i < 2; i++) // get the lenght of the record
1240          {
1241              objectLineLenght_string[i] = input[index];
1242              index++;
1243          }
1244          objectLineLenght_int = (int)strtol(objectLineLenght_string, NULL, 16);
1245
1246          inputLenght = strlen(input) - 1;
1247          while (index < inputLenght)
1248          {
1249              for (int i = 0; i < 2; i++) //get the object code a BYTE at a time
1250              {
1251                  objectLineCode_string[i] = input[index];
1252                  index++;
1253              }
1254              objectLineCode_int = (int)strtol(objectLineCode_string, NULL, 16);
1255
1256              value = (BYTE)objectLineCode_int; //convert the object code to type
                    BYTE
1257
1258              PutMem(objectLineAddress_int, &value, 0);
1259              objectLineAddress_int++;
1260          }
1261
1262          fgets(input, 100, object_file);
1263          index = 1;
1264      }
1265
1266      for (int i = 0; i < 6; i++) // get the ending record to store the starting
          address
1267      {
1268          programEnd_string[i] = input[index];
1269          index++;
1270      }
1271      programEnd_int = (int)strtol(programEnd_string, NULL, 16);
1272  }
1273
```