

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <ctype.h>
5
6  /*
7      System's Programming Phase 2 Algorithm
8
9      Author: Jesus M. Morales
10     Due Date: 3/25/2018
11
12
13 */
14
15 typedef struct
16 {
17     char label[10];
18     int memoryAddress;
19 } LABELS;
20
21 typedef struct
22 {
23     char *label;
24     char *mnemonic;
25     char *opcode;
26 } TOKEN;
27
28 typedef struct
29 {
30     char mnemonic[5];
31     int opcode;
32 } OPCODE;
33
34 void breakupLine(char *input, char *command, char *param1, char *param2, int ↗
    *numParams);
35 void loadFile(char *fileName);
36 void passOne(char * fileName);
37 void passTwo();
38 void executeFile();
39 void debugFile();
40 void dumpFile();
41 void helpFile();
42 void assembleFile();
43 void errorFile();
44 int programLenght;
45
46 int main(void)
47 {
48     char input[50];
```

```
49     char command[50];
50     char param1[50];
51     char param2[50];
52
53     printf("Hello welcome to Jesus Morales Personal Assembler\n\n");
54     while (1)
55     {
56         int numParams = 0;
57         int len = 0;
58         printf("Command ----> ");
59         fgets(input, 50, stdin);
60
61         len = strlen(input) - 1;
62         if (input[len] == '\n')
63         {
64             input[len] = '\0';
65         }
66
67         breakupLine(input, command, param1, param2, &numParams);
68         numParams--;
69
70         if (strcmp(command, "load") == 0)
71         {
72             if (numParams == 1)
73             {
74                 loadFile(param1);
75             }
76             else
77                 errorFile();
78         }
79         else if (strcmp(command, "execute") == 0)
80         {
81             executeFile();
82         }
83         else if (strcmp(command, "debug") == 0)
84         {
85             debugFile();
86         }
87         else if (strcmp(command, "dump") == 0)
88         {
89             if (numParams == 2)
90             {
91                 dumpFile();
92             }
93             else
94                 errorFile();
95         }
96         else if (strcmp(command, "help") == 0)
97         {
```

```
198         helpFile();
199     }
200     else if (strcmp(command, "assemble") == 0)
201     {
202         if (numParams == 1)
203         {
204             assembleFile();
205         }
206         else
207             errorFile();
208     }
209     else if (strcmp(command, "dir") == 0)
210     {
211         system("dir");
212     }
213     else if (strcmp(command, "exit") == 0)
214     {
215         break;
216     }
217     else
218     {
219         printf("Invalid Command , for any help type 'help' to display the  ↗
220             command list. \n \n");
221     }
222     numParams = 0;
223 }
224 return 0;
225 }
226
227 void breakupLine(char *input, char *command, char *param1, char *param2, int  ↗
228     *numParams)
229 {
230     command[0] = param1[0] = param2[0] = '\0';
231     *numParams = sscanf(input, "%s %s %s %*s", command, param1, param2);
232 }
233
234 void loadFile(char *param1)
235 {
236     printf("Loading file: %s\n", param1);
237     passOne(param1);
238     passTwo();
239     printf("The Programg lenght of this file is:  %d Bytes\n\n", programLenght);
240     programLenght = 0;
241 }
242 void executeFile()
243 {
244     printf(" is not yet avaibalbe.\n");
```

```
145 }
146 void debugFile()
147 {
148     printf("debug is not avaialabe.\n");
149 }
150 void dumpFile()
151 {
152     printf("dump is not avaiblable.\n");
153 }
154 void helpFile()
155 {
156     printf("\n");
157     printf("\tWelcome to the Help menu. \n");
158     printf("\tCommands are the following: \n \n");
159     printf("\tload [file_name]\n");
160     printf("\texecute \n");
161     printf("\tdebug \n");
162     printf("\tdump [start] [end] \n");
163     printf("\thelp \n");
164     printf("\tassemble [file_name] \n");
165     printf("\tdirectory \n");
166     printf("\texit \n\n");
167     printf("\t**ALL COMMANDS ARE CASE SENSITIVE.**\n\n");
168 }
169 void assembleFile()
170 {
171     printf("assemble not avaibalbe. \n");
172 }
173 void errorFile()
174 {
175     printf("You typed the wrong number of parameters try again. \n");
176 }
177 void passOne(char *param1)
178 {
179     char input[500];
180     char *tokenizer = input;
181
182     int start = 0;
183     int locctr = 0;
184     int memLenght = 0;
185     int numLabels = 0;
186     int numMnemonics = 25;
187     int index = 0;
188
189     int labelPresentFlag = 0;
190     int duplicateLabelFlag = 0;
191     int illegalLabelFlag = 0;
192     int illegalOperationFlag = 0;
193     int missingDataDirectiveFlag = 0;
```

```

194     int missingStartFlag = 0;
195     int missingEndFlag = 0;
196     int tooManyLabelsFlag = 0;
197     int programTooLongFlag = 0;
198     int errorCode = 0;
199
200     FILE *source_file, *symbol_file, *intermediate_file, *opcode_file;
201     LABELS labelStructure[500];
202     TOKEN tokenStructure;
203     OPCODE opcodeStructure[] = { { "ADD", 0x18 }, { "AND", 0x58 }, { "COMP", 0x28 },
    { "DIV", 0x24 },
204                                     { "J", 0x3C }, { "JEQ", 0x30 }, { "JGT", 0x34 },
    { "JLT", 0x38 },
205                                     { "JSUB", 0x48 }, { "LDA", 0x00 }, { "LDCH",
    0x50 }, { "LDL", 0x08 },
206                                     { "LDX", 0x04 }, { "MUL", 0x20 }, { "OR", 0x44 },
    { "RD", 0xD8 },
207                                     { "RSUB", 0x4C }, { "STA", 0x0C }, { "STCH",
    0x54 }, { "STL", 0x14 },
208                                     { "STX", 0x10 }, { "SUB", 0x1C }, { "TD", 0xE0 },
    { "TIX", 0x2C }, { "WD", 0xDC } };
209
210     source_file = fopen(param1, "r");
211     intermediate_file = fopen("intermediate.txt", "w");
212     symbol_file = fopen("symbolTable.txt", "w");
213
214     if (source_file == NULL)
215     {
216         printf("Error opening file does not exist: %s\n", param1);
217         return;
218     }
219
220     tokenStructure.label = (char *)malloc(6);
221     tokenStructure.mnemonic = (char *)malloc(6);
222     tokenStructure.opcode = (char *)malloc(6);
223
224     while (fgets(input, 500, source_file))
225     {
226         labelPresentFlag = 0;
227         duplicateLabelFlag = 0;
228         illegalLabelFlag = 0;
229         illegalOperationFlag = 0;
230         missingDataDirectiveFlag = 0;
231         missingStartFlag = 0;
232         missingEndFlag = 0;
233         tooManyLabelsFlag = 0;
234         programTooLongFlag = 0;
235         errorCode = 0;
236         memLenght = 0;

```

```
237
238     /* Check if label is present in the string line */
239     if (input[0] == ' ' || input[0] == '\t')
240     {
241         labelPresentFlag = 0;
242     }
243     else
244     {
245         labelPresentFlag = 1;
246     }
247
248     /* Check if comment is present in the string line */
249     if (input[0] == '.')
250     {
251         continue;
252     }
253
254     /* Tokenize the input string */
255     tokenizer = strtok(input, " \t\r\n\v\f");
256
257     /* Remove of the trailing newLine at the end of the string */
258     int counter = 0;
259     while (input[counter - 1] != '\n')
260     {
261         counter++;
262     }
263     input[counter] = '\0';
264
265     /* If there is a label */
266     if (labelPresentFlag == 1)
267     {
268         /* Tokenize the label into the structure */
269         strcpy(tokenStructure.label, tokenizer);
270
271         /* Tokenize the mnemonic into the structure */
272         tokenizer = strtok(NULL, " \t\r\n\v\f");
273         strcpy(tokenStructure.mnemonic, tokenizer);
274
275         /* Tokenize the opcode of the mnemonic into the structure */
276         tokenizer = strtok(NULL, " \t\r\n\v\f");
277         strcpy(tokenStructure.opcode, tokenizer);
278
279         /* Add the labels to the structure to create a list of existing
280            labels/symbols */
281         strcpy(labelStructure[numLabels].label, tokenStructure.label);
282         labelStructure[numLabels].memoryAddress = locctr;
283
284         /* Check if there are labels in the list */
285         if (numLabels > 0)
```

```
285     {
286         /* Check if limit of labels has been reached */
287         if (numLabels > 500)
288         {
289             tooManyLabelsFlag = 1;
290         }
291
292         /* Inefficiently scan the label/symbol list to check for
293            duplicate labels/symbols */
294         for (int i = 0; i < numLabels; i++)
295         {
296             if (strcmp(labelStructure[i].label, tokenStructure.label) ==
297                 0)
298             {
299                 duplicateLabelFlag = 1;
300             }
301         }
302
303         /* Check if the label is legal */
304         if (!isalpha(tokenStructure.label[0]))
305         {
306             illegalLabelFlag = 1;
307         }
308
309         /* Check if we have a START directive in the beginning of the
310            program */
311         if (index == 0 && strcmp(tokenStructure.mnemonic, "START") != 0)
312         {
313             missingStartFlag = 1;
314             locctr = 0;
315         }
316
317         /* Check if we have a END directive in the end of the program */
318         if (missingEndFlag == 1 && errorCode == 0)
319         {
320             if (strcmp(tokenStructure.mnemonic, "END") != 0)
321             {
322                 missingEndFlag = 1;
323             }
324         }
325
326         /* If directive START initialize LOCCTR to the starting address
327            */
328         if (strcmp(tokenStructure.mnemonic, "START") == 0) // if start
329             directive initialize locct to the start(convert the string to
330             integer)
331         {
332             start = atoi(tokenStructure.opcode);
```

```
328         locctr = start;
329     }
330
331     /* Check if program is too long */
332     if (locctr > 6700)
333     {
334         programTooLongFlag = 1;
335     }
336     /* Lenght size in memory from the directives to increment the LOCCTR ↗
337        */
338     if (strcmp(tokenStructure.mnemonic, "WORD") == 0)
339     {
340         memLenght += 3;
341     }
342     if (strcmp(tokenStructure.mnemonic, "RESB") == 0)
343     {
344         memLenght += atoi(tokenStructure.opcode);
345     }
346     if (strcmp(tokenStructure.mnemonic, "RESW") == 0)
347     {
348         memLenght += 3 * atoi(tokenStructure.opcode);
349     }
350
351     if (strcmp(tokenStructure.mnemonic, "BYTE") == 0)
352     {
353         /* Check if operand is set to read a string (C) or a hexadecimal ↗
354            (X) */
355         if (tokenStructure.opcode[0] == 'C')
356         {
357             int bufferSize = 0;
358             int counter = 2;
359             while (tokenStructure.opcode[counter] != '\\' && bufferSize ↗
360                 < 30)
361             {
362                 bufferSize++;
363                 counter++;
364             }
365             memLenght += bufferSize;
366         }
367         else if (tokenStructure.opcode[0] == 'X')
368         {
369             char hexInput[16];
370             int bufferSize = 0;
371             int counter = 2;
372             while (tokenStructure.opcode[counter] != '\\' && bufferSize ↗
373                 < 16)
```



```
373         {
374             hexInput[bufferSpace] = tokenStructure.opcode[counter];
375             bufferSpace++;
376             counter++;
377         }
378         hexInput[bufferSpace] = '\\0';
379         memLenght = (int)strtol(hexInput, NULL, 16);
380     }
381
382     /* Check for errors in the input for the BYTE directive */
383     else
384     {
385         illegalOperationFlag = 1;
386     }
387
388     if (tokenStructure.opcode[1] != '\\'' || tokenStructure.opcode
389         [strlen(tokenStructure.opcode) - 1] != '\\')
390     {
391         missingDataDirectiveFlag = 1;
392     }
393
394     /* Error Flag conditions */
395     if (duplicateLabelFlag == 1 && errorCode == 0)
396     {
397         errorCode = 1;
398     }
399     else if (illegalLabelFlag == 1 && errorCode == 0)
400     {
401         errorCode = 2;
402     }
403     else if (illegalOperationFlag == 1 && errorCode == 0)
404     {
405         errorCode = 3;
406     }
407     else if (missingDataDirectiveFlag == 1 && errorCode == 0)
408     {
409         errorCode = 4;
410     }
411     else if (missingStartFlag == 1 && errorCode == 0)
412     {
413         errorCode = 5;
414     }
415     else if (missingEndFlag == 1 && errorCode == 0)
416     {
417         errorCode = 6;
418     }
419     else if (tooManyLabelsFlag == 1 && errorCode == 0)
420     {
```

```

421         errorCode = 7;
422     }
423     else if (programTooLongFlag == 1 && errorCode == 0)
424     {
425         errorCode = 8;
426     }
427
428     /*Print to the intermediate file and symbol file */
429     fprintf(intermediate_file, "%d\t%s\t%s\t%s\t%d\n", locctr,
430         tokenStructure.label, tokenStructure.mnemonic,
431         tokenStructure.opcode, errorCode);
432     fprintf(symbol_file, "%d\t %s\n", locctr, tokenStructure.label);
433
434     /* Search for the mnemonic in the opcode table and add 3 to it */
435     for (int i = 0; i < numMnemonics; i++)
436     {
437         if (strcmp(opcodeStructure[i].mnemonic, tokenStructure.mnemonic)
438             == 0)
439         {
440             locctr += 3;
441         }
442     }
443
444     /*Update the memory locations after LOCCTR is printed in the file
445     */
446     if (strcmp(tokenStructure.mnemonic, "BYTE") == 0 || strcmp
447         (tokenStructure.mnemonic, "RESB") == 0 || strcmp
448         (tokenStructure.mnemonic, "RESW") == 0 || strcmp
449         (tokenStructure.mnemonic, "WORD") == 0)
450     {
451         locctr += memLenght;
452     }
453
454     /* Increment the number of labels in the system */
455     numLabels++;
456 }
457
458 /*If there is no label in the input line do the same as above but without
459 labels */
460 else
461 {
462     /* Tokenize the mnemonic into the structure */
463     strcpy(tokenStructure.mnemonic, tokenizer);
464
465     /* Tokenize the opcode into the structure */
466     tokenizer = strtok(NULL, " \t\r\n\v\f");
467     strcpy(tokenStructure.opcode, tokenizer);
468
469     /* Check if we have a START directive in the beginning of the

```

```
        program */
462     if (index == 0 && strcmp(tokenStructure.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(tokenStructure.mnemonic, "END") != 0)
472         {
473             missingEndFlag = 1;
474         }
475     }
476
477     /* If directive START initialize LOCCTR to the starting address */
478     /* if start directive initialize locct to the start(convert the string to integer) */
479     {
480         start = atoi(tokenStructure.opcode);
481         locctr = start;
482     }
483
484     /* Check if program is too long */
485     if (locctr > 32000)
486     {
487         programTooLongFlag = 1;
488     }
489
490     /* Lenght size in memory from the directives to increment the LOCCTR */
491     /* */
492     if (strcmp(tokenStructure.mnemonic, "WORD") == 0)
493     {
494         memLenght += 3;
495     }
496     if (strcmp(tokenStructure.mnemonic, "RESB") == 0)
497     {
498         memLenght += atoi(tokenStructure.opcode);
499     }
500     if (strcmp(tokenStructure.mnemonic, "RESW") == 0)
501     {
502         memLenght += 3 * atoi(tokenStructure.opcode);
503     }
504
505     if (strcmp(tokenStructure.mnemonic, "BYTE") == 0)
```

```
506     {
507         /* Check if operand is set to read a string (C) or a hexadecimal ↗
508            (X) */
509         if (tokenStructure.opcode[0] == 'C')
510         {
511             int bufferSpace = 0;
512             int counter = 2;
513             while (tokenStructure.opcode[counter] != '\\' && bufferSpace ↗
514                 < 30)
515             {
516                 bufferSpace++;
517                 counter++;
518             }
519             memLenght += bufferSpace;
520         }
521         else if (tokenStructure.opcode[0] == 'X')
522         {
523             char hexInput[16];
524             int bufferSpace = 0;
525             int counter = 2;
526             while (tokenStructure.opcode[counter] != '\\' && bufferSpace ↗
527                 < 16)
528             {
529                 hexInput[bufferSpace] = tokenStructure.opcode[counter];
530                 bufferSpace++;
531                 counter++;
532             }
533             hexInput[bufferSpace] = '\\0';
534             memLenght = (int)strtol(hexInput, NULL, 16);
535         }
536         /* Check for errors in the input for the BYTE directive */
537         else
538         {
539             illegalOperationFlag = 1;
540         }
541     }
542     if (tokenStructure.opcode[1] != '\\' || tokenStructure.opcode ↗
543         [strlen(tokenStructure.opcode) - 1] != '\\')
544     {
545         missingDataDirectiveFlag = 1;
546     }
547 }
548 /* Error Flag conditions */
549 if (duplicateLabelFlag == 1 && errorCode == 0)
550 {
```

```

551         errorCode = 1;
552     }
553     else if (illegalLabelFlag == 1 && errorCode == 0)
554     {
555         errorCode = 2;
556     }
557     else if (illegalOperationFlag == 1 && errorCode == 0)
558     {
559         errorCode = 3;
560     }
561     else if (missingDataDirectiveFlag == 1 && errorCode == 0)
562     {
563         errorCode = 4;
564     }
565     else if (missingStartFlag == 1 && errorCode == 0)
566     {
567         errorCode = 5;
568     }
569     else if (missingEndFlag == 1 && errorCode == 0)
570     {
571         errorCode = 6;
572     }
573     else if (tooManyLabelsFlag == 1 && errorCode == 0)
574     {
575         errorCode = 7;
576     }
577     else if (programTooLongFlag == 1 && errorCode == 0)
578     {
579         errorCode = 8;
580     }
581
582     /*Print to the intermediate file and symbol file */
583     fprintf(intermediate_file, "%d\t\t\t%s\t%s\t%d\n", locctr, tokenStructure.mnemonic, tokenStructure.opcode, errorCode);
584
585     /* Search for the mnemonic in the opcode table and add 3 to it */
586     for (int i = 0; i < numMnemonics; i++)
587     {
588         if (strcmp(opcodeStructure[i].mnemonic, tokenStructure.mnemonic) == 0)
589         {
590             locctr += 3;
591         }
592     }
593
594     /*Update the memory locations after LOCCTR is printed in the file */
595     if (strcmp(tokenStructure.mnemonic, "BYTE") == 0 || strcmp(tokenStructure.mnemonic, "RESB") == 0 || strcmp

```

```
(tokenStructure.mnemonic, "RESW") == 0 || strcmp
(tokenStructure.mnemonic, "WORD") == 0)
596     {
597         locctr += memLenght;
598     }
599 }
600     index++;
601 }
602
603     programLenght = locctr - start;
604     printf("Pass One complete successfully. \n");
605
606     fprintf(intermediate_file, "\n\n\t Printing Error Code List: \n\n");
607     fprintf(intermediate_file,
608         "*=====*\n");
609     fprintf(intermediate_file, "\tNo Error = 0\n");
610     fprintf(intermediate_file, "\tDuplicate Label = 1\n");
611     fprintf(intermediate_file, "\tIllegal Label = 2\n");
612     fprintf(intermediate_file, "\tIllegal Operation = 3\n");
613     fprintf(intermediate_file, "\tIllegal Data Storage Directive = 4\n");
614     fprintf(intermediate_file, "\tMissing START Directive = 5\n");
615     fprintf(intermediate_file, "\tMissing END Directive = 6\n");
616     fprintf(intermediate_file, "\tToo Many Symbols = 7\n");
617     fprintf(intermediate_file, "\tProgram Too Long = 8\n");
618     fprintf(intermediate_file,
619         "*=====*\n");
620
621     fclose(intermediate_file);
622     fclose(source_file);
623     fclose(symbol_file);
624 }
625 void passTwo()
626 {
627     printf("Pass Two is still in development. \n\n");
628 }
```